Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

an der Universität Hamburg eingereichte
kumulative Dissertation

Emerging Threats to Online Security: Securing Systems Against Unauthorized Automation and Web Bots

submitted by
Richard August See

Dissertation to receive the title *Dr. rer nat.*

Faculty of Mathematics, Informatics and Natural Sciences
Departments of Informatics
Computer Networks (NET) Group

Hamburg, March 25, 2025

Reviewers

Prof. Dr. Mathias Fischer
Prof. Serge Egelman, PhD

Disputation date: June 02, 2025

URN: urn:nbn:de:gbv:18-ediss-130048

Whoever said there's no such thing as a free lunch was never a grad student.

— Serge Egelman

Abstract

Internet services increasingly suffer from unwanted automation through bots, which pose significant challenges including financial losses, security breaches, and diminished user trust. The ability of bots to convincingly emulate human interactions complicates detection efforts, particularly as advancements in machine learning enable increasingly sophisticated automated agents. Bots are used to carry out a wide range of attacks, including credential stuffing, web scraping, and distributed denial-of-service attacks. Traditional countermeasures, such as CAPTCHAs, have become increasingly ineffective due to advances in artificial intelligence, highlighting the need for alternative detection approaches.

This cumulative dissertation addresses key challenges in bot detection, analysis, and prevention, with the goal of mitigating bot-related risks through novel, non-intrusive, and scalable solutions.

For bot detection, approaches are introduced that leverage the interaction behaviors of humans with web-based services, such as mouse movements, typing patterns, and website navigation. Unlike traditional methods that rely on static identifiers (e.g., IP addresses) or explicit user challenges (e.g., CAPTCHAs), these methods passively distinguish humans from bots by analyzing user interaction patterns using machine learning based detection models trained on behavioral data. Evaluations of synthetic and human behavior demonstrate the effectiveness of these approaches.

Effective bot defense also requires the analysis of bot software to uncover operational strategies and vulnerabilities. A method is presented to accelerate the reverse engineering of closed-source applications, a critical yet resource-intensive task. Specifically, dynamic binary instrumentation is employed to systematically identify and prioritize critical code segments (Points-of-Interest) related to sensitive data, as specified by the analyst (Items-of-Interest). Empirical validation on complex malware, including ransomware and peer to peer botnets, demonstrates substantial efficiency improvements and reliable identification of key functionalities.

To address the scalability challenge posed by automated bots, a preventive strategy is proposed that targets API-based automation. Existing obfuscation techniques primarily hinder the initial creation of bots but fail to prevent subsequent large-scale deployments. The proposed method obfuscates client-server communication protocols by assigning distinct protocols to each client. Consequently, attackers are required to reverse engineer each instance individually, significantly increasing the cost and complexity of large-scale bot operations without disrupting legitimate user interactions.

This thesis presents methods to enhance bot defenses across detection, analysis, and prevention. While it advances all three areas, challenges remain. As defenses improve, a shift toward UI-based bots that circumvent APIs is expected. Moreover, ongoing progress in AI will further blur the distinction between human users and automated agents, undermining current detection techniques. A promising direction lies in strong, yet privacy-preserving authentication mechanisms that robustly bind virtual identities to human identities.

Zusammenfassung

Internetdienste leiden zunehmend unter unerwünschter Automatisierung durch Bots. Die sich daraus ergebenden Herausforderungen sind immens. Es können finanzielle Verluste entstehen, Sicherheitsverletzungen auftreten und das Vertrauen der Nutzenden kann Schaden nehmen. Die Fähigkeit von Bots, menschliche Interaktionen überzeugend zu imitieren, erschwert die Erkennung von Bots, insbesondere da Fortschritte im maschinellen Lernen zunehmend ausgeklügelte automatisierte Agenten ermöglichen. Bots werden eingesetzt, um eine Vielzahl von Angriffen durchzuführen, darunter Credential Stuffing, Web-Scraping und verteilte Denial-of-Service Angriffe. Traditionelle Gegenmaßnahmen, wie CAPTCHAs, sind durch die Fortschritte in der künstlichen Intelligenz zunehmend unwirksam geworden, was den Bedarf an alternativen Erkennungsmethoden aufzeigt.

Diese kumulative Dissertation behandelt zentrale Herausforderungen in der Bot-Erkennung, -Analyse und -Prävention, mit dem Ziel, botbedingte Risiken durch neuartige, passive und skalierbare Lösungen zu mindern.

Für die Bot-Erkennung werden Ansätze vorgestellt, welche die Interaktionsmuster von Menschen mit webbasierten Diensten wie Mausbewegungen, Tippmuster und Website-Navigation, nutzt. Im Gegensatz zu herkömmlichen Methoden, die auf statischen Identifikatoren (z.B. IP-Adressen) oder expliziten Aufgaben (z.B. CAPTCHAs) beruhen, unterscheidet diese Methode passiv zwischen Menschen und Bots, indem sie die Interaktionsmuster der Nutzenden mithilfe von auf maschinellem Lernen basierenden Erkennungsmodellen analysiert, die auf Verhaltensdaten trainiert wurden. Evaluierungen von synthetischem und menschlichem Interaktionsverhalten belegen die Effektivität dieser Technik.

Eine effektive Bot-Abwehr erfordert zudem die Analyse von Bot-Software, um operative Strategien und Schwachstellen aufzudecken. Es wird eine Methode vorgestellt, die das Reverse Engineering von Closed-Source-Anwendungen beschleunigt, eine kritische, jedoch ressourcenintensive Aufgabe. Konkret wird Instrumentierung eingesetzt, um systematisch kritische Code-Segmente (Points-of-Interest) in Bezug auf Daten (Items-of-Interest), wie von der analysierenden Person festgelegt, zu identifizieren und zu priorisieren. Empirische Validierungen an komplexer Malware, einschließlich Ransomware und peer-to-peer-Botnets, belegen eine zuverlässige Identifikation zentraler Funktionalitäten.

Um die Skalierbarkeitsproblematik durch automatisierte Bots anzugehen, wird eine präventive Strategie vorgeschlagen, die sich gegen API-basierte Automatisierung richtet. Existierende Obfuscationstechniken behindern hauptsächlich die anfängliche Erstellung von Bots, verhindern jedoch nicht deren nachfolgenden großflächigen Einsatz. Die vorgeschlagene Methode verschleiert Client-Server-Kommunikationsprotokolle, indem sie jedem Client unterschiedliche Protokolle zuweist. Folglich sind Angreifende gezwungen, jede Instanz einzeln zu reverse-engineeren, was die Kosten und die Komplexität großflächiger Bot-Operationen erheblich erhöht, ohne dabei legitime Aktionen von Nutzenden zu stören.

Diese Arbeit präsentiert Methoden zur Verbesserung der Bot-Abwehr in den Bereichen Erkennung, Analyse und Prävention. Obwohl in allen drei Bereichen Fortschritte erzielt wurden,

bleiben Herausforderungen bestehen. Mit der Verbesserung der Abwehrmaßnahmen ist ein Übergang zu UI-basierten Bots zu erwarten, die APIs umgehen. Darüber hinaus wird die fortschreitende Weiterentwicklung im Bereich der künstlichen Intelligenz die Unterscheidung zwischen menschlichen Nutzern und automatisierten Agenten weiter verwischen und die derzeitigen Erkennungstechniken unterminieren. Ein vielversprechender Ansatz liegt in robusten, jedoch datenschutzfreundlichen Authentifizierungsmechanismen, die virtuelle Identitäten zuverlässig an menschliche Identitäten binden.

Acknowledgement

Someone once told me a PhD thesis is like navigating through the sea—you see islands along the way without knowing where you'll eventually land. I was lucky to have a great crew helping me steer this journey.

I would like to thank Mathias for his unwavering support throughout my Bachelor's and Master's studies, during every DISScussion, and for helping me stay focused, even when I was tempted to pursue new directions. You were there until the end, tirelessly helping me to improve. I am also grateful to Hannes, whom I met during my first semester. He was one of the reasons I chose to pursue security and privacy! I would further like to thank Serge, whose humor and cheerful spirit brought much, needed levity to academic work, and whose approach, often mirroring my own, was a source of inspiration.

My gratitude extends to my fellow colleagues and crewmates. I would like to thank first Doğanalp, who warmly welcomed me and was a supportive crewmate and seat neighbor. I learned a great deal about research from him, and he was always open to both fun and athletic challenges. The same appreciation goes to my ever-changing office quartet—Doğanalp, Flo, Nurefşan, Max, and Conny, whose presence made daily work both productive and enjoyable. The same holds for Tatjana, the cook of our ship. My spirited crew of four, whose camaraderie and light-hearted banter were a continual source of encouragement. A special thank you goes to my office mate Kevin, whose willingness to take on a variety of tasks significantly eased our collective workload.

I also thank Anne and Pascal, friends from my first semester, whose parallel journeys in our research group have been both inspiring and enjoyable. My appreciation extends to my students; supervising your theses has been as rewarding as it has been challenging, and your dedication has significantly enriched my own work.

Thank you, Ulf, whose remark back in 2012, predicting that I would eventually come to appreciate LaTeX, sparked both my enduring interest in science and a lasting appreciation for typesetting. Wilfried, who was the first to show me a path beyond school mathematics and encouraged me to explore further. I also thank Oliver for teaching me the value of balance and the importance of taking time to simply "chill." I also thank my friends Merle, Lukas, Nils, and Kim, whom I occasionally burdened with my research problems and involuntary discussions. The same goes for my privacy-conscious friends who prefer not to be mentioned by name.

Finally, I thank my family and my girlfriend, who never doubted me for a second and simply said, "You will do it." Your unwavering confidence and support have meant a great deal to me.

Now at the shore, I hope my crewmates will not linger too long at sea—so that we will see each other again for the next trip.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The Internet has transformed into a vast ecosystem of interconnected services that relies heavily on automation to enable functionality. For example, real-time financial market trackers aggregate stock prices and news updates, fully automated price comparison engines allow consumers to instantly gauge market prices, and search engines operate by automatically consuming and indexing all content on the Internet. While such automation brings efficiency, it also creates opportunities for misuse. Bots exploit free access to resources for various malicious activities, such as scraping proprietary data to train AI models or collecting personal information, such as names, addresses, and email addresses, for use in phishing attacks. They are also used to automatically scan for and exploit security vulnerabilities in applications.

Beyond the operational costs of traffic and repairing attacked systems, bots impose a significant social cost. In user-centric applications such as social media, gaming, and dating platforms, malicious bots distort perceived popularity and crowd out genuine users. This undermines the authenticity of interactions and erodes user trust, ultimately threatening the platform's success. Bots can also be used to spread opinions and misinformation on social media, making them appear more popular than they actually are. This effect is amplified by recommender algorithms that prioritize and promote popular content. As a result, bots can be leveraged to influence public opinion, including electoral processes [BF16].

The problem of bots is significant. Recent statistics indicate that bots accounted for nearly 50% of Layer 7 global Internet traffic in 2023, with approximately one-third of this traffic classified as malicious [Imp24]. Beyond merely inflating traffic, bots impose substantial operational costs by overloading servers, consuming bandwidth, and straining support systems.

Early defense strategies often targeted fixed indicators such as IP addresses or user-agent strings. However, modern bots readily evade these tactics by rotating through IPs or spoofing legitimate browser signatures. Meanwhile, more advanced methods leverage behavioral and interaction-based analysis to detect automated scripts masquerading as human users. Most of these approaches are proprietary and lack thorough evaluation. Additionally, they often assume low attacker capabilities, particularly in areas such as mouse or keyboard dynamics. The tension between maintaining a seamless user experience and implementing robust bot defenses becomes increasingly pronounced as the threat landscape evolves.

1.2 Problem Statement

This dissertation addresses core challenges in detecting, analyzing, and preventing malicious bots. This sections highlights current shortcomings and guides the research questions posed in Section 1.3.

Traditional defenses like CAPTCHAs represent the most widely adopted mechanism against bots. However, CAPTCHAs increasingly fail to provide adequate protection due to advancements in AI. These systems rely on tasks that are easy for humans but computationally challenging for machines. Recent progress in AI, particularly in areas like image recognition and logical reasoning, has diminished the effectiveness of CAPTCHAs [SPK16, AA20]. This has made tasks that were once bottlenecks for bots relatively trivial, requiring novel approaches to address the challenge of bot scalability effectively.

Bot Detection: Limitations of Static Features and Active Challenges Most existing bot detection methods rely on static features, such as IP addresses, user-agent strings, and device fingerprints. Although practical to deploy, these signals can be easily spoofed. Modern bots often utilize full browser environments or automated frameworks to appear indistinguishable from legitimate traffic.

A further limitation is the risk of false positives, which can increase user friction and potentially lead to lost revenue due to user frustration. Therefore, detection approaches should ideally be transparent and avoid active challenges such as CAPTCHAs.

While behavioral analysis, such as mouse movements, keystroke timings, or browsing patterns, offers a more promising alternative, it raises privacy concerns when outsourced to third-party services [Liu18, Mac18], as behavioral data can also be used for tracking [AMM+21]. Consequently, detection must evolve to meet the privacy, scalability, and adaptability requirements of real-world scenarios.

Another fundamental challenge is that detection is inherently retrospective. In the absence of static features, a sufficient amount of behavioral data is required for accurate classification. This raises the question of whether detection can occur quickly enough to prevent damage. For example, if five minutes of user behavior are needed per session to classify a bot, the response may be too late for effective mitigation on a per-session basis.

Bot Analysis: Reverse Engineering Effective bot detection and prevention might require tailored defenses based on a deep understanding of the bot's inner workings. This includes, for example, understanding how bots behave in order to design detection methods that specifically target these behaviors. This makes reverse engineering an essential step in developing targeted countermeasures. Through the analysis of the closed-source and obfuscated bot application, defenders can uncover a bot's communication protocols, control logic, and attack strategies. However, this process is typically slow, requires substantial expert knowledge, and is resource-intensive. Despite the availability of general-purpose tools such as `IDA`[1] and `Ghidra`[2], analyzing

[1] `https://hex-rays.com/ida-pro`
[2] `https://ghidra-sre.org/`

sophisticated, obfuscated bots still demands considerable manual effort. As a result, the high cost and limited scalability of reverse engineering hinder the timely development of customized defense mechanisms. Automating parts of this workflow could reduce overhead and accelerate the development of effective, and adaptive bot prevention mechanisms.

Bot Prevention: Scaling Challenges Once a bot has been developed, it can be replicated with minimal effort and deployed at scale. While a single automated account on a social platform may have limited impact, thousands of such accounts can collectively manipulate engagement metrics, such as likes, comments, and shares, amplifying certain posts or viewpoints far beyond the reach of any individual bot. This scalability enables large-scale influence operations, spam campaigns, and data harvesting efforts that are difficult to counter once initiated.

Existing prevention strategies, including anti-reverse engineering and code obfuscation techniques, are primarily designed to hinder the *initial* reverse engineering process and thus the creation of the *initial* bot. Tools such as Tigress [Col01] are effective at obfuscating individual binaries, making it more difficult for adversaries to analyze or instrument them. However, their protective value diminishes once a bot has been successfully developed and replicated. Moreover, these techniques do not protect the underlying application protocol. Any protocol change must be synchronized with all legitimate communication partners which cannot currently be achieved through obfuscation techniques. Further, in many cases, bot developers do not require a complete understanding of the application's internal logic, extracting the protocol alone is often sufficient to build a bot. Consequently, new strategies are required to limit the scalability of bot deployments without affecting legitimate user activity.

1.3 Research Questions

Automation enables scalable and efficient Internet services but is increasingly exploited by malicious bots. These abuses include resource misuse, protocol mimicry to evade defenses, and evasion of static-feature detection. Reverse engineering remains slow, expert-heavy, and costly. Existing prevention mechanisms lack scalability and often disrupt legitimate users.

Guided by open problems in detection, analysis, and prevention, the following research questions aim to explore solutions to improve their integration.

RQ1: What are the limitations of passive anomaly detection based on user behavior for identifying bots? Effective bot detection must balance accuracy with usability. Active challenges, such as CAPTCHAs, are increasingly ineffective due to advances in machine learning and are often detrimental to the user experience. For instance, Cloudflare CAPTCHAs alone are estimated to waste 500 years of user time per day [Meu21]. To avoid frustrating legitimate users and reducing engagement, passive detection techniques, those that operate transparently in the background, are essential.

Historically, passive detection has relied on static signals such as IP addresses or user-agent strings. However, these are easily spoofed and require continuous maintenance of blacklists or heuristics. In contrast, behavioral data, such as mouse movements, keystroke

dynamics, and navigation patterns, offers a more robust signal, as it is harder to mimic and does not rely on static identifiers.

This research question investigates the effectiveness and limitations of passive detection methods that leverage user behavior. Subquestions include: How accurately can bots be distinguished from humans based on various types of behavioral data? How do different classification approaches perform across behavioral modalities such as mouse dynamics, keyboard timing, and browsing patterns? Can behavioral detection be made privacy-preserving, for example by removing user-identifiable features or applying data minimization techniques?

Another important aspect is the timeliness of detection. Since behavioral data accumulates over time, an open challenge is determining how much interaction is required to make a reliable decision. The generalizability of behavioral models is also examined: can detection models trained on one application be transferred to another, or is application-specific training necessary?

These questions highlight the need for non-intrusive, privacy-aware detection mechanisms that are both effective and scalable without compromising user experience.

RQ2: How can reverse engineering techniques be optimized to accelerate the analysis of closed-source bot binaries? Sophisticated bots can mimic human-like behavior, which undermines the effectiveness of traditional detection methods. Understanding the internal workings of bot applications is essential for uncovering specific behavioral patterns that can inform both detection and prevention strategies.

Reverse engineering enables analysts to identify how bots automate interactions, whether through direct API access, UI manipulation, or instrumentation of running processes. It also facilitates the discovery of operational logic, communication protocols, and software dependencies. Those information can be used for developing tailored mitigation techniques. However, analyzing closed-source, compiled binaries is particularly challenging due to the lack of source code and the inherent complexity of such analysis. This process is often time-consuming, requires expert knowledge, and can be resource-intensive. In practice, a key challenge lies not only in locating relevant sections of the binary but also in assessing their relevance. Effective techniques are needed to rank and match binary components with the analyst's specific interests, while minimizing analysis overhead to avoid breaking application functionality.

RQ3: How can the scalability of bots be limited? The primary threat posed by bots lies not in isolated instances but in their capacity to scale with minimal effort. Once a bot is developed, it can be effortlessly replicated and deployed across a large number of clients or sessions, amplifying its overall impact. This scalability enables adversaries to perform widespread attacks, such as large-scale misinformation campaigns on social media platforms [BF16], with relatively low overhead.

Existing work has primarily focused on detecting and blocking individual bots, for example by flagging suspicious IP addresses or deploying active challenges. However, these techniques can often be bypassed and do not directly address the root problem, scalability. To effectively mitigate large-scale bot threats, it is necessary to embed scalability limitations directly into the architecture of the targeted application. Rather than relying solely on detection after deployment, prevention strategies aim to restrict the feasibility

of bot proliferation from the outset. Such approaches must be lightweight, transparent to legitimate users, and straightforward to integrate for application developers.

This research question therefore investigates approaches that shift the defensive focus from detecting individual bots to inherently preventing large-scale deployment, thereby tackling the scalability challenge at its root.

1.4 Contributions

The contributions of this thesis address the challenges outlined in the research questions by proposing novel techniques for bot detection, reverse engineering, and prevention. These contributions are grouped according to the primary objectives of this work:

C1: Passive Bot Detection A passive bot detection method is proposed, leveraging behavioral signals—such as website traversal patterns, mouse movement dynamics (e.g., acceleration and trajectory), and keystroke timing profiles—to differentiate between human users and bots. In contrast to active detection approaches, this technique imposes no interaction burden on users, thereby maintaining a frictionless user experience. The method relies on self-trained machine learning models for analyzing user input behavior, addressing limitations of prior work that often assumed overly simplistic bot interactions. Privacy can be enhanced by excluding keystroke identities from the classification process, with only minimal impact on accuracy. The approach also exhibits low detection latency, enabling robust classification within a few seconds of user activity. Furthermore, evaluation shows that models benefit from larger training datasets, even when collected across different applications, indicating the potential for cross-application generalization.

C2: Accelerating Program Analysis When generic detection techniques prove insufficient, an in-depth understanding of the behavior and structure of existing bots becomes essential for enhancing bot detection and prevention mechanisms. This contribution addresses *RQ2* by providing tools to facilitate reverse engineering processes and thus making binary analysis more targeted. Specifically, we introduce Points-of-Interest beacons, which guide reverse engineering by highlighting critical instructions in binary code. Analysts provide input data, such as contacted IP addresses or file content, and the approach identifies the instructions accessing this data. Consequently, reverse engineering efforts, both manual and automated, become more focused. We validate our approach using real-world malware and botnet binaries and integrate it into established reverse engineering frameworks such as IDA and Ghidra.

C3: Preventing Bot Scalability A significant contributor to bot-related issues is the bots' inherent capability to scale effortlessly: once developed, they can be duplicated easily, amplifying their impact. To counter this, we propose an obfuscation technique that specifically targets application communication, complicating scalability for bots reliant on automating interactions via communication protocols. Existing binary hardening methods primarily aim to impede initial reverse engineering but typically neglect obfuscation at the communication protocol level. Currently, attackers need only reverse engineer a single, static communication protocol

to create a first bot and subsequently scale effortlessly. In contrast, the proposed approach requires attackers to reverse engineer a unique communication protocol for each individual application instance, substantially increasing the effort required to scale bot deployments. This mechanism remains transparent to users and introduces minimal overhead for service providers.

1.5 Thesis Organization

This cumulative thesis is structured to present a comprehensive approach to bot detection, binary analysis, and prevention through a series of contributions detailed in the previous section. The structure and associated contributions are summarized in Table 1.1. Each chapter is designed to build upon the foundational challenges and solutions introduced in the research questions and contributions.

Table 1.1: The organization of the thesis.

Chapter	**Section**	**Publication**	**Contribution**
Chapter 1: Introduction			
Chapter 2: Background			
Chapter 3: Bot Detection using Behavioral Analysis	Section 3.1	[bWRF23]	C1
	Section 3.2	[bWWF24]	
	Section 3.3	[KbSF24]	
Chapter 4: Automated Analysis and Control of Binaries	Section 4	[bGFK23]	C2
Chapter 5: Bot Prevention through Endpoint and Protocol Obfuscation	Section 5.1	[bFF22]	C3
	Section 5.2	[bRF24]	
Chapter 6: Conclusion			

Chapter 2 provides the necessary background by reviewing existing methods and highlighting their limitations within the domains of bot detection, reverse engineering, and bot prevention. This chapter establishes the foundational context for the contributions presented in the subsequent chapters.

Chapter 3 investigates bot detection through behavioral analysis techniques. Specifically, it describes approaches based on mouse dynamics, keystroke dynamics, and high-traffic behavioral analysis frameworks aimed at effectively identifying bots accross different applications (C1).

Chapter 4 addresses the necessity of efficient binary analysis, introducing Points-of-Interest beacons that facilitate reverse engineering and enhance the understanding of bot and malware behavior (C2).

Chapter 5 emphasizes bot prevention through endpoint and protocol obfuscation. It presents novel techniques, including encrypted endpoints and polymorphic protocols, that increase complexity and cost for attackers attempting to scale their bots (C3).

Finally, **Chapter 6** concludes the thesis by summarizing the key findings and their implications, and outlines potential directions for future research in bot prevention and detection.

1.6 List of Publications

Thesis Contributions

[bFF22] **August See**, Leon Fritz, and Mathias Fischer. Polymorphic protocols at the example of mitigating web bots. In *European Symposium on Research in Computer Security*, pages 106–124. Springer, 2022.

[bGFK23] **August See**, Maximilian Gehring, Mathias Fischer, and Shankar Karuppayah. Binary sight-seeing: Accelerating reverse engineering via point-of-interest-beacons. In *Proceedings of the 39th Annual Computer Security Applications Conference*, pages 594–608, 2023.

[bRF24] **August See**, Kevin Röbert, and Mathias Fischer. Encrypted endpoints: Defending online services from illegitimate bot automation. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 166–180, 2024.

[bWRF23] **August See**, Tatjana Wingarz, Matz Radloff, and Mathias Fischer. Detecting web bots via mouse dynamics and communication metadata. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 73–86. Springer, 2023.

[bWWF24] **August See**, Adrian Westphal, Cornelius Weber, and Mathias Fischer. Detecting web bots via keystroke dynamics. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 423–436. Springer, 2024.

[KbSF24] Jan Kadel, **August See**, Ritwik Sinha, and Mathias Fischer. Botracle: A framework for discriminating bots and humans. In *European Symposium on Research in Computer Security, SecAI workshop*, 2024.

Further Contributions

[bb21] **August See**. Polymorphic protocols for fighting bots. *Electronic Communications of the EASST*, 80, 2021.

[bF23] **August See** and Mathias Fischer. Binding bot resources and uncovering their behavior. Extended Abstract, presented at the International Conference on Networked Systems (NetSys), 2023.

[bGLF25] **August See**, Thimo Grußendorf, Jona Laudan, and Mathias Fischer. Rubber ducky station: Advancing hid attacks with visual data exfiltration. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2025.

[bOSF25] **August See**, Benedikt Ostendorf, Lilly Sell, and Mathias Fischer. Flatdc: Automatic schema reverse engineering of flatbuffers. In *IFIP International Conference on ICT Systems Security and Privacy Protection, WNDSS workshop*, 2025.

[WbF23] Pascal Wichmann, **August See**, and Hannes Federrath. Secpassinput: Towards secure memory and password handling in web applications. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 236–249. Springer, 2023.

[WbGF24] Tatjana Wingarz, **August See**, Florian Gondesen, and Mathias Fischer. Privacy-preserving network anomaly detection on homomorphically encrypted data. In *2024 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2024.

Chapter 2

Background and Related Work

This chapter provides an overview of key concepts and challenges related to web bots. Specifically, it examines bot creation, bot prevention, and bot detection.

2.1 Bot Creation and Analysis

The development of bots involves various methods and tools aimed at automating interactions with networked services. This section provides a detailed analysis of different types of bots, the techniques used in their implementation, and the role of reverse engineering in creating but also analyzing bots.

2.1.1 Definition and Overview

Bots are automated programs designed to perform specific tasks by interacting with systems. This includes scraping bots that collect data and bots that simulate user interactions on social media platforms. Additionally, botnets, malicious software that turns compromised systems into coordinated bot instances, also fall within the broader definition of bots. The effectiveness and complexity of bots vary depending on their implementation. The major categories of bots are as follows:

API-Based Bots

API-based bots interact directly with the application programming interface (API) of a service, bypassing the graphical user interface (GUI) to achieve faster execution and lower computational overhead. By reverse engineering APIs, bot developers can extract endpoints and the necessary data structures to facilitate automated interactions. This method of automation enables high scalability and operational efficiency, making it particularly effective for tasks such as automating social media interactions[1] or data scraping.

A key advantage of API-based bots is their ability to scale efficiently. For example, a single bot can generate thousands of interactions and significantly impact content visibility with minimal effort once API endpoints for liking or commenting on a social media platform are identified.

[1] Example bot: `https://github.com/subzeroid/instagrapi`

This scalability is a key reason for their widespread impact, as a single bot can evolve into a network of automated agents capable of overwhelming a service.

Existing obfuscation techniques, such as those implemented by Themida [Tec22], can impede reverse engineering of individual protocols or APIs. However, they do not address the fundamental scalability of API-based bots, which remains their greatest threat. These obfuscation methods focus on increasing the difficulty of creating an initial bot, but fail to significantly increase the effort required to deploy multiple bot instances.

UI-Based Bots

User interface (UI)-based bots simulate human interactions with graphical user interfaces. These bots use tools such as computer vision to identify interface elements or rely on fixed coordinates to simulate mouse movements and clicks[2]. Unlike API-based bots that interact directly with application endpoints, UI-based bots operate by interacting with the visual components of an application. This approach typically has a higher computational overhead and slower execution speed due to the need to render graphical elements.

UI-based bots are particularly useful in scenarios where APIs are unavailable or highly restricted. For example, an e-commerce bot designed to purchase high-demand items mimics a human user by navigating through product pages, adding items to the shopping cart, and completing the checkout process. They are also not dependent on structured API endpoints, making them versatile for automating tasks in different environments.

However, UI-based bots are constrained by their reliance on GUI interactions, which limits their effectiveness in environments where user interfaces are limited or costly, such as in games. Another significant drawback is the amount of maintenance required to keep them functional. User interfaces tend to change more frequently than structured APIs, which are often versioned (e.g., `/api/v1/resource`) and thus more stable. These frequent UI changes require constant updates to the bot's logic, making UI-based bots more labor-intensive and less scalable than their API-based counterparts.

Common Use Cases

Bots are used in a variety of domains, with different levels of sophistication depending on the application.

Web Scraping Bots often extract data from websites, such as product prices, reviews, or other publicly available information. These bots typically rely on HTTP requests or web automation frameworks like `Puppeteer`[3] to retrieve and parse data. Because web scraping involves limited interaction, reverse engineering is generally not required. Instead, these bots target specific endpoints or HTML elements, making them relatively easy to develop and deploy.

[2]Example bot: `https://github.com/Xian55/WowClassicGrindBot`

[3]`https://pptr.dev/`

Crawling and Indexing Legitimate bots, such as those used by search engines, systematically crawl and index Web content. These bots are typically granted implicit permissions, such as when a `robots.txt` file is absent or does not restrict access, and they are not considered malicious. They provide essential services, including search engine indexing and digital archiving.

Spamming Spamming bots automate repetitive tasks such as account creation, login processes, and posting messages or comments. They are common on social media platforms, forums, and e-commerce sites. These bots typically automate only the functionality necessary for their specific purpose, such as posting messages, and require minimal reverse engineering focused on identifying key interaction points.

Task Automation Task automation bots perform complex, interactive activities that often require extensive reverse engineering of application protocols or binaries. Examples include

- **Gaming:** Bots automate gameplay activities such as resource gathering, quest completion, or PvP combat [LWK+16].
- **Social Media:** Bots manage accounts, engage with posts, and run targeted campaigns [OMAAK20, BF16]. This often requires direct interaction with an API.
- **Application Testing:** Bots automate tasks such as UI testing using browser automation tools such as Selenium[4] or Puppeteer.
- **Fraud and Scalping:** Bots facilitate fraudulent activity, including the creation of fake accounts and the automated purchase of high-demand items such as concert tickets or limited edition products [Par19].

Focus on API-Based Bots

API-based bots are the most scalable and impactful form of automation, making them a critical focus of this research. In contrast to UI-based or binary bots, API-based bots make use of structured and often stable application programming interfaces that allow for efficient automation with minimal computational overhead. Their scalability amplifies their threat, allowing a single bot to grow into a large-scale network that can manipulate content or overwhelm services.

Modern application architectures rely on APIs as the primary communication layer between clients and servers. APIs are an integral part of web applications, mobile apps, and IoT devices, providing standardized mechanisms for data exchange and interaction. This ubiquity makes APIs a prime target for automation, as their structured interfaces allow for predictable and efficient use by bot developers.

[4] `https://www.selenium.dev/`

2.1.2 Bot Creation and Analysis Techniques

Bot development requires a deep understanding of the target application, including its protocol and deployed defense mechanisms. Analysis of such applications often involves reverse engineering techniques, including binary analysis and traffic inspection.

These techniques are equally valuable for bot detection and analysis, allowing researchers to uncover a bot's internal logic and potential weaknesses. By analyzing a bot's internal logic, researchers can identify effective detection and prevention strategies. Conceptually, both bot creators and analysts function similarly to man-at-the-end (MATE) attackers [ASA+15], who have unrestricted access to the application. This level of access allows them to inspect, modify, and manipulate the software without restriction.

MATE Attackers

MATE attackers [ASA+15] have complete control over a client device, including access to software binaries, network traffic, and run-time memory. This access allows them to reverse engineer applications, extract sensitive information (such as API keys or communication protocols), and modify application behavior to their advantage.

Defending against MATE attackers is particularly challenging because, given sufficient time and resources, they can successfully reverse engineer most applications. For example, a mobile application that relies on hard-coded API keys for authentication is vulnerable to decompilation. A MATE attacker can extract these keys and use them to generate malicious requests that mimic legitimate client behavior.

Development Tools

Browser Automation Frameworks Browser automation frameworks such as Selenium and Puppeteer make it easier to interact with Web applications by replicating human-like behavior. These tools are particularly effective for automating interactions with dynamic or JavaScript-heavy content.

Mimicking Browsers Through the rendering of JavaScript, these frameworks can interact with dynamic web applications. This capability allows bots to process and manipulate dynamic content inaccessible to simpler HTTP request libraries.

Stealth Techniques To evade detection, these frameworks often integrate stealth solutions such as Puppeteer Stealth[5], which reduce fingerprint discrepancies between automated and real browser sessions. In some cases, full-headed browser instances are used to further simulate authentic user behavior.

Reverse Engineering and Instrumentation Tools Reverse engineering and instrumentation tools help analyze applications, especially when dealing with obfuscated APIs or native applications. Static analysis tools such as `IDA Pro`, `Ghidra`, and `Radare2`[6] decompile and

[5] `https://www.npmjs.com/package/puppeteer-extra-plugin-stealth`
[6] `https://rada.re/`

analyze binaries, while dynamic tools such as `Frida`[7] provide runtime inspection and function hooking. Platforms such as `PANDA`[8] and `Qiling`[9] facilitate dynamic analysis by monitoring and replaying system execution. These tools are often used to bypass protections such as HTTPS pinning or custom encryption by intercepting and modifying function calls.

There are also more targeted approaches. BotWatcher [BDYGP15] is an example of a dynamic analysis tool for botnet malware. It performs forensic analysis to identify key events, such as file creation or network connections, by periodically capturing memory snapshots of a bot's execution. By applying inference rules to these events, it can determine higher-level bot behavior, including payload downloads and command-and-control activities.

Protocol Analysis Tools Protocol analysis tools play a critical role in examining how applications communicate over networks. Tools such as Wireshark[10] capture and analyze communication at the packet level, while mitmproxy[11] and Proxyman[12] allow HTTPS traffic to be intercepted. Automated tools such as mitm2swagger[13] generate API documentation by analyzing intercepted traffic. These tools help extract API endpoints, headers, and payload structures.

Reverse engineering also facilitates analysis of bot communication protocols and input formats. Approaches, such as ReFormat [WJC+09], reconstruct protocol structures even in the presence of encryption. ReFormat achieves this by identifying cryptographic operations through bitwise analysis and using taint tracking to follow data flows within binaries. However, encrypted or obfuscated protocols, such as those used in Marionette [DCS15], present significant challenges to these techniques.

API Bot Creation Process

Bot creation involves analyzing and understanding a target application's communication protocols or interfaces. In the case of API-based bots, the goal is to circumvent user interfaces entirely by interacting directly with the application's APIs.

1. *Bypassing Encryption and Obfuscation:* Many applications implement custom encryption and obfuscation layers to impede analysis. These protections must be bypassed using static or dynamic analysis techniques, code modification, or disabling HTTPS pinning to facilitate traffic interception.
2. *Protocol Reverse Engineering:* Once traffic is decrypted, API structures can be reverse engineered through manual analysis or automated tools such as mitm2swagger.
3. *Reimplementation:* Extracted API calls are reimplemented in lightweight scripts using libraries such as Python's `requests`.

[7] `https://frida.re/`
[8] `https://github.com/panda-re/panda`
[9] `https://github.com/qilingframework/qiling`
[10] `https://www.wireshark.org/`
[11] `https://mitmproxy.org/`
[12] `https://proxyman.io/`
[13] `https://github.com/alufers/mitmproxy2swagger`

2.2 Bot Detection

Detecting bots is essential to mitigate their negative impact on Web services, including increased operational costs, degraded user experience, and abuse of systems designed for human interaction. Bot detection involves identifying automated behavior and distinguishing it from legitimate human actions. This applies not only to web-based systems, but also to desktop or mobile environments.

2.2.1 Static Detection

Static detection methods identify bots based on predefined system or network characteristics, such as IP address, browser fingerprint, and device attributes. These approaches rely on fixed identifiers rather than behavioral patterns. A basic static detection method involves blocking IP addresses associated with spamming or malicious activity. Services such as *abuseipdb*[14] maintain databases of such addresses for proactive filtering.

Many CAPTCHA systems integrate static detection into their risk assessment workflows [Mac18, Liu18]. They analyze user-agent strings, screen resolution, cookies, and IP addresses to assign a risk score that determines the CAPTCHA challenge level. However, these features can also be used to track users across multiple sites, raising privacy concerns. In addition, tracking mouse movements and keystrokes can identify not only the browser or system, but also individual users [AMM+21], raising ethical concerns about user privacy.

Privacy-preserving alternatives include honeypot-inspired techniques. Brewer et al. [BLRP10] propose embedding invisible fake links in web pages. Bots that interact with these links expose themselves as automated agents, achieving low false positive rates while preserving user privacy.

Cabri et al. [C+18] propose a heuristic-based static detection approach. Their analysis of HTTP logs uses known bot user agents and IP addresses to classify sessions. Using multi-layer perceptrons (MLPs) and Wald's sequential probability ratio test [Wal45], they categorize sessions as *Bot*, *Human*, or *None*, with ambiguous cases re-evaluated as more data becomes available. Their method achieved an F1 score of 0.96; however, its reliance on external databases and static identifiers makes it vulnerable to circumvention by sophisticated bots that spoof user agents.

Static detection methods are particularly effective against bots with consistent characteristics. However, their reliance on fixed attributes makes them vulnerable to evasion by adaptive bots that mimic real browsers or users.

2.2.2 Behavioral Detection

Behavioral analysis distinguishes bots from humans by examining patterns in user interactions. Bots can be detected by analyzing request data, e.g., pages requested, request sequences, and time intervals, and behavior such as keystroke dynamics and mouse movement patterns. This

[14] `https://www.abuseipdb.com/`

section divides these approaches into three subtopics: request data, mouse dynamics, and keystroke dynamics.

Request Data

Request data analysis is a widely used bot detection method that leverages HTTP metadata such as request frequency, page access sequences, referrer information, and session attributes (e.g., total bytes transferred, percentage of image requests). This approach is appealing due to its simplicity and stringer privacy-preserving nature, as it does not rely on cross-site tracking or biometric features. Furthermore, numerous approaches exist for detecting bots based on request characteristics [IKT+19, Mac18, Liu18, SCRM21, LARN21, JKV19].

Iliou et al. [IKT+19] introduced a comparison of different request-based detection machine learning models and features. Their method, evaluated on a year-long dataset of HTTP logs from MK-Lab's public web server, analyzed request paths, user-agent strings, and timestamps. They categorized bots as *simple* or *advanced* based on browser agent names and previous malicious activity. While simple bots were detected with an AUC of 1.00, advanced bots were more difficult to identify, achieving an AUC of 0.64. The authors recommended incorporating more sophisticated features to improve detection accuracy, especially for advanced bots where false positive rates remain a concern.

Mouse Dynamics

Mouse movement analysis is a possibility to distinguish bots from humans. Automated mouse movements often exhibit unnatural patterns, such as linear or exaggerated trajectories, which contrast with the fluid and irregular nature of human interactions [AMFVR22].

Shen et al. [SCG12] studied mouse-based authentication using features such as click types (e.g., single-click, double-click, drag-and-drop), motion ranges, and timestamps. They compared classification models and found that Support Vector Machines (SVMs) performed best, achieving low false positive and false negative rates over long authentication durations.

Acien et al. [AMFVR22] introduced a GAN-based approach to generate human-like mouse trajectories and trained classifiers such as Random Forests, K-Nearest Neighbors, and Recurrent Neural Networks (RNNs). Their method achieved 98.7% accuracy using only a single mouse trajectory. However, research on GAN-generated mouse movements also highlights the challenges of evading such detection methods [AFA19].

Iliou et al. [IKT+21] incorporated mouse trajectory analysis into their request-based detection system, using convolutional neural networks (CNNs) to classify raw mouse positions. However, their dataset included simplistic bot behaviors, such as perfectly linear movements, which are easily distinguishable from human patterns. Figure 2.1 illustrates such basic bot-generated movements.

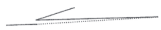
Advanced bot
✓ Heuristic hyperlink selection ✓ Advanced mouse movements


Figure 2.1: Simplified bot-generated mouse trajectories from Iliou et al. [IKT+21] (page 18, table 7).

Keystroke Dynamics

Keystroke dynamics analyzes typing patterns, including key press durations, inter-key latencies, and typing rhythms, to distinguish humans from bots.

TypeNet [AMM+21], an RNN-based model, demonstrated the effectiveness of keystroke analysis for user authentication. By processing keystroke sequences to generate feature embeddings, TypeNet achieved an Equal Error Rate (EER) of 2.2% for desktops and 9.2% for smartphones, outperforming previous CNN-RNN and SVM models.

De Alcala et al. [DMT+22] synthesized keystroke data using Kernel Density Estimation (KDE) and tested SVM and TypeNet classifiers. While both approaches yielded high accuracy, their reliance on synthetic data limited their real-world applicability, as the features generated were independent of contextual typing behavior.

Ceker et al. [ÇU17] used CNNs for keystroke-based user classification, augmenting datasets with synthetic Gaussian-generated features. Their model performed well, but highlighted the challenge of accurately replicating nuanced human typing behavior.

Despite advances, the detection of behavioral anomalies still faces inherent limitations: bots that are able to perfectly mimic human behavior remain undetectable. Forcing bots to mimic human behavior reduces their efficiency, making them slower and limiting some of their advantages. At the same time, scaling remains crucial for large-scale operations. Both enhancing human-like behavior and achieving scalability increase development costs, making widespread bot deployment more challenging.

2.3 Bot Prevention

Bot prevention is essential to maintaining the security and usability of online services. Effective bot prevention strategies aim to deter bot creation, limit scalability, while minimizing disruption to legitimate users.

2.3.1 Prove of being a Human

Approaches to proving humanity attempt to distinguish between human users and automated systems that are attempting to mimic human behavior. These methods typically rely on challenges or certifications of being a human.

One simple but controversial method is to require users to present government-issued identification. While this provides strong guarantees against automation, it is highly privacy-invasive. Many users are reluctant to provide sensitive personal information, which can reduce participation and deter new users from completing the registration process. Ultimately, this approach is impractical for widespread adoption because of user friction [Hea10].

The most common human verification mechanism is the CAPTCHA [Mac18, Liu18]. CAPTCHAs present challenges that are designed to be easy for humans but difficult for machines, such as recognizing distorted characters, identifying objects in images, or solving logic-based tasks. However, CAPTCHAs have two major limitations. First, they introduce significant user friction-Cloudflare estimates that CAPTCHAs collectively consume approximately 500 years of human effort per day [Meu21]. Second, advances in machine learning have significantly improved automated CAPTCHA solving capabilities, reducing their effectiveness [KJK22, HTR+20, SPK16].

Alternative approaches have been proposed to mitigate these challenges. Privacy Pass [DGS+18] allows users to solve a CAPTCHA once and receive cryptographic tokens that can be redeemed to bypass future CAPTCHAs. These tokens enhance privacy by preventing repeated challenges without tracking user identities. However, the continued decline of tasks that are trivial for humans but difficult for machines raises concerns about the long-term viability of CAPTCHA-based methods.

An emerging alternative uses cryptographic techniques and hardware-based attestations to prove humanity without requiring interactive challenges. These methods use Trusted Platform Modules (TPMs), Trusted Execution Environments (TEEs), or secure hardware devices to provide cryptographic attestations of human presence [WMK+22, Inca]. Unlike CAPTCHAs, these approaches eliminate direct user interaction while ensuring that attestations cannot be transferred. They can also be integrated with existing privacy mechanisms such as Privacy Pass [DGS+18], providing a seamless and secure verification method.

2.3.2 Obfuscation

Obfuscation techniques aim to hinder bot development by increasing the difficulty and cost of reverse engineering. By obfuscating code, protocols, or API endpoints, these techniques force attackers to expend significantly more effort to extract the information necessary for automation.

A fundamental challenge in assessing the effectiveness of obfuscation is that, when operating on their own devices, attackers always retain full control of their execution environment (cf. Section 2.1.2) . Consequently, obfuscation can increase the difficulty of reverse engineering, but cannot provide absolute protection [ASA+15, TIF19, BCG+16].

There are several tools and frameworks for obfuscating web content, ranging from academic research prototypes [Bin24, Yun24, Kac] to commercial solutions [Incb, Jsc, Mob]. These tools obfuscate HTML, JavaScript, and CSS while claiming minimal performance overhead. However, independent evaluations of their real-world impact remain limited.

Tools like NOMAD dynamically randomize HTML form element names and IDs on a per-session basis, forcing bots to dynamically extract attributes during execution [VYG13]. Similarly, We-

bRanz applies randomized obfuscation to HTML attributes, disrupting predefined automation patterns while preserving the visual and functional integrity of a web page [WZX+16]. While effective in specific contexts, such as form submission or DOM manipulation, these techniques have limited applicability to broader automation scenarios.

Beyond web applications, obfuscation is widely used to protect binary software. Tools such as Tigress [Col01], VMProtect [Sof21], and Themida [Tec22] use advanced obfuscation techniques to protect compiled binaries from reverse engineering. However, a key limitation of these tools is their inability to effectively obfuscate network endpoints. Attackers can intercept and analyze network traffic using system-wide proxies, allowing them to identify API endpoints and bypass obfuscation-based protections.

In addition to code obfuscation, developers often use API keys to encrypt traffic between client and service in addition to HTTPs [See20b]. This serves as an authentication mechanism and add a layer of complexity for bots. Bots must extract these keys to interact with protected endpoints and thus the cost of creating bots is increased.

2.4 Summary

The increasing sophistication of AI has enabled bots to mimic human behavior more convincingly, posing significant challenges to traditional detection mechanisms such as CAPTCHAs. This trend highlights a fundamental trade-off between usability, ensuring seamless access for legitimate users, and security, preventing automated abuse.

Many current detection methods rely on easily spoofable static features, such as IP addresses and user-agent strings. While behavioral detection offers a more robust alternative, its effectiveness can be further improved by adopting more realistic threat models that account for advanced simulation of human-like inputs, such as mouse movements and keystroke dynamics.

Bot analysis remains a critical, yet resource-intensive component of defense. Reverse engineering techniques are essential for understanding bot functionality and serve as a prerequisite for developing effective detection and prevention strategies. However, they typically require significant manual effort and expert knowledge, limiting their accessibility and scalability.

On the prevention side, many mechanisms aim to increase the cost of initial bot development, for example through binary obfuscation or proof-of-humanity techniques. However, these approaches typically do not scale against large bot deployments. Once a bot is created, it can often be replicated and deployed with minimal additional effort, particularly when targeting automation-friendly interfaces such as APIs. To address this limitation, recent work has proposed prevention strategies based on cryptographic attestations and trusted execution environments. These approaches move away from traditional challenge-response mechanisms by providing verifiable proofs of human presence. While promising, they require significant infrastructure support and widespread adoption to be effective in practice.

Chapter 3

Bot Detection Using Behavioral Analysis

Automated programs, commonly referred to as web bots, systematically make requests to online endpoints, increasing operational costs and degrading the experience of legitimate users. A notable example is the disruption caused by such bots in the e-commerce sector, where they made popular items (e.g., graphics cards and gaming consoles) unavailable for extended periods of time by purchasing them faster than human customers could. Commercial countermeasures, such as intrusion detection systems (IDS), are designed to detect anomalous or malicious behavior. However, bots often evade detection by interacting with websites as intended, for example by navigating to product pages and clicking the "buy now" button.

CAPTCHAs have long served as a commonly used mechanism to combat automated bot activity. These challenges require users to complete tasks designed to verify their human identity. However, despite their widespread use, CAPTCHAs are generally disliked due to the inconvenience they cause [Hea10], as they disrupt the user experience and consume considerable time. In modern implementations, most CAPTCHA systems are tightly coupled with risk assessment techniques [Mac18, Liu18]. Although the specific features used in these assessments are not disclosed, there is evidence that Google's reCAPTCHA, for example, assigns lower trust scores to users who are not logged into a Google account [Liu18]. Despite these advances, even risk-based CAPTCHA systems still result in significant wasted time - Cloudflare alone reports approximately 500 years of cumulative time lost every day [Meu21]. In addition, ongoing advances in artificial intelligence and image recognition have steadily eroded the effectiveness of CAPTCHA-based defenses [SPK16].

These limitations underscore the need for more transparent, non-intrusive bot detection methods that do not disrupt the user experience. Building on the related work discussed in Section 2.2, this chapter presents an improved classification approach for distinguishing between human and automated behavior and input patterns.

The main contributions of this chapter are as follows:

- **Mouse Dynamics Detection (Section 3.1):** This section examines mouse movement patterns to distinguish between human and automated interactions. It extends previous research on simple, bot-like cursor patterns by incorporating more nuanced features to improve detection performance.
- **Keystroke Dynamics Detection (Section 3.2):** This section analyzes keystroke patterns to distinguish between human and automated input. It examines dependencies between keystrokes and applies state-of-the-art neural network architectures to model and detect artificial typing behavior.

- **Cascaded Detection Framework (Section 3.3):** This section introduces a layered detection approach that combines lightweight heuristics with more detailed behavioral analysis. Initial indicators, such as window size, provide fast detection, while subsequent steps analyze browsing behavior and navigation patterns to improve accuracy. The performance of the approach is evaluated using a real-world e-commerce dataset.

3.1 Bot Detection Using Mouse Dynamics

Existing approaches to bot detection have several limitations: many are proprietary, rely solely on HTTP request data, or make overly simplistic assumptions, such as the assumption that advanced bots produce only straight-line mouse movements (see Figure 3.1). Another common assumption is that mouse behavior is consistent across sites and applications, which is often not the case.

To address these challenges, we investigate the use of mouse dynamics for bot detection under realistic conditions. Specifically, we contribute the following:

- **Mouse Dynamics-Based Bot Classification.** We perform a comparative analysis of bot classifiers that use mouse dynamics alongside those that rely solely on HTTP request data. To facilitate this comparison, we create a unified dataset[1] containing mouse and HTTP request data collected from the same user sessions, including advanced bots that are able to simulate human-like cursor movements.
- **Cross-Site Training for Bot Detection.** We evaluate the generalizability of mouse dynamics-based detection by training classifiers on datasets obtained from external sources. This analysis evaluates whether detection models can be trained on third-party data to reduce the need for site-specific training efforts. For this purpose, we use external tools to simulate mouse movements (see Figure 2.1).
- **Impact of Data Volume and Time Until Detection.** We examine how detection accuracy varies with the amount of available data, including the number of requests and recorded mouse movements. In addition to overall accuracy, we analyze how quickly a reliable classification can be made, quantifying the time until detection.

To structure our research, we address the following research questions:

RQ1.1: How does the performance of our proposed detection approach vary with different amounts of available data, especially with respect to the number of requests and mouse movements?

RQ1.2: To what extent does the detection performance change when the model is trained with mouse dynamics data collected from external sources? In other words, can we build generic detection models that remain effective across different Web sites?

[1]Due to privacy concerns related to the potential re-identification of users across applications, we cannot make this dataset publicly available.

Building on these research questions, the following publication[2] summarizes our findings on the effectiveness of mouse dynamics for bot detection. Our results indicate that mouse dynamics can serve as a reliable indicator, with our classifiers achieving competitive performance compared to related work, even under more stringent and realistic conditions. Specifically, we report a classification accuracy of 0.966 and an AUC of 0.994.

> August See, T. Wingarz, M. Radloff, M. Fischer. Detecting Web Bots via Mouse Dynamics and Communication Metadata. IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC), 2023.

The following sections describe our approach to classifying and generating synthetic mouse movements, summarize the results, and outline limitations.

3.1.1 Method

Our approach improves web bot detection by integrating request-based features with mouse dynamics data and training machine learning models to classify interactions as human or automated. While request metadata (e.g., user agent or screen size) can be easily spoofed, continuous mouse trajectories are much more difficult to replicate. As a result, attackers attempting to evade detection must generate realistic, human-like cursor movements for each interaction.

Figure 3.1 illustrates that human users do not exhibit strictly linear mouse movements. The left panel shows a user interacting with a browser, the center panel depicts synthetic mouse movements, and the right panel presents a human playing a rhythm game. Further, even advanced bots can generate non-linear trajectories, indicating that the naive assumption of perfectly straight paths is insufficient for bot detection. Additionally, mouse movement patterns vary across applications, emphasizing the importance of domain-specific considerations. In this work, we use:

- *HTTP request-based features*, adapted from Iliou et al. [IKT+19], with the removal of attributes inappropriate for our setting.
- *Mouse dynamics-based features*, derived from Acién et al. [AMFVR22] and extended by using feature engineering techniques from previous studies [GF04, AEZ19].

The following sections describe these features and their role in bot detection.

[2]The core concept, methodological design, and evaluation strategy of this paper originate from the author of this dissertation. The third author implemented and conducted the experiments, while the second and fourth authors contributed to the refinement of the final publication.

[3]`https://github.com/Xetera/ghost-cursor`

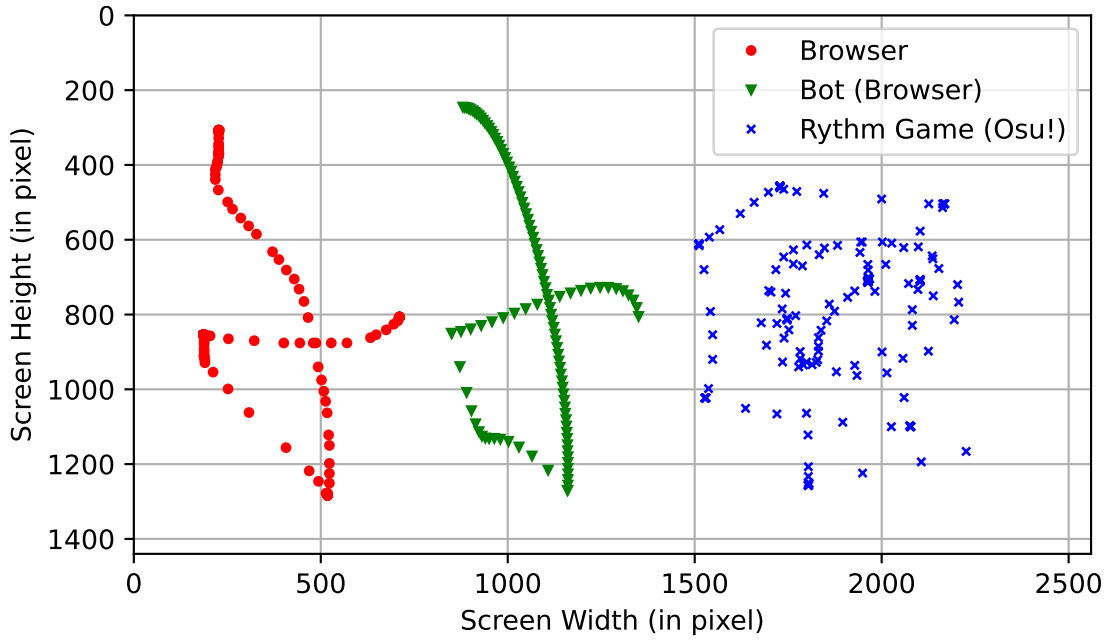


Figure 3.1: Examples of mouse tracks from different applications: a browser-based chat application (left), an advanced bot using ghost-cursor[3](center), and a human user playing a rhythm game (right).

Request Data

We refine the feature set proposed by Iliou et al. [IKT+19], omitting attributes irrelevant to our experimental setup (e.g., search engine detection via the `Referer` header). The final set includes:

1. Fraction of HTTP requests that result in 4xx errors.
2. Fraction of requests for CSS files.
3. Fraction of requests for JavaScript files.
4. Fraction of requested URLs that contain the previously requested URL as a substring.
5. Total session time (time elapsed from first to last request).
6. Standard deviation of requested page depth (number of "/" characters in the URL path).
7. Mean and standard deviation of intervals between requests.

These features capture patterns that distinguish between human and automated web interactions.

Mouse Features

Following previous work [AMFVR22, GF04, AEZ19], we extract:

- **Base Signals**: Resampled coordinates (x'_t, y'_t), path length s'_t, tangential velocity v, angular velocity ω, and higher order terms such as acceleration and jerk.
- **Statistical Descriptors**: Mean, standard deviation, minimum, maximum, and range for each base signal.
- **Global Characteristics**: Total duration of each action, total raw path length, and additional motion descriptors such as *straightness* (ratio of the Euclidean distance between the first and last points to the total path length) and *jitter* (ratio of the original path length to the smoothed path length).

Each action is represented as a 50-dimensional feature vector that serves as input to machine learning models trained to distinguish between human and bot-generated mouse movements. By leveraging continuous motion data, this approach provides a more robust defense against automated agents, as generating highly realistic human-like trajectories remains a challenging task for attackers.

3.1.2 Key Results

In the following, we summarize evaluation of our bot detection approach and summarize the key results. We focus on two main aspects: (i) performance under varying amounts of available data (**RQ1.1**) and (ii) effectiveness when incorporating external mouse dynamics data (**RQ1.2**). Our experiments use a random forest classifier for both request and mouse-based features, following recommendations from previous work [IKT+19, AMFVR22]. Random forests provide interpretable measures (e.g., feature importance) that help us understand the model's decision-making process.

Evaluation Setup

Our dataset consists of user query data along with the corresponding mouse trajectories. Since no publicly available dataset integrates both aspects [IKT+21], we created our own by setting up two dedicated websites. We recruited 322 participants to visit the first website and 163 to visit the second, logging every request and mouse event. Mobile devices and sessions without mouse data were excluded.

Figure 3.2 illustrates the distribution of collected data points across participants.

To simulate bot behavior, we used `Puppeteer` to emulate automated browsing with random delays of 0–2,s. The bots accessed top-level pages, randomly selected subpages, and attempted account registration to mimic the behavior of a moderately throttled web scraper. Two bot variants were evaluated:

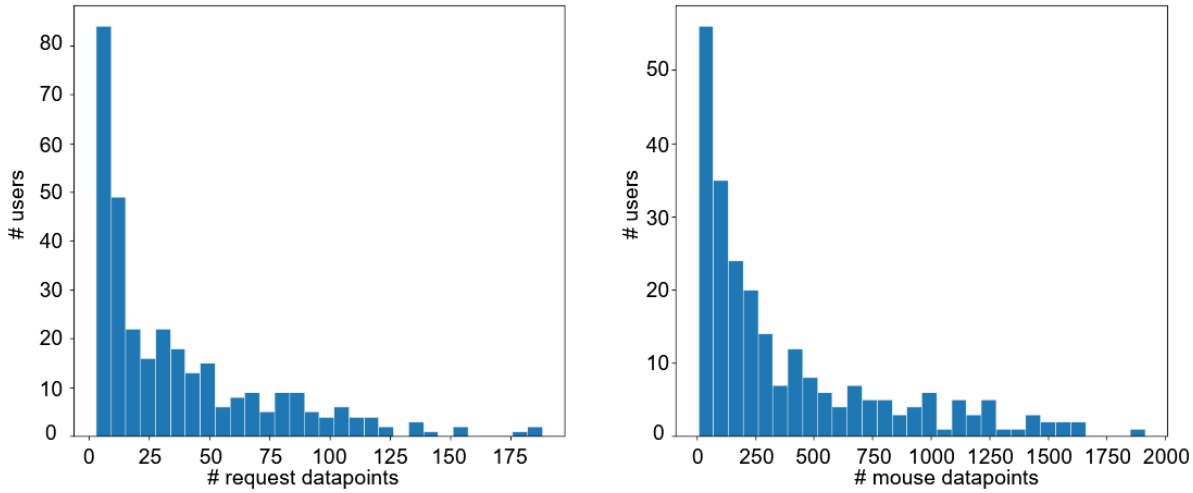


Figure 3.2: Distribution of participants by number of data points collected.

- **Basic Mouse Bot:** Moves the cursor in a straight line at a constant speed, similar to approaches in related work (see Figure 2.1).
- **Advanced Mouse Bot:** Uses `ghost-cursor`[4] to generate smoother, human-like trajectories.

We ensured an equal number of bot and human sessions to allow for balanced dataset.

For features based on mouse dynamics, each recorded mouse trace is segmented into *actions* sequences of up to 50 data points or a maximum of two seconds, typically ending with a click. To ensure consistency, each action is resampled at fixed 20 ms intervals, producing time-aligned signals for x and y coordinates. This allows additional motion-based metrics to be derived, including velocity, acceleration, and angular velocity.

Results RQ1.1: Bot detection under varying data availability

Request Data Classifier The Table 3.1 shows how the performance of the request-based classifier varies with the number of HTTP requests per session. Even with only 10–20 requests, the model achieves an accuracy of 0.89–0.91. With 200 or more requests, the performance approaches 0.98, indicating that increasing request data significantly improves classification reliability.

Mouse Dynamics Classifier Table 3.2 shows the performance of the mouse-based classifier at different event counts. Even with only 50 recorded events, the model achieves an accuracy better than 0.93 against advanced bots. Since users typically generate about 30 mouse events per second, collecting 50 events takes only about 1.66 seconds, making the approach practical for real-time applications such as CAPTCHAs.

[4] `https://github.com/Xetera/ghost-cursor`

Table 3.1: Request data model performance with varying number of requests.

Requests	Acc	Prec	Recall	AUC	Time (s)
200	0.980	0.980	0.980	0.982	0.209
100	0.970	0.961	0.980	0.985	0.209
No limit	0.960	0.960	0.960	0.972	0.215
50	0.950	0.941	0.960	0.976	0.208
20	0.910	0.918	0.900	0.975	0.221
10	0.890	0.842	0.960	0.956	0.220
5	0.900	0.885	0.920	0.945	0.220
4	0.880	0.913	0.840	0.922	0.211

Table 3.2: Performance of the mouse-based model with different event counts.

Events	Acc	Prec	Recall	AUC	Time (s)
No limit	0.966	0.964	0.968	0.993	0.449
50	0.933	0.943	0.922	0.979	0.124
200	0.930	0.951	0.906	0.979	0.189
100	0.920	0.942	0.895	0.975	0.149
20	0.855	0.846	0.868	0.949	0.114
10	0.850	0.862	0.833	0.903	0.120
5	0.846	0.909	0.769	0.916	0.099
4	0.826	0.895	0.739	0.879	0.098

These results indicate that both request-based and mouse-based models benefit from increased data availability. However, mouse dynamics provides faster improvements, suggesting its suitability for low-latency scenarios such as authentication checks.

Basic vs. Advanced Mouse Bot Table 3.3 compares classifier performance on basic (linear) and advanced (non-linear) mouse bots. As expected, detection of basic bots is trivial (accuracy $\approx$0.995, AUC=1.000), while advanced bots slightly reduce the accuracy to 0.966.

Table 3.3: Mouse based classifier performance against basic and advanced bots.

Scenario	Acc	Prec	Recall	AUC	Time (s)
Basic Mouse	0.995	0.997	0.994	1.000	0.857
Advanced Mouse	0.966	0.962	0.970	0.994	1.293

Combining Request and Mouse Features Next, we evaluated a combined model that averages the predictions from the request-based and mouse-based classifiers. Table 3.4 shows that this combined approach achieves high accuracy (0.95–0.96) and produces *zero false positives*

in our test sets, suggesting potential for production environments where false positives are costly [Hea10].

Table 3.4: Combined model performance on unseen users.

Test Ratio	Acc	Recall	AUC	TP	FP	TN	FN
0.1	0.960	0.953	0.976	122	0	22	6
0.2	0.950	0.940	0.970	252	0	49	16

Results RQ1.2: Incorporating External Mouse Dynamics

To evaluate whether external mouse datasets can improve detection, we incorporated data from Antal et al. [ADF19], which contains 1.54 million mouse movement records from 21 users performing common computing tasks. The combined dataset increased accuracy to 99.71% and reduced false positive and false negative rates to below 1%. This suggests that general mouse dynamics can aid in bot detection without the need for site-specific patterns.

3.1.3 Discussion and Implications

Our results indicate that mouse dynamics combined with request-based data can effectively distinguish between human and automated activity.

Limitations The primary limitation of our work is its reliance on synthetic bot behavior, similar to previous work [IKT+21]. While we used tools designed to mimic human-like movements, the lack of a public dataset of verified bot behavior limits the generalizability of our results. Furthermore, advanced attackers could record and replay authentic user movements to evade detection.

In addition, the current approach focuses exclusively on mouse interactions, excluding users who rely solely on keyboards, touchscreens, or assistive technologies. Future research should address these limitations by extending detection mechanisms to accommodate different input modalities.

Higher-Level Implications As bot developers continue to refine their methods-for example, by training models on human-generated mouse traces-the detection task will become more challenging. Privacy concerns also arise with the collection of detailed mouse trajectories, as such data can potentially enable user re-identification. Therefore, future implementations should use techniques such as differential privacy or anonymization to balance security and user privacy.

3.2 Bot Detection Using Keystroke Dynamics

While mouse dynamics provide valuable signals for bot detection, they are not always available, e.g., there are no mouse movements in SSH. This section extends the work presented in Section 3.1 by using keystroke dynamics to distinguish between automated and human input. Previous research in this area has primarily focused on user authentication [AMM+21] rather than bot detection. In addition, some approaches rely on simplistic assumptions, such as treating keystroke events as independent or ignoring character-level dependencies [DMT+22]. In contrast, we employ classifiers that analyze keystroke timing patterns across multiple dimensions, capturing temporal dependencies within keystroke sequences, e.g., through bigram and trigram distributions. In addition, we introduce generative models that synthesize realistic keystroke sequences to augment training datasets.

Our main contributions are:

- **Keystroke Timing for Bot Detection.** We demonstrate that keystroke timing alone, without access to keycodes, can effectively discriminate between human and automated typing patterns. Our results show that omitting key identity results in only a small performance penalty of about 2% in accuracy.
- **Character-Dependent Keystroke Synthesis.** We introduce a keystroke synthesis approach that generates character-dependent timing data using a diffusion model. By associating specific key codes with more human-like timing patterns, this method improves upon prior key-independent models. It leverages the success of diffusion models in other domains, such as image generation [HJA20], and adapts them to the domain of keystroke dynamics.
- **Synthetic Data for Improved Classification.** We evaluate the impact of augmenting training datasets with synthetic keystroke data. Our results indicate that classifiers trained on more diverse synthetic samples achieve higher detection accuracy.

To structure our research, we address the following research questions:

RQ1.3: How well can classifiers discriminate between human and artificially generated keystrokes?

RQ1.4: How does character-level dependency modeling affect the accuracy of bot detection?

RQ1.5: How accurately can generative models replicate human keystroke dynamics?

Building on these research questions, the following publication[5] summarizes our findings on keystroke-based classification. Our results indicate that classifiers trained on datasets containing both human and artificially generated keystrokes can achieve high performance using only timing information, without relying on keycode data. In this configuration, we observe an

[5]The core idea, methodological design, and evaluation strategy for this publication originate from the author of this dissertation. The second author implemented the models and experiments, while the third and fourth coauthors contributed to refining the final paper.

accuracy and F1 score of approximately 0.98. Incorporating keycode information further improves classification performance beyond this level.

> August See, A, Westphal, C. Weber, M. Fischer. Detecting Web Bots via Keystroke Dynamics. IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC), 2024.

The following sections describe our approach to classifying and generating synthetic keystrokes, summarize the results, and outline limitations.

3.2.1 Method

This section presents a method for bot detection based on keystroke patterns and the generation of synthetic data that closely resembles human typing behavior. The method consists of four main components: data preparation, feature extraction, synthetic data generation, and classification.

Data Preparation and Cleansing

High-quality, representative data sets is essential for training classifiers and generating realistic synthetic samples. For our analysis, we used the Dhakal dataset [DFKO18], which consists of 136 million keystrokes collected from over 168,000 users. Each participant typed 15 predefined sentences, e.g., *"Have I mentioned how much I love Houston traffic?"*, with the exact press and release times recorded per keystroke.

We applied automated preprocessing to remove records with missing values or null timestamps. In addition, we filtered out human errors that resulted in abnormally high or low latencies, as such anomalies could cause the model to misinterpret irregular typing patterns as indicative of automated input. While this filtering slightly reduces the natural error rate of the dataset, it ensures that the model learns primarily from intentional, human-like typing behavior.

Feature Extraction for Keystroke Analysis

Building on established approaches [AMM+21, DMT+22], we characterize each keystroke by extracting the following temporal features:

- *Hold latency*: Time between pressing and releasing the same key.
- *Press latency*: Time between pressing a key and pressing the next key.
- *Release latency*: Time between releasing a key and releasing the next key.
- *Inter-key latency*: The time between releasing a key and pressing the next key.

To investigate the role of key identity in detection performance, we consider two different feature sets:

- L_{nokey}: *hold, press, release, inter-key* latencies without key identity information.
- L_{key}: The same latencies, plus the ASCII value of each key.

Key-Independent Modeling Key-independent models ignore the identities of individual keys and rely solely on timing patterns. While this simplification can reduce model complexity and improve generalizability [DMT+22], it also ignores potential differences in typing behavior between characters.

Key-Dependent Modeling Typing patterns often vary depending on the character and its context. For example, transitions such as `ab` or `aµ` can have different latency profiles. To capture these dependencies, we model key transitions using bigrams, inverted bigrams, and trigrams, incorporating the ASCII keycodes into the feature set.

Synthetic Data Generation

The generation of synthetic keystroke data serves two purposes: to augment training data sets and to test the robustness of classifiers against artificially generated keystrokes. We use two complementary approaches:

Kernel Density Estimation (KDE) Following the methodology of [DMT+22], we fit Gaussian KDE models to observed latency distributions. This technique allows us to generate new samples by drawing from these learned distributions.

- *Key-Independent Generation*: Latencies are sampled from a single distribution per feature, regardless of character identity.
- *Key-Dependent Generation*: Separate KDE models are trained for different bigram and trigram transitions (e.g., `ab` and `abc`). If an unknown transition occurs during sampling, the model defaults to a more general latency distribution.

Diffusion Model Inspired by recent advances in generative modeling [DN21], we apply a probabilistic denoising diffusion model (DDPM) [HJA20] using the *denoising diffusion pytorch* library [Phi]. Unlike KDE, this method learns to reconstruct entire keystroke sequences (e.g., sequences of 30 consecutive keys) rather than isolated events.

The model first applies a hyperbolic tangent normalization to the timing features. During inference, it starts with Gaussian noise and iteratively denoises it to produce realistic keystroke patterns. This approach naturally captures long-range dependencies between characters and sequences.

Classifier Training and Evaluation

Differentiating human keystrokes from those generated by automated processes is the final step. We evaluate the performance of the following classifier architectures:

- **Long Short-Term Memory (LSTM):** We implement a two-layer LSTM network with dropout regularization [SHK+14] and batch normalization [IS15] to prevent overfitting. The hidden states of the LSTM are passed through a fully connected layer to predict human or bot input. The model is optimized using cross-entropy loss.
- **Transformer:** We use a transformer architecture with multi-head self-attention. Keystroke sequences are embedded and augmented with sinusoidal position encodings. The model is trained with cross-entropy loss and uses early stopping to avoid overfitting.
- **Support Vector Machine (SVM):** For a baseline comparison, we train an SVM with an RBF kernel on the extracted feature vectors. Although SVMs do not inherently model sequential dependencies, prior work suggests that they can achieve strong performance with well-defined timing features.

Each model is trained on a balanced dataset of real and synthetic keystroke sequences. Performance is evaluated in terms of accuracy, precision, recall, and F1 score. Through this evaluation, we aim to determine the feasibility of distinguishing automated from human-generated keystrokes, and to assess the utility of better synthetic data in improving model robustness.

3.2.2 Key Results

This subsection presents the evaluation and results of our synthetic keystroke generation and bot detection experiments. We address three primary research questions: How well can classifiers distinguish synthetic from real keystrokes (RQ1.3)? How does the inclusion of keycode information affect classification accuracy (RQ1.4)? How closely do the generated keystrokes resemble human typing patterns (RQ1.5)?

Overview of Generator Models The Table 3.5 gives an overview of the considered generative models. The simplest, *KDE Universal* [DMT+22], independently models each latency type (hold, press, release, inter-key) without considering character identity. In contrast, the *Diffusion* model processes sequences of up to 30 keystrokes, allowing it to capture temporal dependencies over longer contexts.

Results RQ1.3: Differentiating Human and Synthetic Keystrokes

To evaluate recognition performance, we trained support vector machines (SVMs), LSTMs, and transformers on datasets containing 7,500 real and 7,500 synthetic samples (one sentence per sample). In addition, we included a *Random Sequences* dataset in which keystroke latencies were drawn uniformly at random to serve as a naive baseline.

Table 3.5: Generator models with varying degrees of dependence on surrounding keys.

Model	**Timing Context**	**Abbreviation**
KDE Universal	Based on general key timing; No dependency on surrounding keys	KDE Univ.
KDE Bigram	Dependent on the current and successor key.	KDE Bi.
KDE Bigram Reversed	Dependent on the current and predecessor key.	KDE Bi. Rev.
KDE Trigram	Dependent on the current, predecessor and successor key.	KDE Tri.
Diffusion	Considers up to 30 keystrokes.	Diff.

SVM Detection Performance Table 3.6 shows the accuracy (ACC) and F1 scores of the SVM on different training and test datasets. The model performs well when evaluated on similarly generated data, but struggles with cross-generator generalization. For example, performance degrades significantly when trained on *Diffusion*-based samples but tested on random sequences, suggesting that the model relies heavily on generator-specific timing patterns. As expected, incorporating more realistic assumptions about key surroundings improves detection. However, the impact is less pronounced than anticipated. For example, an SVM trained on KDE Univ. achieves only slightly lower performance compared to one trained on KDE Tri.

Table 3.6: Excerpt of SVM performance results (subset shown for brevity).

Train \ Test		KDE Univ.	KDE Bi.	KDE Bi. Rev.	KDE Tri.	Diff.	Rand. Seq.
KDE Univ.	ACC	0.9869	0.9440	0.9871	0.9436	0.4949	0.9866
	F1	0.9870	0.9422	0.9873	0.9418	0.0270	0.9868
KDE Tri.	ACC	0.9810	0.9794	0.9816	0.9774	0.4968	0.9809
	F1	0.9813	0.9797	0.9819	0.9777	0.0549	0.9812
Diff.	ACC	0.5317	0.6100	0.5304	0.5907	0.7348	0.3311
	F1	0.4624	0.5891	0.4600	0.5599	0.7530	0.0033

LSTM and Transformer Performance Next, we evaluated the LSTM and Transformer models on a larger *Mixed* dataset, consisting of 16000 synthetic and 16000 real samples (sentences). Table 3.7 shows that both architectures outperform SVM, especially when tested on more complex generators. The LSTM model achieves F1 values close to 0.98 on all test datasets, highlighting the effectiveness of sequence-aware models in capturing keystroke dynamics.

Table 3.7: Performance of Transformer and LSTM trained on a mixed dataset wit 32k samples.

Model \ Test		KDE Univ.	KDE Bi.	KDE Bi. Rev.	KDE Tri.	Diff.	Rand. Seq.
Transformer	ACC	0.9332	0.8919	0.9154	0.9363	0.9270	0.9293
	F1	0.9314	0.8880	0.9145	0.9350	0.9271	0.9268
LSTM	ACC	0.9807	0.9776	0.9797	0.9810	0.9808	0.9803
	F1	0.9802	0.9778	0.9799	0.9806	0.9810	0.9797

Results RQ1.4: Impact of Keycode Inclusion

To enhance privacy, some evaluations were conducted with classifiers that had access only to timing data, excluding keycode identity. To assess the impact of incorporating *keycode* information (ASCII values), we trained LSTMs on the same datasets both with and without this feature. Table 3.8 shows that performance consistently improves when keycodes are available. This finding aligns with previous research [DMT+22], which evaluated simpler synthetic data and reported that key-dependent models better capture character-specific typing behavior.

Table 3.8: LSTM performance with keycodes included.

Train \ Test		KDE Univ.	KDE Bi.	KDE Bi. Rev.	KDE Tri.
KDE Univ.	ACC	1.000	0.998	0.998	0.998
	F1	1.000	0.998	0.998	0.998

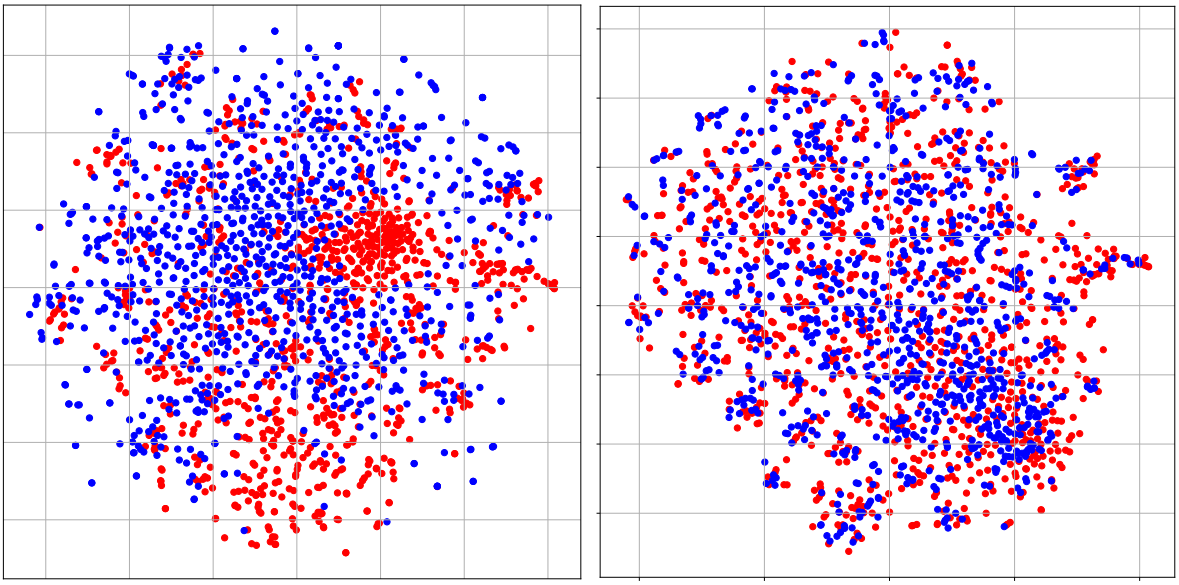
Results RQ1.5: Authenticity of Generated Keystrokes

We first assessed how closely the generated keystrokes resembled human typing patterns. Figures 3.3a and 3.3b show t-SNE embeddings comparing real human data (red), sourced from the Dhakal dataset [DFKO18], with synthetic samples (blue). While the KDE-based models produce distinct clusters that are separate from the real data, the *Diffusion* model produces distributions that more closely resemble human typing patterns.

Additional t-SNE analysis for the *KDE Bigram*, *Bigram Reversed*, and *Trigram* models showed no significant improvement in clustering quality over the *Universal* variant. In contrast, the *Diffusion* model captures both local and global context, allowing for more realistic keystroke timing distributions.

3.2.3 Discussion and Implications

Our results demonstrate that trained classifiers can successfully distinguish generated keystroke patterns from real human keystrokes, achieving higher accuracy when keycode information



(a) KDE Universal (b) Diffusion

Figure 3.3: t-SNE visualization of real (red) vs. synthetic (blue) keystroke distributions.

is available and when trained on a diverse dataset. While more advanced generative models, such as the *Diffusion* model and *KDE Trigram*, produce keystroke patterns that more closely resemble human behavior compared to simpler models, they remain detectable by our approach.

Limitations The performance of both the KDE and Diffusion models is heavily influenced by the quantity and diversity of the training data. In particular, KDE models struggle when training samples are insufficient, especially for rare keystroke transitions. Diffusion models, on the other hand, face scalability limitations, currently supporting sequences of only up to 30 keystrokes. Moreover, the generated samples often reflect generic typing behavior rather than the distinct patterns of individual users, which may introduce detection biases. Additionally, the detection models remain highly dependent on the training distributions, making them less reliable when encountering novel timing patterns.

Higher-Level Implications Keystroke dynamics offer a promising direction for bot detection, but they also pose privacy risks. Collecting timing data can reveal individual typing habits and potentially be used to re-identify users. Future research should explore privacy-preserving techniques (e.g., differential privacy) and investigate the potential of combining keystroke and mouse dynamics to create more robust detection systems.

3.3 Bot Detection Using Behavioral Analysis in High-Traffic Applications

To counter automated threats in high-traffic environments, we introduce a fast, multistep detection pipeline designed to minimize bot interactions as early as possible. The system first applies lightweight heuristics based on static features, e.g., IP addresses, before escalating to more advanced machine learning models that analyze user behavior through website traversal patterns.

Our contributions can be summarized as follows:

- **Multi-Stage Bot Detection.** We present a multi-stage detection pipeline that combines computationally efficient heuristics with advanced machine learning models for bot detection.
- **Large-Scale Evaluation.** We present a comprehensive evaluation based on a large-scale, real-world e-commerce dataset containing raw HTTP request data of customers.
- **Behavioral and Graph-Based Analysis.** We provide a detailed analysis of the impact of various behavioral attributes and graph-based features on detection performance.

To guide our investigation, we address the following research questions:

RQ1.6: What is the performance of each stage within the multi-stage detection approach in distinguishing between bots and human users?

RQ1.7: Which attributes have the greatest impact on detection performance?

RQ1.8: How does the dimensionality of a website traversal (WT) graph influence classification accuracy?

Building on these research questions, the following publication[6] summarizes an evaluation conducted on a proprietary e-commerce dataset. Our results demonstrate that the proposed approach achieves high performance, with precision, recall, and AUC exceeding 98%, outperforming alternative methods such as Botcha.

> Jan Kadel, August See, R. Sinha, M. Fischer. BOTracle: A Framework for Discriminating Bots and Humans. European Symposium on Research in Computer Security, SecAI Workshop, 2024.

The following sections describe our approach to multi-state detection using heuristics on individual requests and across a user session, summarize the results, and outline limitations.

[6]The core concept, website traversal graph approach, and evaluation design originated from the author of this dissertation. The first author implemented the methodology and performed the evaluation. The third and fourth co-authors contributed to the refinement of the manuscript.

3.3.1 Method

In this section, we summarize our approach to a multi-stage bot detection pipeline. Our methodology integrates multiple classification strategies, with some applied to the first request and others analyzing entire sessions.

Bot Detection

The proposed approach combines heuristic methods, a Semi-Supervised Generative Adversarial Network (SGAN), and a Deep Graph Convolutional Neural Network (DGCNN) within a multi-stage detection pipeline. Figure 3.4 illustrates the overall process. An incoming hit (i.e., a web server request) is first evaluated using lightweight heuristics. If the heuristic analysis confidently classifies the hit, a bot or human label is assigned. Otherwise, the hit is preprocessed: irrelevant features are discarded, and the remaining attributes are converted into a one-dimensional numeric vector using labeling, integer encoding, and one-hot encoding. Rare categorical values are aggregated into special categories to reduce dimensionality. The resulting vector is fed into the SGAN, which generates a probability score. If this score exceeds a confidence threshold $\lambda \in [0, 1]$, a label is assigned. Hits below the threshold are aggregated into sessions, converted into WT graphs, and then analyzed by the DGCNN. The DGCNN also produces a probability score, which is compared to λ, with additional hits collected as needed to improve the session representation.

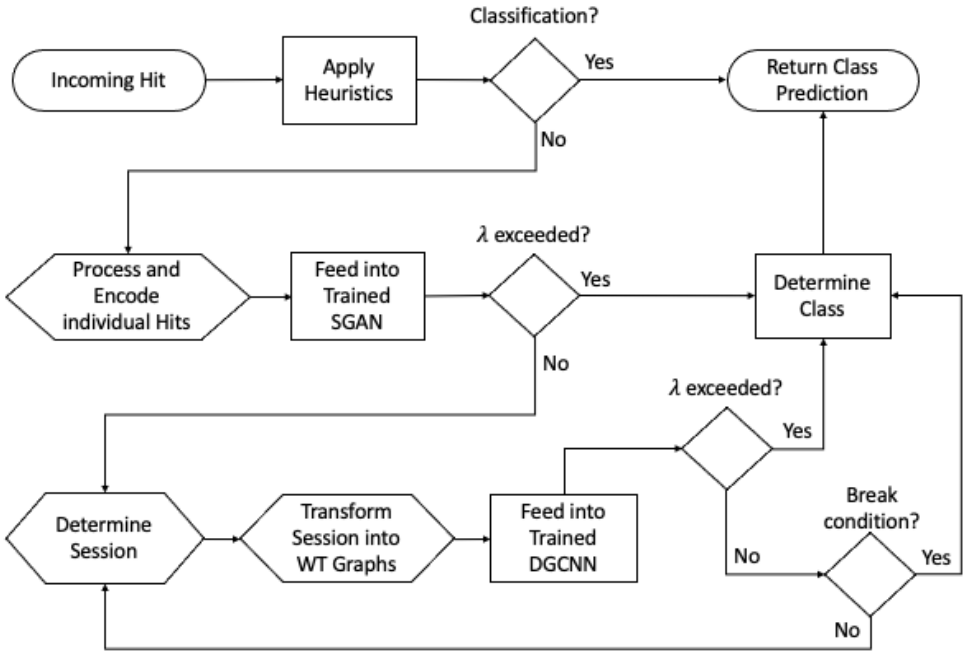


Figure 3.4: Multi-Stage bot detection pipeline process.

Bot Attribute Analysis

A thorough analysis of client attributes is essential for distinguishing between bots and human users. We divide these attributes into two groups:

Non-Behavioral Attributes:

- *Identity Attributes:* Contains static identifiers such as IP addresses.
- *Technical Attributes:* Includes browser properties such as user agents and screen sizes.

Behavioral Attributes:

- *Traversal Attributes:* Captures the navigation path through the site.
- *Interaction Attributes:* Records interactions with elements on the page.
- *Visit Attributes:* Reflects visit frequency and timing patterns.

Because identity attributes are easily spoofed, our detection approach relies primarily on behavioral and technical attributes, which are more difficult for bots to mimic.

Heuristic Detection of Obvious Bots

Heuristic methods are used to detect obvious bot activity and augment the labeled training dataset. The heuristics used by us are:

Forged User Agent: User agents such as `python-request` strongly indicate automated behavior. If the reported capabilities differ from those of legitimate browsers, the match is classified as a bot [Z$^+$15, DS11].

Regular Inter-Hit Timing: Consistent, evenly spaced requests often indicate automation, as human interactions typically exhibit temporal variability [T$^+$20, S$^+$21].

Unrealistic window sizes: Extremely small window sizes (e.g., below 50 pixels) are unusual for human users, but are often used by bots to reduce computational overhead [JKV19].

Bot Detection Using Technical Features

This stage uses a semi-supervised Generative Adversarial Network SGAN [Ode16] to classify individual hits based on both labeled and unlabeled data. The classifier transforms the preprocessed vector into a probability distribution over bot and human classes using a softmax activation. The loss of the classifier is computed using the cross-entropy function:

$$L(Y_k, p_k) = -\sum_{k=1}^{n} Y_k \cdot \log(p_k)$$

Here Y_k is the true label and p_k is the predicted probability for class k. Both the discriminator and the classifier are trained using backpropagation [RHW86]. To improve the model's ability to discriminate between real and synthetic data, the discriminator uses the ExpSum [SGZ+16] activation function:

$$E(Z) = \frac{F(Z)}{F(Z)+1} \quad \text{with } F(Z) = \sum_{z_k \in Z} e^{z_k}$$

Bot Detection Using Web Site Traversal Graph Analysis

WT graphs model user navigation behavior by representing website pages as nodes and navigation actions as edges. Edges are weighted based on visit frequency, and nodes are annotated with relevant attributes, e.g., timestamps and page types. We hypothesize that bots exhibit distinct navigation patterns, such as exhaustive, breadth-first traversal, that are rarely observed in human behavior.

The WT-based detection process consists of three steps:

1. **Feature Extraction:** Relevant attributes are extracted from raw web server logs.
2. **Graph Construction and Transformation:** The extracted features are structured as graph representations, ensuring compatibility with graph-based learning. Dynamic edge updates account for variations in session behavior.
3. **Graph-Based Classification:** A Dynamic Graph Convolutional Neural Network (DGCNN) [ZCNC18] learns hierarchical features from the constructed graphs, capturing both local and global traversal patterns to classify sessions effectively.

Table 3.9 shows the primary features considered within the WT graphs.

Table 3.9: Features included in WT graphs.

Attribute	**Description**	**Component**
First hit page name	initial page of a session	node
Detailed Pagename	Specific page of a hit	Node, Edge
Previous page name	Previous page in a hit	Node, Edge
Timestamp	Time of visit	Node Label
Page type	category of visited page	node label
Benchmark label	hit-specific benchmark identifier	node label

Additional graph-based features include node degree, node count, edge count, page type distribution, session topics, identified using the RAKE algorithm [R+10], hits per session, hits per subpage, degree centrality, and betweenness centrality.

3.3.2 Key Results

This section summarizes the evaluation of our detection pipeline and addresses three research questions: the effectiveness of each component of the pipeline (RQ1.6), the importance of individual features (RQ1.7), and the influence of the dimensionality of the WT graph on the classification performance (RQ1.8).

Data Labeling and Ground Truth

Our evaluation is based on a real-world dataset provided by Adobe from an e-commerce website with approximately 40 million monthly visits. We analyzed approximately 1.4 million visits acquired over two consecutive days. The dataset captures a diverse range of previously unknown bot types; however, it lacks definitive ground truth. To address this limitation, we labeled the data based on the following assumptions:

Human Assumption: Traffic is labeled human if it originates from verified user accounts within the hosting organization. This assumption is based on the expectation that employees are unlikely to use bots on their own site.

Bot Assumption: Traffic is labeled as bot if it originates from IP addresses associated with known cloud providers, despite the potential presence of legitimate users relying on proxies or VPNs.

This labeling strategy is more conservative than that of previous work (e.g., Botcha [DSV+21]), which assumed that bots do not engage in purchases-a claim invalidated by recent evidence showing that bots purchase high-demand products [Bro21, Par19].

Table 3.10 summarizes the number of hits classified under our initial assumptions and after applying heuristic refinement. The heuristics, designed to identify additional bot instances without affecting human classifications, increased the number of bot hits from 51,462 to 65,018, while the number of human hits remained unchanged at 7,630. Unknown classifications decreased from 723,579 to 710,023. Notably, nine human interactions were misclassified as bots, resulting in a recall rate of 0.9988.

Table 3.10: Ground truth based on initial assumptions and heuristic refinement.

Class	Initial Assumption (#Hits)	After Heuristics (#Hits)
Bot	51,462	65,018
Human	7,630	7,630
Unknown	723,579	710,023

Results RQ1.6: Detection Performance

We evaluated two primary models: one based on technical features using a Semi-Supervised GAN (SGAN), and another based solely on behavioral data using a Deep Graph Convolutional

Neural Network (DGCNN). Their performance was compared to the baseline method Botcha, configured as described in the Botcha Paper [DSV+21]. Table 3.11 shows the key metrics - accuracy, recall, precision, F1-score and AUROC - for each model. Both SGAN and DGCNN achieved high detection rates (accuracies above 98%), with SGAN showing slightly better performance in terms of accuracy and AUROC. However, the DGCNN model is more resilient against bots that mimic technical attributes, as behavior patterns are inherently more difficult to spoof. Although Botcha-RAM outperformed the other models in certain metrics, its reliance on static cues may limit its effectiveness in dynamically evolving threat environments. The DGCNN operating on WT graphs does not rely on these static cues.

Table 3.11: Comparison of detection performance between models.

Model	**Accuracy**	**Recall**	**Precision**	**F1-Score**	**AUROC**
SGAN	0.9895	0.9875	0.9189	0.9519	0.9886
DGCNN	0.9845	0.9833	0.9791	0.9812	0.9892
Botcha-MAM	0.9364	0.8383	1.0000	0.9120	0.9437
Botcha-RAM	0.9952	0.9663	0.9807	0.9735	0.9996

Results RQ1.7: Importance of Technical Feature

To identify the technical features that have the most impact on the detection, we applied the permutation importance algorithm [ATSL10] to the SGAN classifier. In this procedure, each feature is shuffled multiple times (with $K = 50$) to measure the corresponding drop in classifier performance relative to a baseline score. The importance of each feature i_d is calculated as

$$i_d = s - \frac{1}{K} \sum_{k=1}^{K} s_{k,d}$$

Table 3.12 lists the average decrease in accuracy (reported via the R^2-score and negative MSE) for the most influential features. These include browser height and width, whether Java was enabled, the user agent (excluding standard browsers such as Firefox, Chrome, and Safari), the number of (sub)page visits, and the total visit count. Notably, `post_browser_height` and `post_browser_width` emerged as the most critical features. However, these attributes are relatively easy to spoof, highlighting the need to incorporate behavioral metrics that are more resistant to manipulation.

Results RQ1.8: Importance of WT Graph Size

The evaluation dataset was partitioned using overlap clustering (70% training, 30% testing). Table 3.13 presents the results. The number of nodes represents the number of subpages accessed, while #Graphs indicates the number of instances (sessions) that accessed only a single subpage. Note that additional activity, such as page refreshes, is incorporated into the graph.

Table 3.12: Excerpt of technical feature importance scores for the SGAN classifier.

Feature	R2-Score		Negative MSE	
	μ_i	σ_i	μ_i	σ_i
post_browser_height	0.542	± 0.008	0.051	± 0.001
post_browser_width	0.287	± 0.010	0.027	± 0.001
post_java_enabled_N	0.082	± 0.003	0.008	± 0
post_java_enabled_Y	0.061	± 0.002	0.006	± 0
user_agent_Other	0.024	± 0.002	0.002	± 0
visit_page_num	0.022	± 0.003	0.002	± 0
visit_num	0.012	± 0.004	0.001	± 0

Even graphs with only one to three nodes exhibited high classification accuracy, recall, and precision, with performance further improving as graph size increased. The near-perfect accuracy was unexpected, given the classifier had access to only a limited set of features. We assume this is due to the dataset primarily containing simple bots that, for example, repeatedly refresh a single page to monitor products. A controlled test with a dataset containing more realistic bots would be beneficial to further evaluate this behavior.

Table 3.13: Classification performance as a function of WT graph size.

Nodes	# Graphs	ACC	Recall	Precision	F1-Score
1	26137	0.998	0.981	0.998	0.990
2	17066	0.973	1.000	0.974	0.986
3	3533	1.000	1.000	1.000	1.000
4	371	0.998	0.999	0.999	0.999
5	251	0.998	1.000	0.998	0.999
6	101	0.998	1.000	0.998	0.999
7	526	0.997	1.000	0.997	0.998
8	1579	1.000	1.000	1.000	1.000
9	1175	1.000	1.000	1.000	1.000
10	108	1.000	1.000	1.000	1.000

3.3.3 Discussion and Implications

Our evaluation indicates that the proposed bot detection framework is both robust and effective across different detection modules. The integration of technical and behavioral analysis achieves high accuracy and provides adaptability to the evolving characteristics of automated traffic.

Limitations There are several limitations to consider. First, the inability to share the underlying dataset due to privacy constraints restricts external validation and comprehensive benchmarking against related work. While we benchmarked against another approach, we could not test on a dataset with ground truth. At the time of writing, no such dataset

was available. The primary challenge is that publishing such a dataset would expose user behavior and potentially personalized data, raising privacy concerns and potentially violating data collection agreements or e-commerce platform policies. Alternative approaches, such as generating synthetic datasets or using federated evaluation methods, could mitigate this limitation but may not fully capture real-world bot behavior. Finally, the approach may fail to identify bots that perfectly mimic human behavior. However, such bots inherently operate less efficiently, as they are slowed down by the need to replicate natural user interactions.

Higher-Level Implications The results suggest that forcing bots to mimic human behavior significantly reduces their efficiency, as they are slowed down by the constraints of human-like interactions. This undermines key advantages such as fast task execution. Consequently, bot developers must design increasingly complex systems, which raises operational costs. While adversaries can partially offset these costs by deploying a larger number of bots, these bots, too, are ultimately slowed down, making large-scale automation less effective.

3.4 Summary

This chapter examines the advantages and limitations of passive anomaly detection methods for identifying bot traffic through user behavior analysis. The results demonstrate that machine learning can reliably differentiate between genuine human interactions and synthetic bot behavior.

In particular, we achieved improved classification accuracy for mouse and keyboard data compared to existing approaches, especially in scenarios involving more realistic synthetic bot behavior. The synthetic mouse movements we generated deliberately mimic human-like trajectories, incorporating slight curvatures and irregular paths rather than unrealistic straight-line movements. Regarding keyboard inputs, our method accounts for key dependencies, recognizing that keystroke timing and rhythm are influenced by adjacent characters. For instance, typing sequences such as *aa* exhibit different timing characteristics than sequences like *a@*. For classification on request data, we analyzed a real e-commerce dataset, comparing our approach to an existing method. We found that heuristics, technical features, and session-wide detection of website traversal yielded strong performance in identifying bots.

A significant challenge in bot detection research is the lack of labeled, standardized, and publicly accessible datasets containing both human and bot behaviors. This limitation hinders direct performance comparisons across different methodologies. To address this issue, we utilized publicly available datasets documenting authentic human mouse movements and keystrokes. When suitable datasets were unavailable, we reimplemented existing methods from literature or used third-party tools to generate the necessary data. Where applicable datasets existed, we evaluated multiple detection methods to enable comprehensive benchmarking and ensure methodological rigor.

Detection methods, including the approaches presented in this dissertation, effectively identify bots by distinguishing them from real user behavior. However, they rely on the assumption that bots exhibit detectable differences, which may change as adversaries develop more sophisticated evasion techniques. For mouse and keyboard data, we generated synthetic datasets from scratch

and successfully detected bots. While attackers could attempt to record and replay or augment human-like input to evade detection. Similarly, for website traversal graphs, recorded patterns could be reused to simulate authentic user behavior, but this would limit a bot's adaptability and efficiency.

While these approaches perform well, future improvements may be necessary to address evolving attack strategies. However, by forcing bots to mimic human behavior more precisely, these methods inherently reduce their effectiveness, increase operational costs, and limit large-scale automation. Further, all detection approaches in this work were generic, meaning they were designed to identify bots in general and could be applied to different application scenarios.

Chapter 4

Automated Analysis and Control of Binaries

In the previous chapter, user interaction patterns, such as mouse and keyboard behavior, were utilized for bot detection. While this approach remains effective against many automated scripts, its long-term viability is challenged by advancements in machine learning, which at some point will replicate human input behavior with such high fidelity that it becomes indistinguishable from genuine user activity. As a result, such generic detection approaches alone become less reliable. Consequently, more targeted reactive defense techniques are required, focusing on the distinct behavior and operational characteristics of specific bots. For example, vulnerabilities can be identified within a bot's communication protocol, enabling the development of tailored detection and prevention mechanisms. In web-based scenarios, this may involve analyzing how the bot constructs requests and interacts with the service. Similarly, on gaming or social media platforms, detection efforts can focus on the bot's request patterns and communication behavior. While obtaining the binary is not always possible, many bots are publicly available[1][2] or can be purchased.

Unfortunately, most malicious bots remain closed source and are often deliberately obfuscated. Although powerful reverse engineering tools exist (e.g. `IDA`[3] and `Ghidra`[4]), analyzing these binaries still requires significant expertise and time (see Section 2.1). To address this challenge, our work introduces an automated approach to accelerate the reverse engineering process. The core of our methodology involves identifying *Points of Interest (POIs)* in binaries. POIs are specific instructions or memory addresses that meaningfully interact with specific data elements, which we refer to as *Items of Interest (IOIs)*. These IOIs can be derived from sandbox observations, existing threat intelligence, or expert knowledge. For example, in the case of ransomware that reads and encrypts files, providing plaintext or ciphertext content as an IOI allows our approach to identify the corresponding addresses of instructions, thereby labeling these instructions as POIs.

We investigate the applicability of this approach by applying it to representative malware. Specifically, we focus on identifying key functionality, such as ransomware encryption routines, and instrumenting malware for runtime monitoring, for instance, to enumerate infected peers in a P2P botnet. More specifically, our contributions are as follows:

[1] Example bot: `https://github.com/subzeroid/instagrapi`
[2] Example bot: `https://github.com/Xian55/WowClassicGrindBot`
[3] `https://www.hex-rays.com/`
[4] `https://ghidra-sre.org/`

- **Method for POI Identification.** We present a systematic method for identifying binary instructions that interact with relevant IOIs, such as file content or network requests. This approach allows analysts to quickly focus on core components of the executable.
- **Confidence-Based POI Ranking.** Our methodology assigns each POI a confidence score that reflects the degree to which it interacts with the target data. This scoring mechanism helps to filter out routine functions, e.g., `memcpy` while prioritizing more relevant ones, e.g., `encrypt_file`.
- **Automated Reverse Engineering Support.** We implement a workflow that integrates with popular reverse engineering suites, `IDA` and `Ghidra`, to assist analysts in their investigations. In addition, we present a prototype that can instrument malware binaries at runtime, facilitating tasks such as botnet enumeration without extensive manual effort.
- **Empirical Evaluation.** We evaluate the effectiveness of our approach in two primary areas:
  1. *Ransomware Analysis:* We apply our POI-based technique to the *Locky* and *WannaCry* ransomware families and demonstrate that we can quickly identify their file encryption routines.
  2. *P2P Botnet Monitoring:* Our runtime instrumentation is tested on four different P2P botnets (*ZeroAccess*, *Sality*, *Nugache*, and *Kelihos*), allowing us to enumerate peers by tracking POIs that interact with known IP addresses.

We structure our investigation around the following research questions:

- **RQ2.1 Identification:** *How effective is our method at identifying high-quality POIs?* We aim to evaluate the accuracy of our approach in detecting instruction addresses that interact *exclusively* with IOIs, thereby identifying high quality POIs and reducing the manual effort required to analyze numerous irrelevant POIs.
- **RQ2.2 Performance Impact:** *What performance overhead does our instrumentation introduce, and how does it affect malware execution?* Since instrumentation can degrade performance or disrupt time-sensitive routines such as TCP streams, we evaluate potential side effects, including socket timeouts and anti-instrumentation defenses.
- **RQ2.3 Confidence Score Quality:** *How accurately does the confidence score reflect the relevance of the extracted indicators?* We investigate whether higher-scoring POIs improve IP address extraction, using P2P botnets as a representative case due to their obfuscated and closed-source nature, which makes them particularly challenging to analyze.

Building on these research questions, the following publication[5] summarizes our findings on POI-based detection. Our findings indicate that POIs effectively highlight critical bot functionality,

[5]The approach to multi-instruction POIs, including the corresponding confidence scores and evaluations (including ransomware), originates from the author of this dissertation. The multi-instruction POI detection methodology was refined from the author's master's thesis. The concept, implementation, and evaluation of single-instruction POIs were contributed equally by the first and second authors. The second author developed the confidence scoring mechanism and conducted the practical evaluation using automated botnet monitoring. The third author proposed the core idea, while the third and fourth co-authors provided feedback and helped to refine the final paper.

accelerating reverse engineering efforts. Empirical validation confirms that identified POIs consistently correspond to essential operations, such as encryption and network communication logic.

> August See, M. Gehring, M. Fischer, S. Karuppayah. Binary Sight-Seeing: Accelerating Reverse Engineering via Point-of-Interest-Beacons. Annual Computer Security Applications Conference, 2023.

In the following sections, we discuss our design choices, prototype implementation, and evaluation results. The evaluation demonstrates that the POI-based approach effectively identifies relevant instruction addresses while maintaining a manageable performance overhead that does not interfere with the execution of the monitored software.

4.1 Method

Our methodology for identifying and exploiting points of interest (POIs) in binaries consists of five main steps, as summarized in Figure 4.1. These steps are designed to minimize the manual effort involved in reverse engineering and to facilitate automated analysis scenarios, such as P2P botnet monitoring. We then describe two concrete applications of our approach: first, a generic discovery tool that integrates with Ghidra and IDA, and second, an automated botnet monitoring system, *PinPuppet*. The latter serves as an example of applying POIs to a complex reverse engineering task, demonstrating how peer-related information in P2P botnets can be uncovered without prior protocol knowledge, despite their obfuscated and closed-source nature.

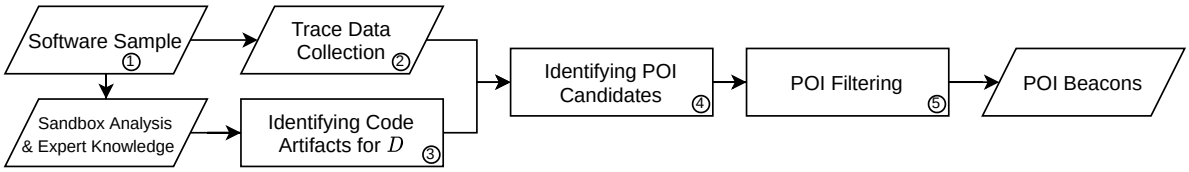


Figure 4.1: Overview of the POI discovery process.

Step 1: Obtaining a Software Sample

The first step is to obtain a sample of the software of interest. For malicious software, samples are often acquired from honeypots, malware repositories[6], infected machines, or underground forums where bot-related tools are traded and distributed. Since our research is primarily focused on malicious applications, we assume that the sample under investigation is typically closed source and obfuscated.

[6] https://github.com/ytisf/theZoo

Step 2: Collecting Trace Data

Next, we record the interactions between the binary and its environment by instrumenting the software using a dynamic binary instrumentation (DBI) tool. Examples of such frameworks include Intel Pin [LCM+05], DynamoRIO, and Frida. During execution, we log each *(data, address, access_type)* tuple, collectively referred to as *trace data*. Here, *data* represents the values read from or written to memory or registers. The *address* corresponds to the instruction pointer. The *access_type* indicates whether the operation is a read or a write. This fine-grained view of the program's runtime behavior captures low-level data manipulations that can later be matched against indicators provided by the analyst.

Step 3: Identifying IOIs

The next step is to identify a set D of Items of Interest (IOIs) that contain data elements relevant to the analyst's objectives. For example, when investigating how malware contacts its command and control (C2) servers, IP addresses or domain names extracted from a sandbox report [Gua10] could be used as IOIs. To maximize detection accuracy, analysts must consider various representations (e.g., little-endian versus big-endian integers) and potential encodings (e.g., binary or ASCII). However, not all values are equally appropriate: indicators that are too generic (e.g., 80, for port 80) run the risk of matching irrelevant code paths. Therefore, analysts should choose distinctive values, such as full IP addresses or unique registry keys, to identify instructions of functional significance.

Step 4: Identifying POI Candidates

We define a POI as an instruction address that *reads or writes* an IOI from D. However, matching these data elements in the trace is not always straightforward. Some indicators (e.g., a 32-bit integer) may align with the read or write operation of a single instruction, while others (e.g., a string) may be processed across multiple instructions. To handle this variability, we distinguish between two search strategies:

1. Single-Instruction POI Search This approach compares each data access in the recorded trace with each IOI in D. If an instruction directly processes an IOI in a single operation (e.g., `mov eax, [esp]`), the corresponding instruction address is marked as a *POI candidate*.

2. Multi-Instruction POI Search Identifying multi-instruction POIs is computationally challenging due to the variability in the order and manner in which IOI-related data is accessed. With access patterns that can be sequential, reverse, or random, and an unknown number of involved instructions, a brute-force approach quickly becomes infeasible. To efficiently address this problem, we employ two custom lookup tables: a memory map and an identifier map. The memory map associates addresses with the instructions that read or write data, while the identifier map tracks memory region sizes and dynamically updates them as memory usage changes. Together, these tables enable a fast memory lookup algorithm that efficiently compares memory contents to IOIs and records the corresponding instruction addresses as POI candidates.

Step 5: Ranking and Filtering POIs

Once a set of POI candidates has been identified, we assign each one a *confidence score* to help analysts to prioritize high-quality POIs. The core idea is to measure how *exclusively* a POI processes elements from D compared to unrelated data. Scores range from 0 to 1, where a score of 1.0 indicates that the POI interacts exclusively with IOI-related data. In contrast, general-purpose functions (e.g., `memcpy`) often appear as POIs but are associated with diverse data, resulting in lower scores. A threshold-based filter then excludes POIs with low scores to reduce noise.

- *Single Instruction Score (*score_S*):* For each instruction p, we count how many times an IOI appears in the trace and compare it to the total number of data accesses made by p:

$$\text{score}_\text{S}(p) = \sum_{d \in D} \frac{C(d, \text{trace}(p))}{|\text{trace}(p)|}$$

  Here, $C(x, L)$ counts the occurrences of x in the list L.

- *Multiple-Instruction Score: (*score_C*)* For multi instruction POIs, we track how many IOI bytes a candidate instruction reads or writes, relative to its total data accesses:

$$\text{score}_\text{C}(p) = \sum_{s \in P} \frac{C(\text{addr}(p), s)}{\#p}$$

  where P is the set of memory write sequences that reconstruct IOIs from D, and $\#p$ is the total number of bytes accessed by instruction p.

4.1.1 Generic POI Discovery

We developed a generic tool to automate the five-step process shown in Figure 4.1, using Intel Pin as the DBI framework along with Python scripts to search for single and multiple instruction POIs. The tool requires three inputs from the analyst:

1. A *binary sample* (e.g., a suspicious executable).
2. A set of IOIs (D) obtained from sandbox logs or domain knowledge.
3. (optionally) a confidence score threshold to exclude low quality IOIs.

Upon execution, the tool outputs a ranked list of POI candidates along with detailed trace information. Custom plugins for IDA and Ghidra highlight these POIs, allowing analysts to navigate directly to the instructions associated with the relevant functionality (e.g., file encryption routines in ransomware). The flexibility of the POI framework also enables automated applications beyond manual reverse engineering.

4.1.2 Use Case: Automated P2P Botnet Monitoring

In addition to supporting manual reverse engineering, POIs can facilitate automated analysis. As an example, *PinPuppet* is a prototype tool designed to dynamically instrument live malware samples and enumerate P2P botnets at runtime.

P2P Botnet Characteristics

In a P2P botnet, infected machines maintain peer lists that allow for decentralized communication. This mechanism, often referred to as *membership management* (MM), complicates botnet tracking due to the absence of a centralized command and control server. MM typically operates as follows: each bot maintains an internal peer list and attempts to contact entries from this list. Upon successfully reaching a live peer, the bots exchange information about other active peers, thereby updating their respective peer lists dynamically. However, by identifying POIs that handle peer-related data, PinPuppet can iteratively probe and map the structure of the botnet.

PinPuppet Workflow

Figure 4.2 illustrates the architecture of the system:

1. **Puppet VM**: Runs the malware sample under Intel pin instrumentation.
2. **Agent**: A lightweight service that allows remote upload, execution, and retrieval of results (e.g., trace logs and POI information).
3. **Router**: Intercepts and manipulates network traffic to redirect bot communication at runtime.
4. **Puppeteer**: Coordinates the crawling process, including snapshot management, peer selection, and result aggregation.

Step-by-Step Crawling Process The following steps describe how PinPuppet leverages POIs to automate the exploration of a live P2P botnet during execution.

1. *Trace Collection and Baseline IOIs:* The malware is first executed to generate a trace log. In parallel, any observed peers (e.g. IP addresses or ports) are extracted to form the initial IOI sets D.
2. *POI Candidate Identification:* The Puppeteer scans the trace to identify statements that manipulate D.
3. *Filtering and Confidence Scores:* Confidence scores are calculated, and POIs with low scores are discarded.

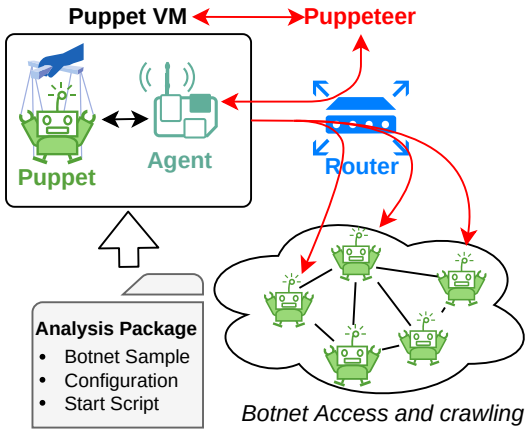


Figure 4.2: Architecture of the PinPuppet system.

4. *Crawling Primitive:* The router redirects outgoing connections to selected peers, allowing PinPuppet to iteratively explore the network by following newly discovered peers.

By iteratively applying this process, PinPuppet enumerates significant portions of the botnet without requiring explicit protocol knowledge. All critical information is inferred at runtime from the identified POIs.

4.2 Key Results

This section summarizes the evaluation results of our reverse engineering methodology applied to malicious program binaries-specifically, ransomware and botnets. We address three primary research questions: (RQ2.1) the effectiveness of identifying high-quality POIs, (RQ2.2) the performance impact introduced by the instrumentation, and (RQ2.3) the reliability of the confidence score in improving the quality of results. To demonstrate the applicability of POIs in reverse engineering challenging binaries, we use examples of complex, hard-to-analyze malware. Specifically, *Locky* and *WannaCry* serve as ransomware samples for identifying encryption routines, while four real-world P2P botnets (*ZeroAccess*, *Sality*, *Nugache*, and *Kelihos*) are analyzed within an isolated local testbed as a ground-truth-based case study.

The evaluation was conducted in two stages. First, we performed a manual analysis focused on ransomware. Subsequently, we conducted a more in-depth evaluation on P2P botnets to gather information regarding the quality of POIs.

4.2.1 Setup

For the ransomware evaluation, we executed samples such as *Locky* and *WannaCry* within a virtual machine, monitoring file access to identify the affected files. From these, we extracted segments of the encrypted file contents to serve as IOIs.

To obtain quantitative insights, we analyzed the behavior of *PinPuppet*, which uses POIs to extract peers from P2P botnets, as described in Section 4.1.2. All experiments were conducted in virtual machines running on a Proxmox hypervisor within a network isolated from the Internet. Multiple VMs were infected with instances of the botnets. One instance was instrumented prior to any communication between the bots. We recorded the IP addresses and ports contacted by the instrumented bot from its fixed peer list, using them as IOIs (e.g., IP *1.2.3.4* or port *60124*). Based on this a priori information, we aimed to identify and score the instructions responsible for processing the peer list. Subsequently, the bots were allowed to discover each other on the network. By monitoring the behavior of the previously identified POIs, we observed which data they accessed and verified whether valid IP addresses were involved, leveraging the known ground truth from our controlled infection and IP distribution.

4.2.2 Results RQ2.1: Identification

This section presents the results for the research question: *How effective is our method at identifying high-quality POIs?*

The ransomware analysis was straightforward. Manual reverse engineering confirmed that the function containing multiple POIs with the highest aggregated score was responsible for the encryption process.

For P2P botnet monitoring, Figure 4.3 shows the distribution of confidence scores for IP-related POIs across the evaluated botnets. It displays the number of register-based, memory-based, and contiguous memory (multi-instruction) POIs identified for each botnet, grouped into 10% confidence score bins.

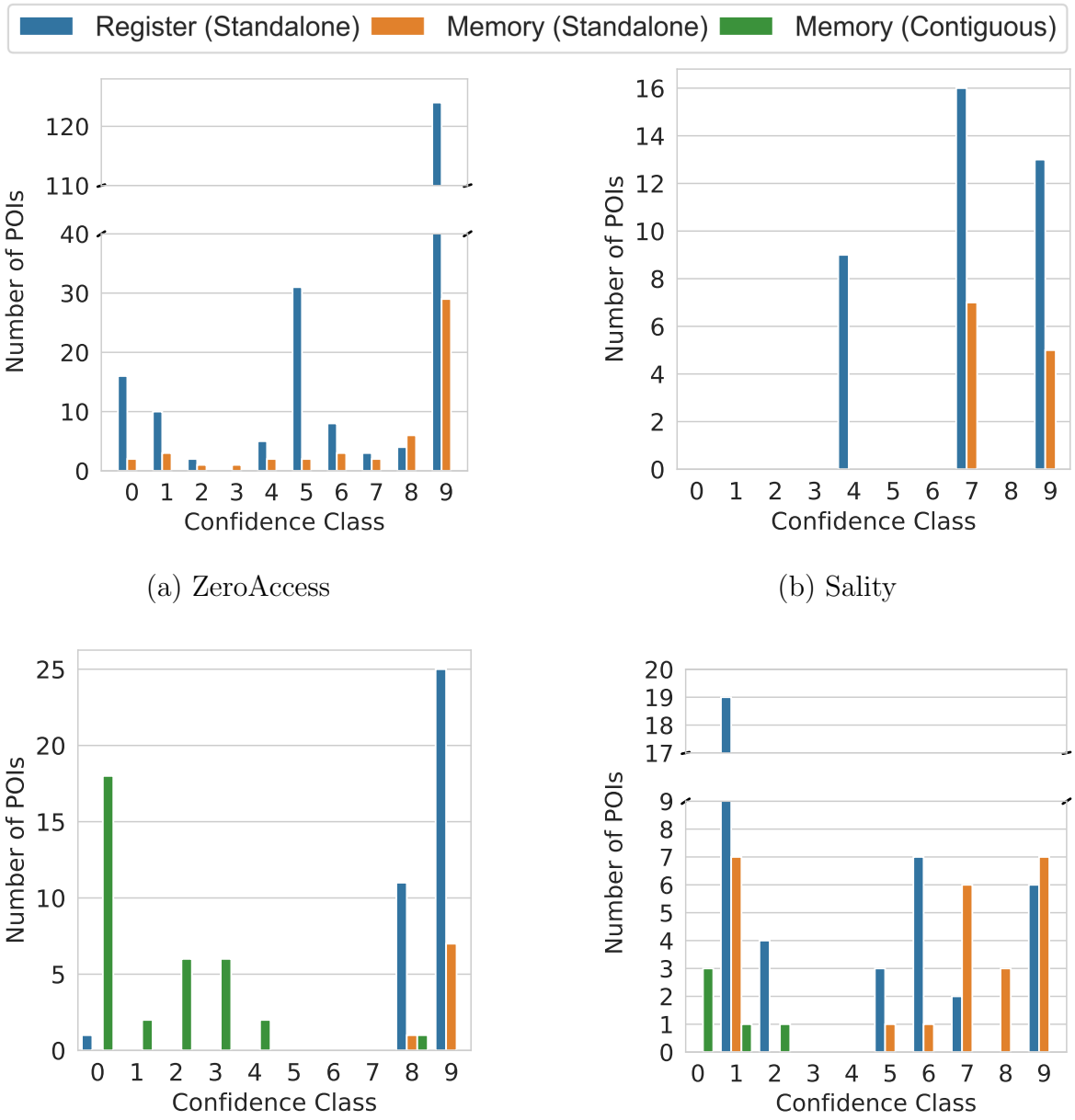
We see that each botnet contains a subset of POIs with confidence scores greater than 0.8 (i.e., confidence class 8 or higher), suggesting a specialization in handling specific IP addresses. Furthermore, multi-instruction POIs were detected in *Nugache* and *Kelihos*, attributed to the storage of peer data as ASCII sequences.

Overall, this high-confidence subset of POIs was sufficient to locate key code regions, such as peer-handling routines and encryption functions. The validity of POIs was verified through manual inspection. This result confirms that our method effectively identifies meaningful POIs while minimizing the inclusion of irrelevant instruction addresses. Section 4.2.4 describes the role of the confidence score in more detail.

4.2.3 Results RQ2.2: Slowdown

This subsection addresses the overhead introduced by our instrumentation. Excessive overhead may lead to two issues: the malware binary may detect that it is being monitored, or it may fail to execute correctly due to timing-related disruptions, such as TCP connection timeouts during data exchange. *What performance impact does our instrumentation introduce, and does it disrupt malware execution?*

We measured performance overhead by running each sample under Intel Pin [LCM+05] and comparing its execution time to the uninstrumented baseline. Our measurements showed



(a) ZeroAccess

(b) Sality

(c) Nugache

(d) Kelihos

Figure 4.3: Distribution of confidence scores for IP-based POIs in ZeroAccess, Sality, Nugache and Kelihos. Each plot shows the number of POIs per confidence score range (horizontal axis).

moderate overhead without any functional breakdowns. For example, in one of its membership management cycles, *Kelihos* exhibited a 6.5% runtime increase (approximately 13 additional seconds). Other botnets exhibited similar or smaller slowdowns, and none of the samples failed to execute or lost network connectivity. While the added instrumentation inevitably increases execution time, it did not trigger any observed anti-instrumentation mechanisms or led to network communication failures.

4.2.4 Results RQ2.3: Quality of Confidence Scores

This section analyzes the influence of our quality score. Given that our approach identifies multiple POIs (Section 4.2.2), we evaluate whether POIs with high confidence scores effectively prioritize relevant behavior, that is, whether they access all correct IPs within the network on which we deployed the botnets, or whether the results include noise. *How accurately does the confidence score reflect the reliability of the extracted indicators?*

To assess the relationship between confidence scores and extraction accuracy, we compared the set of peers extracted by each POI to the botnet's ground-truth peer list as well as from our knowledge of the network. Figure 4.4 shows a scatter plot of correctness scores, indicating whether a POI actually accessed an IP address from our ground truth, in relation to its confidence score. The vertical green dotted line indicates the 0.8 threshold we used when filtering POIs. The majority of POIs with scores above 0.8 exhibit high correctness, with few false positives. Furthermore, most data points lie on or above the diagonal line, indicating that the actual correctness often matches or exceeds the predicted confidence. This trend suggests that the confidence score is conservative, it tends to underestimate rather than overestimate performance. As a result, it provides a reliable filtering mechanism that allows analysts to focus on the most relevant parts of the code. This makes the confidence score particularly useful for both automated tasks, such as botnet crawling, and for prioritizing functions during manual reverse engineering.

4.3 Summary and Discussion

Our reverse engineering approach allows data-driven reverse engineering of closed-source binaries by identifying and ranking critical instructions, referred to as *Points of Interest (POIs)*. These POIs directly or indirectly interact with specific data elements, termed *Items of Interest (IOIs)*, which can be derived from sandbox analyses, execution traces, or expert-defined criteria. By leveraging automated dynamic binary instrumentation, our approach reduces manual effort by capturing detailed runtime traces to pinpoint relevant code regions.

We validated our methodology through practical security applications, including the automated crawling of peer-to-peer botnets (e.g., *ZeroAccess*, *Sality*, *Nugache*, and *Kelihos*) and the analysis of ransomware behavior (e.g., *Locky*, *WannaCry*). Evaluation results demonstrated that high-confidence POIs consistently identified critical malware functionalities, such as file encryption routines and peer management mechanisms, thereby significantly reducing manual reverse engineering efforts. Performance tests indicated a modest instrumentation overhead of approximately 6.5%, with no noticeable impact on malware execution.

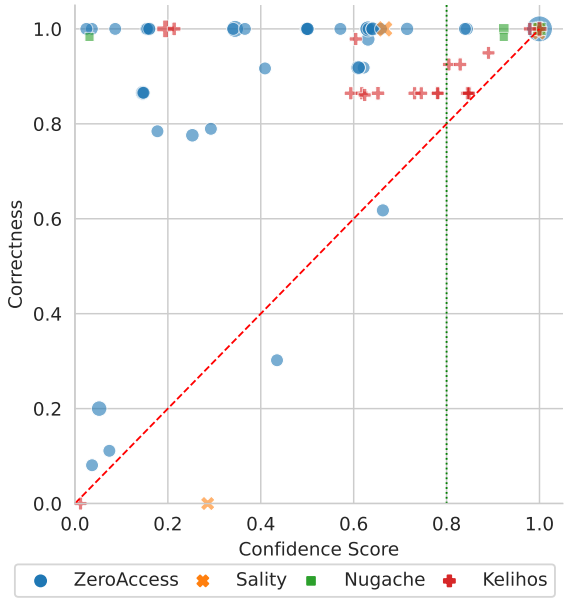


Figure 4.4: Correctness versus confidence score for identified POIs. Points above the dashed diagonal indicate that the actual correctness exceeds the POI score, suggesting a conservative, non-overestimating metric. The green dotted line marks the filtering threshold of 0.8.

Limitations Despite its ability to reduce analyst workload, our method remains susceptible to certain adversarial techniques, such as code virtualization obfuscation [Col01], which causes the same instructions to be used for different data, thereby limiting the reliability of our confidence score. Adversaries may also insert irrelevant instructions to mimic legitimate IOI interactions, generating extraneous and misleading POIs. Although such tactics increase analysis complexity, our method remains effective, as critical data must still be accessible in memory before system calls or library function executions.

Higher-Level Implications This approach facilitates the analysis of closed-source binaries in a data-driven manner, significantly reducing manual effort. However, it also raises dual-use concerns. Adversaries could exploit similar methodologies to study legitimate software or refine their obfuscation techniques to evade detection.

Chapter 5

Bot Prevention through Endpoint and Protocol Obfuscation

Rather than detecting bots, prevention offers a proactive approach to defending against them. Although detection strategies and binary analysis, as described in the previous chapters, help us identify and understand how bots operate, adversaries continue to refine their methods to mimic legitimate user behavior, thus countering detection approaches. To address this, prevention techniques aim to prevent or mitigate bot activity before it becomes practical or scalable.

One prevention approach focuses on increasing the difficulty of initial bot development and deployment by making the targeted application harder to analyze. This strategy employs obfuscation techniques. The related work is further discussed in Section 2.3.2. Existing obfuscation approaches [Sof21, Tec22] primarily focus on impeding reverse engineering of client software but do not address the threat posed by API-based bots (Section 2.1.1). The problem lies in the fact that the protocol itself cannot be obfuscated only on client-side, as it must remain consistent between the client and server to ensure proper communication. Since both rely on mutually understandable protocols and endpoints to maintain synchronization, these methods only hinder initial bot creation by obfuscating client-side components. However, API-based bots typically bypass obfuscation that exclusively focuses on the client-side, by directly extracting protocols, endpoints, and payload formats from client software. Moreover, traffic interception techniques available at both device and router levels further diminish the effectiveness of traditional client obfuscation.

Recognizing these limitations, this chapter proposes extending obfuscation, initially devised as a defense against reverse engineering (discussed in Section 2.1.2), to the network endpoints and communication protocols themselves. Assigning unique endpoints and protocols to individual clients forces each bot to adapt to a client-specific protocol, significantly increasing the cost and complexity of sclaing bots and thus large-scale automated attacks.

The main contributions of this chapter are summarized below:

- Section 5.1 extends the concept of obfuscation to protocols, proposing the assignment of unique communication protocols to each client. This technique ensures bot prevention relying solely on protocol-level obfuscation to prevent bot scaling.
- Section 5.2 introduces a novel and more lightweight method for obfuscating endpoints. This method is designed for lightweight bot prevention strategies specifically designed to network services.

5.1 Polymorphic Protocols for Limiting Protocol Analysis and Bot Scalability

Communication protocols are a common target for bot developers aiming to automate interactions with services. While existing defenses often focus on obfuscating client-side components such as binaries, HTML, or JavaScript [Col01, Jsc, Sof21, Tec22], the underlying protocols typically remain static and predictable. To address this gap, we explore the use of *polymorphic protocols* as a means to increase the effort required for bot replication. By assigning each client a customized protocol variant, our approach requires attackers to reverse engineer individual instances. For each bot they create, a different protocol must be analyzed, thereby increasing the cost of automated abuse. This approach is conceptually similar to current practices such as embedding API keys in applications and rotating them upon release, which forces attackers to extract the key in order to build a bot[1]. However, polymorphic protocols generalize this idea by potentially including API keys while also obfuscating the protocol logic itself. As a result, they offer at least the same level of protection as API keys, while additionally requiring attackers to extract the protocol structure, rather than simply updating a single data dependency in their bot code.

Polymorphic protocols are particularly useful for services with stateful user interactions, such as social media platforms and online games. In these contexts, large numbers of automated accounts are often required to achieve meaningful impact. For example, a single bot may be sufficient to scrape data from a website, but on social media, one bot can typically like or share a post only once. This creates a strong incentive to scale bot deployments. By constantly changing data formats and field definitions per client, the attacker's cost of scaling bots increases. In the following sections, we investigate the technical feasibility and evaluate the performance and overhead of implementing polymorphic protocols in real-world applications.

Our primary goal is to increase the cost of duplicating bots while keeping performance and organizational overhead low:

- **Client-Specific Protocols.** We introduce *Polymorphic Protocols*, which assign each client a unique protocol variant. The core idea is that protocol messages can be encoded and ordered differently across multiple rounds, resulting in complex and diverse protocol structures. Our approach builds on strong obfuscation techniques for binaries [Col01, Tec22] and censorship-resistant protocol obfuscation [DCS15, MMLDG12]. To the best of our knowledge, we are the first to apply protocol obfuscation specifically to combat bot replication at scale.
- **Reference Implementation.** We provide an implementation for Java and Protobuf,[2] that can be easily integrated into continuous integration (CI) pipelines. Given a standard Protobuf definition file, our system automatically generates custom protocol variants.
- **Performance Evaluation.** We measure the computational overhead of polymorphic protocols and show that our approach can be applied without affecting user experience.

[1] `https://github.com/8mas/SINoALICE-API/`
[2] `https://github.com/UHH-ISS/polymorphic-protocols`

Our research is guided by the following questions:

RQ3.1: Can polymorphic protocol obfuscation limit bot scalability, and how does it compare to unique, client-specific API keys?

RQ3.2: What is the performance overhead of polymorphic protocols? Does this affect legitimate users and service providers?

Building on these research questions, the following publication[3] summarizes our findings on limiting bot scalability. We have integrated our approach into an existing Java/Protobuf ecosystem with minimal effort. Depending on deployment choices and protocol configurations, polymorphic protocols introduce only a small performance penalty for end users, while increasing the cost of scaling the bot. The ability to generate client-specific protocols on-demand further complicates the attacker's task of reusing a single protocol implementation across multiple accounts.

> August See, L. Fritz, M. Fischer. Polymorphic Protocols at the Example of Mitigating Web Bots. European Symposium on Research in Computer Security, 2022.

In the following sections, we discuss the construction and application of polymorphic protocols, summarize the key findings of our evaluation, and outline limitations and potential improvements.

5.1.1 Method

This section presents our approach to generate and deploy *polymorphic protocols*. By assigning a unique protocol variant per client, our method significantly increases the cost to duplicate API bots.

Overview of Polymorphic Protocols

Polymorphic protocols customize the message formats that clients use to communicate with a service. Instead of relying on static tokens (such as API keys) that can be easily duplicated, polymorphic protocols generate structurally different protocols for each client. This forces adversaries to reverse engineer or recreate a new protocol per bot, which quickly becomes too expensive and time-consuming. Figure 5.1 outlines the main components of the method:

A *Protocol Specification* defines the *base protocol*, including the structure and semantics of messages. Each client is associated with a *Client Identifier*, such as a user ID, which is commonly used in many applications (see Section 5.2.1 for more examples). Additionally, a *Secret Seed*, a private random value, ensures that the resulting client protocol is both unique

[3]The core idea, solution approach, and evaluation design for this publication stem from the author of this dissertation. The second author performed the implementation and evaluation. The third author contributed to the refinement of the work.

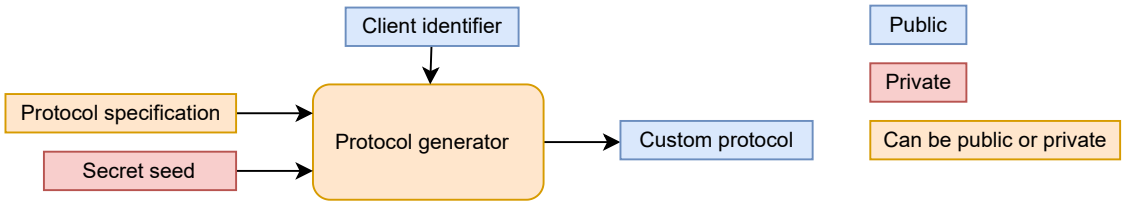


Figure 5.1: High-level view of polymorphic protocol generation. A base protocol, client ID, and secret seed are fed into a *protocol generator*, which outputs a client-specific custom protocol.

and unpredictable. Building on this foundation, the protocol is instantiated for each client through a generation process.

Protocol Generation. The *protocol generator* processes the base protocol and applies transformations such as permutations, re-encodings, or insertions to it, to derive a *custom protocol*. These transformations are chosen deterministically based on the client ID and the secret seed. When an adversary duplicates a client, all clones share the same protocol signature, which can be detected and then blocked. To reason about these transformations formally, we introduce a simple model of protocol structure and semantics.

Formal Model

We define a protocol as $\mathcal{P} = \{F, S\}$, where $F = \{f_1, f_2, \ldots, f_n\}$ denotes the set of message format specifications and $S = s_1, f_2, \ldots, s_m$ represents their semantics. A semantic s can be associated with one or multiple messages. A message format specification f may take the form $f = \{\textit{field1} : \text{int}, \textit{field2} : \text{string-base64}, \ldots\}$. The deterministic generator $G(\mathcal{P}, \text{client}, \text{seed}) \mapsto \mathcal{P}'$ applies transformations T_x that convert format specifications F, or subsets thereof $N \subseteq F$, into new variants N'. Transformations must be computationally efficient and unambiguously invertible, ensuring that valid messages can be easily decoded into the original protocol message.

Protocol Transformations

There are various transformation methods, each with distinct properties. We distinguish between *unconditional transformations* and *conditional transformations*.

Unconditional Transformations. These transformations do not require any semantic knowledge of the message and can be applied to any message without affecting cross-message functionality. This includes all possible permutations, encodings, or additions to a format specification.

Permutation Shuffle fields or field bytes within a message.

Radix Convert fields to a different base (2–255).

Encryption Encrypt selective fields.

Dummy Bytes Append random bytes anywhere in the message.

Hash Hash data in the message and append the digest (e.g., SHA-1).

Such transformations may include cryptographic operations (e.g., field encryption) to further complicate reverse engineering [WJC+09].

Conditional Transformations. These require knowledge of message semantics and thus cannot be applied to every message without considering semantics:

Delay Delay non-urgent messages.

Swap Rearrange fields of a messages.

Split Split a logical message into several smaller ones.

Merge Merge messages into one.

Custom Logic Delegate domain-specific calculations to the client.

Semantics (e.g., ordering dependencies, timing constraints) must be modeled to ensure that delayed or reordered messages do not break the application's logic.

Transformation Metrics. To evaluate the effectiveness of transformations, we use three metrics derived from a sampled subset of messages, $M_s \subseteq M$, where M denotes the set of valid messages. Let T_{xi} and T_{xj} represent two transformations within the same transformation class x, e.g., radix transformation, and let $C(\cdot)$ be a compression function. Furthermore, let D_{1x} denote a set of transformation indices, and D_{2x} denote a set of transformation index pairs, e.g., two different transformations of the same class like radix i radix j. The difference between two transformations is quantified by the normalized compression distance (NCD) [LCL+04].

ΔT_x – *Average distance between transformed messages:*

$$\Delta T_x \;=\; \sum_{m \in M_s} \sum_{(i,j) \in D_{2x}} \frac{\big|\, T_{xi}(m) \;-\; T_{xj}(m) \big|}{|M_s| \;\cdot\; |D_{2x}|}$$

A lower value indicates that messages transformed by different methods are relatively similar, making them potentially more guessable.

$\varnothing T_x$ – *Average difference in compressed length:*

$$\varnothing T_x \;=\; \sum_{m \in M_s} \sum_{i \in D_{1x}} \frac{\big|\, C(T_{xi}(m)) \;-\; C(m) \big|}{|M_s| \;\cdot\; |D_{1x}|}$$

This metric quantifies the extent to which each transformation inflates or shrinks the data when compressed.

δT_x – Uniqueness of transformations:

$$\delta T_x \;=\; \frac{|X|}{|D_{1x}| \cdot |M_s|}, \qquad X \;=\; \left\{ T_{xi}(m) \;\middle|\; i \in D_{1x},\, m \in M_s \right\}$$

This metric measures how often different transformations produce identical outputs. A lower value indicates that transformations can map different inputs to the same output, thereby reducing protocol diversity.

Randomizing Protocols

Protocols can be randomized by selecting appropriate transformations. Algorithm 1 illustrates a typical procedure for generating a customized protocol. For each message format $f \in F$ (line 2), we *select* (lines 3–4) one unconditional (line 5) and one conditional transformation (line 6) using a pseudorandom generator (*prg*) initialized with the client ID and a secret seed (line 4). If no semantics are specified, conditional transformations are omitted to avoid potential application failures. The transformed format is then added to the new format set F' (line 7). Unconditional transformations do not depend on semantics and can be randomly applied on top of previously applied ones. In contrast, conditional transformations require tracking the applied semantics (line 8), as they may alter message behavior, for example, by combining or delaying messages, which can modify or violate the original communication semantics.

Algorithm 1: Selecting transformations for each message format.

Input : $\mathcal{P} = \{F, S\}$ - base protocol
Input : *prg* - seeded with client ID and secret seed
Input : T_c - set of conditional transformations
Input : T_u - set of unconditional transformations
Result: $\mathcal{P}' = \{F', S'\}$ - custom protocol

```
1 F' ← {},   S' ← S;
2 for f ∈ F do
3     (T'_c, T'_u) ← getAllowedTransformations(T_c, T_u, f, S');
4     (t_c, t_u) ← prg.choice(T'_c, T'_u);
5     f' ← t_c(f) ; // apply conditional transformation
6     f' ← t_u(f') ; // apply unconditional transformation
7     F'.add(f');
8     S'.update(f, t_c) ; // track multi-msg transforms if needed
9 return {F', S'};
```

This results in a different protocol per client. The method also allows for partial or multi-round transformations, although these may introduce complexity if transformations conflict.

Deployment of Polymorphic Protocols

Once generated, a custom protocol can be integrated into the client and server via a standalone proxy or by tightly integrating the generated code into the application. Figure 5.2 illustrates

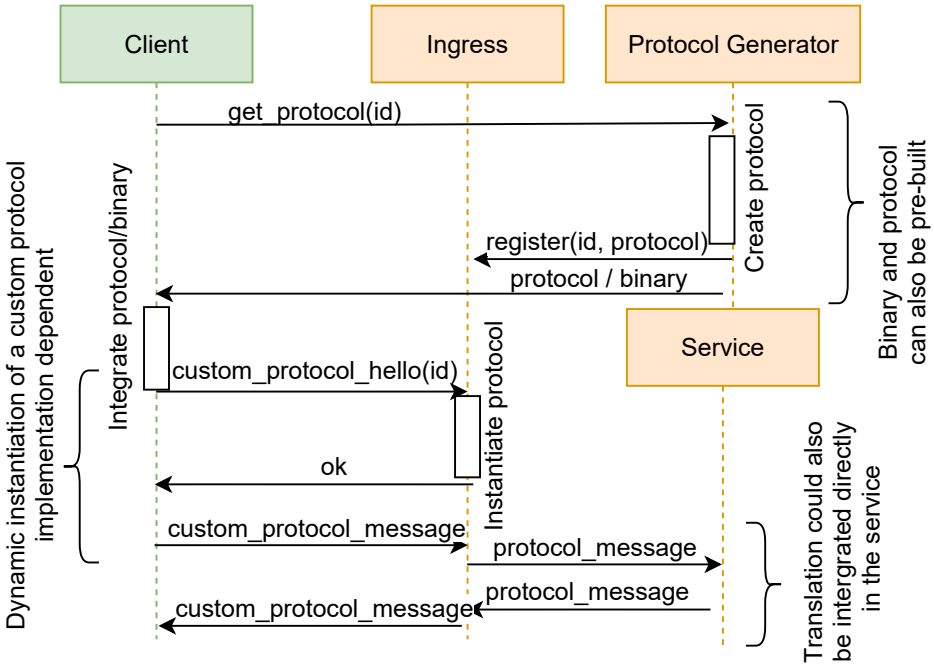


Figure 5.2: Example on how a custom protocol can be requested, delivered, and deployed between a client and an ingress server acting as a proxy.

an example of dynamically instantiating and using a polymorphic protocol. The client initiates the process by requesting a custom protocol using its client ID. The protocol generator creates the custom protocol, registers it with the ingress server, and returned to the client, either as a standalone binary, as direct machine code that is streamed and executed, or integrated into the main application. To communicate with the server, the client sends a custom protocol hello message. The ingress server locates and instantiates the protocol. Once the client is notified of the successful instantiation, it can start to send custom protocol messages to the ingress, which acts as a proxy and translates these messages into a format that the service understands. This procedure represents only one possible deployment approach; other methods are equally feasible. For example, the protocol can be generated a priori and integrated into the client during application compilation.

Deployment Strategies. In addition to the actual deployment implementation, there are different modes for using the polymorphic protocol.

Full-Polymorphic Each client runs a unique protocol. This provides maximum diversity, but comes with a higher overhead.

Time-Polymorphic All clients share the same protocol, which is replaced periodically (e.g., weekly or after suspicious traffic is detected).

All deployment strategies and modes introduce overhead, as the applied transformations incur additional computational cost and increase bandwidth usage. Furthermore, generating and streaming protocol instances is not free and contributes to the overall resource consumption.

Reducing Overheads. Polymorphic protocols inevitably introduce some performance and maintenance overhead. To mitigate these we recommend the following:

- *Use polymorphic protocols only for high-risk clients*, such as those running on emulators or rooted devices[4].
- *Use grouped deployments*, assigning protocol variants by region, operating system, or device integrity checks.
- *Coordinate with bot detection systems* so that legitimate users experience fewer forced updates.

5.1.2 Key Results

This section summarizes our key findings. First, we evaluate how our approach increases the attacker's cost of scaling bots (RQ3.1). We then evaluate the performance overhead of using our approach (RQ3.2).

Results RQ3.1 Attacker Cost and Security Discussion

Estimating the cost to a man-at-the-end (*MATE*) attacker (see Section 2.1.2) is inherently difficult [ASA+15, TIF19], and existing metrics for evaluating obfuscation [BCG+16, BCP17] are often too narrow for our broader goal: raising the cost of *duplicating bots* rather than just thwarting a single, one-time reverse engineering. We compare our approach to the typical practice of using custom API keys, focusing on two deployment models (time-polymorphic and full-polymorphic) and two types of attackers:

Restricted-MATE (R-MATE). This attacker has access to the binary (as usual in MATE scenarios [ASA+15]), but lack sophisticated means or tools to *automatically* extract embedded secrets, e.g., API keys. This attacker is considered the most common, as fully automated code/data extraction for encrypted protocols is *non-trivial* [WJC+09, LJLW13, NSC15].

Unrestricted-MATE (U-MATE). This attacker can employ a full range of advanced static and dynamic analysis, code slicing, or code reuse techniques without limitation. However, large-scale automation requires reliable extraction of *only* the minimal protocol transformation logic, which remains a complex task [TIF19, CLGJ19].

Comparison to Client-Only Obfuscation. Client-side obfuscation techniques, such as virtualization and control flow transformations [Col01, Jsc, Tec22, Sof21], aim to make reverse engineering more difficult by obfuscating the binary. However, these approaches ignore the underlying communication protocol, which remains exposed and consistent. As a result, attackers

[4] `https://developer.android.com/google/play/integrity/overview`

like R-MATe and U-MATE can bypass such protections by simply recording and replaying network interactions, without the need for reverse engineering. In contrast, our approach addresses this limitation by obfuscating the protocol itself. This prevents straightforward recording and replaying, thereby offering protection even when the client is not reverse engineered.

Comparison to API Keys. In practice, many services rely on a single API key embedded in clients, which is occasionally updated. An attacker who extracts this key can replicate bots at scale as long as the key remains valid. For the sake of a fair comparison, we assume that each client is assigned a unique API key. Polymorphic protocols change not only the secret but also the *structure* (e.g., permutations, dummy bytes) and can embed cryptographic keys. This applies to both time-polymorphic (periodic protocol updates) and full-polymorphic (per-client) variants:

- *R-MATE attackers* must actively re-extract *entire protocols* instead of simply swapping a key. This makes bot duplication more expensive, as each protocol variant introduces new transformations.
- *U-MATE attackers* face the additional challenge of automating protocol extraction (beyond just the key). Since polymorphic protocols may also contain standard encryption steps, extracting embedded keys is essentially a *subset* of inverting the entire protocol. Tools that slice or reuse code (e.g., hooking [BB10]) struggle when large transformations are scattered or heavily obfuscated.

Although U-MATE adversaries may ultimately succeed in extracting partial or complete protocols, each protocol update (time-polymorphic) or unique variant (full-polymorphic) forces repeated reverse engineering. In contrast, simply replacing an API key in a known protocol often remains a simple adaptation.

Scaling Bots. Once attackers have a functioning bot, they typically replicate it at scale. Polymorphic protocols require a new protocol variant for each additional bot instance. We recommend, and have implemented, an additional encryption layer around each message specification. This makes it infeasible for an attacker to guess the protocol, as the search space for the key is equivalent to that of a 256-bit key.

Code Reuse and Hardening. Attacks on polymorphic protocols may attempt to *slice* transformation logic from the binary and reuse it. However, reliably extracting *executable* protocol transformations without extraneous dependencies remains a non-trivial and error-prone process [TIF19, CLGJ19]. In addition, commercial obfuscation solutions [Sof21, Tec22] and publicly available tools [Col01] can further hinder automated code slicing, e.g., via function virtualization. Although these defenses are outside the core scope of our polymorphic design, they can be integrated to increase the cost and complexity for U-MATE attackers, effectively reducing them to the R-MATE scenario if the automated approach fails.

Overall, polymorphic protocols slow down adversaries more than API keys alone by coupling protocol structure changes with secret data. In scenarios where bots pose a significant business or security risk, these additional layers can be critical in preventing large-scale bot creation.

Table 5.1: Mean program build properties: Difference of polymorphic compared to the original protocol in percent (N=100).

	$G(P_{1,15})$	$G(P_{100,15})$	$G(P_{1,150})$	$G(P_{100,150})$
Build Time	+3%	+72%	+9%	+128%
Memory	-14%	+18%	+33%	+94%
Protocol Size	+138%	+144%	+202%	+295%

Results RQ3.2: Performance Overhead

To evaluate the performance overhead of polymorphic protocol generation, we applied our approach on top of Protocol Buffers (protobuf). The build process involves transforming message formats using various transformations, including radix conversion, field encryption, and insertion of dummy bytes, followed by the generation of wrapper classes and the compilation of the resulting protocol using the protobuf compiler. For transformation we included only one round.

Table 5.1 summarizes the overhead introduced during this process. Each entry $G(P_{x,y})$ denotes a protocol consisting of x messages, each containing y fields. The reported values indicate the relative difference compared to the original, unmodified protocol. The build time for the base protocol from 500ms to 600 ms, with memory usage ranging from 3 MB to 131 MB. The protocol binary size ranged from 0.57 MB for the base version to 42 MB for the largest variant.

The table reports relative differences compared to the original, unmodified protocol. For smaller protocols such as $G(P_{1,15})$, the overhead remains modest: build time increases by only 3%, memory usage decreases by 14%, and the protocol size grows by 138%. However, for larger configurations such as $G(P_{100,150})$, the overhead becomes more pronounced, with build time increasing by 128%, memory usage by 94%, and protocol size by 295%.

These results demonstrate that the overhead scales with the number of messages and fields in the protocol. While the increase in protocol size is significant, it is primarily due to the structural complexity introduced by transformations. Build time overhead is most relevant for large-scale deployments or environments with tight build-time constraints.

Runtime overhead is more difficult to quantify precisely, as it strongly depends on the specific transformations applied and the fields they target. For example, inserting dummy bytes or applying base64 encoding increases message size, whereas transformations like hashing or radix conversion introduce varying computational costs.

To provide a clearer view of runtime characteristics, Table 5.2 lists the average time required to apply each transformation class, measured in seconds per one million applications. These values represent isolated transformation costs and serve as a reference for understanding their relative impact during protocol execution.

In practical scenarios, polymorphic protocols approximately double the CPU time required for serialization and deserialization. This overhead arises because the applied transformations are reversed through an additional wrapper layer executed alongside standard protobuf decoding. These costs remain acceptable for non-real-time applications or systems with moderate traffic

Table 5.2: Runtime cost per transformation class

Transformation class	Time (s per 1 million applications)
Permutation of message bytes	413.23
Dummy bytes (4)	1.07
Hash	2.91
Radix (2–255)	160.85

volumes. Developers can further mitigate the overhead by excluding large or static fields, such as media payloads, from transformation, allowing for more fine-grained control over performance.

5.1.3 Discussion and Implications

Our approach increases the cost of large-scale bot creation by frequently changing the structure of the application layer protocol. Polymorphic protocols apply transformations, such as permutations and the insertion of dummy bytes, to generate client-specific protocol variants. As a result, each bot instance requires separate reverse engineering, significantly increasing the effort needed to analyze or replicate the protocol at scale. While these protocols do not *guarantee* permanent deterrence of attackers, they render the scaling attacks much more difficult for services that experience significant bot abuse.

Limitations As with other obfuscation schemes, it remains difficult to accurately quantify the additional overhead that polymorphic protocols impose on different adversaries [BCG+16, BCP17]. In addition, attackers can still use techniques such as code slicing to extract protocols. To mitigate this, polymorphic protocols can be complemented with anti-slicing techniques [CLGJ19], which further complicate the extraction of minimally functional code. Given that the operational overhead introduced by polymorphic protocols is manageable but present, this approach is most effective for services already burdened by bot attacks, where the increase in adversary cost outweighs the moderate performance or deployment overhead.

Higher-Level Implications Malware often communicates with external entities, for example, to exfiltrate data or interact with command and control servers. While this communication is often encrypted, it can exhibit recognizable patterns that intrusion detection systems can use to identify and block such activity. By randomizing protocol formats, polymorphic protocols reduce the predictability of patterns such as packet timing, message lengths, and static field signatures. This unpredictability can hinder IDS systems that rely on detecting consistent data structures and behaviors. Beyond security monitoring, polymorphic protocols provide a flexible and effective way to disrupt mass bot replication across application domains. However, they may force bot builders to turn to alternative techniques, such as UI automation, while increasing the complexity of code and deployment processes.

5.2 Encrypted Endpoints for Limiting Bot Scalability

Bots that interact with APIs scale easily because the API, and specifically its endpoints, remain consistent across clients. To counter this, we introduce an approach that assigns client-specific communication endpoints. These endpoints must first be extracted from the client, such as a web application or binary, before they can be used. Since each instance has unique endpoints, the extraction process must be repeated for every new bot, significantly increasing the effort required for replication. We refer to this approach as *encrypted endpoints*, a lightweight and practical variant of polymorphic protocols. Encrypted endpoints effectively impede bot scalability by breaking the uniformity of the communication interface, while remaining straightforward to implement and incurring minimal overhead. The overhead is limited to a slightly increased URL length and a single encryption or decryption step at runtime.

Our contributions are as follows:

- **Encrypted Endpoints.** We assign unique endpoints to each account to thwart bots that rely on static, reusable APIs. Even if attackers reverse engineer the client or capture network traffic, the extracted endpoints remain tied to a specific account, limiting their reuse across multiple bot instances. This feature can be combined with traditional obfuscation techniques [Col01, VYG13, DB18, XZZ12] and defenses against traffic capture attacks [See20a, Koc23] to further impede automated abuse.
- **Adaptability.** Our approach is not limited to services that require a login. It can also be used in applications that do not require authentication. For user convenience, URLs can be shared without compromising bot protection. In addition, service owners do not need to predefine all endpoints; they can be generated as needed.
- **Secondary Security Benefits.** By making URLs user-specific and unguessable, potential attacks such as URL guessing or directory traversal are effectively mitigated. This mechanism effectively acts as a per-user allowlist for valid URLs, restricting access to only those endpoints explicitly issued to a given client.
- **Implementation and Evaluation.** We implemented our method as middleware for the FastAPI framework and the Jinja2 template engine.[5] For the evaluation, we measured end-to-end latency and used a profiler to obtain reliable results, as the individual operations were too fast and subject to variability due to system jitter. Our measurements show that the middleware introduces less than 1% overhead, corresponding to an added latency of under 0.1 ms per request. The implementation is thus lightweight and can be easily integrated into existing projects.

The following research questions guide our investigation:

RQ3.3 To what extent do encrypted endpoints limit the scalability of bots, and how do they compare to alternative bot detection or mitigation strategies?

RQ3.4 What is the computational and operational overhead associated with using encrypted endpoints?

[5] `https://github.com/8mas/encrypted-endpoints`

RQ3.5 What types of secondary security threats, beyond bot replication, are mitigated by encrypted endpoints?

Building on these research questions, the following publication[6] summarizes our approach.

> August See, K. Röbert, M. Fischer. Encrypted Endpoints: Defending Online Services from Illegitimate Bot Automation. International Symposium on Research in Attacks, Intrusions and Defenses, 2024.

In the following sections, we discuss the construction of encrypted endpoints, strategies to mitigate potential drawbacks, summarize the key findings of our evaluation, and outline their limitations.

5.2.1 Method

This section outlines our approach to obfuscating network service endpoints and mitigating bot scalability. The goal is to prevent automated extraction of API endpoints and ensure that even if a bot is developed, it cannot be easily reused across multiple accounts or sessions.

Core Idea and Workflow

Each client (user) is provided with a uniquely customized Web site (or application resources) where all endpoints (URLs) are encrypted and signed with a client-specific key.

Formal Model (Brief). Given I_c as the client identifier, such as an IP address or user ID, shared between the client and the service, a client-specific key k_c is derived using a key derivation function (KDF) and the master key k_m, both of which are known only to the service:

$$k_c = \mathrm{KDF}(k_m, I_c).$$

Each URL $u = p \| a$ (path and parameters) is encrypted using an authenticated encryption scheme:

$$u' = \langle e, t_e \rangle = AE_{k_c}(u),$$

where e is the ciphertext and t_e is the message tag (MAC). The client cannot forge valid URLs because it does not have access to k_c or k_m. In the following, $\{message\}_{k_a}$ is shorthand for $\mathrm{AE}_{k_c}(message)$.

[6]The solution approach and full implementation and evaluation for this publication originated from the author of this dissertation. The second author is credited with the core idea. The second and third authors contributed to the refinement of the work.

Workflow. The basic function of encrypted endpoints is as follows: the middleware maintains a master key k_m and derives a client-specific key k_c based on a client identifier I_c. When the client accesses the site, all server-generated URLs are replaced with encrypted and signed versions using k_c. This binds unique endpoints to the client. These URLs are only valid for the client associated with I_c, thereby preventing large-scale bot replication.

When a user visits a page for the first time during browsing, the initial functionality is triggered as illustrated in Figure 5.3. The user requests the root page without any encrypted URL, as this is the initial visit. However, a client identifier is still transmitted in some contexts, such as the IP address or a session cookie from a previous interaction. The middleware derives the client-specific key and forwards the request to the backend, including the client key k_a of client A as shown in the figure. The backend responds with the requested webpage and encrypts all embedded endpoints or URLs using k_a.

The functionality of browsing is shown in Figure 5.4, when a client accesses an endpoint via its URL, the respective encrypted endpoint is used. The middleware verifies that the endpoint comes from the backend by checking its signature, since only the middleware and backend have access to the master key k_m and the client key k_c derived from it. This ensures that the URL was issued to the specific client, effectively preventing simple duplication of bots. Since bots must extract valid URLs for each individual client, large-scale replication is significantly hindered. Both binary programs and web applications can be further hardened against URL extraction by applying established obfuscation techniques [VYG13, DB18, BLRP10], which complicate parsing and reverse engineering efforts.

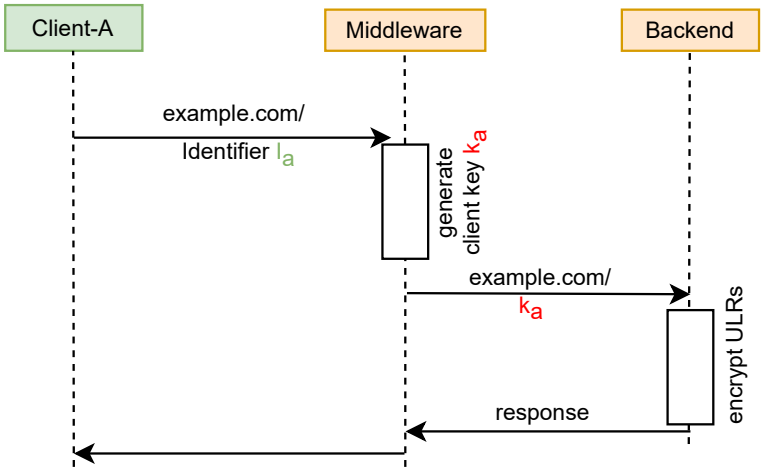


Figure 5.3: Encrypted Endpoint usage in the context of webpages and browsing.

Error Handling and Security. If a URL that was not generated for the current client is accessed, decryption fails and the middleware returns an error response. As shown in Figure 5.5, attempts to reuse another user's URLs are blocked by the MAC verification. Similarly, attackers scanning for sensitive files will only be blocked, as the URLs are invalid (see Figure 5.6), providing an effective URL allowlist.

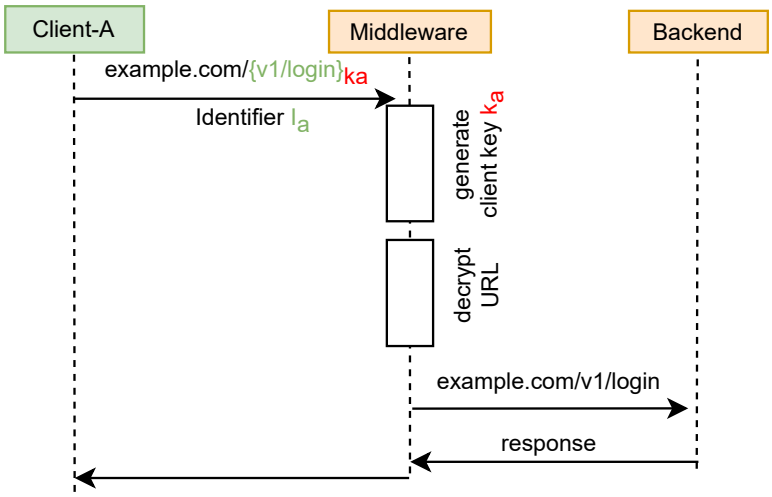


Figure 5.4: Encrypted Endpoint usage (backend only).

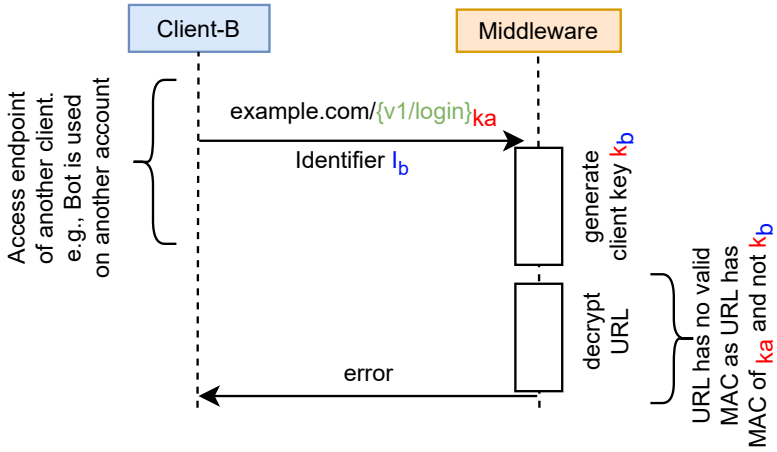


Figure 5.5: Attempt to use a URL generated for a different client: The MAC verification fails.

Client Identifier

A robust, tamper-proof identifier I_c is critical to ensuring the integrity of the system. Possible implementations include:

- *User/Session ID:* This approach is tied to user login and ensures that URLs are unique per user. It is particularly effective when session state can be persistently associated with the user, for example, through an authentication mechanism.
- *IP Address:* While simple, this method is susceptible to address changes, for example, when switching from cellular data to Wi-Fi, which can disrupt functionality. Additional rate-limiting mechanisms are recommended to mitigate potential abuse. Furthermore,

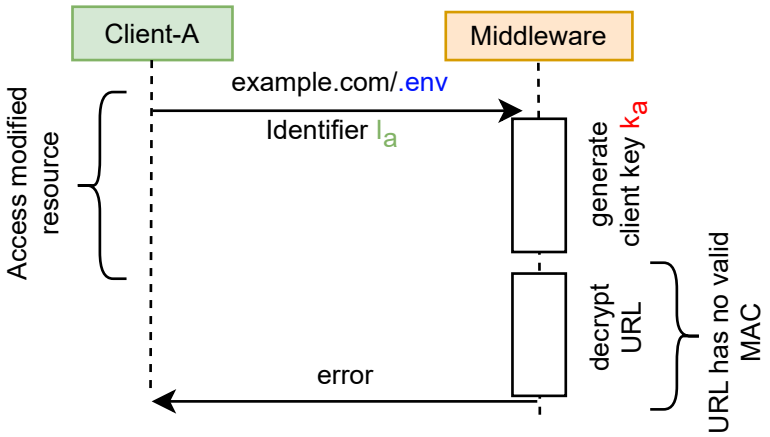


Figure 5.6: Directly accessing a non-issued URL also fails due to MAC mismatch.

the use of shared IP addresses, such as in NAT scenarios, may cause issues when multiple users appear under the same identifier and must be appropriately accounted for.

- *Browser Fingerprint:* This approach is effective when combined with client-side code execution, as it allows for detailed client identification.

For services without login functionality, a combination of IP address and browser fingerprinting is useful. Session resumption techniques can account for changes in I_c, such as those caused by network switches.

Practical Considerations

Encrypted endpoints alter fundamental aspects of how the Internet currently operates. Consequently, certain interleaving technologies are also affected by these endpoint modifications. In this section, we summarize some of the affected principles and discuss how their drawbacks can be mitigated.

URL Sharing. Encrypted URLs are inherently client-specific. In cases where read-only or public sharing is required, certain URLs or HTTP methods (such as safe GETs) can be left unencrypted or re-encrypted for a different client. This allows selective URL sharing while still protecting sensitive endpoints.

Session Resumption. Change-prone identifiers, such as IP-address identifiers, can invalidate existing URLs when networks are switched. A token-based session resumption mechanism can store the old key, allowing the middleware to decrypt and re-encrypt URLs using a new key k_c, thereby maintaining session continuity.

Partial Encryption of Endpoints. When certain parameters (such as search fields) are fully user-controlled, only known segments of the URL are encrypted. The remaining parameters are appended in clear text. The application or middleware must clearly define which segments should remain obfuscated to limit the enumeration of endpoints.

URL Validity and Refresh. To prevent long-term bot operation, URLs can expire by incorporating a time-based or version-based nonce into the key derivation process:

$$k_c = \mathrm{KDF}(k_m, I_c, \mathrm{nonce}_t).$$

This forces bots to periodically re-extract valid links, increasing their maintenance costs.

Performance Optimizations

While encrypting endpoints is expected to be fast, certain performance optimizations can further enhance efficiency.

Stateful vs. Stateless. A stateless approach redetermines k_c and re-encrypts URLs as needed, reducing memory requirements but increasing computational load. In contrast, a stateful system caches keys or pre-generated URLs, reducing latency at the expense of additional storage requirements.

Risk Assessment. The server can selectively apply encrypted endpoints to high-risk clients or suspected bots [WZC19, SCRM21], reducing overhead for normal traffic. This provides a balance between performance and the need to prevent automated endpoint extraction and large-scale bot creation.

5.2.2 Key Results

This section provides a summary of our findings with respect to the defined research questions. The primary goal is to evaluate the impact of encrypted endpoints on bot scalability (RQ3.3), assess the overhead they introduce (RQ3.4), and examine the additional protection they provide (RQ3.5). We implemented our approach as middleware in `FastAPI`, requiring minimal changes to both back-end and front-end code. The middleware provides functionality to encrypt and decrypt URLs, including partial encryption support to accommodate dynamic web pages and templating engines such as `Jinja2`. The implementation is available as open source[7].

Results RQ3.3: Limiting Bot Scalability

Our evaluation closely follows that defined in Section 5.1.2. We consider two attackers: *(i) Endpoints-Only*, where adversaries rely on intercepting and replaying traffic without analyzing client-side code to retrieve endpoints, for example by using tools such as [See20a, Koc23]; and *(ii) Endpoints and Data Parsing*, where adversaries extend their efforts to parsing HTML, JavaScript, or binaries to extract encrypted endpoints.

In the *Endpoints-Only* model, encrypted endpoints renders captured URLs ineffective for reuse across accounts, thereby neutralizing replay-based automation attacks. In contrast, traditional code obfuscation provides no resistance to such attacks.

[7] `https://github.com/8mas/encrypted-endpoints`

In the *Endpoints and Data Parsing* model, attackers can theoretically extract encrypted URLs by decompiling or analyzing the application. However, combining encrypted endpoints with established code obfuscation techniques [Kac, Sof21, Tec22], which alone cannot prevent these attacks, significantly increases the effort required do extract and exploit such URLs, making it difficult to sustain large-scale bot operations.

Compared to alternative strategies such as CAPTCHAs or risk-based detection, encrypted endpoints are *complementary*: they do not actively interact with or challenge legitimate users, but instead prevent bots from reusing or replaying links across multiple clients. As a result, this approach raises the barrier to scalable automated abuse, especially when used in conjunction with other security measures.

Results RQ3.4: Overhead of Encrypted Endpoints

In terms of performance overhead, our experiments show only a small increase in response time and CPU usage. All measurements were conducted on a Linux laptop equipped with an Intel i7-10700K processor.

Processing Overhead We evaluated the runtime performance of encrypted endpoints using a local test setup. The application served HTML pages with varying numbers of encrypted URLs. To ensure accuracy, we used profiling in addition to end-to-end measurements, which are subject to millisecond-level jitter.

For backend processing, we observed that handling encrypted URLs introduces only minimal latency. In the *ENC* configuration, where the client-specific key is derived per request, the added latency was +0.031 ms. Caching the key on the server (*S-Key*) reduced this to +0.020 ms, and caching the encrypted URLs (*S-URLs*) further lowered it to +0.002 ms. All configurations used AES-GCM with hardware acceleration via AES-NI [Gue10].

Table 5.3 presents the overhead observed when rendering encrypted URLs using the Jinja2 template engine. We tested responses with 10 and 100 unique URLs. As expected, the overhead increases with the number of encrypted URLs, but remains within acceptable limits. Caching client keys and URLs significantly reduces this overhead, especially in high-traffic or large-template scenarios.

Table 5.3: Latency overhead when rendering encrypted URLs in Jinja2, measured via profiling over 100,000 requests.

URLs	Base	ENC	S-Key	S-URLs
10	0.033ms	+0.158ms	+0.076ms	+0.023ms
100	0.087ms	+1.470ms	+0.662ms	+0.128ms

These performance measurements demonstrate that encrypted endpoints introduce only negligible overhead, even at scale, and can be efficiently integrated into existing web applications. However, one practical consideration when deploying encrypted URLs is the potential increase in their length. We examine this aspect in the following section.

URL Stretch Encrypted endpoints increase URL length due to encryption and encoding overhead. Although HTTP/1.1 does not specify a maximum URL length [FGM+99], browsers enforce practical limits, e.g., Chromium supports up to 2MB.[8] Our tests (April 2024) show that major desktop and mobile browsers handle URLs longer than 20,000 characters.

In our implementation, length expansion comes from base64 encoding (ca 33% overhead), padding and MAC (16 bytes each), and 2-byte separators for partial encryption. The total URL length is calculated as:

$$L_{\text{final}} = L_{\text{data}} \cdot 1.33 + n_{\text{blocks}} \cdot (16 + 16 + 2)$$

This overhead is negligible and below current browser limits.

Results RQ3.5: Protection Against Other Attacks

While our primary focus is on limiting bot scalability, we find that encrypted endpoints also serves as an implicit URL allow list. By requiring a valid cryptographic signature from the server for each link served, this mechanism can be used to prevent bot attacks:

- Prevent directory traversal attacks (e.g., `/../../etc/passwd`) and accidental file exposures, since such URLs are never served by the server.
- Reduces the risk of local file inclusion exploits and certain types of reflected XSS or SQL injection attacks, provided that these exploits rely on modifying predefined URL parameters.

However, fully user-controlled URL parameters remain outside the scope of this approach: if the server cannot anticipate or restrict a parameter, encryption alone will not provide sufficient protection. Furthermore, this mechanism only applies to URLs explicitly issued by the server. If attackers force the server to generate malicious URLs, those URLs will still be valid. Therefore, while encrypted endpoints do not provide a comprehensive security solution, they do limit the attack surface for specific URL-based exploits.

5.2.3 Discussion and Implications

Our primary goal is to prevent bot scalability across multiple user accounts by assigning unique, encrypted endpoints to each user. This forces attackers to continuously extract valid endpoints from server responses, making bot reuse and mass automation much more difficult. In this way, we aim to reduce the proliferation of simple scripts and automated tools, and further restrict advanced attackers when combined with proper code obfuscation.

[8] `https://chromium.googlesource.com/chromium/src/+/HEAD/docs/security/url_display_guidelines/url_display_guidelines.md`

Limitations Despite the increased difficulty for bots that rely on direct endpoint knowledge, resource-rich attackers can still resort to UI-based automation (Section 2.1.1), such as scripting interactions at the front-end layer. Although this process is slow and often requires significant computational overhead (e.g., via VMs or sequential bot launches), determined adversaries may still succeed. In addition, sites that require shareable URLs - such as online retailers, payment providers, or social media sites - must first identify which URLs can remain unencrypted so that legitimate external linking can continue to function. This identification process places an up-front burden on developers and system architects and can add complexity to deployment.

Higher-Level Implications Encrypted endpoints help to hide URL paths and parameters, limiting third-party visibility into resource details and thus oppose anti-tracking add-ons such as ClearURLs.[9] However, some level of tracking is inherently tied to these encrypted endpoints, as any parameter embedded for authentication or customization effectively becomes a persistent identifier. In some cases, this can be more invasive than tracking cookies, as the tracking information is stored directly in the URL itself. When such links are shared, the embedded parameters may inadvertently reveal sensitive details or user-specific tokens, exacerbating privacy concerns. This highlights the need for further research on the privacy implications of encrypted and user-specific URLs, particularly in contexts where links are exposed or shared across users or systems.

On a broader level, encrypted endpoints help make bot creation more resource-intensive and technically challenging. This raises the entry barrier for attackers, potentially reducing bot proliferation. However, it also encourages the development of more sophisticated attack methods, fueling an ongoing arms race between bot creators and defenders. In addition, widespread adoption of such methods may shift adversaries' focus to alternative attack vectors, such as UI automation.

5.3 Summary

This chapter examines techniques for bot prevention. Unlike detection, which by definition occurs after a bot has been deployed, prevention approaches aim to hinder bot creation in the first place.

We focus on limiting the scalability of API-based bots, whose key strength lies in their ability to scale very efficiently once developed. Our core idea is to require different clients or users of a service to communicate using their unique application protocols. As a result, for a bot to scale, it must extract valid configurations from legitimate clients. This impedes large-scale bot operations, restricting not only bot operators but also individuals attempting to use public bots, such as in online gaming or automated shopping.

We propose two approaches. The first, called polymorphic protocols, randomizes the full application protocol between the client and the service. This applies both at the representation level, e.g., how data is encoded or ordered, and at the semantic level, where message sequences can be randomized when permitted. This approach forces bot creators to extract valid protocols

[9] `https://github.com/ClearURLs/Addon`

from legitimate clients, a challenging task that can and should be combined with existing obfuscation techniques to complicate reverse engineering. Our approach is configurable to be lightweight in terms of CPU usage and message size overhead.

The second approach, called encrypted endpoints, can be seen as a more specific version of polymorphic protocols, where only the endpoint, often a URL, is obfuscated. This counters adversarial techniques such as traffic recording and replay attacks, as well as protocol reverse engineering. Like polymorphic protocols, encrypted endpoints require extraction from legitimate clients to scale bot operations. Application obfuscation should be employed to further complicate the extraction process for adversaries. This approach is lightweight, as it increases URL size by only 33% plus a constant overhead for authentication tags and padding.

A key challenge in obfuscation research is assessing the extent to which such techniques hinder adversaries. The difficulty arises from the fact that man-at-the-end (MATE) attackers have full control over the device running the application (client) and can employ arbitrary analysis techniques. We demonstrate that our approaches impose at least the same level of complexity as extracting API keys, which are a subset of our approach. That is, extracting protocols and endpoints is at least as difficult as extracting API keys and likely imposes greater implementation overhead on bot developers, as they must reimplement the complex protocol transformations.

Our approaches may shift the advantage in favor of defenders by limiting the greatest strength of bots, their scalability. While attackers may turn to UI bots to circumvent API-based restrictions, such methods are significantly more expensive, require direct interaction with the user interface, and are harder to automate efficiently. By forcing adversaries to rely on such less scalable and more labor-intensive techniques, our approach makes large-scale bot operations increasingly impractical.

Chapter 6

Conclusion

Web bots are becoming increasingly common, as they allow to automate tasks for various purposes. They can serve beneficial roles, such as web crawling for search engines, or be used for profit, by scraping data to train proprietary AI models. However, they can also be employed maliciously, e.g., by executing automated attacks or by pretending to be humans to influence public opinion on social media. As AI technology advances, creating sophisticated bots becomes easier. Autonomous AI agents are capable of making decisions, further complicating the task of distinguishing human users from bots. The most common defense against bots are CAPTCHA systems. However, these can be bypassed using AI techniques, while simultaneously causing frustration for legitimate users. Alternative approaches, such as requiring strong authentication (e.g., government-issued IDs), increase user friction, leading to reduced usability and raise privacy concerns. Therefore, there is a need for alternative methods that do not impose direct challenges on human users.

This cumulative thesis examines these challenges and proposes potential solutions. Guided by the research questions RQ1-3, three groups of contributions C1-3 are derived. Detection methods based on human behavior are introduced to differentiate humans from bots (C1). An analysis technique is presented to accelerate the reverse engineering of closed-source binaries (C2), which can aid in understanding bot behavior, which is a pre-requisite for novel detection and prevention strategies. Finally, bot prevention methods are proposed that specifically target the scalability of bots (C3).

In the following, we answer the research questions defined in Section 1.3, which guided the development of these contributions.

RQ1: What are the limitations of passive anomaly detection based on user behavior for identifying bots? To address this question, we developed and evaluated several machine learning models and detection strategies based on user behavior. These include classification models for mouse dynamics on web pages, various modeling strategies for keystroke dynamics, such as character-independent, character-dependent, and sentence-level models, as well as heuristics for individual request classification. Additionally, a machine learning model was designed to operate on graph representations of website traversal behavior for bot detection based on this behavioral data.

Our evaluation shows that behavior-based detection is generally effective, achieving an accuracy of over 0.95 across all approaches. In the case of mouse dynamics, bots can be reliably distinguished from human users with high accuracy after approximately 1.66 seconds of interaction. Our models outperform related work, under stricter conditions that account for more sophisticated mouse movements, such as those that do not follow straight-line patterns.

For keystroke dynamics, our models also demonstrate strong performance. We evaluate both character-identity-dependent and character-agnostic classifiers. While including key identity achieves near-perfect accuracy (close to 1), we achieve 0.98 accuracy in a more privacy-preserving variant that omits key identity information. Compared to related work, our models perform better even against more sophisticated synthetic keystroke sequences, which incorporate dependencies on preceding and following keys. We further show that both mouse and keyboard behavior models generalize across different applications, indicating that application-specific training is not strictly necessary for effective detection. Also the graph-based bot detection framework, which captures user website traversal behavior, performs well with over 0.98 accuracy. However, the high accuracy observed may partly result from the relative simplicity of the bots included in the evaluation.

These findings confirm that passive behavioral detection is currently effective. However, the results for mouse and keyboard input indicate that as the synthesis of bot behavior becomes more advanced, detection accuracy decreases.

RQ2: How can reverse engineering techniques be optimized to accelerate the analysis of closed-source bot binaries? To address this question, we developed a novel, data-driven reverse engineering approach. Instead of manually identifying relevant instructions, analysts specify the data they are interested in, and our approach performs two tasks: it locates the instructions that access this data and ranks them based on how exclusively they interact with it. This allows general-purpose instructions to be filtered out, significantly reducing the analysis scope. The analysis is not limited to single-instruction access but considers data interactions across multiple instructions.

Our method is implemented using Intel Pin and supported by plugins for Ghidra and IDA, enabling integration with standard reverse engineering workflows and allowing detailed inspection of binary behavior and resource usage. To validate our confidence-based filtering mechanism, we applied the approach to four P2P botnets. By analyzing a local bot instance, we attempted to automatically crawl the botnet and collect the IP addresses of all infected peers. This setup served as ground truth for evaluating the filtering accuracy. Our results show that instructions with a high confidence score consistently interacted with correct peers, demonstrating the utility of the ranking for identifying relevant code.

Overall, the proposed approach supports more efficient and targeted reverse engineering by narrowing down the analysis to the most relevant binary components, thereby reducing reverse engineering overhead.

RQ3: How can the scalability of bots be limited? To address this question, we developed novel obfuscation techniques aimed at limiting the scalability of API-based bots, which represent the most easily replicated class of bots. The core idea is to increase the overhead for bot creators by requiring them to reverse engineer multiple distinct client applications rather than a single one to scale their bots. This is achieved by assigning each client a unique, randomized application-layer protocol, thereby preventing bots from reusing the same automation logic across instances.

We explored two approaches. The first, termed polymorphic protocols, randomizes the entire application-layer protocol through structural transformations such as field reordering and insertion of dummy bytes. This approach offers strong obfuscation, though at a higher deployment

cost. The second, more lightweight method involves randomizing API endpoints through encrypted URLs. These encrypted endpoints maintain compatibility with user expectations and established conventions, for instance, enabling URL sharing in e-commerce scenarios, while still hindering bot scalability.

Both approaches were implemented and evaluated. Polymorphic protocols require bot developers to reverse engineer each randomized protocol variant, as even elements such as API keys are embedded within the transformations. This significantly increases the effort required to scale bots. Compared to traditional defenses, such as binary obfuscation or API key rotation, our approach targets the protocol itself, which is the primary interface used by scalable bots. However, both approaches benefit significantly when combined with application obfuscation techniques. These techniques make reverse engineering more difficult, thereby hindering application slicing and targeted protocol extraction. Encrypted endpoints incur minimal overhead, adding only a small increase in URL length and approximately one millisecond of computational cost for encryption, decryption, and validation. These endpoints are thus fitting for modern desktop and mobile browsers and can additionally serve as an allowlist mechanism, mitigating attacks that exploit non-server-generated URLs, such as directory brute-forcing or path traversal.

Overall, both methods raise the cost and complexity of large-scale bot creation. By forcing attackers to invest in repeated reverse engineering, they shift the balance in favor of defenders and provide scalable, lightweight protections that are harder to circumvent than conventional techniques.

Effective Bot Mitigation While the approaches were evaluated independently, they are complementary and can be combined for more robust bot mitigation. Effective protection requires both prevention and detection mechanisms that hinder the creation of individual bots and limit their scalability. For binary applications that face significant bot abuse, we recommend deploying polymorphic protocols in conjunction with standard obfuscation and application hardening techniques. For web applications or scenarios where performance overhead must be minimal, encrypted endpoints offer a lightweight and practical solution. In addition to prevention, a multi-stage detection pipeline is essential. Initial stages should rapidly filter out obvious bots to conserve computational resources. Ongoing risk assessment should leverage website traversal patterns, mouse dynamics, and keystroke behavior to detect more subtle or evasive bots. Finally, understanding adversaries is important. This includes collecting usage data from application logs and acquiring bot samples when possible. Such insights enable the development of more targeted and adaptive detection and prevention strategies.

Future Work

This doctoral thesis contributes novel solutions for bot detection, analysis, and prevention. However, further potential for future work remains in the following areas:

In-Depth, Privacy-Friendly Authentication Services must defend against bots to protect their human users. Users have a right to privacy, and services aim for frictionless usage. The solution is a privacy-preserving authentication approach, conceptually similar to Single Sign-On

(SSO), in which an identity provider certifies user attributes, e.g., that the user is human and over 18 years old. Unlike current SSO systems, the identity provider should not learn which services the user is accessing. To prevent bots, there should be a limit on the number of different authentications or accounts per human user on a given service that are non-linkable to the service. This could also be combined with physical tokens, e.g., national ID cards to bind the user to an identity.

UI-Based Bot Prevention and Detection With increasing resources and state-level actors leveraging the user interface (UI) of applications to create bots that mimic human behavior, traditional prevention approaches become less effective. These approaches, which focus on lower levels such as APIs, are insufficient against UI-level threats. Prevention strategies could involve minor randomization of the UI, which is imperceptible to human users but disrupts bots that rely on recorded clicks and keystrokes. Additionally, detection approaches could combine behavioral and request-based monitoring by analyzing UI event patterns to identify bot activity.

Web Tarpits and Bot Fingerprinting Instead of immediately blocking detected bots, an alternative approach is to redirect them into controlled environments similar to honeypots, so-called web tarpits. These environments serve two purposes. First, they help reduce bot activity in the wild by consuming bot resources such as bandwidth, CPU, and memory. Second, they enable the collection of detailed behavioral data to build bot fingerprints, which can inform future detection strategies. As a potential added benefit, scraping bots used for AI training could later be reidentified if the tarpit serves distinct, traceable content, such as intentionally embedded false facts.

Compiler-Level Network Obfuscation A common limitation of obfuscation approaches is that they operate at the source-to-source level. For example, *C++* code is transformed into obfuscated *C++* code. However, this introduces a dependency on the programming language, limiting the general applicability of the obfuscation. By leveraging compilers, for example the *LLVM* infrastructure, it would be possible to perform obfuscations directly on the intermediate representation (*LLVM-IR*). This could be implemented, for instance, as an *LLVM* optimization pass, enabling language-independent transformations at the compiler level. As a result, obfuscation would be applicable to many, or even all, *LLVM* target architectures, significantly broadening its reach. This line of work aligns with moving target defense, which relies on frequent reconfigurations to stay ahead of attackers. While some existing approaches move in this direction, they have not yet been applied to networked services in the context of the proposed polymorphic protocols and endpoints.

Identifying Large-Scale Bot Campaigns Detection approaches that only examine single user instances are losing effectiveness. To counter large-scale bot campaigns, solutions could focus on clustering similar behavior, such as mouse movements, website traversal, posts or likes, to identify bot networks. This approach could render large-scale campaigns less effective, as it would facilitate the identification and disruption of bot clusters.

Statement on the Use of AI-based Tools

The author used ChatGPT-4 as a language editing tool to check the text of this thesis. The tool was employed solely to correct typographical errors, improve grammar and refine phrasing. The prompts instructed the model to act as a computer science professor reviewing a PhD dissertation, with a focus on expert-level scientific American English and adherence to LaTeX formatting. The model was explicitly instructed not to add any information or content beyond the original text. Additionally, Grammarly and DeepL were used for spelling and grammar corrections, including rephrasing.

Bibliography

[AA20] Fatmah H Alqahtani and Fawaz A Alsulaiman. Is image-based captcha secure against attacks based on machine learning? an experimental study. *Computers & Security*, 88:101635, 2020.

[ADF19] Margit Antal and Lehel Denes-Fazakas. User verification based on mouse dynamics: a comparison of public data sets. In *2019 IEEE 13th International Symposium on Applied Computational Intelligence and Informatics*, pages 143–148. IEEE, 2019.

[AEZ19] Margit Antal and Elöd Egyed-Zsigmond. Intrusion detection using mouse dynamics. *IET Biometrics*, 8(5):285–294, 2019.

[AFA19] Ismail Akrout, Amal Feriani, and Mohamed Akrout. Hacking google recaptcha v3 using reinforcement learning. *arXiv preprint arXiv:1903.01003*, 2019.

[AMFVR22] Alejandro Acien, Aythami Morales, Julian Fierrez, and Ruben Vera-Rodriguez. Becaptcha-mouse: Synthetic mouse trajectories and improved bot detection. *Pattern Recognition*, 127:108643, 2022.

[AMM+21] Alejandro Acien, Aythami Morales, John V Monaco, Ruben Vera-Rodriguez, and Julian Fierrez. Typenet: Deep learning keystroke biometrics. *IEEE Transactions on Biometrics, Behavior, and Identity Science*, 4(1):57–70, 2021.

[ASA+15] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48:44–57, 2015.

[ATSL10] André Altmann, Laura Toloşi, Oliver Sander, and Thomas Lengauer. Permutation importance: a corrected feature importance measure. *Bioinformatics*, 26(10):1340–1347, 2010.

[BB10] Jan Berdajs and Z Bosnić. Extending applications using an advanced approach to dll injection and api hooking. *Software: Practice and Experience*, 40(7):567–584, 2010.

[BCG+16] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200, 2016.

[BCP17] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *26th USENIX Security 17*, pages 661–678, 2017.

[BDYGP15] Thomas Barabosch, Adrian Dombeck, Khaled Yakdan, and Elmar Gerhards-Padilla. Botwatcher: Transparent and generic botnet tracking. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*, pages 565–587. Springer, 2015.

[BF16] Alessandro Bessi and Emilio Ferrara. Social bots distort the 2016 us presidential election online discussion. *First monday*, 21(11-7), 2016.

[Bin24] BinBashBanana. BinBashBanana/html-obfuscator, June 2024. Accessed: 2024-07-01. URL: `https://github.com/BinBashBanana/html-obfuscator`.

[BLRP10] Douglas Brewer, Kang Li, Laksmish Ramaswamy, and Calton Pu. A link obfuscation service to detect webbots. In *2010 IEEE International Conference on Services Computing*, pages 433–440. IEEE, 2010.

[Bro21] Steven Brock. Scalping in ecommerce: ethics and impacts, 2021. Accessed: 2024-11-10. URL: `https://ssrn.com/abstract=3793357`.

[C+18] A. Cabri et al. Online web bot detection using a sequential classification approach. In *IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2018.

[CLGJ19] Xiaoyang Cheng, Yan Lin, Debin Gao, and Chunfu Jia. Dynopvm: Vm-based software obfuscation with dynamic opcode mapping. In *International Conference on Applied Cryptography and Network Security*, pages 155–174. Springer, 2019.

[Col01] Christian Collberg. the tigress c obfuscator, 2001. Accessed: 2024-12-07. URL: `https://tigress.wtf/about.html`.

[ÇU17] Hayreddin Çeker and Shambhu Upadhyaya. Sensitivity analysis in keystroke dynamics using convolutional neural networks. In *2017 IEEE workshop on information forensics and security (WIFS)*, pages 1–6. IEEE, 2017.

[DB18] Ahmed Diab and Tawfiq Barhoum. Prevent xpath and css based scrapers by using markup randomizer. *Int. Arab. J. e Technol.*, 5(2):78–87, 2018.

[DCS15] Kevin P Dyer, Scott E Coull, and Thomas Shrimpton. Marionette: A programmable network traffic obfuscation system. In *24th USENIX Security 15*, pages 367–382, 2015.

[DFKO18] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. Observations on typing from 136 million keystrokes. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pages 1–12, 2018.

[DGS+18] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proceedings on Privacy Enhancing Technologies*, 2018.

[DMT+22] Daniel DeAlcala, Aythami Morales, Ruben Tolosana, Alejandro Acien, Julian Fiérrez, Santiago Hernandez, Miguel A Ferrer, and Moisés Díaz. Statistical keystroke synthesis for improved bot detection. *CoRR*, 2022.

[DN21] Prafulla Dhariwal and Alex Nichol. Diffusion Models Beat GANs on Image Synthesis. In *Advances in Neural Information Processing Systems*, volume 11, 2021.

[DS11] D. Doran and S. Swapna. Web robot detection techniques: overview and limitations. *Data Mining and Knowledge Discovery*, 22:183–210, 2011.

[DSV+21] Sunny Dhamnani, Ritwik Sinha, Vishwa Vinay, Lilly Kumari, and Margarita Savova. Botcha: detecting malicious non-human traffic in the wild. *arXiv preprint arXiv:2103.01428*, 2021.

[FGM+99] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999.

[GF04] Hugo Gamboa and Ana Fred. A behavioral biometric system based on human-computer interaction. In *Biometric Technology for Human Identification*, volume 5404, pages 381–392. SPIE, 2004.

[Gua10] Claudio Guarnieri. Cuckoo cuckoo sandbox, 2010. Accessed: 2020-04-11. URL: `cuckoosandbox.org`.

[Gue10] Shay Gueron. Intel advanced encryption standard (aes) new instructions set. *Intel Corporation*, 128, 2010.

[Hea10] Nick Heath. Expedia on how one extra data field can cost $12m, 2010. Accessed: 2021-10-18. URL: `https://www.zdnet.com/article/expedia-on-how-one-extra-data-field-can-cost-12m/`.

[HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems*, volume 2020-December, 2020.

[HTR+20] Md Imran Hossen, Yazhou Tu, Md Fazle Rabby, Md Nazmul Islam, Hui Cao, and Xiali Hei. An object detection based solver for google's image recaptcha v2. In *RAID 2020*, pages 269–284, 2020.

[IKT+19] Christos Iliou, Theodoros Kostoulas, Theodora Tsikrika, Vasilis Katos, Stefanos Vrochidis, and Yiannis Kompatsiaris. Towards a framework for detecting advanced web bots. In *Proceedings of the 14th international conference on availability, reliability and security*, pages 1–10, 2019.

[IKT+21] Christos Iliou, Theodoros Kostoulas, Theodora Tsikrika, Vasilis Katos, Stefanos Vrochidis, and Ioannis Kompatsiaris. Detection of advanced web bots by combining web logs with mouse behavioural biometrics. *Digital threats: research and practice*, 2(3):1–26, 2021.

[Imp24] Imperva. Bad Bot Report | Evasive Bots Drive Online Fraud | Imperva, 2024. Accessed: 2024-11-18. URL: `https://www.imperva.com/resources/resource-library/reports/bad-bot-report/`.

[Inca] Apple Inc. Replace CAPTCHAs with Private Access Tokens - WWDC22 - Videos. Accessed: 2022-11-21. URL: `https://developer.apple.com/videos/play/wwdc2022/10077/`.

[Incb] ProtWare Inc. Encrypt HTML source, Javascript, ASP. Protect links & images. HTML encryption. Accessed: 2024-07-01. URL: `https://www.protware.com/`.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *32nd International Conference on Machine Learning, ICML 2015*, volume 1, 2015.

[JKV19] H. Jonker, B. Krumnow, and G. Vlot. Fingerprint surface-based detection of web bot detectors. In *European Symposium on Research in Computer Security*, pages 586–605. Springer, 2019.

[Jsc] Jscrambler. Webpage Integrity: Manage Third-party Risks. Accessed: 2024-07-01. URL: `https://jscrambler.com/webpage-integrity`.

[Kac] Timofey Kachalov. javascript-obfuscator/javascript-obfuscator: A powerful obfuscator for JavaScript and Node.js. Accessed: 2024-07-01. URL: `https://github.com/javascript-obfuscator/javascript-obfuscator`.

[KJK22] Mohinder Kumar, MK Jindal, and Munish Kumar. A systematic survey on captcha recognition: types, creation and breaking techniques. *Archives of Computational Methods in Engineering*, 29(2):1107–1136, 2022.

[Koc23] Albert Koczy. Mitmproxy2swagger, January 2023. Accessed: 2023-01-13. URL: `https://github.com/alufers/mitmproxy2swagger`.

[LARN21] X. Li, B. A. Azad, A. Rahmati, and N. Nikiforakis. Good bot, bad bot: Characterizing automated browsing activity. In *IEEE Symposium on Security and Privacy (SP)*, pages 1589–1605. IEEE, 2021.

[LCL+04] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul MB Vitányi. The similarity metric. *IEEE transactions on Information Theory*, 50(12):3250–3264, 2004.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[Liu18] Wei Liu. Introducing recaptcha v3: the new way to stop bots, 2018. Accessed: 2021-05-20. URL: https://developers.google.com/search/blog/2018/10/introducing-recaptcha-v3-new-way-to.

[LJLW13] Min Liu, Chunfu Jia, Lu Liu, and Zhi Wang. Extracting sent message formats from executables using backward slicing. In *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, pages 377–384. IEEE, 2013.

[LWK+16] Eunjo Lee, Jiyoung Woo, Hyoungshick Kim, Aziz Mohaisen, and Huy Kang Kim. You are a game bot!: Uncovering game bots in mmorpgs via self-similarity in the wild. In *Ndss*, pages 1–15, 2016.

[Mac18] Intuition Machines. Stop more bots. start protecting user privacy., 2018. Accessed: 2021-05-20. URL: https://www.hcaptcha.com/.

[Meu21] Thibault Meunier. Humanity wastes about 500 years per day on CAPTCHAs. It's time to end this madness, May 2021. Accessed: 2022-11-21. URL: http://blog.cloudflare.com/introducing-cryptographic-attestation-of-personhood/.

[MMLDG12] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 97–108, 2012.

[Mob] Genesis Mobile. JavaScript Obfuscator - Protect your JavaScript Code. Accessed: 2024-07-01. URL: https://jasob.com/.

[NSC15] John Narayan, Sandeep K Shukla, and T Charles Clancy. A survey of automatic protocol reverse engineering tools. *CSUR*, 48(3):1–26, 2015.

[Ode16] Augustus Odena. Semi-supervised learning with generative adversarial networks. *arXiv preprint arXiv:1606.01583*, 2016.

[OMAAK20] Mariam Orabi, Djedjiga Mouheb, Zaher Al Aghbari, and Ibrahim Kamel. Detection of bots in social media: a systematic review. *Information Processing & Management*, 57(4):102250, 2020.

[Par19] Debra Parma. At ticketmaster, scalpers score and fans come last. *JL & Com.*, 38:463, 2019.

[Phi] Phil Wang. Denoising Diffusion Probabilistic Model, in Pytorch. Accessed: 2025-03-25. URL: https://github.com/lucidrains/denoising-diffusion-pytorch.

[R+10] S. Rose et al. *Automatic keyword extraction from individual documents*, volume 1. Text mining: applications and theory, 2010.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[S+21] G. Suchacka et al. Efficient on-the-fly web bot detection. *Knowledge-Based Systems*, 223:107074, 2021.

[SCG12] C. Shen, Z. Cai, and X. Guan. Continuous authentication for mouse dynamics: A pattern-growth approach. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, 2012. doi:10.1109/DSN.2012.6263955.

[SCRM21] Grażyna Suchacka, Alberto Cabri, Stefano Rovetta, and Francesco Masulli. Efficient on-the-fly web bot detection. *Knowledge-Based Systems*, 223:107074, 2021.

[See20a] August See. Charles-Extractor, July 2020. Accessed: 2023-01-13. URL: https://github.com/8mas/Charles-Extractor.

[See20b] August See. Jodelapi, 2020. Accessed: 2021-05-20. URL: https://github.com/see-aestas/JodelApi.

[SGZ+16] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *Advances in neural information processing systems*, 29, 2016.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15, 2014.

[Sof21] VMProtect Software. VMProtect Software Protection, 2021. Accessed: 2022-12-07. URL: https://vmpsoft.com/.

[SPK16] Suphannee Sivakorn, Iasonas Polakis, and Angelos D Keromytis. I am robot:(deep) learning to break semantic image captchas. In *2016 IEEE EuroS&P*, pages 388–403. IEEE, 2016.

[T+20] T. Tanaka et al. Bot detection model using user agent and user behavior for web log analysis. *Procedia Computer Science*, 176:1621–1625, 2020.

[Tec22] Oreans Technologies. Oreans Technologies : Software Security Defined., 2022. Accessed 2021-12-07. URL: https://www.oreans.com/Themida.php.

[TIF19] Mahin Talukder, Syed Islam, and Paolo Falcarin. Analysis of obfuscated code with program slicing. In *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pages 1–7. IEEE, 2019.

[VYG13] Shardul Vikram, Chao Yang, and Guofei Gu. Nomad: Towards non-intrusive moving-target defense against web bots. In *CNS*, pages 55–63. IEEE, 2013.

[Wal45] A. Wald. Sequential tests of statistical hypotheses. *Ann. Math. Statist.*, 16(2):117–186, 1945.

[WJC+09] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *European Symposium on Research in Computer Security*, pages 200–215. Springer, 2009.

[WMK+22] Tara Whalen, Thibault Meunier, Mrudula Kodali, Alex Davidson, Marwan Fayed, Armando Faz-Hernández, Watson Ladd, Deepak Maram, Nick Sullivan, Benedikt Christoph Wolters, et al. Let the right one in: Attestation as a usable {CAPTCHA} alternative. In *Eighteenth Symposium on Usable Privacy and Security (SOUPS 2022)*, pages 599–612, 2022.

[WZC19] Ang Wei, Yuxuan Zhao, and Zhongmin Cai. A deep learning approach to web bot detection using mouse behavioral biometrics. In *Biometric Recognition: 14th Chinese Conference, CCBR 2019, Zhuzhou, China, October 12–13, 2019, Proceedings 14*, pages 388–395. Springer, 2019.

[WZX+16] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. Webranz: web page randomization for better advertisement delivery and web-bot prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 205–216, 2016.

[XZZ12] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 9–16. IEEE, 2012.

[Yun24] Jason Yung. json2d/obscure, June 2024. Accessed: 2024-07-01. URL: `https://github.com/json2d/obscure`.

[Z+15] Y. Zhang et al. Detecting malicious activities with user-agent-based profiles. *International Journal of Network Management*, 25(5):306–319, 2015.

[ZCNC18] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, 2018.

Appendices

Copyright Notice

This appendix presents publications as originally published and reprinted with permission from the corresponding publishers. The copyright of the original publications is held by the respective copyright holders; see the following copyright notices.

- ©2023 Springer. Reprinted, with permission, from Richard August See, T. Wingarz, M Radloff, and M.Fischer, Detecting Web Bots via Mouse Dynamics and Communication Metadata, IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC), 2023.
- ©2024 Springer. Reprinted, with permission, from Richard August See, A. Westphal, C. Weber, and M.Fischer, Detecting Web Bots via Keystroke Dynamics, IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC), 2024.
- ©2024 Springer. Reprinted, with permission, from Jan Kadel, Richard August See, R. Sinha and M. Fischer, BOTracle: A Framework for Discriminating Bots and Humans, European Symposium on Research in Computer Security, SecAI Workshop, 2024.
- ©2023 ACM. Reprinted, with permission, from Richard August See, M. Gehring, M. Fischer and S.Karuppayah, Binary Sight-Seeing: Accelerating Reverse Engineering via Point-of-Interest-Beacons, Annual Computer Security Applications Conference (ACSAC), 2023.
- ©2022 Springer. Reprinted, with permission, from Richard August See, L. Fritz and M. Fischer, Polymorphic Protocols at the Example of Mitigating Web Bots, European Symposium on Research in Computer Security (ESORICS), 2022.
- ©2024 ACM. Reprinted, with permission, from Richard August See, L. Fritz and M. Fischer, Encrypted Endpoints: Defending Online Services from Illegitimate Bot Automation, International Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2024.

Appendix A

Detecting Web Bots via Mouse Dynamics and Communication Metadata

Abstract

The illegitimate automated usage of Internet services by web robots (bots) is an ongoing problem. While bots increase the cost of operations for service providers and can affect user satisfaction, e.g., in social media and games, the main problem is that some services should only be usable by humans, but their automated usage cannot be prevented easily. Currently, services are protected against bots using visual CAPTCHA systems, the de facto standard. However, they are often annoying for users to solve. Typically, CATPCHAs are combined with heuristics and machine-learning approaches to reduce the number of times a human needs to solve them. These approaches use request data like IP and cookies but also biometric data like mouse movements. Such detection systems are primarily closed source, do not provide any performance evaluation, or have unrealistic assumptions, e.g., that sophisticated bots only move the mouse in straight lines. Therefore we conducted an expeiment to evaluate the usefulness of detection techniques based on mouse dynamics, request metadata, and a combination of both. Our findings indicate that biometric data in the form of mouse dynamics performs better than request data for bot detection. Further, training a mouse dynamic classifier benefits from external and not only website-specific mouse dynamics. Our classifier, which differentiates between artificial and human mouse movements, achieves similar results to related work under stricter and more realistic conditions.

Reference

August See, T. Wingarz, M. Radloff, M. Fischer. Detecting Web Bots via Mouse Dynamics and Communication Metadata. IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC), 2023.

Contribution

The core concept, methodological design, and evaluation strategy of this paper originate from the author of this dissertation. The third author implemented and conducted the experiments, while the second and fourth authors contributed to the refinement of the final publication.

Detecting Web Bots via Mouse Dynamics and Communication Metadata

August See, Tatjana Wingarz, Matz Radloff, and Mathias Fischer

Universität Hamburg, Hamburg, Germany
{richard.august.see,tatjana.wingarz,mathias.fischer}@uni-hamburg.de
matzradloff@gmail.com

Abstract. The illegitimate automated usage of Internet services by web robots (bots) is an ongoing problem. While bots increase the cost of operations for service providers and can affect user satisfaction, e.g., in social media and games, the main problem is that some services should only be usable by humans, but their automated usage cannot be prevented easily. Currently, services are protected against bots using visual CAPTCHA systems, the de facto standard. However, they are often annoying for users to solve. Typically, CATPCHAs are combined with heuristics and machine-learning approaches to reduce the number of times a human needs to solve them. These approaches use request data like IP and cookies but also biometric data like mouse movements. Such detection systems are primarily closed source, do not provide any performance evaluation, or have unrealistic assumptions, e.g., that sophisticated bots only move the mouse in straight lines. Therefore we conducted an experiment to evaluate the usefulness of detection techniques based on mouse dynamics, request metadata, and a combination of both. Our findings indicate that biometric data in the form of mouse dynamics performs better than request data for bot detection. Further, training a mouse dynamic classifier benefits from external and not only website-specific mouse dynamics. Our classifier, which differentiates between artificial and human mouse movements, achieves similar results to related work under stricter and more realistic conditions.

Keywords: web bots, mouse dynamics, captchas

1 Introduction

Programs that can automatically request endpoints increase operating costs and can frustrate users. For example, web bots made popular items such as graphics cards and new game consoles unavailable for years because they were purchasing them automatically. Commercial countermeasures, such as IDS solutions, prove ineffective against web bots because these bots operate within the constraints of the targeted e-commerce website's existing APIs or user interface. For instance, web bots may mimic human behavior by navigating to a product and clicking on the "buy now" button. The problem is that the endpoints are used as intended.

A defense that scans, e.g., for malicious payload in requests is pointless here. One solution is CAPTCHAs, which give users tasks that are difficult for a computer but easy for a human to solve. However, this introduces user friction that can cost a company customers and thus revenue [9]. Modern CAPTCHAs are used together with risk assessment methods. Depending on the risk score, a certain hard CAPTCHA or no CAPTCHA at all is presented [15]. While we see this as a step in the right direction, the problem remains that such solutions are not privacy friendly as request data and biometric features are passed on to third parties. Commercial CAPTCHA providers, understandably, do not disclose which features they use for detection and which are most beneficial to identify bots.

Existing approaches are subject to various limitations, such as being closed-source, considering only request data or mouse dynamics, or having shallow assumptions in their evaluation. For instance, some approaches assume that advanced web bots only produce mouse movements in straight lines rather than curved human-like movements. Moreover, these approaches fail to address other real-world problems, such as whether website-specific mouse data or any mouse data can be used for bot detection. We address this in our paper.

Our main contribution is evaluating the usefulness of mouse dynamics for detecting bots in realistic settings. In more detail:

- We evaluate the performance of classifiers for bot detection based on mouse dynamics and compare them to the performance of using request data. We do this on a consistent dataset that contains mouse- and request data belonging to the same user. Our evaluation includes advanced bots that mimic human mouse movements utilizing third-party software.
- We show that bot detection based on mouse dynamics can benefit from not being solely trained on website-specific mouse movements, indicating that website operators do not need to train on the mouse movements of their users exclusively but can leverage third-party datasets.
- We investigate the relationship between the number of data points and the performance of bot detection, thus allowing us to determine the amount of data required for good performance and, consequently, the speed at which a classification can take place.

The rest of this paper is structured as follows: Section 2 discusses web bot detection using request data and mouse dynamics. Section 3 explains how and which features we used to train classifiers from related work. Section 4 describes our evaluation, how we created our dataset, and the limitations of our work. Finally, section 5 concludes the paper.

2 Related Work

We divide the related work into approaches that detect bots based on request data and ones detect bots based on mouse data. Note that there are also approaches that recognize bots based on other biometric data [5, 6], or approaches that are based on trusted platforms [8].

Modern CAPTCHA systems like hCaptcha and reCAPTCHA [15, 16] are already using biometric data like mouse movements in addition to request data. However, they do not disclose if the detection of bots is website specific and how well which factor performs. This is most likely due to protecting business secrets and denying bot creators information about where improvements need to be made. Google itself doesn't even specify what data they use exactly for reCAPTCHA. However, there is related work that tries to break this down [19].

2.1 Bot Detection via Request Data

A simple way for bot detection is to block IPs that are known for spamming or cyber attacks, for example, using *abuseipdb*[1]. Furthermore, there are many approaches for the detection of bots based on request data [11, 12, 14–16, 20].

Iliou et al. [11] present a comparison of different machine learning algorithms and combinations of various attributes used in previous literature. Their approach does not rely on cross-website tracking or using external resources like IP databases. This makes their approach simple to reimplement and validate. Their methods were tested on a year's worth of HTTP log data from MK-Lab's public web server[2]. The data included IP addresses, the request method, the request path, referrers, user agent strings, and timestamps. The attributes mainly include request metadata that would be suitable for a privacy-friendly bot detection system, for example, the percentage of image requests or the number of total bytes per session. The authors split the bot data in their dataset into simple and advanced bots, which are determined by whether the requests have a browser agent name and, in case they do, whether the IPs have shown malicious activity before. Their results show that different sets of attributes perform best depending on the classification algorithm used. The best machine learning methods are Random Forest and Multilayer Perceptron, although the paper concludes that using an ensemble classifier that averages over all used methods would be more stable. Additionally, simple web bots can be detected very easily, while detecting advanced bots is significantly harder, with areas under the ROC curve of 1.00 and 0.64, respectively. Especially in false positive intolerant use cases, the performance of detecting advanced bots is too poor to be used in the real world. The authors conclude that future work would need to incorporate more advanced features that bots cannot easily simulate.

2.2 Bot Detection via Mouse Dynamics

There is a lot of related work in the area of authentication using biometric data using mouse dynamics [13, 17, 18].

The work by Shen et al. [18] shows that it is possible to use mouse and trackpad actions to verify the authenticity of users. For this purpose, they group

[1] https://www.abuseipdb.com/

[2] Multimedia Knowledge and Social Media Analytics Laboratory, `https://mklab.iti.gr/`

mouse events such as single-click, double-click, or drag-and-drop. This data is combined with information about the current application type, e.g., web browsing or gaming, the screen area where the mouse movement occurred, the window position, and the timestamp. The data is transformed into a feature vector containing data such as the time it took a user to click a button, the speed of movement, or the acceleration. Finally, three different one-class classifiers (Nearest Neighbor, Single-Layer Neural Network, Support Vector Machine) are compared, with the Support Vector Machine method performing best with false positive and false negative rates of 0.37% and 1.12%, respectively. This was achieved only when 3000 operations and 30 minutes of processing time for successful authentication are considered. With a more feasible authentication time of one minute, the values for FPR and FNR increase to 44.65% and 34.78%, respectively. This drastically limits the applicability of this approach, which the authors also note.

Acien et al. [1] show the feasibility of using biometric features for bot detection. They use both function-based and GAN-based mouse trajectory synthesis methods to generate training and evaluation data. Six different classifier types (Support Vector Machine, K-Nearest Neighbor, Random Forest, Multi-Layer Perceptron, and 2 Recurrent Neural Networks with Long Short-Term Memory and Gated Recurrent Units, respectively) are compared to each other, with Random Forest performing the best. Combined, their method can distinguish between humans and bots with up to 98.7% accuracy with only one mouse trajectory as input. They conclude that, compared to state-of-the-art works, the usage of mouse data has unexploited potential in the context of bot detection. While there are other approaches for bot detection based on mouse dynamics [5, 21], they are all quite similar to each other. There are even approaches and projects that try to synthesize human mouse movements [1, 2].

2.3 Bot Detection via Request Data and Mouse Dynamics

Iliou et al. [10] present a method of using request data together with mouse data for bot detection, building on their previous work on bot detection using requests [11]. When classifying mouse dynamics, they do not build on existing work but create a new model that performs the classification using a convolutional neural network (CNN) on the raw mouse positions. As they do not have a labeled bot dataset, they create bots themselves. While the general idea of their approach has merit, they perform their evaluation with advanced bots that only move the mouse in straight lines. Figure 1 shows an example of the advanced bot behavior used for evaluation in their paper.

The mouse movements created by such a bot are not realistic enough to challenge human behavior and are very easy to classify as a bot by, e.g., looking at the straightness/curvature or angular velocity of the purported mouse movements. While our approach is similar to the one presented by Iliou et al., we utilize more advanced bots for

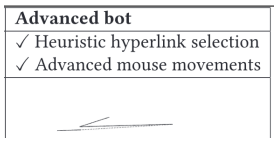


Fig. 1. Excerpt of the advanced mouse movements of [10] (Page 18, Table 7).

our evaluation that do not only move in straight lines and thus mimic human-like behavior more closely. We describe our advanced bot setup in Section 3.

In summary, there is already relevant research on bot detection using biometric features, including mouse dynamics, but most of it is closed-source. Most publicly available approaches tackle the problem of bot detection independent of the website being defended, i.e., the bot detection systems are trained on a dataset that does not originate from the website being defended. Additionally, the majority of approaches consider only request data or mouse data for detection, while the ones combining both techniques only evaluate their approaches with easily detectable bots.

3 Bot Detection using Requests and Mouse Dynamics

Our core idea is to improve the detection of web bots by using mouse dynamics in addition to request data. While request (meta)data like user-agent or screen size can easily be faked, mouse movements are continuous and contain many features, making them significantly harder to replicate. Thus an attacker would need valid, human-like mouse movements for each action. An example of several mouse movements is given in Figure 2, which shows three different recordings of mouse movements, two from using a chat app (one activity produced by a human, the other by the advanced bot used in our evaluation) and one from using a rhythm game[3] where the mouse is used heavily. This image depicts that the mouse dynamics are different per application and that advanced bots exist which do not exclusively move in straight lines.

To be consistent with prior work, we use machine learning models and request features proposed by [11] for bot detection on request data. For bot detection on mouse dynamics, we use machine learning models and mouse features proposed by [1]. Both papers are described in Section 2.

3.1 Request Data

Iliou et al. [11] ranked the best-performing metrics for simple and advanced bots per classification algorithm. However, some attributes used in their analysis are not suitable for this paper. For example, the authors include a Boolean indicating whether a request has a known search engine in their "Referer" header. Because we asked participants to visit the websites directly, this attribute is omitted. We use the following selection of metrics from Iliou et al.'s work [11] for our analysis:

1. The percentage of HTTP requests that led to an HTTP 4xx code response.
2. The percentage of HTTP requests that requested a CSS file.
3. The percentage of HTTP requests that requested a JavaScript file.

[3] `https://osu.ppy.sh/home`
[5] `https://github.com/Xetera/ghost-cursor`

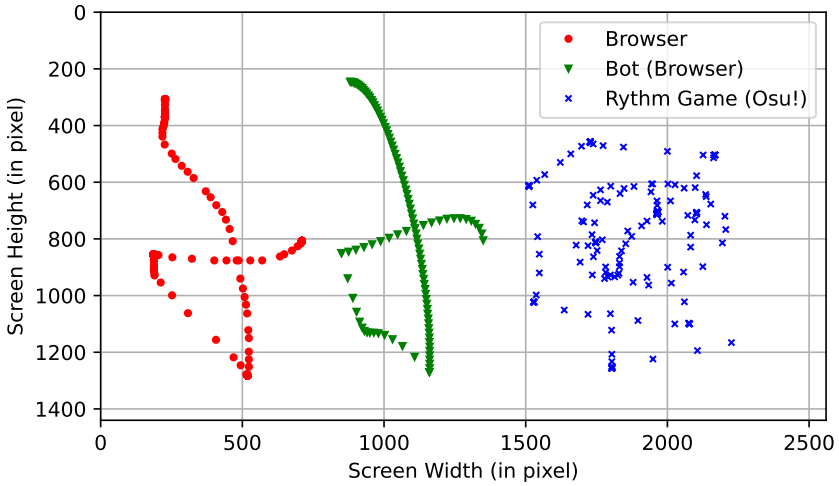


Fig. 2. Mouse dynamics example. Browser activity in a chat app on the left. In the middle movements of ghost-cursor[5]creating a path between four given points. On the right a human playing a rythm game (Osu!).

4. The percentage of HTTP-requested URLs that contain the previously requested URL as a subpart.
5. The total time between the first and the last HTTP request of the session.
6. Standard deviation of requested pages' depth (number of "/" in URL path).
7. Mean and Standard Deviation of times between successive requests.

3.2 Mouse Features

Mouse dynamic features consist of the relative x- and y-coordinates as well as a time value for each mouse event (e.g., left-click). Single mouse data points are grouped based on the following rules: They either end with a click, have a maximum of 50 data points, or span a maximum of two seconds. The features are calculated for each group. As a basis for all derived values, the time, x and y coordinates are linearly interpolated such that vectors with uniformly spaced values x'_t and y'_t every $20ms$ are generated. All indices start at zero.

In the following, we describe the additional used features engineered similarly to Gamboa et al. [7] and [4]. We include the path length from the origin, angle of the path tangent, horizontal, vertical, and overall velocity, acceleration, jerk, and angular velocity. Additionally, the type of action, length of the movement, and time needed to complete the action will be used.

The path length from the origin s'_t, i.e., the accumulated sum of previous segment lengths:

$$s'_t = \sum_{k=0}^{t-1} \sqrt{(x'_{k+1} - x'_k)^2 + (y'_{k+1} - y'_k)^2}$$

The angle of the path tangent with the x-axis θ_t is the arctangent ($atan2$ is used, which returns only values $-\pi < \theta < \pi$) of the segment at time $t > 0$. At $t = 0$ an angle of 0 is assumed.

$$\theta_t = atan2((y'_{t+1} - y'_t), (x'_{t+1} - x'_t))$$

The temporal features horizontal (v_x), vertical (v_y), tangential (v) and angular velocity (ω) as well as tangential acceleration ($\dot{v}$) and jerk ($\ddot{v}$) are computed as follows:

$$v_x = \frac{\delta x}{\delta t}; \quad v_y = \frac{\delta y}{\delta t}; \quad v = \sqrt{v_x^2 + v_y^2};$$
$$\omega = \frac{\delta \theta}{\delta t}; \quad \dot{v} = \frac{\delta v}{\delta t}; \quad \ddot{v} = \frac{\delta \dot{v}}{\delta t}$$

For each of the 9 vectors $(x'_t, y'_t, s'_t, v_x, v_y, v, \omega, \dot{v}, \ddot{v})$ the mean, standard deviation, minimum, maximum and value range (max-min) is calculated and yields the first 45 feature values.

Additionally, the time t_{total} and length s_{n-1} of the stroke (i.e. group of n data points), its straightness and jitter are computed. The time is the difference between the first and last data points' timestamps and the length can is the accumulated sum of segment lengths but using the raw instead of the interpolated data.

$$t_{total} = t_{n-1} - t_0$$
$$s_{n-1} = \sum_{k=0}^{n-1} \sqrt{(x'_{k+1} - x'_k)^2 + (y'_{k+1} - y'_k)^2}$$

Analogous to Gamboa et.al.'s definition [7], the *straightness* is defined as the ratio of the Euclidian distance between the first and last points of each group, and the total distance:

$$straightness = \frac{\sqrt{(x_0 - x_{n-1})^2 + (y_0 - y_{n-1})^2}}{s_{n-1}}$$

The *jitter* is the ratio between the original and smoothed path lengths:

$$jitter = \frac{s'_{n'-1}}{s_{n-1}}$$

In total, these 50 values make up the input vector that is computed for each mouse action group. We base our detection on whether a mouse movement is human on the features described above.

4 Evaluation

In this section, we summarize the evaluation results of our approach. We employed the best classifiers that were identified in related work. We utilized a random forest classifier [11] to analyze the request data, and for the mouse data,

we also employed a random forest classifier [1]. One of the advantages of using a random forest classifier for the mouse data is its explainability, which helps in understanding how the model arrived at its predictions. We used the dataset described in Section 4.1 for our training and evaluation. The features used for the classifiers are described in Section 3.1 and Section 3.2. Further, we answer the following research questions:

RQ1: What is the performance of the detection depending on the available data, i.e., the number of requests and mouse movements?

RQ2: How does the performance of the machine learning model change when trained additionally with mouse dynamics from an external dataset, i.e., unrelated to mouse dynamics on our websites?

4.1 Data Collection and Augmentation

To train and evaluate bot detection approaches, we need a dataset of request data together with related mouse dynamics, i.e., the combination of requests and matching mouse movements of a user. However, such a dataset does not exist to our knowledge [10]. While some datasets on mouse dynamics exist [3], they are obtained by users that repeatedly perform specific mouse-intensive tasks. Since such a type of mouse dynamics differs from the mouse dynamics of users visiting a website, it could affect a classifier's performance. We use this dataset in a second step to explore whether this is true.

Since no suitable dataset combining both features is publicly available [10], we need to build our dataset. For this, we invited users to visit and browse our two websites that log each request and every mouse movement. We announced our experiment with a link to our websites via a mailing list and had 322 participants visiting the first and 163 participants visiting the second website mentioned in the mail. We excluded mobile users, as well as users with no recorded mouse movements. Figure 3 shows the distribution of all users on both websites and their generated data points.

To integrate bot data we had to write our own. For this we use puppeteer. The behavior looks like this:

1. Accepting the initial prompt dialogue to start the experiment
2. Visiting the top-level pages {About, Blog, Contact/Imprint, Login, Register}
3. Visiting 10 randomly selected single blog pages
4. Visiting 100 randomly selected pages
5. Registering an account

All actions are configured to wait for the target element to be visible and clickable, scrolling it into view, if not. A random delay between 0 and 2 seconds is applied before each action. The behavior should reflect a scraper that does not scrape at full speed and in a specific order, e.g., width search.

Further, we distinguish between a basic mouse bot and an advanced mouse bot. The basic mouse bot moves the mouse on a direct path and at a constant speed to the target (links, blog posts, ...). The advanced mouse bot does not

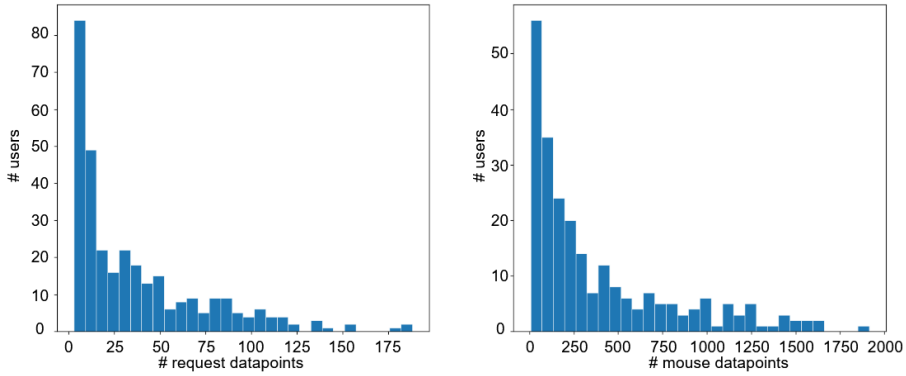


Fig. 3. Distribution of users in terms of data point count

do this but uses bezier curves implemented in the popular javascript library ghost-cursor[6], which promises human-like mouse movements. An example of such movements created by ghost-cursor is depicted in Figure 2. We sample as many bot users as human users in the experiment.

4.2 Results

This section presents our results. Limitations are described in Section 4.3.

RQ1 - Bot Detection Performance The performance of our model for the detection of bots depends on the data available. With more data available, the request data model's evaluation metrics show increasingly good performance. Table 1 shows the detailed results. Note that there are many cases where fewer data points than the limit are available (cf. Figure 3). The mouse data model generally performs better the more data is available. Table 2 lists the performance for different amounts of data points per user. A big advantage of this approach is that more data points can potentially be acquired in a shorter amount of time compared to request data. For example, when sampling at 30 events per second, as this work's implementation does, it only takes on average 1.66s (50 samples) to capture the number of data points needed to surpass the request data model's performance. When, for example, considering a potential application for a CAPTCHA, this time does not represent a significant disruption of most user interactions, e.g., filling in a registration form.

Parameter Tuning We used combinations of the following parameters to determine the best random forest parameters for bot detection empirically. Note that the number of bots and users in the dataset is the same, i.e., the same number of sessions. For mouse movements, we use the advanced bot that mimics human mouse movements.

[6] `https://github.com/Xetera/ghost-cursor`

Table 1. Request data model performance with varying amounts of data points per user

Data Points / User	Acc	Precision	Recall	AUC	Time
200	0.980	0.980	0.980	0.982	0.209
100	0.970	0.961	0.980	0.985	0.209
No limit	0.960	0.960	0.960	0.972	0.215
50	0.950	0.941	0.960	0.976	0.208
20	0.910	0.918	0.900	0.975	0.221
5	0.900	0.885	0.920	0.945	0.220
10	0.890	0.842	0.960	0.956	0.220
4	0.880	0.913	0.840	0.922	0.211

Table 2. Mouse data model performance with varying amounts of data points per user (Advanced Mouse Bot)

Data Points / User	Acc	Precision	Recall	AUC	Time
No limit	0.966	0.964	0.968	0.993	0.449
50	0.933	0.943	0.922	0.979	0.124
200	0.930	0.951	0.906	0.979	0.189
100	0.920	0.942	0.895	0.975	0.149
20	0.855	0.846	0.868	0.949	0.114
10	0.850	0.862	0.833	0.903	0.120
5	0.846	0.909	0.769	0.916	0.099
4	0.826	0.895	0.739	0.879	0.098

1. Number of estimators (10, 50, 100, 150, 200, 1000)
2. Maximum number of features (None, log2, sqrt)
3. Maximum tree depth (None, 1, 2, 3, 4, 6, 7)

Tables 3 and 4 show the 10 best-performing combinations for mouse and request data. The data is sorted by accuracy. The additional scores Precision, Recall, AUC, and training time are computed as well. Their values of the top-performing results lie close together except for training time. The different values for the number of estimators and the maximum number of features for split consideration perform very similarly. For request data, the values for the following experiments were chosen to be 100 and *None*, respectively, as their result had the same accuracy and AUC as the top result. Analogously for the mouse data result, 200 and *sqrt* were chosen. All results have in common that no restriction to the decision trees' maximum depth is applied, which is also the default value of *scikit-learn's* implementation. This is expected as the tree depth is directly correlated with the ability to classify multi-dimensional input data.

Basic vs. Advanced Mouse Bot The first direct comparison used all available human mouse data and the generated basic and advanced mouse data for train-

Table 3. Model accuracy for different parameters (request data)

Features	Estimators	Acc	Prec.	Recall	AUC	Time
None	1000	0.910	0.887	0.940	0.973	4.348
log2	200	0.910	0.887	0.940	0.973	0.732
log2	1000	0.910	0.887	0.940	0.975	3.965
None	100	0.910	0.887	0.940	0.973	0.486
sqrt	1000	0.910	0.887	0.940	0.972	3.912
log2	100	0.910	0.887	0.940	0.971	0.389
log2	150	0.910	0.887	0.940	0.972	0.694
None	50	0.900	0.870	0.940	0.971	0.273
sqrt	200	0.900	0.870	0.940	0.972	0.727
None	150	0.900	0.870	0.940	0.972	0.703

Table 4. Model accuracy for different parameters (mouse data)

Features	Estimators	Acc	Prec.	Recall	AUC	Time
sqrt	150	0.967	0.964	0.969	0.994	11.305
log2	1000	0.966	0.962	0.970	0.993	70.796
sqrt	200	0.966	0.962	0.970	0.994	5.899
sqrt	50	0.966	0.964	0.968	0.993	9.280
sqrt	100	0.966	0.962	0.969	0.994	9.815
sqrt	1000	0.966	0.960	0.971	0.994	82.889
log2	200	0.964	0.963	0.965	0.993	20.381
log2	150	0.962	0.963	0.960	0.993	5.950
None	150	0.961	0.954	0.969	0.991	91.586
None	200	0.961	0.953	0.969	0.991	110.588

ing and testing datasets. Table 5 shows that the model performs better in every aspect and can classify inputs more reliably when using data generated only by the basic mouse bot with linear movements. This is expected as the bots' linear movements are uniquely identifying properties that result in very specific outcomes for many input features. The model only correctly differentiated between humans and bots 96.6% of the time but still has a very high AUC.

Combining Mouse and Request Data for Advanced Bot detection Since we have a dataset containing request data and a user's matching mouse dynamics, we can explore whether a bot detection system that combines both mouse and request data may improve the performance compared to using them individually. Therefore, we apply the classifiers mentioned above to the mouse and request data individually. Afterward, we combine the calculated predictions of the two classifiers and average them to determine the final result. We split the data

Table 5. Simple and Advanced Mouse Data Performance

Scenario	Acc	Prec	Recall	AUC	Time
Basic mouse	0.995	0.997	0.994	1.000	0.857
Advanced mouse	0.966	0.962	0.970	0.994	1.293

into training and test sets on the user level, i.e., each user instance (human or bot) is only part of either the training or test set. We use two test ratios for this experiment, namely 0.1 and 0.2. Table 6 shows the overall performance, the precision of 1.0 was omitted from the table for readability. The most valuable difference is that there are no false positive results, while the false negative rates of 0.272 and 0.327 are higher in contrast to using the classifiers separately. However, the lower false positive rate and same overall performance favor using the combined approach as it is a priority not to disrupt the user experience [9].

Table 6. Combined Mouse and Request Data Performance

Test ratio	Acc	Recall	AUC	TP	FP	TN	FN
0.1	0.960	0.953	0.976	122	0	22	6
0.2	0.950	0.940	0.970	252	0	49	16

RQ2 - Using Unrelated Mouse Dynamics for Training We used Antal et al.'s dataset [3] to compare the performance to different real-world data. Inputs from 21 users were collected during their normal computer activities on one desktop and 20 laptop devices. Both mice and touchpads were used. Their raw mouse movement and interaction data are preprocessed similarly to the experiment's data. The whole dataset yielded $1.54M$ input vectors. The initial assumption was that these mouse movements do not match the interaction with our website and the performance decreases when using these mouse movements. However, when trained additionally with the external dataset, the accuracy increases to 99.71%, and the values for FPR and FNR decrease to 0.55% and 0.15%, respectively. This indicates that the origin of the mouse movements is not important with regards to their effectiveness.

4.3 Limitations

The strongest limitation of our approach is that we only have access to a synthetic bot dataset, similar to approaches like [10]. Further, we create bots using

fitting third-party projects. However, an external, labeled dataset with bot- and real-human traffic would be more suitable. The lack of such a realistic dataset leads to the performance of our model likely being worse outside our lab setting. Bots may act more camouflaged, e.g., by using recorded mouse movements. Those bots would probably escape detection. However, this is universal. Bots that behave completely like humans cannot be distinguished from humans. At the same time, making a bot behave like a human increases the costs for an attacker because the bot cannot work at full performance, e.g., scrape all data available or monitor a website for a long duration.

Further, while we were able to show that mouse data can easily be used for the detection of bots, another limitation is the focus on mouse data. This excludes users who interact without a mouse, e.g., only via keyboard, mobile devices, or screen readers.

5 Conclusion

We demonstrated the value of incorporating both mouse dynamics and request data when detecting bots. Unlike previous research that relied solely on one of the two data types or made unrealistic assumptions about the capability of advanced bots, we used a consistent dataset that included both mouse movements and request data belonging to the same user. Furthermore, we utilized a third-party library to create bots that performed human-like mouse movements. We used classifiers that performed best in literature for these tasks. We achieved better results with similar performance but in a more realistic setting. Thus mouse dynamics remain a useful tool for identifying even advanced bots. An interesting finding was that mouse data from third-party sources can be used to train the classifiers while achieving similar performance, thus simplifying the usage process. By leveraging third-party mouse data, operators can minimize the need to save and train on potentially sensitive user data. In the future, we intend to test our approach on a larger e-commerce dataset. Further, we want to consider more combinations of alternative approaches to bot detection, e.g., by including typing behavior as well as touch events from smartphones.

References

1. Acien, A., Morales, A., Fierrez, J., Vera-Rodriguez, R.: BeCAPTCHA-Mouse: Synthetic Mouse Trajectories and Improved Bot Detection. arXiv:2005.00890 [cs] (Mar 2021), http://arxiv.org/abs/2005.00890
2. Akrout, I., Feriani, A., Akrout, M.: Hacking google recaptcha v3 using reinforcement learning. arXiv preprint arXiv:1903.01003 (2019)
3. Antal, M., Denes-Fazakas, L.: User verification based on mouse dynamics: a comparison of public data sets. In: 2019 IEEE 13th International Symposium on Applied Computational Intelligence and Informatics. pp. 143–148. IEEE (2019)
4. Antal, M., Egyed-Zsigmond, E.: Intrusion detection using mouse dynamics. IET Biometrics **8**(5), 285–294 (2019)

5. Chu, Z., Gianvecchio, S., Wang, H.: Bot or human? a behavior-based online bot detection system. In: From Database to Cyber Security, pp. 432–449. Springer (2018)
6. Dee, T., Richardson, I., Tyagi, A.: Continuous transparent mobile device touchscreen soft keyboard biometric authentication. In: 2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID). pp. 539–540. IEEE (2019)
7. Gamboa, H., Fred, A.: A behavioral biometric system based on human-computer interaction. Proc SPIE **5404**, 381–392 (08 2004). https://doi.org/10.1117/12.542625
8. Gummadi, R., Balakrishnan, H., Maniatis, P., Ratnasamy, S.: Not-a-bot: Improving service availability in the face of botnet attacks. In: NSDI. pp. 307–320 (2009)
9. Heath, N.: Expedia on how one extra data field can cost $12m. `https://www.zdnet.com/article/expedia-on-how-one-extra-data-field-can-cost-12m/` (2010), accessed: 2021-10-18
10. Iliou, C., Kostoulas, T., Tsikrika, T., Katos, V., Vrochidis, S., Kompatsiaris, I.: Detection of advanced web bots by combining web logs with mouse behavioural biometrics. Digital threats: research and practice **2**(3), 1–26 (2021)
11. Iliou, C., Kostoulas, T., Tsikrika, T., Katos, V., Vrochidis, S., Kompatsiaris, Y.: Towards a framework for detecting advanced web bots. In: Proceedings of the 14th International Conference on Availability, Reliability and Security. ARES '19, Association for Computing Machinery, New York, NY, USA (2019)
12. Jonker, H., Krumnow, B., Vlot, G.: Fingerprint surface-based detection of web bot detectors. In: European Symposium on Research in Computer Security. pp. 586–605. Springer (2019)
13. Jorgensen, Z., Yu, T.: On mouse dynamics as a behavioral biometric for authentication. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. pp. 476–482 (2011)
14. Li, X., Azad, B.A., Rahmati, A., Nikiforakis, N.: Good bot, bad bot: Characterizing automated browsing activity. In: 2021 IEEE symposium on security and privacy (sp). pp. 1589–1605. IEEE (2021)
15. Liu, W.: Introducing recaptcha v3: the new way to stop bots. `https://developers.google.com/search/blog/2018/10/introducing-recaptcha-v3-new-way-to` (2018), accessed: 2021-05-20
16. Machines, I.: Stop more bots. start protecting user privacy. `https://www.hcaptcha.com/` (2018), accessed: 2021-05-20
17. Sayed, B., Traoré, I., Woungang, I., Obaidat, M.S.: Biometric authentication using mouse gesture dynamics. IEEE systems journal **7**(2), 262–274 (2013)
18. Shen, C., Cai, Z., Guan, X.: Continuous authentication for mouse dynamics: A pattern-growth approach. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012). pp. 1–12 (2012). https://doi.org/10.1109/DSN.2012.6263955
19. Sivakorn, S., Polakis, J., Keromytis, A.D.: I'm not a human: Breaking the google recaptcha. Black Hat **14** (2016)
20. Suchacka, G., Cabri, A., Rovetta, S., Masulli, F.: Efficient on-the-fly web bot detection. Knowledge-Based Systems **223**, 107074 (2021)
21. Wei, A., Zhao, Y., Cai, Z.: A deep learning approach to web bot detection using mouse behavioral biometrics. In: Chinese Conference on Biometric Recognition. pp. 388–395. Springer (2019)

Appendix B

Detecting Web Bots via Keystroke Dynamics

Abstract

The increasing presence of malicious web bots within the digital ecosystem has not only escalated operational costs for web services but also significantly deteriorated user satisfaction, especially in sectors like online gaming and social media. These bots engage in a range of unauthorized activities, contributing to a complex threat landscape. The situation is further complicated by rapid advancements in AI, which not only increase the demand for data, often sourced through unauthorized web scraping by bots, but also blur the lines between human and computer interactions. Enhanced image recognition capabilities allow AI to effectively bypass CAPTCHA systems, and the deployment of large language models provides sophisticated reasoning abilities. In response, this paper introduces an approach to mitigate these challenges by leveraging keystroke dynamics as a means for bot detection. Classifiers that differentiate between humans and bots based on keystroke patterns necessitate the use of synthetically generated keystroke data for training purposes. Such data is typically generated using generative AI models; however, it often lacks contextual accuracy, leading to easily distinguishable keystrokes and consequently affecting the classifiers' efficacy. To address this issue, our method models keystroke dynamics in a key-dependent fashion, employing a state-of-the-art diffusion model to produce data of superior quality. We further illustrate that classifiers, particularly those based on Transformer and LSTM architectures, exhibit enhanced performance when retrained with this refined data. We furthermore assess the feasibility of bot detection by exclusively analyzing keystroke timings without including key identity, which we find feasible with only a marginal performance decrease.

Reference

August See, A, Westphal, C. Weber, M. Fischer. Detecting Web Bots via Keystroke Dynamics. IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC), 2024.

Contribution

The core idea, methodological design, and evaluation strategy for this publication originate from the author of this dissertation. The second author implemented the models and experiments, while the third and fourth coauthors contributed to refining the final paper.

Detecting Web Bots via Keystroke Dynamics

August See, Adrian Westphal, Cornelius Weber, and Mathias Fischer

Universität Hamburg, Germany
{richard.august.see,adrian.westphal
cornelius.weber,mathias.fischer}@uni-hamburg.de

Abstract. The increasing presence of malicious web bots within the digital ecosystem has not only escalated operational costs for web services but also significantly deteriorated user satisfaction, especially in sectors like online gaming and social media. These bots engage in a range of unauthorized activities, contributing to a complex threat landscape. The situation is further complicated by rapid advancements in AI, which not only increase the demand for data, often sourced through unauthorized web scraping by bots, but also blur the lines between human and computer interactions. Enhanced image recognition capabilities allow AI to effectively bypass CAPTCHA systems, and the deployment of large language models provides sophisticated reasoning abilities. In response, this paper introduces an approach to mitigate these challenges by leveraging keystroke dynamics as a means for bot detection. Classifiers that differentiate between humans and bots based on keystroke patterns necessitate the use of synthetically generated keystroke data for training purposes. Such data is typically generated using generative AI models; however, it often lacks contextual accuracy, leading to easily distinguishable keystrokes and consequently affecting the classifiers' efficacy. To address this issue, our method models keystroke dynamics in a key-dependent fashion, employing a state-of-the-art diffusion model to produce data of superior quality. We further illustrate that classifiers, particularly those based on Transformer and LSTM architectures, exhibit enhanced performance when retrained with this refined data. We furthermore assess the feasibility of bot detection by exclusively analyzing keystroke timings without including key identity, which we find feasible with only a marginal performance decrease.

Keywords: web bots, keystroke dynamics, captchas

1 Introduction

Web bots present significant challenges in digital environments, impacting user experience and escalating operational costs. Their widespread presence in social media, online gaming, and dating platforms necessitates effective strategies to differentiate between human and bot interactions.

The emergence of sophisticated AI has fueled an increase in bot deployment across digital platforms. This is largely due to the high demand for training data

for AI algorithms, often sourced through unauthorized scraping. This practice not only incurs ethical and legal issues but also highlights an urgent need for robust bot detection mechanisms.

One approach to addressing this issue is the implementation of CAPTCHAs, which present tasks that are generally straightforward for humans but challenging for computers to complete. However, the effectiveness of this class of problems is diminishing due to advancements in AI technologies [7]. Furthermore, CAPTCHAs inadvertently create barriers to user engagement, potentially leading to customer loss and, consequently, a reduction in revenue [8].

A more recent approach to addressing this issue is the analysis of keystroke dynamics. Research has shown that the unique typing patterns of individuals can serve as reliable identifiers [2], providing a potential means to differentiate human users from bots. For that, machine learning models have been at the forefront. Examples include TypeNet, which employs a Recurrent Neural Network (RNN) for classification [2], and methods that use the Support Vector Machine (SVM) for classification and the Kernel Density Estimation (KDE) algorithm for feature synthesis [4]. However, a common limitation of these approaches is the generation of synthetic data without considering specific keycodes [4], which may compromise the realism and effectiveness of the models.

The main contribution of this paper is the exploration of innovative generative methods aimed at producing more realistic synthetic data to enhance the accuracy of bot classifiers. We further make the following contributions:

- We introduce a method using a Diffusion Model to synthesize character-dependent keystroke data, linking keycode with its biometric features, a significant advancement over prior key-independent techniques, inspired by diffusion models' success in image generation [9].
- Our evaluation of bot detection through keystroke timing analysis, excluding key identity, shows feasibility with a slight performance drop of about 0.03% in accuracy.
- Our results indicate that training classifiers with newly generated synthetic data enhances their performance.

The remainder of this paper is organized as follows: Section 2 reviews existing methodologies for web bot detection, with a focus on keystroke dynamics. Section 3 details the features and methods employed to train classifiers, drawing from insights in related research. Section 4 outlines the evaluation process and limitations of our approach. Finally, Section 5 concludes the paper.

2 Related Work

Contemporary CAPTCHA systems, including hCaptcha [14] and reCAPTCHA [13], employ biometric indicators such as mouse movements, along with request data, for bot detection. Their effectiveness in distinguishing bots from human users remains undisclosed, and the specifics of how this is achieved are also omitted. This is likely due to the protection of trade secrets and the intention to withhold

critical information from bot developers. Google, the developer of reCAPTCHA, has not publicly detailed the data utilized in their system, although attempts to demystify this have been made in related research [19].

Bot detection using behavioral data, such as mouse movements, is a known concept [1,10]. In our study [17], we address limitations—such simplistic assumptions about mouse movements, e.g., straight lines [10]. Our method differentiates between bots and humans, including those bots that simulate human mouse movements via third-party software. We incorporate mouse movement data not only collected on our websites but also from external sources. Our results show significant improvement in bot detection using a diverse set of mouse movement data, eliminating the need for website-specific user data. Additionally, we investigate the relationship between data volume and detection accuracy, discovering that with a sampling rate of 30 points per second, we require only 1.66 seconds (50 samples) to detect a bot with 99.71% accuracy. This finding highlights the minimal amount of data necessary for effective bot detection.

TypeNet [2], an RNN-based neural network, is developed for individual identification through keystroke dynamics. It operates by analyzing M keystroke sequences and corresponding features F, creating embeddings to authenticate individual keystrokes. The network processes a $M \times F$ vector, with F comprising various latencies: Hold Latency (time between key press and release), Inter-key Latency (time between releasing a key and pressing the next), Press Latency (time between consecutive presses), Release Latency (time between consecutive releases), and normalized ASCII code of keystrokes. TypeNet, consisting of two LSTM layers with 128 units and a tanh activation function, was trained using the Dhakal and Palin datasets, incorporating both physical and smartphone keystrokes. In performance comparisons for free text scenarios, TypeNet's Equal Error Rate (EER) of 2.2% on desktop data and 9.2% on smartphone data significantly outperformed the CNN-RNN model and the SVM model, demonstrating its effectiveness in keystroke-based individual authentication.

A recent study [4] aimed to distinguish human keystrokes from bot keystrokes, examining the role of key codes. They used the Dhakal [5] dataset and features from TypeNet. Two methods were tested: The first used Kernel Density Estimator (KDE) to generate synthetic keystrokes based on feature distributions for a non-specific group of persons. KDE is a non-parametric way to estimate the probability density function of a random variable. KDE works by smoothing sample data points with a kernel function, typically a Gaussian, to produce a continuous density estimate from discrete data. The second synthesized keystrokes for specific individuals using the Dhakal dataset. Both methods produced synthetic keystrokes to train two classifiers, a SVM and TypeNet, for separating human and non-human keystrokes. The study balanced synthetic and human keystroke data and found similar performance between the SVM and TypeNet. However, a limitation was the synthetic features' generation independent of the input text.

A study [3] investigated CNNs for classifying individuals based on keystroke dynamics across three datasets: CMU labs (51 users), GREYC KeyStroke (133 users), and GREYC web-based (83 users). Synthetic data were generated using

Gaussian parameters from these datasets, effectively tripling the volume. Features such as hold, inter-key, and press latencies served as feature vectors. The CNN architecture included two convolutional layers, two max pooling layers, and a fully connected layer, trained on an 80/20 split using the Adam optimizer at a 0.001 learning rate in TensorFlow, achieving high accuracy.

GANs demonstrate potential in bypassing mouse movement-based bot detection and may be applicable in creating human-like keystrokes, as indicated in *Web bot detection evasion using generative adversarial networks* [11]. This paper used GANs to emulate human mouse movements, utilizing a web dataset of Wikipedia viewer mouse movements and a subset of the HuMidb dataset. Mouse movements were represented as images. For evaluating bot detection, balanced accuracy was used, and recall measured bot evasion. Training a CNN with datasets X (human images) and Y (GAN images), the study found effective bot detection. However, retraining the GAN with new human data unseen by the detection model significantly decreased recall in both datasets.

The current research partly focuses on detecting synthetic data to challenge classification systems, with methods like SVMs and CNNs. A study [11] revealed the potential of generative models in creating synthetic mouse movements. There's room for improvement in synthetic keystroke generation, notably through advanced generative models like diffusion models, which have shown promising results in image synthesis [6]. A key limitation in existing approaches is the independence of synthetic keystrokes from input keystrokes, ignoring the potential impact of the input on typing behavior. Thus, the research questions for this paper are as follows:

RQ1: How accurately can generator models replicate keystroke dynamics?
RQ2: To what extent can classifier models distinguish between human and non-human keystrokes?
RQ3: How does key-dependency modeling affect bot detection accuracy?

3 Bot Detection via Keystroke Dynamics Analysis

The aim of this paper is to devise a robust model proficient in distinguishing between human and artificial keystroke dynamics. This necessitates a model proficient in generating keystrokes that mimic human behavior for training data purposes. The initial phase involves preprocessing the data to make it amenable for subsequent analysis. This is followed by the aggregation of datasets that typify human keystrokes, which serve as a standard for the model. Next, we fabricate synthetic keystroke data using an array of generative models. The culminating phase includes the development and training of classifiers that discern between human-generated and synthetic keystrokes, utilizing both datasets for this purpose.

3.1 Dataset and Data Cleansing

The generation of synthetic data and the development of an effective classifier necessitate a significant amount of high-quality data. In this context, 'high qual-

ity' denotes a dataset that comprehensively encompasses a wide spectrum of scenarios, facilitating robust generalization capabilities. For keystroke analysis, this entails incorporating a diverse cohort of participants, employing a variety of sentences or characters, and utilizing different keyboard layouts. A prime example of a dataset fulfilling these requirements is the Dhakal dataset [5]. The Dhakal dataset encompasses 136 million keystrokes, collated from 168,000 participants. Data collection entailed participants typing 15 distinct predefined sentences, such as "Have I mentioned How much I love Houston traffic?", as swiftly and accurately as possible using their keyboards. Each sentence was displayed individually, with a new sentence appearing only after the participant pressed "continue". The dataset recorded each keystroke, including the press and release times.

We scrutinized the dataset automatically. The aim was to identify and eliminate incomplete data, focusing particularly on rows with null values or any missing values. Given that human typing is inherently imperfect, errors such as mistyping the intended key (e.g., pressing "s" instead of "a") are common. However, training the model on such data could skew its accuracy. Accidental keystrokes, like typing "aq" instead of "a@", introduce noise into the training data, not only in terms of incorrect characters but, more critically, in the timing and rhythm that diverge from the typical pattern for these characters. If such data are not filtered out, the model might erroneously associate these timing patterns with the typed characters. This is comparable to typing random letters without intention; the rhythm and speed would differ from deliberate typing. To ensure the model's accuracy, it is essential to exclude these anomalies, thereby guaranteeing that the timing data the model learns from truly represents intentional, deliberate keystrokes. This may lead to the unintended consequence that human typing errors could be more likely misidentified as bot-generated text. The preprocessed dataset was thus curated to encompass a broad spectrum of inputs from all participants.

3.2 Data Preparation for Keystroke Classifiers

Key-Independent Modeling Previous research, such as *Statistical Keystroke Synthesis for Improved Bot Detection* [4], has explored keystroke modeling and synthesis without distinguishing sentences or letters, yet achieving notable success in bot detection. Nonetheless, these authors acknowledged a limitation in their approach: the modeling does not account for the specific keys being pressed. For instance, the timing sequence for "$1.5\mu^2$" would be identical to that of "hello" under their model. This approach overlooks the reality that latency features do differ depending on the specific key pressed.

Inter-Key Dependent Modeling In contrast to key-independent models, timing features in reality are influenced by the specific key pressed. Moreover, the timing can be affected by adjacent keys. For example, the latency of the letter "m" in the sequences "@m" and "mm" would differ due to the influence of preceding and succeeding characters.

For our analysis, data is transformed into a sequence of features. These latencies are well-established in keystroke dynamics literature [2] [4] and facilitate comparison with related work.

Based on these features, we define the following feature lists:

- L_{nokey}: A feature list comprising:
  hold latency, *press latency*, *release latency*, and *inter-key latency*.
- L_{key}: A feature list that extends L_{nokey} by including *keycode*.

3.3 Synthetic Keystroke Generation

The aim of generating synthetic keystrokes is to closely emulate human keystroke patterns, rendering them indistinguishable from actual human input by existing detection systems. Consistent with contemporary research, we employed kernel density estimation for synthesis, as utilized in [4]. Additionally, we explored the application of diffusion models, inspired by their success in image synthesis domains, such as DDPM [9] and Stable Diffusion [16]. The methodologies and training protocols for both models are elaborated below.

Kernel Density Estimation Utilizing the scikit-learn library[1], we adhered to the hyperparameters established in existing literature, particularly those in [4], which recommends a bandwidth of 1.0 using the Gaussian Kernel. Given the unimodal, approximately normal distribution of all features, we employed the Gaussian kernel for optimal parameter search, applying Silverman's rule of thumb [18] in conjunction with a grid search method.

Key-Independent Generation For a given sentence X, the generator aims to produce a synthetic sequence mimicking human keystroke features (*hold-latency*, *inter-key-latency*, *press-latency*, *release-latency*, *keycode* (cf. Section 2, Section 3). In key-independent modeling, separate KDE models are trained for each latency type based on optimal parameters identified earlier. Synthetic sequences are then generated for each letter using these KDE models.

Letter-Dependent Generation This approach employs bigrams, reversed bigrams, and trigrams to analyze correlations between consecutive keystrokes in human typing. Bigrams and trigrams are defined by corresponding pairs and triplets of keystrokes, each characterized by various latencies and *keycodes* (the ASCII code of a letter). In the case of bigrams, the latencies of the current key are influenced by the successor key, while for reversed bigrams, they depend on the predecessor. In trigrams, the latencies of the current key are affected by both the predecessor and successor keys. A dictionary of KDE estimators is created for each unique bigram and trigram, trained on the corresponding latencies. When encountering elements not present in the training data, we resort to separate KDE estimators trained on the overall latency data.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KernelDensity.html

The generation process starts by decomposing input text into bigrams, reversed bigrams, and trigrams. Synthetic latencies are generated for each of these elements, adjusting for the number of letters in a sentence. If a KDE estimator exists for a specific sequence, a sample is drawn from it and modified with random deviations. Otherwise, samples are drawn from the unigram KDE estimators. The synthetic latencies, once generated and adjusted to include the ASCII code of the respective letter, are compiled into the final synthetic latency list. To maintain dataset balance, an equivalent number of synthetic keystroke sequences to the number of human sequences are generated.

Denoising Diffusion Probabilistic Models The integration of diffusion models for generating synthetic keystroke sequences is particularly intriguing, given their success in other generative tasks, notably surpassing state-of-the-art GANs in image generation [6]. We utilize the *denoising diffusion pytorch* package [15], that implements Denoising Diffusion Probabilistic Model [9]. Optimal hyperparameters, such as batch size and learning rate, were determined through a grid search approach. Differing from the KDE method, this model processes and generates sequences in the format $Press\_time_0$, $Release\_time_0$, ..., $Press\_time_n$, $Release\_time_n$, corresponding to the list L_1 from Section 3.2. As these data require normalization for DDPM usage, we applied the tangent hyperbolic function for normalization, due to its effective mapping of positive values into the 0-1 range and the existence of an inverse function.

The diffusion model, unlike KDE generators, is trained on full sequences of human keystroke timings, not just unigrams, bigrams, or trigrams. It trains on sequences of up to 30 keystrokes. New data generation occurs through model sampling. Sampled values, ranging from 0-1 and derived from normalized data, are reverted to their original scale using the inverse of the hyperbolic tangent. Since the sampling basis of a DDPM model is Gaussian noise, the process is inherently non-deterministic. These sampled sequences are converted into latencies, which are then used as input for classifiers.

3.4 Classification Models

The primary objective of the classifiers is to determine whether a given sequence of features originates from human or synthetic input. Architectures capable of integrating temporal dependencies, such as Long Short-Term Memory (LSTM) networks and Transformers, are particularly suitable for this task. Additionally, SVMs have shown promising results in related literature. Thus, in this paper, we evaluate LSTM, SVM, and Transformer-based models as classifiers. All models are trained on diverse datasets comprising both human and synthetic keystroke sequences, with each set containing roughly equal distributions of both data types. Each sequence is associated with a label: 1 for bot-generated data and 0 for human data.

Long Short-Term Memory For effective sequence processing and temporal data consideration, we utilized a LSTM-based model, implemented using Py-

torch Lightning. The architecture comprises two LSTM layers, each followed by a dropout layer [20] to prevent overfitting, and a batch normalization layer [12] to stabilize training. The model employs Relu and Elu activation functions, with the output of the second LSTM layer passing through a linear layer, sized for two classes. Outputs are converted into logistic probabilities using logarithmic softmax, and Xavier initialization is applied to linear layers' weights.

The model is optimized using cross-entropy loss. To mitigate overfitting, an early stopping mechanism halts training upon no significant improvement in validation dataset performance. Training employs batch processing for efficiency and robustness against noise, and checkpoints are saved for model state restoration or continued training.

Transformer The Transformer model, implemented similarly using PyTorch Lightning, starts with a linear embedding layer for feature sequence transformation, supplemented by positional encoding. The encoded embeddings pass through a transformer encoder, comprising multiple identical layers with multi-head self-attention and feed-forward networks.

The output of the transformer encoder undergoes global average pooling for a final fixed-size output. Xavier initialization is used for weights, and ELU functions are the primary activation method. Like the LSTM model, the Transformer uses cross-entropy loss, with early stopping and checkpointing implemented to optimize training and prevent overfitting.

4 Evaluation

In this section, we present our findings addressing three research questions: the authenticity of generated keystrokes, the discrimination between human and non-human keystrokes, and the impact of keycode inclusion on classifier performance. An overview of the evaluated generator models is given by Table 1. These models are employed in our evaluation to generate synthetic keystrokes for training and testing classifiers, which are designed to differentiate between human and synthetic keystrokes. We employ a cross-evaluation approach, training classifiers on one generator model's data and testing them on synthetic data generated by a different generator.

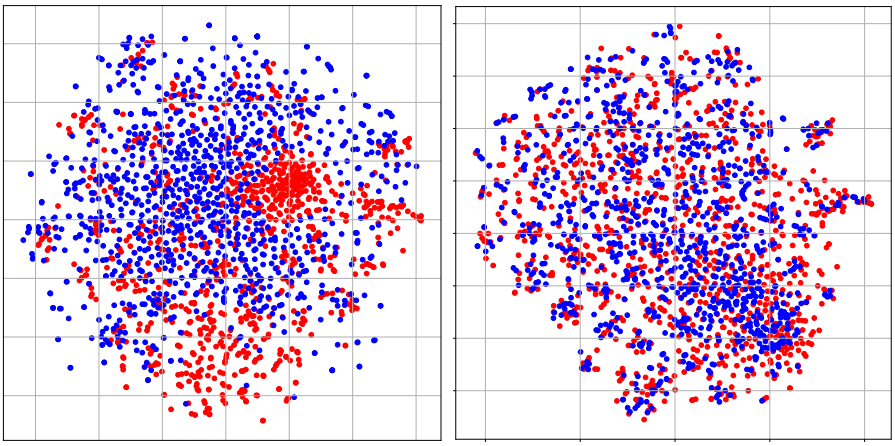
4.1 RQ1: Authenticity of Generated Keystrokes

To evaluate the generator models (cf. Table 1), we first employed data visualization techniques, including t-SNE analysis, to compare synthetic and human data. t-SNE, a technique for reducing dataset dimensionality, helps ascertain if the synthetic data closely mirrors the human data's distribution, a key aspect in evading existing classifiers and aligns with methods used in current literature [4].

The t-SNE analysis presented in Figures 1a and 1b provides initial evidence suggesting the superiority of the diffusion model over alternative generators like KDE in emulating human keystroke dynamics. The synthetic keystroke timings

Model	**Generation of timing information**	**Abbreviation**
KDE Universal [4]	Based on normal key timing, independent of current and surrounding keys.	KDE Univ.
KDE Bigram	Dependent on the current and successor key.	KDE Bi.
KDE Bigram Reversed	Dependent on the current and predecessor key.	KDE Bi. Rev.
KDE Trigram	Dependent on the current, predecessor, and successor keys.	KDE Tri.
Diffusion	Based on a 30-keystroke input.	Diff.

Table 1: Models for Generating Timing Information: Models are different by key sequence context, ranging from independent (KDE Universal) to dependent on immediate (KDE Bigram, KDE Bigram Reversed) and extended (KDE Trigram) key sequences, as well as sentence-level context (Diffusion).



(a) Generator Model: Universal KDE (b) Generator Model: Diffusion

Fig. 1: t-SNE visualization comparing the Synthetic (blue) and Human (red) keystroke timing features for two models.

generated by the diffusion model are notably more indistinguishable from human keystrokes. This observation warrants further investigation in the ensuing evaluation.

In relation to the other t-SNE analyses of KDE, particularly for bigram and trigram models, the lack of significant divergence from the universal KDE results suggests that the keystrokes generated by the universal, bigram, and trigram models are more similar than previously assumed. It is important to note that Figures depicting these analyses were omitted due to space constraints and their similarity.

4.2 RQ2: Discrimination Between Human and Non-Human Keystrokes

This section assesses the capability of distinguishing between human and non-human keystroke sequences. We evaluate three classifiers—Support Vector Machine, LSTM, and Transformer—across various datasets to determine their efficacy with different data types. Training datasets, each comprising 7500 human and 7500 synthetic samples, where one sample represents a single written sentence randomly selected from the Dhakal dataset, were generated using various generators. As per similar research [4], we consider two scenarios: Using four features (hold-latency, inter-key-latency, press-latency, release-latency) and, RQ3, including keycodes. The first scenario could also be somewhat advantageous from a privacy perspective, as classifiers do not have direct access to the keycode information but can indirectly infer it based on the keystroke timings.

An additional *Random Sequences* dataset, simulating unsophisticated attackers with random number sequences, was also created. The *Mixed* dataset comprises an equal blend of synthetic data from all generators and human data.

The SVM classifier demonstrates a comparable proficiency in distinguishing synthetic from human keystrokes in Universal KDE, akin to the performance observed in related work [4]. Furthermore, the classifier exhibits robust generalizability to random sequences and extends this capability to both bigram and trigram KDEs.

The SVM classifier's ability to differentiate between synthetic and human keystrokes in Bigram, Bigram Reversed, (and Trigram) configurations demonstrates a significant reliance on the predecessor key in a sequence. This finding is crucial in understanding the classifier's performance. The classifier showed exceptional accuracy on data resembling its training set but its effectiveness markedly declined when confronted with unfamiliar data. This is most pronounced in scenarios involving where the SVM was trained using sequences generated by the Diffusion Model. Then the classifier struggled considerably to distinguish synthetic keystrokes with random timings (Rand. Seq.), from human-generated sequences. Further insights were gained from the t-SNE analysis, which highlighted that keystrokes synthesized by the Diffusion Model closely mimic human keystrokes, more so than those generated by other models. Despite this, even when the classifier was trained with data from the Diffusion Model, it was unable to effectively differentiate these from human keystrokes in the Random Se-

Trained on \ Tested on		KDE Univ.	KDE Bi.	KDE Bi. Rev.	KDE Tri.	Diff.	Rand. Seq.
KDE Univ.	ACC	0.9869	0.9440	0.9871	0.9436	0.4949	0.9866
	F1	0.9870	0.9422	0.9873	0.9418	0.0270	0.9868
KDE Bi.	ACC	0.8824	0.8569	0.9871	0.8261	0.4993	0.9014
	F1	0.8908	0.8640	0.9873	0.8298	0.2790	0.9101
KDE Bi. Rev.	ACC	0.9869	0.9474	0.9871	0.9439	0.4949	0.9865
	F1	0.9870	0.9459	0.9873	0.9420	0.0274	0.9867
KDE Tri.	ACC	0.9810	0.9794	0.9816	0.9774	0.4968	0.9809
	F1	0.9813	0.9797	0.9819	0.9777	0.0549	0.9812
Diff.	ACC	0.5317	0.6100	0.5304	0.5907	0.7348	0.3311
	F1	0.4624	0.5891	0.4600	0.5599	0.7530	0.0033
Mixed	ACC	0.7814	0.7927	0.7792	0.7444	0.5192	0.9343
	F1	0.7600	0.7753	0.7571	0.7075	0.2590	0.9383

Table 2: Accuracy (ACC) and F1-Score (F1) of the SVM Classifier when trained and tested on various synthetic keystrokes datasets. Training and testing were conducted on datasets corresponding to the generator used for creation, such as KDE Univ, which includes synthetic keystrokes generated by the KDE Universal generator. Keycode information was intentionally omitted during the training and testing processes, compelling the classifier to rely solely on timing characteristics for differentiation. Each dataset consisted of 15.000 samples.

quences test. This underlines a notable limitation in the classifier's generalizability, raising questions about its reliance on predecessor keys and its adaptability to diverse keystroke patterns.

This generalization issue persisted even when the classifier was trained and tested on the Mixed dataset, which included a variety of data types. The consistent occurrence of this challenge across different datasets suggests a fundamental limitation in the model's adaptability to diverse data types. Given these results, it's possible that the volume of training and test data was insufficient for the classifier to develop a robust and generalizable model. This hypothesis merits further exploration. Therefore, we plan to investigate the effects of varying data sizes on classifier performance in subsequent sections, utilizing a different classifier to gain a more comprehensive understanding.

In consideration of space limitations, we will focus on the results of two advanced classifiers: LSTM and Transformer, as shown in Table 3. Both models were trained on a balanced mixed dataset with 16,000 synthetic and 16,000 human data points. The results indicated significant performance improvements over models trained on smaller datasets, with the Transformer model achieving an average accuracy of 0.90 and the LSTM model an impressive 0.97.

The increased training data volume notably enhanced model performance, particularly evident in the sophisticated LSTM and Transformer classifiers. The LSTM model was especially effective in differentiating between human and syn-

Model \ Tested on		KDE Univ.	KDE Big.	KDE Big. Rev.	KDE Tri.	Diff.	Rand. Seq.
Transformer	ACC	0.9332	0.8919	0.9154	0.9363	0.9270	0.9293
	F1	0.9314	0.8880	0.9145	0.9350	0.9271	0.9268
LSTM	ACC	0.9807	0.9776	0.9797	0.9810	0.9808	0.9803
	F1	0.9802	0.9778	0.9799	0.9806	0.9810	0.9797

Table 3: Performance metrics of the LSTM and Transformer Classifier trained and evaluated using the Mixed dataset. Keycode information was intentionally omitted during training and testing. Samples: 32000.

thetic keystrokes, including the challenging Diffusion dataset that had stumped other models.

4.3 RQ3: Assessing the Impact of Keycode Inclusion on Classifier Performance

In our analysis, the decision to omit keycodes could potentially affect classifier performance. This approach however aligns with methodologies observed in other studies [4]. Thus, our research includes a focused investigation into the impact of incorporating keycodes on the classifier's effectiveness. Table 4 presents the results of the LSTM classifier, our best performer, in differentiating between human and synthetic keystroke data, using a balanced dataset of 32,000 samples (half human, half synthetic).

Trained on \ Tested on		KDE Univ.	KDE Bi.	KDE Bi. Rev.	KDE Tri.
KDE Univ.	ACC	1.000	0.998	0.998	0.998
	F1	1.000	0.998	0.998	0.998
KDE Tri.	ACC	1.000	1.000	1.000	1.000
	F1	1.000	1.000	1.000	1.000

Table 4: Performance of the LSTM Classifier on various KDE synthetic samples with keycodes in feature vectors, evaluated across different datasets. Samples: 32000.

As illustrated in Table 4, the inclusion of keycodes significantly enhances the performance of the LSTM classifier, underscoring the pivotal role of keycode information in augmenting classifier efficacy. The improvement achieved with Trigram training over the Universal training approach underscores the effectiveness of Trigram data in enabling the classifier to discern between synthetic and

human keystrokes more accurately than when trained with universal synthetic keystrokes. This effectiveness might be attributed to the trigram model's capacity to capture more intricate details regarding timing, which stems from its sensitivity to key positions and sequences. These observations are in harmony with findings presented in related literature [4], reinforcing the recommendation to integrate keycodes into bot detection models.

4.4 Limitations

The primary limitation of KDE generators stems from their focus on key dependencies, especially considering the challenge that arises from the rarity of certain sequences in trigrams or higher-order models. The scarcity of human data for these longer sequences results in a lack of training data, rendering KDE-based approaches less feasible as they cannot infer knowledge about unseen sequences. This limitation significantly affects the sophistication of the synthetic data generated. The diffusion model currently is confined to a sequence length of 30, thus prohibiting the generation of extended synthetic keystroke sequences. This limitation is also crucial for classifiers, as they are restricted to a maximum sequence length of 30 in the present framework. A pervasive limitation among contemporary generators is their capacity to emulate a blend of human typing styles, yet they thus fail to accurately replicate the typing behavior of a specific individual. This shortfall can adversely affect classifier performance, as classifiers may excel by detecting this particular discrepancy.

5 Conclusion

This paper focused on enhancing synthetic keystroke generation and evaluating classifiers ability to discern this improved data. It introduced key-dependent generation using bigrams and trigrams, employing Kernel Density Estimator and, notably, a diffusion model trained on the Dhakal dataset. Evaluation of various classifiers showed that LSTM and Transformer models outperform the SVM, especially when trained with ample data. However, these models' effectiveness decreased with limited data. Future research could aim to refine the recreation of individual typing patterns. Ultimately, the progression of these technologies may culminate in generators that can flawlessly replicate or generate new human typing behaviors.

References

1. Acien, A., Morales, A., Fierrez, J., Vera-Rodriguez, R.: BeCAPTCHA-Mouse: Synthetic Mouse Trajectories and Improved Bot Detection. arXiv:2005.00890 [cs] (Mar 2021), http://arxiv.org/abs/2005.00890
2. Acien, A., Morales, A., Monaco, J.V., Vera-Rodriguez, R., Fierrez, J.: TypeNet: Deep Learning Keystroke Biometrics (1 2021), http://arxiv.org/abs/2101.05570

3. Çeker, H., Upadhyaya, S.: Sensitivity Analysis in Keystroke Dynamics using Convolutional Neural Networks. Tech. rep.
4. DeAlcala, D., Morales, A., Tolosana, R., Acien, A., Fierrez, J., Hernandez, S., Ferrer, M.A., Diaz, M.: Statistical Keystroke Synthesis for Improved Bot Detection (7 2022), https://arxiv.org/pdf/2207.13394v2.pdf
5. Dhakal, V., Feit, A.M., Kristensson, P.O., Oulasvirta, A.: Observations on typing from 136 million keystrokes. In: Conference on Human Factors in Computing Systems - Proceedings. vol. 2018-April. Association for Computing Machinery (4 2018). https://doi.org/10.1145/3173574.3174220
6. Dhariwal, P., Nichol, A.: Diffusion Models Beat GANs on Image Synthesis. In: Advances in Neural Information Processing Systems. vol. 11 (2021)
7. Guerar, M., Verderame, L., Migliardi, M., Palmieri, F., Merlo, A.: Gotta captcha'em all: a survey of 20 years of the human-or-computer dilemma. ACM Computing Surveys (CSUR) **54**(9), 1–33 (2021)
8. Heath, N.: Expedia on how one extra data field can cost $12m. https://www.zdnet.com/article/expedia-on-how-one-extra-data-field-can-cost-12m/ (2010), accessed: 2021-10-18
9. Ho, J., Jain, A., Abbeel, P.: Denoising diffusion probabilistic models. In: Advances in Neural Information Processing Systems. vol. 2020-December (2020)
10. Iliou, C., Kostoulas, T., Tsikrika, T., Katos, V., Vrochidis, S., Kompatsiaris, I.: Detection of advanced web bots by combining web logs with mouse behavioural biometrics. Digital threats: research and practice **2**(3), 1–26 (2021)
11. Iliou, C., Kostoulas, T., Tsikrika, T., Katos, V., Vrochidis, S., Kompatsiaris, I.: Web bot detection evasion using generative adversarial networks. In: Proceedings of the 2021 IEEE International Conference on Cyber Security and Resilience, CSR 2021. pp. 115–120. Institute of Electrical and Electronics Engineers Inc. (7 2021). https://doi.org/10.1109/CSR51186.2021.9527915
12. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: 32nd International Conference on Machine Learning, ICML 2015. vol. 1 (2015)
13. Liu, W.: Introducing reCAPTCHA v3: the new way to stop bots. https://developers.google.com/search/blog/2018/10/introducing-recaptcha-v3-new-way-to (2018), accessed: 2021-05-20
14. Machines, I.: Stop more bots. start protecting user privacy. https://www.hcaptcha.com/ (2018), accessed: 2021-05-20
15. Phil Wang: https://github.com/lucidrains/denoising-diffusion-pytorch
16. Rombach, R., Blattmann, A., Lorenz, D., Esser, P., Ommer, B.: High-resolution image synthesis with latent diffusion models. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 10684–10695 (June 2022)
17. See, A., Wingarz, T., Radloff, M., Fischer, M.: Detecting web bots via mouse dynamics and communication metadata (2023), accepted for publication at International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)
18. Silverman, B.: Density Estimation for Statistics and Data Analysis. Routledge (2 1998). https://doi.org/10.1201/9781315140919
19. Sivakorn, S., Polakis, J., Keromytis, A.D.: I'm not a human: Breaking the Google reCAPTCHA. Black Hat **14** (2016)
20. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research **15** (2014)

Appendix C

BOTracle: A framework for Discriminating Bots and Humans

Abstract

Bots constitute a significant portion of Internet traffic and are a source of various issues across multiple domains. Modern bots often become indistinguishable from real users, as they employ similar methods to browse the web, including using real browsers. We address the challenge of bot detection in high-traffic scenarios by analyzing three distinct detection methods. The first method operates on heuristics, allowing for rapid detection. The second method utilizes, well known, technical features, such as IP address, window size, and user agent. It serves primarily for comparison with the third method. In the third method, we rely solely on browsing behavior, omitting all static features and focusing exclusively on how clients behave on a website. In contrast to related work, we evaluate our approaches using real-world e-commerce traffic data, comprising 40 million monthly page visits. We further compare our methods against another bot detection approach, Botcha, on the same dataset. Our performance metrics, including precision, recall and AUC, reach 98 percent or higher, surpassing Botcha.

Reference

Jan Kadel, August See, R. Sinha, M. Fischer. BOTracle: A Framework for Discriminating Bots and Humans. European Symposium on Research in Computer Security, SecAI Workshop, 2024. ©2024 Springer.

Contribution

The core concept, website traversal graph approach, and evaluation design originated from the author of this dissertation. The first author implemented the methodology and performed the evaluation. The third and fourth co-authors contributed to the refinement of the manuscript.

BOTracle: A framework for Discriminating Bots and Humans

Jan Kadel[1⋆], August See[2⋆], Ritwik Sinha[3], and Mathias Fischer[2]

[1] Adobe Inc. {kadel,risinha}@adobe.com
[2] Universität Hamburg, Germany
{richard.august.see,mathias.fischer}@uni-hamburg.de

Abstract. Bots constitute a significant portion of Internet traffic and are a source of various issues across multiple domains. Modern bots often become indistinguishable from real users, as they employ similar methods to browse the web, including using real browsers. We address the challenge of bot detection in high-traffic scenarios by analyzing three distinct detection methods. The first method operates on heuristics, allowing for rapid detection. The second method utilizes, well known, technical features, such as IP address, window size, and user agent. It serves primarily for comparison with the third method. In the third method, we rely solely on browsing behavior, omitting all static features and focusing exclusively on how clients behave on a website. In contrast to related work, we evaluate our approaches using real-world e-commerce traffic data, comprising 40 million monthly page visits. We further compare our methods against another bot detection approach, Botcha, on the same dataset. Our performance metrics, including precision, recall, and AUC, reach 98 percent or higher, surpassing Botcha.

Keywords: web bots, bot detection, website navigation, user behavior

1 Introduction

The automation of work through software is increasingly common, and used to alleviate the burden of monotonous tasks, particularly in internet communications, where bots automate repetitive functions. A prime example is search engine bots indexing the internet autonomously. However, the prevalence of malicious bots is concerning, with over half of internet traffic being bot-related [43]. Trends indicate a rise in malicious activities by bots, including spamming, scalping, and influencing elections [9,43,6].

Differentiating between bot and human traffic is challenging due to the diverse and sophisticated nature of bots, some of which can mimic complete browser environments to evade detection. The issue is compounded when detection systems incorrectly identify humans as bots or fail to detect bots, especially as bots increasingly evade measures like CAPTCHAs [14,3,15,42]. Such errors

⋆ These authors contributed equally to this work.

can lead to customer dissatisfaction and security risks. Therefore, bot detection systems should prioritize identifying bots based on characteristics impervious to common evasion techniques, focusing on passive monitoring and analyzing inherent traits of web clients, such as behavior or intent, rather than relying solely on challenge-response tests [14,3,15,42].

The primary contribution of this paper is the development of a multi-stage web bot detection pipeline with distinct detectors, evaluated on a real-world e-commerce dataset. In greater detail, our contributions include:

- We propose a layered bot detection approach suitable for business environments, given our industry focus. This approach combines multiple techniques into an efficient pipeline. It begins with heuristic methods that utilize static features, such as IP addresses, to make quick decisions, primarily to conserve computing resources. For more detailed per-request analysis, a semi-supervised Generative Adversarial Network (SGAN) is employed. Additionally, a Deep Graph Convolutional Neural Network (DGCNN) evaluates potential bots using website traversal graphs, emphasizing session-wide behavior. Notably, this approach concentrates solely on behavior, without relying on features such as IP addresses, user agents, or window sizes. Bots are generally unaware of the average behavior on a specific website, making it challenging for them to mimic it. Furthermore, even if a bot attempts to mimic typical user behavior, it loses its inherent advantages, such as tirelessness and speed.
- Our approach and its individual components are evaluated using a proprietary, anonymized dataset from a real-world e-commerce platform[3]. This platform attracts approximately 40 million monthly page visits and generates an estimated annual revenue of between 500 million and 1 billion USD. We benchmark the performance of our method against related works that have utilized the same dataset.
- We conduct an in-depth analysis to ascertain which bot features most significantly influence the SGAN's effectiveness. Furthermore, we explore how the size of website traversal graphs impacts the performance of the DGCNN.

In this paper, we first review existing bot detection methods, assessing their strengths and weaknesses (Section 2). We then introduce our novel approach for effective bot detection (Section 3), followed by an evaluation of its performance against established methods using real-world data (Section 5). The paper concludes with a summary of our findings and their implications (Section 6).

2 Related Work

Research in this field can be categorized into two main streams: active and passive approaches. Active methods, such as CAPTCHAs, present direct challenges

[3] Unfortunately, we cannot release this dataset or disclose the company's name, as this may violate our business contract with them

to users. Passive methods typically involve risk assessment strategies that utilize heuristics or machine learning algorithms. This section will review a selection of prior studies, highlighting their respective advantages and limitations.

2.1 Active Approaches

Modern CAPTCHA systems, like hCaptcha and reCAPTCHA [22,24], blend biometric data, including mouse movements, with request information. Despite this integration, the specifics of their bot detection strategies, such as website-specific tailoring or the effectiveness of various elements, remain undisclosed. This opacity is likely strategic, aimed at safeguarding proprietary methods and hindering bot developers from adapting their strategies. Google, for instance, keeps the data utilized in reCAPTCHA under wraps. Research efforts, including those by Sivakorn et al. [37], are dedicated to deciphering these CAPTCHA mechanisms. Moreover, CAPTCHAs, despite their technological advances, still detract from user experience and can be time-consuming. This is true even in modern systems that employ risk assessment to reduce the frequency of CAPTCHA challenges for users [2].

This category also includes registration with personal data, such as using telephone numbers or presenting an official ID, are included. These methods, however, may deter users due to the additional hurdles they present [13] and their lack of privacy friendliness.

2.2 Passive Approaches

Bot detection can be straightforwardly implemented by blocking IPs known for malicious activities, utilizing resources such as *abuseipdb*[4]. Moreover, various methods focus on analyzing request data to detect bots [16,24,22,38,20,18,12]. However, there is an issue wherein a bot might abandon browser automation frameworks like Selenium or Puppeteer and opt instead for a headful Chrome instance. This approach could circumvent fingerprints that are specific to browser automation frameworks.

There are also obfuscation approaches that complicate bot creation. See et al. [33] proposed polymorphic protocols that aim to generate a unique protocol for each client (user). While this approach is effective for native applications, it does not perform well for web clients and within the HTTP context.

Iliou et al. [16] compared machine learning algorithms using attributes from previous studies, avoiding cross-website tracking or external resources like IP databases for a more replicable and privacy-friendly approach. Analyzing a year's worth of HTTP log data from MK-Lab's web server[5], they categorized bots as simple or advanced based on browser agent names and IP history. Their findings indicate that the effectiveness of attribute sets varies with the chosen machine learning method, with Random Forest and Multilayer Perceptron performing

[4] https://www.abuseipdb.com/
[5] https://mklab.iti.gr/

best. An ensemble classifier provided more stable results. While simple bots were easily detectable, advanced bot detection proved challenging, especially in scenarios requiring low false positive rates.

Related research also explores biometric authentication using mouse dynamics or typing behavior[35,34,36,31,19,32]. See et al. [35,34] demonstrated the feasibility of distinguishing humans from bots based on mouse dynamics and typing behavior, utilizing open-source datasets for human data and generating bot data from third-party projects. They concluded that, while differentiation is currently possible, the sophistication of methods for generating artificial human-like keystrokes and mouse movements is likely to increase.

The Botcha framework [10] centers on identifying malicious bots, employing Positive Unlabeled Learning (PU Learning) for data labeling. It assumes e-commerce platform clients making purchases are human, while others remain unlabeled. A machine learning classifier then categorizes these unlabeled sessions, determining human-like characteristics. Dhamnani et al. modified the traditional PU classifier to increase accuracy. Their validation used cloud provider IP addresses as negative benchmarks, with the model incorrectly classifying 2.5% of these as human but correctly identifying 99% of human-labeled data. The framework tagged 82% of sessions as human, leaving 18% unclassified. While effective in data-scarce scenarios, using purchasing behavior as a sole human indicator is problematic, given some bots are designed for purchases [21] or content scraping [4]. Additionally, Botcha's reliance on authentication mechanisms to deter bots may be ineffective against advanced bots capable of autonomous authentication. Although Botcha's adaptation of PU learning improves performance with specific datasets, this approach may limit its broader applicability. We later compare our approach to Botcha's results.

Cabri et al. [9] analyzed web sessions from HTTP logs, using heuristics for initial classification. Sessions were identified as bots based on factors like known bot user agents and IP addresses, or as humans if matching Udger database browsers [1]. Their approach used a multi-layer perceptron (MLP) and Wald's sequential probability ratio test [41] to classify sessions as Bot, Human, or None, with uncertain cases reassessed using more data. This method achieved a notable F1 score of approximately 0.96. However, the lack of external benchmarking and reliance on database labels for human-bot distinction, which could be exploited by bots altering user agents, are limitations. Yet, its innovative use of common HTTP features and reassessment of ambiguous cases demonstrate its potential effectiveness.

BotGraph [23] is a web bot detection method, analyzes client sessions through website traversal patterns. It constructs a website's sitemap, then maps session requests onto a sitemap subgraph, representing client navigation. These subgraphs are transformed into images to train a Convolutional Neural Network (CNN), classifying sessions as bot or non-bot using supervised learning. Over 30 experts manually labeled the data, focusing on behavioral and fingerprinting features. The model achieved 93.5% accuracy on a search engine dataset, 95.7% on a news site, and 98.4% on a university site, with overall precision and recall

around 95%. BotGraph effectively differentiates bot and human website usage patterns. However, it faces challenges in sessions with few requests (¿ 3), where both bots and humans show similar patterns [23]. Despite its reliance on manual labeling, which might introduce biases [26], BotGraph's adaptability across websites and its efficient performance (adding 2-6 milliseconds of latency on GPUs) are notable strengths.

The reviewed bot detection methods, while effective to some degree, each have notable limitations. Approaches relying on user agents are vulnerable to manipulation [9]. Botcha assumes bots do not make purchases, a premise easily challenged by bots capable of transactions [10]. BotGraph's focus on client traversal patterns overlooks critical page metadata, leading to potential false positives [23]. Additionally, CAPTCHAs, commonly used for bot detection, degrade user experience and can be bypassed [3,14]. These issues highlight the need for more robust, non-circumventable bot detection systems with high precision and recall, which current methodologies do not fully offer.

3 Detecting Bots

Our methodology integrates multiple classification strategies, including heuristic techniques, a Semi-Supervised Generative Adversarial Network (SGAN), and a Deep Graph Convolutional Neural Network (DGCNN). This integration aims to automate captcha resolution, thereby enhancing user experience by minimizing human intervention.

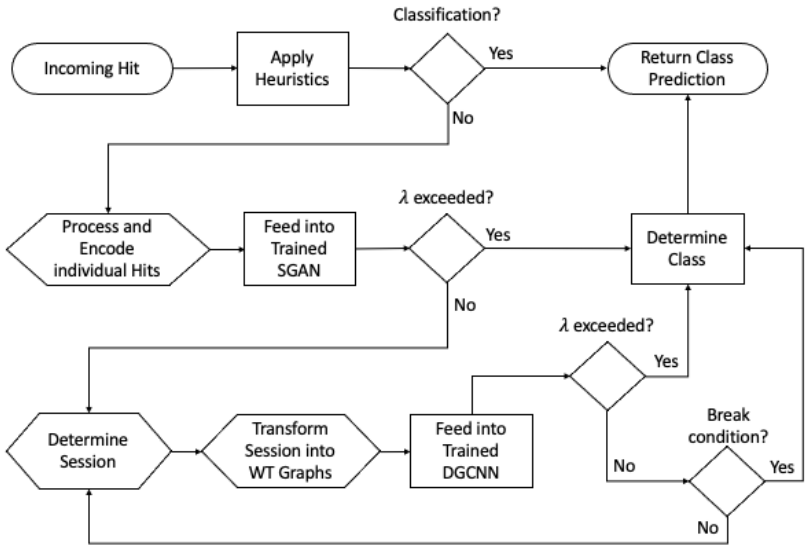


Fig. 1. Multi-Stage Bot Detection Pipeline Process as Flow Chart

Figure 1 depicts our bot detection pipeline. The procedure commences with an incoming hit - a request to a web server record. Initially, rapid heuristic-based bot detection is employed. If the heuristic approach successfully classifies a hit, the process concludes by issuing a class prediction. If not, the hit undergoes processing and encoding to prepare it for the SGAN model. This preprocessing involves discarding irrelevant features and transforming the data into a one-dimensional numerical vector using flagging, integer, and one-hot encoding. Additionally, we aggregate rare categorical values into special categories to compact the input vector. Once transformed, the numerical vector is input into the SGAN, yielding a bot or human probability. This probability is evaluated against a confidence threshold $\lambda \in [0, 1]$. Probabilities below λ lead to a rejection of the prediction due to insufficient confidence. Conversely, exceeding the threshold results in a definitive class label. Hits failing to produce a confident SGAN prediction are accumulated into client sessions, subsequently transformed into Website Traversal (WT) graphs, and analyzed by the DGCNN. Similar to the SGAN, the DGCNN produces a probability compared against λ. Surpassing this threshold finalizes the process with a class prediction, while falling short requires additional hits to augment the session before repeating the analysis.

Subsequent sections detail the pipeline components. We commence by outlining our approach to labeling in an unlabeled dataset from a website with over 50 million monthly visits, differentiating between bot and human characteristics. Generated using Adobe Analytics® within the Adobe Experience Cloud®, this dataset lacks pre-existing bot-related labels. Our labeling strategy is versatile, suitable for various websites. We then describe the heuristic bot detection methods, designed to identify straightforward cases like unmodified user-agent bots, thereby enriching the labeled data for SGAN and DGCNN training. Moreover, we elucidate the concept of WT graphs and their efficacy in bot detection. The machine learning models utilized in the pipeline are not detailed, as our focus was on their application rather than modifying their fundamental principles.

3.1 Bot Feature Analysis

In the task of distinguishing web clients as either bots or humans, it is essential to conduct a comprehensive analysis of client attributes, identifying those most indicative of bot activity. We categorize these attributes into two main groups: behavioral and non-behavioral. Non-behavioral attributes encompass identity, and technical factors, whereas behavioral attributes are defined by patterns in navigation, interaction, and frequency of website visits.

Non-behavioral Attributes :
- Identity Attributes: Consist of user identifiers, such as IP addresses.
- Technical Attributes: Relate to technical aspects, like user agent and screen size.

Behavioral Attributes :
- Traversal Attributes: Capture user's navigation patterns on the website.
- Interaction Attributes: Chronicle user interactions with site elements.

– Visit Attributes: Indicate the frequency and pattern of user visits.

Identity attributes, such as user location, or name, are seldom used in bot detection models due to their ease of manipulation. Focus is instead placed on behavioral and technical attributes, which compel bot developers to mimic human behavior more intricately to evade detection.

Technical attributes, such as user agent, screen size, and Java applet support, provide valuable insights. Models can detect bots through unusual user agents and atypical screen dimensions. Java applet support helps identify outdated or falsified user agents. However, these can also be forged.

Traversal attributes map a client's website journey, highlighting visited pages. Patterns like exhaustive breadth-first traversal suggest bot activity, unlike humans' targeted visits. Interaction attributes chronicle client engagements, such as using promo codes and purchasing. These patterns can distinguish bot types, for instance, crawlers that avoid purchases or scalper bots aiming for bulk acquisitions. Additionally, visit attributes, including visit frequency and patterns, help detect spammer bots, especially with high visit frequency and volume.

Heuristics for Detection of Obvious Bots The primary aim of bot detection heuristics is to identify and filter out apparent bots, a process that also aids in training machine learning models. For example, a user agent self-identified as "python request" is typically indicative of a bot. Labeling all data linked to such an agent as "bot" and integrating this into the training dataset enables the machine learning model to discern patterns, like bots making requests at regular intervals, for specific durations, or targeting certain pages. These heuristics are intentionally broad to ensure high precision in bot detection.

Forged User Agent: Some user-agents, like those from automation libraries (e.g., python-request), indicate bot activity. Bot operators often disguise them as regular browser user-agents like Firefox. If the user-agent's capabilities do not match a genuine agent, it is likely fraudulent. Human users typically do not modify their user-agents in this manner It is generally assumed that human users do not alter their user-agents in this way [45,11].

Time Between Hits Similarity: The interval between website visits can indicate bot activity [40,39]. Consistent timing between requests suggests automated browsing, as human interactions typically exhibit variability due to innate randomness. For example, a monitoring bot might be programmed to visit a website hourly with precise intervals.

Unrealistic Window Sizes: Effective human web browsing requires sufficiently large browser windows for clear graphical display. Bots, conversely, often utilize minimal window sizes, like those used by deep learning bots employing convolutional neural networks [17]. Smaller windows demand less processing power, which is cost-effective. Therefore, very small browser dimensions, such as an axis under 50 pixels, strongly suggest bot traffic.

3.2 Bot Detection Using Technical Features

This approach employs a SGAN trained on a mix of labeled and unlabeled data to analyze individual hits for bot-related characteristics. The choice of an SGAN is strategic due to its proficiency in processing both labeled and unlabeled data. Within this framework, a classifier is trained using the labeled data to differentiate between bots and humans. The SGAN's generator and discriminator are trained using both labeled and unlabeled data; the generator aims to create realistic labeled data points, while the discriminator's objective is to identify fabricated data points. Any generated data points not recognized as artificial are further utilized to refine the classifier. This approach enables the exploitation of features from unlabeled data in classifier training, making the SGAN model particularly suitable for addressing web bot challenges. Web log datasets, like the one used in this research, often contain a significant portion of unlabeled data points. The labels for training are derived from the assumptions regarding bot and human clients outlined in Section 5.1. These labels allow for confident categorization of the behavioral aspects of each data point, which are then employed to train the binary classifier within the SGAN's discriminator.

The primary goal of adversarial learning is to refine the generator to produce synthetic samples indistinguishable from real ones by the discriminator. However, in the context of bot detection, where the focus is on identifying rather than generating realistic bot data, it is necessary to modify the model architecture accordingly. Consequently, the discriminator in our model consists of two components sharing a common neural network framework: the classifier and the discriminator components. The discriminator component fulfills the traditional role of differentiating between fake and real samples. It outputs a probability $p_{real} \in [0, 1]$, indicating the likelihood of a sample being real; values near one suggest a real sample, while values close to zero imply a generated sample. The classifier component, meanwhile, is tasked with determining the class membership of a data point by predicting a vector of class membership probabilities $[p_1, ..., p_n]$ for n classes, utilizing the Softmax activation function. Despite their distinct objectives, the integration of shared weights enhances the classifier's performance by allowing it to benefit from both supervised and unsupervised learning through the discriminator. This is particularly advantageous when labeled data is scarce.

During the training phase, the classifier is trained with supervised samples. Its objective is to predict a vector of class membership probabilities for each sample and compute a loss value L_C using the cross entropy loss function [25]. The cross entropy loss function, represented as L, is generally defined in equation 4.1. Here, n represents the number of classes, and k refers to the k-th class. Additionally, Y_k indicates the true class value, being one if a sample belongs to class k and zero otherwise. p_k denotes the class membership probability resulting from the Softmax-activated output layer. The loss is minimized when the predicted probabilities align closely with their respective true values.

$$L(Y_k, p_k) = (-1) \cdot \sum_{k=1}^{n} Y_k \cdot log(p_k) \tag{1}$$

Both the discriminator and classifier components in our model utilize the cross entropy loss function, leading to their respective loss values being represented as L_D for the discriminator and L_C for the classifier. The class membership probabilities essential to this process are calculated by applying the Softmax activation function to the output layer of the shared neural network [25,8]. The Softmax function, denoted as S, is defined in equation 4.2. Here, Z represents the vector of all output activations, and z_i signifies the activation of the i-th neuron in the output layer. Softmax is responsible for converting these real-valued inputs into normalized probabilities. The output is a categorical probability distribution, indicating the likelihood of each class membership.

$$S(Z, z_i) = \frac{e^{z_i}}{\sum_{z_k \in Z} e^{z_k}} \tag{2}$$

The loss value is critical for optimizing the weights of the shared neural network through the backpropagation algorithm [29]. The discriminator, trained on both real and synthetic samples, aims to enhance its proficiency in distinguishing between these two types. The loss value L_D, derived from the discriminator's predictions, is backpropagated through both the discriminator and generator networks to refine the generator's performance. As the generator produces labeled, realistic data points, it significantly augments the classifier's efficacy by providing a larger pool of labeled data.

Distinct from the classifier, the discriminator employs a unique method for its output layer activation, a technique developed by Salimans et al. to optimize the training of generative adversarial networks [30]. This method, termed the *ExpSum Activation Function* and denoted as E, is formulated in equation 3. Here, Z represents the k activations in the output layer of the shared neural network, each playing a role in the improved discrimination of real versus fake data points.

$$E(Z) = \frac{F(Z)}{F(Z) + 1} \qquad F(Z) = \sum_{z_k \in Z} e^{z_k} \tag{3}$$

3.3 Bot Detection through Analysis of Website Traversal Graphs

The appealing aspect of these graphs is that they are purely based on behavioral features. Website traversal graphs, which depict user navigation on a website with nodes representing sub-pages and edges as navigational links, are further enhanced by weighting edges according to visit frequency and labeling nodes with pertinent attributes. This enriched graphical representation serves as a comprehensive data source for analysis. Our core hypothesis posits that automated web bots exhibit unique navigation patterns, distinct from those of human users, characterized by intensive search tactics and specific page refresh frequencies.

Our approach involves a three-layered methodology. The first layer involves extracting prominent features from the raw data. Following this, a sorting pooling layer restructures these features into a format conducive to deep learning analysis. The final stage employs a one-dimensional Convolutional Neural Network (CNN). Notably, incoming data points, termed as hits, incrementally expand the existing session graph, allowing for dynamic graph development. The classification process thus takes into account the entirety of session hits, rather than isolated events.

Table 1 outlines the specific features utilized in our analysis and their association with the elements of the Website Traversal (WT) graph.

Attribute	**Description**	**Component**
First Hit Pagename	The initial page of a session	Node
Detailed Pagename	The specific page related to a hit	Node, Edge
Previous Pagename	The preceding page in a hit	Node, Edge
Timestamp	The time of page visit	Node Label
Page Type	The category of the visited page	Node Label
Benchmark Label	Associated with a specific hit's benchmark label	Node Label

Table 1. Features Incorporated in WT Graphs

Furthermore, the following metrics are extracted from WT graphs:

Node Degree The count of edges linked to a node.

Node Count The aggregate number of nodes within the graph.

Edge Count The complete count of edges in the graph.

Page Type Distribution This metric calculates the relative frequency of various page types within a set of hits, based on their occurrence and total number of hits for each page type in the set, applicable to a single session or WT Graph.

Session Topics Extracted using the RAKE algorithm [28], this metric comprises keywords from page titles within the WT graph, focusing only on those with a score of 1 or higher. It aims to differentiate between known bots and humans by comparing session topic sets of unidentified clients.

Number of Hits The total visits across all pages in the graph.

Hits per Sub Page The visit count for each subpage.

Degree Centrality A metric indicating a node's centrality in the graph, based on its connected edges.

Betweenness Centrality This measure reflects a node's significance in the graph, determined by the number of shortest paths passing through it among all other nodes.

4 Implementation

4.1 SGAN Architecture and Training

The SGAN used in the evaluation comprises a discriminator and a generator network. The discriminator integrates parallel, aligned layers for discrimination and classification. It employs binary cross-entropy for the discriminator layer and sparse categorical cross-entropy, coupled with a Softmax activation function, for the classifier layer. Both layers utilize the Adam optimization algorithm, with learning rate parameters set at $\alpha = 0.0002$ and $\beta_1 = 0.5$. The architecture features seven shared hidden layers: the first, third, and fifth are dense layers with 100 units each, activated by the Sigmoid function. The second, fourth, and sixth layers are leaky ReLU layers with a fixed scaling parameter of $\alpha = 0.2$. A dropout layer with $p = 0.4$ constitutes the seventh layer. The generator network comprises three layers: an input layer drawing from a 100-dimensional latent vector with 200 units activated by the Sigmoid function, and an output layer with units corresponding to each dataset feature, activated by ReLU. The SGAN employs binary cross-entropy for loss and Adam for optimization, mirroring the discriminator's initialization.

4.2 DGCNN Architecture and Training

For implementing bot detection via WT graphs, we follow the framework presented in the study by [44]. This research introduces a graph-based deep learning mechanism for classification tasks, enabling the encoding of undirected graphs with node features for neural network training.

The DGCNN consists of a GCN and a 1D-CNN. The GCN section includes four graph convolution layers with 32, 32, 32, and 1 hidden units, respectively, followed by a sort pooling layer to format outputs for the 1D-CNN. Activation in each layer is achieved through $tanh$. The GCN parameter k is fixed at 35. The 1D-CNN features two convolutional layers: the first with 16 filters, kernel size, and stride matching the sum of hidden units in all GCN layers; the second with 32 filters, a kernel size of five, and a stride of one. A max pooling layer with a pool size of two separates these layers. Following the second convolutional layer is a dense layer with 128 hidden units activated by ReLU, succeeded by a dropout layer with $p = 0.5$. The final output layer, activated by Sigmoid, uses a single unit. Binary cross-entropy serves as the loss function, and Adam, parameterized with a learning rate of $\alpha = 0.0001$, is the optimization algorithm.

5 Evaluation

The evaluation of the proposed methodologies encompasses both individual and collective assessments. Specifically, this paper aims to elucidate the following research inquiries:

RQ1: What is the efficacy of each module within the detection pipeline in differentiating between bots and human users?

RQ2: Which attributes exert the most significant impact on the outcomes of detection?
RQ3: How does the dimensionality of a WT graph influence the classification accuracy?

5.1 Data Labeling and Ground Truth

Our analysis leverages a dataset from an e-commerce website, which garners approximately 40 million monthly visits. The evaluation is conducted on a subset of around 1.4 million of these visits. To our knowledge, there currently exists no standardized dataset suitable for this type of evaluation.

There are several advantages to utilizing a real-world dataset, such as its realism and the inclusion of unknown types of bots. However, a significant drawback is the lack of ground truth, specifically identifying who is a bot and who is not. We address this issue by using assumptions to provide labeling of the data.

Human Assumption Traffic is labeled as human if it originates from accounts of human users (i.e., not automated test users) within the organization hosting the website. This is based on the assumption that it is highly unlikely for an employee to create a bot to interact with their own company's website, as this would risk potential employment termination for engaging in malicious activities.

Bot Assumption A client is categorized as a bot if their requests originate from an IP address associated with a cloud provider's computational center. While it is acknowledged that some legitimate users may access websites via proxies or VPNs, this is considered a minority scenario.

It is important to note that generating ground truth through assumptions is a practice that has been used previously. However, we adopt a stricter approach than related work. For instance, Botcha's [10] hypothesis that bots do not engage in purchases is increasingly being questioned. Modern examples demonstrate that bots actively purchase items such as graphics cards, concert tickets, and game consoles [7,27].

Table 2 illustrates the quantity of data points labeled under the human and bot assumptions, in conjunction with those identified by the heuristics. It is critical to acknowledge, however, that the heuristics—excluding the human assumption—are incapable of discerning human users. Additionally, our analysis examined potential conflicts between the heuristics and our fundamental assumptions. We discovered that the heuristics erroneously classified 9 instances of human interactions as bot activities. Despite this, the obtained recall rate of 0.9988 was deemed satisfactory for our purposes.

Table 2 presents categorizes individual requests from our dataset based on their identification as either human, bot, or unknown. It is structured to compare the number of hits identified through our initial assumptions (cf. 5.1) with those further refined by the application of our heuristics (cf. 3.1), which are designed to identify more bots but not humans. It shows a refinement in bot detection

Class	Assumption (#Hits)	Enhanced by Heuristics (#Hits)
Bot	51.462	65.018
Human	7.630	7.630
Unknown	723.579	710.023

Table 2. Ground Truth Based on Initial Assumptions and Heuristic Refinement

in our dataset, increasing from 51,462 to 65,018 instances with the application of heuristics, while maintaining consistent human identification and reducing unknown classifications.

5.2 Results

In this section, we present the results of our comparative analysis.

RQ1: Detection Performance We compare the performance of the model relying on technical features like window size (SGAN) and the model relying on behavior only (DGCNN) with another bot detection approach on the same dataset. Botcha was configured with the exact parameters described in its publication [10].

Model	Accuracy	Recall	Precision	F1-Score	AUROC
SGAN	0.9895	0.9875	0.9189	0.9519	0.9886
DGCNN	0.9845	0.9833	0.9791	0.9812	0.9892
Botcha-MAM	0.9364	0.8383	1.0	0.9120	0.9437
Botcha-RAM	0.9952	0.9663	0.9807	0.9735	0.9996

Table 3. Comparison to Related Work (Botcha)

All approaches exhibit strong performance. SGAN and Botcha, which rely on technical (non-behavioral) attributes, might seem more effective. However, the superiority of WT graphs that leverage behavioral features is noteworthy. This aspect is increasingly significant in the dynamic and challenging landscape of bot detection, where bot creators constantly adapt to evade detection mechanisms.

Behavioral features are paramount due to their independence from the underlying automation technology. For instance, while using a real, rendered browser enables bots to mimic human activity and evade detection based on technical, non-behavioral features, achieving this level of mimicry requires sophisticated emulation of human behavior. Bot creators encounter difficulties in replicating the nuanced user behavior unique to each website, which varies with the site's type and structure. Furthermore, programming bots to mimic human-like browsing can significantly hinder their efficiency. For example, adhering to human browsing patterns, such as generating an average of 10 results per minute over

5-minute sessions, compromises the inherent speed and endurance advantages of bots.

In this landscape, while SGAN and DGCNN demonstrate commendable performance, with SGAN slightly ahead in terms of accuracy and AUROC, Botcha-RAM stands out as the most effective model, achieving high scores in accuracy, recall, precision, F1-Score, and AUROC. Botcha-MAM, despite its unmatched precision, shows reduced efficiency in other essential metrics.

RQ2: Technical Feature Importance In our investigation into SGAN's detection capabilities, we aim to identify which features exert the most significant impact. To this end, we employ the Permutation Importance Algorithm [5], a method effective in discerning key features influencing data point classification. Initially, the algorithm calculates a reference score s, assessing the classifier's performance on a particular dataset, using a chosen scoring function.

The process involves iterating over each feature column $d = 1, \ldots, D$, where $D = |features|$. For every column, the algorithm shuffles the data K times, with $k = 1, \ldots, K$, and computes a new score $s_{k,d}$ for the perturbed dataset. It then evaluates how this new score compares with the reference score s. This comparison is crucial as it reveals each feature's effect on classification accuracy. A significant drop in the score due to feature manipulation signals a high importance of that feature, while a negligible change indicates low importance. The importance of a feature is quantified using the equation: $i_d = s - \frac{1}{K} \sum_{k=1}^{K} s_{k,d}$

Applying the SGAN classifier to the test dataset, we determine the feature importances, as detailed subsequently. Notably, the algorithm was configured with a K value of 50, and various scoring functions were employed. Additionally, the application of Onehot encoding to some features accounts for the presence of specific values in feature names.

Feature	**R2-Score**		**Negative MSE**	
	μ_i	σ_i	μ_i	σ_i
post_browser_height	0.542	± 0.008	0.051	± 0.001
post_browser_width	0.287	± 0.010	0.027	± 0.001
post_java_enabled_N	0.082	± 0.003	0.008	± 0
post_java_enabled_Y	0.061	± 0.002	0.006	± 0
user_agent_Other	0.024	± 0.002	0.002	± 0
visit_page_num	0.022	± 0.003	0.002	± 0
visit_num	0.012	± 0.004	0.001	± 0
hourly_visitor	0.010	± 0.001	0.001	± 0
page_type_product	0.005	± 0.001	0	± 0
last_purchase_num	0.004	± 0.001	0	± 0
user_agent_Mozilla/5.0	0.003	± 0.001	0	± 0

Table 4. Feature Importance Scores of the SGAN Classifier for the most important features. MSE: Mean Squared Error. μ: Mean accuracy decrease. σ: Standard deviation.

As shown in Table 4, the most prominent features in bot detection are attributes such as *post_browser_height* and *post_browser_width*. These features, despite their high significance as shown by their R2-scores and Negative MSE values, are relatively easy for bots to falsify. By simply adjusting the browser height and width to mimic those of a typical user, bots can effectively camouflage their non-human nature. This insight highlights the imperative of integrating behavioral characteristics into bot detection mechanisms. Behavioral features explore the intricacies and patterns inherent in human interactions, which are considerably more arduous for bots to emulate. Contrasting with static attributes such as browser dimensions or Java enablement status, behavioral patterns encompass intricate, dynamic, and frequently nuanced human activities. Even if a bot were to mimic these patterns, it would result in a significant loss of efficiency.

RQ3: WT Graph Size Importance For this investigation, we utilized a dataset formed through overlap clustering, assigning 30 percent as test data. The most efficacious DGCNN model, pretrained on 70% of the dataset and employing overlap clustering, was used in this study. It is important to note that the AUROC metric was excluded from use, given its inapplicability in certain graph sizes where only a single class exists.

Nodes	# Graphs	ACC	Recall	Precision	F1-Score
1	26137	0.998	0.981	0.998	0.99
2	17066	0.973	1.0	0.974	0.986
3	3533	1.0	1.0	1.0	1.0
4	371	0.998	0.999	0.999	0.999
5	251	0.998	1.0	0.998	0.999
6	101	0.998	1.0	0.998	0.999
7	526	0.997	1.0	0.997	0.998
8	1579	1.0	1.0	1.0	1.0
9	1175	1.0	1.0	1.0	1.0
10	108	1.0	1.0	1.0	1.0

Table 5. Classification performance depending on WT graph size.

The findings indicate that an increase in graph size correlates with enhanced performance of the model. The data presented in the table reveals that even minimal graphs, comprising one to three nodes, are classified with high accuracy. This underscores the efficacy of WT graphs in encapsulating web client characteristics, thereby facilitating the DGCNN's ability to learn representations, leading to heightened accuracy, precision, and recall. It is noteworthy that this performance is attainable due to the unique feature of WT graphs, which aggregate multiple interactions with the same webpage into a singular node. This characteristic enables the trained DGCNN to identify bots effectively, even in scenarios where they interact with only a single sub-page multiple times. Additionally, the classification's performance augments with the expansion of nodes

in the WT graph, suggesting that the DGCNN's bot detection capabilities are more pronounced as clients engage with a greater number of sub-pages.

5.3 Limitations

There are several limitations to this paper that should be considered. First, we are unable to share the dataset that was used for this research due to data protection reasons. This limitation is also present in many related works, and thus limits our ability to fully benchmark our approach against these studies.

Since we are using a real world dataset we lack an accurate ground truth. We rely instead of the most basic assumptions we can think of (cf. 5.1). However, while we have considered the characteristics and behaviors of both groups in the design of our approach, it is possible that some bots or humans may exhibit behaviors that we have not accounted for. This could potentially impact the accuracy of our method in detecting certain types of bots.

Additionally, it is worth noting that our approach may not be able to detect bots that behave exactly like humans. This however is a common limitation among bot detection approaches. However, a bot that behaves exactly like a human is less effective for the bot operator and makes it more difficult to perform malicious activities.

6 Conclusion

In conclusion, our bot detection framework, applied to a large-scale e-commerce site with a substantial monthly visitor count of 50 million, demonstrates its robustness and effectiveness. The methodology begins with heuristic-based approaches for simple bot detection, and then advances to more sophisticated techniques involving technical features analyzed through a Semi-supervised Generative Adversarial Network (SGAN). Behavioral features, particularly those related to website navigation patterns transformed into a Website Traversal Graph and processed via a Deep Graph Convolutional Neural Network (DGCNN), further strengthen our detection capabilities.

Our assessment, when benchmarked against the Botcha methodology, demonstrates that our approach achieves a level of effectiveness similar to theirs, albeit solely relying on behavioral features. The focus on behavioral analysis not only yields high detection accuracy but also necessitates a greater level of sophistication from bot developers. This requirement for bots to more accurately imitate human behavior leads to a decrease in their effectiveness and operational viability, thus enhancing the efficacy of our bot detection framework.

Future work will explore the development of more reliable measures that require less user interaction up to the point of detection, aiming to enhance the efficiency in identifying bots.

References

1. Udger database for user agents, ip addresses, and other web technologies. https://udger.com. Accessed 2024-04-05.
2. Humanity wastes about 500 years per day on captchas. it's time to end this madness. http://blog.cloudflare.com/introducing-cryptographic-attestation-of-personhood/, 2021. Accessed: 2022-11-21.
3. Commercial image captcha solving service. https://bestcaptchasolver.com/, 2022. Last visited on October 6th.
4. Was ist content scraping? https://www.cloudflare.com/de-de/learning/bots/what-is-content-scraping/, 2022. Last visited on October 6th.
5. A. Altmann et al. Permutation importance: a corrected feature importance measure. *Bioinformatics*, 26(10):1340–1347, 2010.
6. A. Bessi and E. Ferrara. Social bots distort the 2016 us presidential election online discussion. *First Monday*, 21(11-7), 2016.
7. S. Brock. Scalping in ecommerce: ethics and impacts. https://ssrn.com/abstract=3793357, 2021.
8. J. Brownlee. How to implement a semi-supervised gan (sgan) from scratch in keras. https://machinelearningmastery.com/semi-supervised-generative-adversarial-network/, July 24 2019.
9. A. Cabri et al. Online web bot detection using a sequential classification approach. In *IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2018.
10. S. Dhamnani et al. Botcha: Detecting malicious non-human traffic in the wild. https://arxiv.org/abs/2103.01428, 2021.
11. D. Doran and S. Swapna. Web robot detection techniques: overview and limitations. *Data Mining and Knowledge Discovery*, 22:183–210, 2011.
12. D. Goßen et al. Hlisa: Towards a more reliable measurement tool. In *Proceedings of the 21st ACM Internet Measurement Conference*, 2021.
13. N. Heath. Expedia on how one extra data field can cost $12m. https://www.zdnet.com/article/expedia-on-how-one-extra-data-field-can-cost-12m/, October 18 2010.
14. M. I. Hossen and X. Hei. A low-cost attack against the hcaptcha system. In *IEEE Security and Privacy Workshops (SPW)*, pages 422–431, 2021.
15. C. Iliou et al. Web bot detection evasion using generative adversarial networks. In *IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 2021.
16. C. Iliou, T. Kostoulas, T. Tsikrika, V. Katos, S. Vrochidis, and Y. Kompatsiaris. Towards a framework for detecting advanced web bots. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, 2019.
17. H. Jonker, K. Benjamin, and V. Gabry. Fingerprint surface-based detection of web bot detectors. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security*, volume 24. Springer International Publishing, 2019.
18. H. Jonker, B. Krumnow, and G. Vlot. Fingerprint surface-based detection of web bot detectors. In *European Symposium on Research in Computer Security*, pages 586–605. Springer, 2019.

19. Z. Jorgensen and T. Yu. On mouse dynamics as a behavioral biometric for authentication. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 476–482, 2011.
20. X. Li, B. A. Azad, A. Rahmati, and N. Nikiforakis. Good bot, bad bot: Characterizing automated browsing activity. In *IEEE Symposium on Security and Privacy (SP)*, pages 1589–1605. IEEE, 2021.
21. C. Lin and H-C. Hsiao. Need tickets? a case study of bot-enabled ticket scalping. Document reference needed.
22. W. Liu. Introducing recaptcha v3: the new way to stop bots. `https://developers.google.com/search/blog/2018/10/introducing-recaptcha-v3-new-way-to`, 2018. Accessed: 2021-05-20.
23. Y. Luo et al. Botgraph: Web bot detection based on sitemap. `https://arxiv.org/abs/1903.08074`, 2019.
24. Intuition Machines. Stop more bots. start protecting user privacy. `https://www.hcaptcha.com/`, 2018. Accessed: 2021-05-20.
25. M. Martinez and R. Stiefelhagen. Taming the cross entropy loss. In *German Conference on Pattern Recognition*. Springer, Cham, 2018.
26. S. Mohanty. Hand labeling considered harmful. `https://www.oreilly.com/radar/arguments-against-hand-labeling/`, 2021. Last visited on October 6th, 2022.
27. D. Parma. At ticketmaster, scalpers score and fans come last. *JL & Com.*, 38:463, 2019.
28. S. Rose et al. *Automatic keyword extraction from individual documents*, volume 1. Text mining: applications and theory, 2010.
29. D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.
30. T. Salimans et al. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, volume 29, 2016.
31. B. Sayed, I. Traoré, I. Woungang, and M. S. Obaidat. Biometric authentication using mouse gesture dynamics. *IEEE Systems Journal*, 7(2):262–274, 2013.
32. A. See, T. Wingarz, M. Radloff, and M. Fischer. Detecting web bots via mouse dynamics and communication metadata. In *International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*, 2023. Accepted for publication.
33. August See, Leon Fritz, and Mathias Fischer. Polymorphic protocols at the example of mitigating web bots. In *European Symposium on Research in Computer Security*, pages 106–124. Springer, 2022.
34. August See, Adrian Westphal, Cornelius Weber, and Mathias Fischer. Detecting web bots via keystroke dynamics. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 423–436. Springer, 2024.
35. August See, Tatjana Wingarz, Matz Radloff, and Mathias Fischer. Detecting web bots via mouse dynamics and communication metadata. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 73–86. Springer, 2023.
36. C. Shen, Z. Cai, and X. Guan. Continuous authentication for mouse dynamics: A pattern-growth approach. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, 2012.
37. S. Sivakorn, J. Polakis, and A. D. Keromytis. I'm not a human: Breaking the google recaptcha. In *Black Hat*, volume 14, 2016.
38. G. Suchacka, A. Cabri, S. Rovetta, and F. Masulli. Efficient on-the-fly web bot detection. *Knowledge-Based Systems*, 223:107074, 2021.
39. G. Suchacka et al. Efficient on-the-fly web bot detection. *Knowledge-Based Systems*, 223:107074, 2021.

40. T. Tanaka et al. Bot detection model using user agent and user behavior for web log analysis. *Procedia Computer Science*, 176:1621–1625, 2020.
41. A. Wald. Sequential tests of statistical hypotheses. *Ann. Math. Statist.*, 16(2):117–186, 1945.
42. G. Ye et al. Yet another text captcha solver: A generative adversarial network based approach. In *ACM SIGSAC Conference on Computer and Communications Security*, 2018.
43. I. Zeifman. Bot traffic report 2016. `https://www.incapsula.com/blog/bot-traffic-report-2016.html`, 2017. Retrieved: 2017-02-01.
44. M. Zhang et al. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.
45. Y. Zhang et al. Detecting malicious activities with user-agent-based profiles. *International Journal of Network Management*, 25(5):306–319, 2015.

Appendix D

Binary Sight-Seeing: Accelerating Reverse Engineering via Point-of-Interest-Beacons

Abstract

Reverse engineering is still a largely manual and very time-consuming process. To ease this process, beacons in the form of known instructions or code patterns are commonly used to guide reverse engineers in dissecting a binary. However, if done manually, identifying high-quality beacons can be very laborious. This paper introduces a novel method to automatically identify the so-called Points-of-Interests (POIs) in binaries. POIs are instructions that interact with data specified by the analyst known a priori, e.g., via sandbox analysis or expert knowledge. These POIs are then used as beacons to guide analysts to find interesting parts of the binary that interact with the specified data, e.g., the encryption routine. Compared to taint analysis, our approach offers simplicity while delivering a select few, yet high-quality beacons, thereby establishing clear focus points. Based on our proposed method, we implemented two types of prototypes. First, a prototype whose output can be loaded via custom plugins into IDA and Ghidra, i.e., two of the more popular reverse-engineering tools. We show the applicability of our method via the prototype by summarizing the insights of the analysis for the Locky and Wannacry ransomware as one of the potential application domains, i.e., malware reverse engineering. Second, we also introduced a prototype that monitors P2P botnets in a fully automated manner by directly instrumenting the botnet malware without requiring manual reverse-engineering. We demonstrate the effectiveness of our prototype by applying it to the ZeroAccess, Sality, Nugache, and Kelihos botnets and summarize our findings in this paper. Using our approach, we effortlessly found the encryption function in the two analyzed ransomware. For P2P botnets, our monitoring prototype could enumerate the bots in all analyzed botnets, only relying on our POIs

Reference

August See, M. Gehring, M. Fischer, S. Karuppayah. Binary Sight-Seeing: Accelerating Reverse Engineering via Point-of-Interest-Beacons. Annual Computer Security Applications Conference, 2023.

Contribution

The approach to multi-instruction POIs, including the corresponding confidence scores and evaluations (including ransomware), originates from the author of this dissertation. The multi-instruction POI detection methodology was refined from the author's master's thesis. The concept, implementation, and evaluation of single-instruction POIs were contributed equally by the first and second authors. The second author developed the confidence scoring mechanism and conducted the practical evaluation using automated botnet monitoring. The third author proposed the core idea, while the third and fourth co-authors provided feedback and helped to refine the final paper.

Binary Sight-Seeing: Accelerating Reverse Engineering via Point-of-Interest-Beacons

August See*
richard.august.see@uni-hamburg.de
Universität Hamburg
Hamburg, Germany

Maximilian Gehring*
gehring@cs.tu-darmstadt.de
TU Darmstadt
Darmstadt, Germany

Mathias Fischer
mathias.fischer@uni-hamburg.de
Universität Hamburg
Hamburg, Germany

Shankar Karuppayah
kshankar@usm.my
National Advanced IPv6 Centre, Universiti Sains Malaysia
Penang, Malaysia

ABSTRACT

Reverse engineering is still a largely manual and very time-consuming process. To ease this process, beacons in the form of known instructions or code patterns are commonly used to guide reverse engineers in dissecting a binary. However, if done manually, identifying high-quality beacons can be very laborious. This paper introduces a novel method to automatically identify the so-called *Points-of-Interests (POIs)* in binaries. POIs are instructions that interact with data specified by the analyst known a priori, e.g., via sandbox analysis or expert knowledge. These POIs are then used as *beacons* to guide analysts to find interesting parts of the binary that interact with the specified data, e.g., the encryption routine. Compared to taint analysis, our approach offers simplicity while delivering a select few, yet high-quality beacons, thereby establishing clear focus points. Based on our proposed method, we implemented two types of prototypes. First, a prototype whose output can be loaded via custom plugins into IDA and Ghidra, i.e., two of the more popular reverse-engineering tools. We show the applicability of our method via the prototype by summarizing the insights of the analysis for the Locky and Wannacry ransomware as one of the potential application domains, i.e., malware reverse engineering. Second, we also introduced a prototype that monitors P2P botnets in a fully-automated manner by directly instrumenting the botnet malware without requiring manual reverse-engineering. We demonstrate the effectiveness of our prototype by applying it to the *ZeroAccess*, *Sality*, *Nugache*, and *Kelihos* botnets and summarize our findings in this paper. Using our approach, we effortlessly found the encryption function in the two analyzed ransomware. For P2P botnets, our monitoring prototype could enumerate the bots in all analyzed botnets, only relying on our POIs.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; *Malware and its mitigation.*

KEYWORDS

reverse engineering, binary instrumentation, malware, P2P Botnets

ACM Reference Format:
August See, Maximilian Gehring, Mathias Fischer, and Shankar Karuppayah. 2023. Binary Sight-Seeing: Accelerating Reverse Engineering via Point-of-Interest-Beacons. In *Annual Computer Security Applications Conference (ACSAC '23), December 4–8, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627106.3627139

1 INTRODUCTION

The global economic losses caused by cybercrimes were reported to be nearly 1 trillion USD in 2020 [25]. Malware is at the heart of many cybercrime activities. For this reason, security researchers reverse-engineer malware to obtain valuable information about its targets, propagation vectors, and how to mitigate it. However, reverse engineering is a time-consuming and manual effort and requires high-skilled analysts. Given that more than 116 Million new versions of malware have emerged[1] in 2021 alone, any effort towards accelerating or automating the reverse-engineering process would save valuable resources. Hence, various tools and projects that aim to automate parts of the reverse engineering process have emerged recently, e.g., replay of protocols using symbolic execution [23], analysis of data structures [19], or taint tracking [6]. In addition, reverse engineering efforts are also made easier with the availability of comprehensive reverse engineering suites and tools like IDA[2] and Ghidra[3]. However, these advancements have evolved into an arms race between analysts and malware authors. There is no silver bullet for binary analysis. For instance, some of the advanced tools and methods that were proposed can be circumvented via obfuscation techniques, e.g., binary packing to counter static analysis, copying data via a timing side channels to counter taint tracking, or causing path explosions to hinder symbolic execution [1, 2, 5]. Another problem is that many tools are resource intensive and complex to use. Finally, only some tools are interoperable with one another, e.g., software reverse engineering suites. Hence, malware analysts continuously require more advanced tools and techniques to stay abreast with the advancements of malware authors. Our main contribution is a technique designed to accelerate the reverse engineering process. This technique is applicable

*Both authors contributed equally to this research.

ACSAC '23, December 4–8, 2023, Austin, TX, USA

ACM ISBN 979-8-4007-0886-2/23/12…$15.00
https://doi.org/10.1145/3627106.3627139

[1] https://portal.av-atlas.org/malware/statistics
[2] https://www.hex-rays.com/
[3] https://ghidra-sre.org/

to both manual and automated approaches. It relies on data-driven methods, making it more user-friendly and accessible. Compared to taint analysis, our approach offers simplicity while delivering a select few but high-quality beacons, thus offering clear focus points.

- We introduce the concept of Point-of-Interests (POIs) within binaries that can guide analysts as beacons [30] during reverse engineering. POIs are instructions (addresses) within a binary that interact with a set of data items, so-called Data Items of Interest (DIOIs). In other words, our approach allows analysts to input data such as IP addresses or file content (data, referred to as DIOIs), and it retrieves all the addresses that interact with this data (addresses, referred to as POIs).
- We provide a metric *confidence score* which allows us to rate the POIs. It specifies how exclusively a POIs interacts with data from DIOIs. For example, an analyst wants to know where the encryption of files in ransomware takes place. The file content is known and serves as DIOIs. Our approach returns a list of POIs. Among them, POIs that are located in, e.g., `encrypt_file` can be found, but also POIs that are located in functions like `memcopy`. Memcopy is a general-purpose function and is not essential for the analyst. Because the `memcopy` POI interacts (read, write) with data not specified as DIOI, the confidence score is low, and thus the POI can be filtered.
- We implement both approaches and will make them open-source. We further implement plugins for Ghidra and IDA.
- We evaluate our approach. First, in manual reverse engineering, by discovering the encryption routine of two notorious ransomware (Locky, Wannacry). Second, we show that having high-quality POIs can enable the automatic crawling and monitoring of four distinct botnets (ZeroAccess, Sality, Nugache, Kelihos). Given the increasing significance of automated analysis, we focus on the second part. Furthermore, the successful utilization of POIs within an automated framework indicates that discovering and, notably, scoring POIs is effective. Consequently, these findings underscore the applicability of POIs not only in automated scenarios but also in manual settings.

The remainder of this paper is structured as follows. In Section 2, we introduce the necessary reverse engineering background and related work. Section 3 introduces our proposed technique, and Section 4 details the implementation of our prototypes and further discusses the setup for evaluating our prototype in a controlled environment. Then, Section 5 describes our evaluation with the corresponding results, and Section 6 describes the effect an adversary could have on POIs. Finally, Section 7 concludes this paper.

2 RELATED WORK

This section provides an overview of related work to our POI methodology.

Beacons in Reverse Engineering Tools. IDA and Ghidra are two of the most commonly used tools in malware reverse engineering. They provide valuable beacons to aid reverse engineering efforts, e.g., automatic identification of known functions or automatic generation of high-level programming language code. However, an analyst would have to review all beacons exhaustively and hope that something eventually leads to the intended goal of the analyst. In contrast, our proposal empowers the analyst to define their own goals, e.g., discovering ransomware's encryption routine. Based on the specific goals, the corresponding DIOIs can be identified, e.g., the filename of an encrypted file, and used during the analysis of identifying suitable beacons. Hence, the resulting beacons are specific to the goal of the analyst. We created custom plugins to integrate our beacons with the commonly used tools IDA and Ghidra (cf. Appendix F) to strengthen interoperability with other reverse engineering tools.

PANDA [7] is a platform dedicated to dynamic analysis based on QEMU. It stands out because it it allows to record and accurately replay a whole system execution with all running processes. This allows the reproducible analysis and execution of samples. There are many addons avaiable for more specialised analysis. In particular, PANDA cannot be used to resume after a recording or modify the execution of an existing recording[4].

Taint Analysis. Our approach to identifying POIs shares certain similarities to (dynamic) taint analysis [26]. However, they pursue different goals and function differently. Taint tracking is used to identify if and how specific instructions or operations can reach and potentially manipulate data, e.g., sensitive or user-controlled. It helps uncover security risks and vulnerabilities by tracking tainted data flow within a program or system, such as buffer overflows or command injections. It can also be used for analysis or reverse engineering purposes to better understand a system [10, 29]. Initially, there are taint sources and sinks, which refer to addresses within the program or memory. The first step involves identifying these sources and sinks. Subsequently, taint propagation occurs, marking all addresses that interact with the taint sources. If marked instructions write new data, such as in registers, memory, or even status registers, that newly written data becomes tainted. Additionally, instructions that interact with this tainted data are marked again. That is why it is called taint propagation. When execution reaches a taint sink, it is examined whether it currently interacts with tainted data.

Our approach, in comparison, aims to facilitate reverse engineering by finding specific points in the program an analyst is interested in. Furthermore, it focuses solely on the data itself, disregarding its origin. Thus our approach is simpler to use as it does not require the identification of sources and sinks. The major challenge in taint tracking lies in the complex propagation step. Further, many methods hinder or prevent taint analysis from identifying data flows [1, 5]. For instance, tainted data can be written to a file and subsequently read, losing its tainted status. Furthermore, taint analysis often marks a large number of instructions, potentially resulting in the discovery of more POIs, metaphorically speaking. However, marking too many instructions as beacons might diminishes their usefulness as it can be overwhelming to the analyzer.

Ultimately, both techniques operate similarly but have different objectives. Therefore, combining both approaches would be interesting, e.g., using our approach to identify taint sources. The drawbacks of our approach are described in Section 6.

[4] https://github.com/panda-re/panda/blob/master/panda/docs/manual.md

Executable Binary Characteristics. Our approach works by first identifying DIOIs and based on those POIs. This raises the question if and how suitable DIOIs can be obtained. This is more straightforward than it seems at first. A binary generally needs to interact with the host's operating system. It reads and writes files or registery, it spawns threads and processes and sends and receives data from the internet. This is also true for malware. For example, ransomware must read and write files for extortion and botnets must keep contact with the internet to send or receive commands. These interactions are observable through various tools, such as sandboxes [12] or API monitoring[5]. Generally, the interactions themselves can not be hidden by the binary and thus may reveal parts of the binaries core behavior.

Dynamic Analysis of Botnets. BotWatcher [3] is a tool that utilizes dynamic analysis of a botnet malware to infer higher-level behavior. By periodically taking memory snapshots and performing forensic analyses on the snapshots, BotWatcher detects events such as file creations or TCP connections. Based on these events, inference rules are utilized to infer high-level malware behaviors, e.g., malware/payload download. In contrast, our proposal can provide low-level behavioural analysis such as providing direct references to individual instructions and code segments of interest.

Related approaches are found in the field of procotol and input format reverse engineering. While many approaches deal with this problem, only few deal with encrypted data [8]. ReFormat [31] is a system for protocol reverse engineering that can even operate on encrypted messages. It relies on the amount of bitwise operations to identify potential cryptographic functions. Further, it uses taint tracking to mark buffers tainted by the received message. Our approach is similar in that regard, that it can find the buffers holding the encrypted or unencrypted data, if some data is known. However, it does not rely on heuristics to do so. Protocol extraction approaches may have difficulty with new approaches that obfuscate the protocol [9, 27]. While our approach and format reverse engineering have some techniques in common to achieve their respective goals, the actual goals are different. The goal of format reverse engineering approaches is to determine the structure of messages or structured inputs. Our approach, in contrast, is more generic and identifies and assesses code locations within a binary program (POIs) that interact with the data being searched for (DIOIs).

Automated P2P Botnet Monitoring. The core idea of our PinPuppet prototype (see Section 4.3) revolves around reusing the P2P communication module of the malware to replay and respond to membership maintenance (MM)-messages. The idea of reusing existing functionalities of a malware was already proposed in Inspector Gadget[16] by *Kolbitsch et al.* In their work, they identify and extract specific functionalities from an executable, e.g., domain generation algorithm, as a so-called gadget, to speed up their analysis. The main difference between our approach and this particular work is that their approach still requires a human analyst to manually spot and extract the gadgets. Furthermore, Inspector Gadget extracts code, whereas our approach extracts data.

[5]https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

3 POIS AS BEACONS

POIs are instructions that are known to interact with data from a prespecified set of data, the so-called DIOIs. POIs can be represented as tuples *(data, address)*. In this context, interact can, for example, mean reading or writing data from and to memory. The core idea is to use these identified instruction POIs as beacons in reverse engineering. In the identification process for POIs, the DIOIs serve as a ground truth of relevant data.

Listing 1: Example Assembly Code

```
1  // eax, address of the message buffer
2  mov [eax], 0xcafe
3  // further message construction
4  push eax
5  // send message buffer to recipient
6  call send_message
```

For example, consider having a binary that sends several types of messages over the network. The first 4 bytes of the message determine the message type. Let us further assume an analyst is interested to investigate the purpose of a particular message type that is identified by `0xcafecafe` in its payload. The corresponding assembly code could look like the one shown in Listing 1. Using the message type information as the a priori DIOIs, the analyst could focus on the region of the assembly code that is associated to this instruction, i.e., the POI. In the listing, line 2 would, for example, be identified as a POI as it constructs the relevant part of the message, i.e., writes `0xcafecafe` to memory. However, the process of selecting POI candidates is not trivial as several factors may affect the suitability of a candidate. In this example, the DIOIs were specified based on expert knowledge, however, the set of DIOIs can also be constructed based on sandbox analysis results (see Section 2). For example, IP addresses occurring in a packet capture could be used as DIOIs to identify POIs processing IP addresses.

In the following, we provide a high level overview of our methodology to identify and use POIs. Then, we elaborate the details of some of the intermediate steps and finally present two strategies for identifying high-quality POIs.

3.1 Methodology

Our methodology consists of five steps as illustrated in Figure 1. In Step ①, a software sample needs to be obtained. This is relatively straight forward for benign software. For malware, one could use honeypots, malware repositories, or obtain a sample from an infected machine. Next in step ②, by instrumenting the software using dynamic binary instrumentation (DBI) (see Section 2) tools, all interactions of the sample with the Operating System (OS) to access registers and memory regions are traced and logged as a list of *(data, address, access_type)* tuples (here onwards referred to as trace data). In Step ③, the set of DIOIs (D) accessed by the software, e.g., registry modifications and network communication, is identified via sandbox analysis reports or expert knowledge. As D reflects the information that the software processes or interacts, elements DIOI ($d \in D$), e.g., IP addresses or registry filenames, are also expected to appear in the trace data from Step ②. Hence, Step ④ uses the trace data and D to identify potential POI candidates. Finally, Step ⑤ filters out low-quality POI candidates. These are

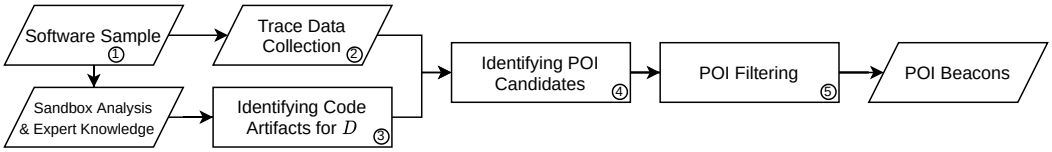


Figure 1: The POI discovery process.

candidates that, instead of mostly processing data in D or related to D, also process other data. By filtering them, we prevent them from adversely affecting the final outcome of using the POIs as beacons. Note that our methodology shares some similarities with taint analysis [26]. We highlight and discuss the differences in Section 6.

3.2 Identifying Code Artifacts

The process of identifying the DIOIs (Step ③) that are unique to the investigated sample is relatively straight forward as every binary needs to interact with the operation system (Section 2). It can be done as follows:

- An analyst can leverage sandboxing environments, e.g., a *Cuckoo Sandbox* [12], to generate reports of changes made by the software during the analysis. From those reports, data that is unique to the software can be identified and used as DIOIs. Depending on the data type, alternative representations and encodings have to be considered. For example, for an IP addresses in binary form, one may include the most significant byte (MSB) and least significant byte (LSB) equivalent representation. For floating point numbers, single and double precision representations have to be added.
- As alternative to sandboxes, an analyst can leverage tools for monitoring OS-APIs, e.g., *API Monitor*[6] or *strace*[7], to identify unique values as potential inputs into D.
- Additional expert knowledge, e.g., from manual analysis or published research, can also be considered as DIOIs.

The choice of information that is considered as DIOIs is driven by the main goals of the analyst in studying the software. For instance, if understanding the network communications of a malware is of interest, the set of outgoing IP addresses and ports used by the malware, e.g., for Command and Control (C2) communication, can be a suitable choice. However, not every information can be used as a DIOI. Consider the following scenario where a sandbox analysis reported an outgoing connection from the software to a remote host with the destination IP address 10.20.30.40 and port 80: only the IP address would be suitable to be considered as DIOIs. The port information is not suitable as the (decimal) value for port 80 can be easily used by other components within the malware, e.g., as an internal counter or as the ASCII representation of the character `'P'`. A poor quality POI would be the result. In contrast, the likelihood of observing a random data access that matches the exact binary representation of the IP address 10.20.30.40 is rather low.

[6]http://www.rohitab.com/apimonitor
[7]https://linux.die.net/man/1/strace

3.3 Identifying POI Candidates

POIs are defined as instruction addresses that interact with any DIOI, i.e., by reading or writing any registers or memory. However, the process of identifying POIs instructions (Step ④) based on the trace data ② and the set of DIOIs③ is not trivial. The main challenge is due to the different methods of how data or values are processed by an OS. Depending on the CPU architecture and available instruction extensions, the maximum length of data associated with an instruction varies. Hence, the elements within D may not necessarily always fit a single recorded instruction execution. For instance, an integer value can be processed in a single instruction, but for example strings require multiple instruction executions for processing. To address this issue, we introduce two search strategies, the first one to identify *single instruction POI* and the second one to identify *multiple instruction POI*.

3.3.1 Identifying single instruction POIs. We define three possible ways for the software to interact with a DIOI via a single instruction: (i) reading from memory, e.g., a `mov eax, [esp]` instruction, (ii) writing to memory, e.g., a `push eax` instruction, and (iii) having DIOIs loaded in a register while being executed (note that this does not need to be a memory read, e.g. by increasing the value in the register).

This search strategy iterates each entry in the execution trace and compares the values of the registers, memory reads, and memory writes for each DIOI. Whenever the data access of a single instruction in the execution trace matches any $d \in D$, the corresponding instruction's address is marked as a potential POI candidate.

3.3.2 Identifying multiple instruction POIs. Searching for a multiple instruction POI is more challenging because the *order* of how a DIOI $d \in D$ is accessed is unknown. Hence, d could be accessed in the memory using a sequential, reverse, or randomized pattern. As it is impossible to know upfront the total number of instructions nor the exact access pattern for a DIOI, the corresponding memory region needs to be searched and matched exhaustively against each DIOI per memory-write operation. Moreover, the memory allocation information provided by the OS can only report the total allocation to an application or process. It does not give information on the exact, i.e., current, utilization of the allocated memory. Therefore, we need to track the current utilization of the allocated memory by the binaries process, i.e., the actual memory writes recorded in the execution trace from Step ②. For this purpose, we leverage lookup tables in our internal tracking mechanism.

Lookup Tables. To track the utilization of each memory block we introduce two custom lookup tables: the *memory map* and the *identifier map*. The memory map dynamically assigns new *memory regions* and continuously tracks them to map each address of a

contiguous block of written memory to the corresponding region. It also tracks which instruction reads and writes which value. The *identifier map* is then used to keep track of the size of each particular tracked memory region. Hence, for each change in the *memory map*, an updated size of the affected memory region is also tracked and reflected via the *identifier map*. The ability to track the current utilization size of a particular memory region enables a targeted DIOI search as described in the following.

Memory Search. For each memory write-operation in the recorded trace data, we only need to search within the exact memory region that we continuously track using our aforementioned lookup tables. Hence, we introduced a custom memory search algorithm as illustrated in Algorithm 1 to leverage the tracking information of the lookup tables to search for POIs that accesses any DIOI. The algorithm first obtains the information of the memory region for the currently accessed memory address from the lookup tables (Lines $1-3$ and $5-6$). Next, the algorithm checks if the size of a DIOI ($d \in D$) fits the size of the currently considered memory region (Line 7). If the size fits, the contents within the memory region are compared to d (Line $8-9$). Upon a successful match, the algorithm stores the address where d was found and continues with the next $d \in D$. Since the memory map tracks which instructions read and wrote which values, all instructions that accessed the searched DIOI d are known. Those instruction are marked as POI candidates.

Algorithm 1: Searching for DIOIs d in memory.

Input: lastAddr, D
Result: The start addresses of the searched DIOIs D if it is found. Otherwise, the start address is -1.

```
1  // A region ID is the start address of a memory region
2  regionID ← memoryMap[lastAddr]
3  regionSize ← identifierMap[regionID]
4  for d in D do
5      startAddr ← max(regionID, lastAddr- d.size)
6      endAddr ← min(regionID + regionSize, lastAddr + d.size)
7      if d.size ≤ (endAddr-startAddr) then
8          buffer ← readMemory(startAddr, endAddr)
9          pos ← buffer.find(d)
10         results [d] ← pos
11     results [d] ← −1
12 return results;
```

3.4 POIs Filtering

The POI candidates identified in Step ④ consists of all instruction addresses that interacted with any DIOIs within the trace data. Thus, the choice of DIOIs can greatly affect the quality of identified POIs. When one, for example, uses the integer 80 as a DIOI, this could refer to a port in the context of networking, but also occur in the context of a loop as a counter. Thus, it is possible that some of the identified POIs are false positives, i.e., misleading beacons. Alternatively, it is also possible that a DIOI is processed in a general purpose function, e.g., the `memcpy`. Even though this does not reveal any code specific to the DIOI, it would still appear as a POI. Of course, such cases could be easily identified and skipped by expert analysts, the performance of automation tools that rely upon these POIs can be adversely affected (see Section 4.3 for a discussion of such an automation tool).

As the quality of a POI cannot be determined upfront without reverse engineering, we introduce a metric called **POI confidence score** to estimate the quality of a POI. It assesses the frequency of an address associated to a particular POI being used to access *only* some set of DIOIs. This metric assigns a score between 0.0 and 1.0 to each POI as an estimation of the quality of the POI. A score of 1.0 indicates that the POI is used exclusively by the code for accessing only DIOIs. A score of < 1.0 indicates that some other contents ($d' \notin D$) are also accessed by the instruction at the POI. Based on the resulting confidence scores, a threshold-based filter can then be applied to select only high-quality POIs as beacons. Next, we elaborate on the required adaptations of the proposed metric for the two search strategies outlined in Section 3.3

3.4.1 Filtering of single instruction POIs. First, function $C(x, L)$ is the operation that returns the number of elements in L that are equal to x. L is a list of elements ($L = e_1, \ldots, e_m$). The POI confidence score metric for a single instruction POIs p is then formulated as:

$$score_S(p) := \sum_{d \in D} \frac{C(d, \text{trace}(p))}{|\text{trace}(p)|} \tag{1}$$

where $\text{trace}(p)$ is the list of data processed by the instruction at POI p. Hence, for each POI we calculate the number of interacted data that is in D, divided by the total number of accessed data.

3.4.2 Filtering of multiple instruction POIs. Meanwhile, the confidence score metric for multiple instruction POIs leverages the tracking information of the lookup tables introduced in Section 3.3. In more detail, let $P = s_1, \ldots s_n$ be a list of lists of instruction addresses. Each s is a list of instruction addresses that wrote the bytes for a DIOI d, where $s = i_1, \ldots, i_m$ and every i is the address of an instruction that wrote some byte of a DIOI. In addition, let p be a POI, $\text{addr}(p)$ be the address corresponding to p, and $\#p$ the number of bytes this POI wrote during the runtime of the process. Then, the confidence score for a POI p is calculated using

$$score_C(p) = \sum_{s \in P} \frac{C(\text{addr}(p), s)}{\#p} \tag{2}$$

Hence, for each POI, we calculate the number of written DIOI bytes, divided by the total number of written bytes. Note that only bytes written that result in a complete DIOI match are considered. The confidence score for a read-memory POIs is also calculated analogously.

4 APPLICATIONS OF POI

Our methodology to identify POIs is mainly envisioned to be used to identify high-quality beacons to accelerate reverse-engineering efforts by improving the usability. However, POI-based beacons can also be integrated into automated tools for more advanced use-cases. In the following, we first describe the implementation details for a generic POI identification tool. Then, to demonstrate the advantages offered by the POIs, we describe a more complex tool: our *PinPuppet*

prototype that attempts to leverage found beacons for automated P2P botnet monitoring without the need for reverse engineering.

4.1 Generic POI Discovery Tool

Our generic POI discovery tool implements our proposal according to the stages depicted in Figure 1. First, an executable binary is needed and a goal or idea of what needs to be achieved, e.g., finding instructions that process a file that is read or are those that are responsible for sending some data. In addition, some DIOIs that are known to be interacted with by the malware sample should also be identified a priori through means of a malware sandbox analysis report or expert knowledge (see Section 3.2). Next, to obtain the execution trace of the binary, we instrument the malware sample using Intel Pin [20] (in the following only Pin) as our choice of the DBI framework. Another script implements the logic to search for single instruction and multiple instruction POIs (see Section 3.3) on the recorded execution trace against the set of data D, POI candidates are then identified. Afterwards, the confidence score filtering is further applied as described in Section 3.4 to omit poor-quality POIs. Finally, the remaining high-quality POIs are loaded into IDA and Ghidra using custom plugins that we developed (see Appendix F). Using these plugins, analysts can focus on the relevant beacons in dissecting the binary.

4.2 Ransomware Analysis

Ransomeware encrypts user files for extortion. One goal of reverse engineering can be to find the function responsible for the encryption. Encryption involves reading a file and writing the encrypted file. The contents of the encrypted and unencrypted file can be used as DIOIs to finds the encryption function. In detail, this can be done as follows. The ransomeware should be executed on a VM with snapshot capabiliites and before executing the malware it is advised to create a new snapshot. When the ransomeware is done encrypting some files, save them to the host machine also back up the associated original files by rolling back the snapshot. Now the file contents can be used as DIOIs. Using the encrypted file content POIs can be found that access this. The encrypted file content is assumed to be in memory for the first time immediately after the encryption process. Thus, the first POI found is likely to be within the encryption function. This can be supplemented by using the cleartext file content to find matching POIs. Here, the last found POI may be of interest, since after encrypting the file content is encrypted, the cleartext file contents no longer have any use to the ransomware and are likely deleted.

4.3 PinPuppet: Botnet Monitoring

To demonstrate the applicability of POIs on advanced use cases, we developed an automated P2P botnet monitoring prototype called PinPuppet. PinPuppet uses the POI mechanism to instrument a malware to automatically perform botnet monitoring. The main goal of PinPuppet is to crawl a P2P botnet without presuming detailed knowledge on the botnet communication protocol.

4.3.1 P2P Botnet Characteristics. P2P botnets do not need to rely on dedicated C2 servers to enable infected machines to communicate with their botmasters [13]. Each bot in the P2P botnet maintains a peer list that contains information, e.g., IP addresses and ports,

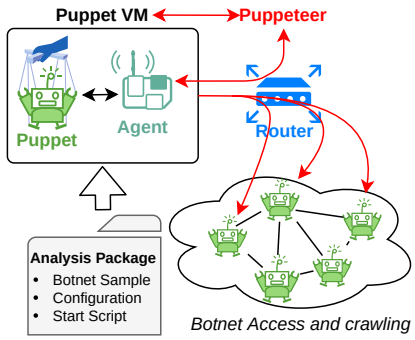


Figure 2: System overview for PinPuppet.

necessary to contact other bots in the botnet. This list is updated via a MM mechanism that periodically probes peers within the peer list and exchanges knowledge on other peers (via *getL* and *retL* messages) with them to ensure a connected botnet overlay. Due to the open nature of P2P botnets, the MM mechanism can be exploited to monitor botnets using custom tools such as *crawlers* and *sensors* [13].

4.3.2 System Overview. PinPuppet consists of four dedicated components as depicted in Figure 2 that interact with each other to perform an automated crawling of P2P botnets.

The **Analysis Package** component bundles all information that is required for crawling a P2P botnet. It consists of the malware sample, the samples bootstrap list $P_{\text{bootstrap}}$, a crawling configuration file, and a script that defines how the malware should be executed while being instrumented by Pin. The two main components that are running on the Virtual Machine (VM) are the **Puppet** and **Agent**. The Agent is our foothold in the Puppet VM. It allows uploading analysis packages, running them, and downloading the results. Meanwhile, the Puppet is the malware sample instrumented by Pin to trace instructions. The **Router** is used for routing the traffic between the Puppet VM, the Puppeteer, and the botnet itself. Finally, the **Puppeteer** is the brain of PinPuppet. It is able to set up and reconfigure the Router and controls the Puppet VM (e.g., creating snapshots, starting, and stopping it). It also communicates with the Agent via the network and sends commands to it. Data collected by the Puppet, is downloaded by the Puppeteer using the Agent for further processing.

4.3.3 Methodology. The automated crawling by PinPuppet is a four-step process. These steps incorporate the processes described in Figure 1. In the first step, trace data is collected. Next, DIOIs are identified from the information gathered in the first step, D is constructed, and POI candidates are identified. In the third step, these POI candidates are filtered based on a confidence score threshold. Finally, the identified POIs are used for crawling.

POI Candidate Identification. In this step, the Puppeteer performs the POI candidate identification (see Section 3.3 and Figure 1) for identifying IP- and port-POIs, i.e., POIs corresponding to instructions frequently associated with IPs or ports. The identification process uses the data set D_i for IP-POIs and D_p for port-POIs. Both sets are constructed based on the IPs and ports in $P_{\text{bootstrap}} \cup P_{\text{socket}}$.

For identifying single instruction-POIs, the 4-byte binary LSB-first and MSB-first representations of the IPs and ports are used. For identifying Contigous-POIs, the ASCII representations of the IPs are used.

The intuition behind using $P_{\text{bootstrap}} \cup P_{\text{socket}}$ as the basis for ground truth of our data sets is that it approximates all peers that the puppet can know. The true set of known peers is $P_{\text{bootstrap}} \cup P_{\text{socket}} \cup P_{\text{shared}}$ where P_{shared} are all peers that were shared with the Puppet in the trace data collection step. Of course, we do not know which peers were shared with the Puppet, however, as bots who request new shared peers oftentimes eventually contact these shared peers, there is an overlap between P_{socket} and P_{shared}. Thus the difference between $P_{\text{bootstrap}} \cup P_{\text{socket}} \cup P_{\text{shared}}$ and $P_{\text{bootstrap}} \cup P_{\text{socket}}$ is small.

POI Filtering. In this step, the Puppeteer performs POI filtering (Section 3.3) for the identified IP-POIs. First, the Puppeteer calculates the confidence score of each IP-POI and then filters them based on the confidence score threshold. To filter port-POIs, we first need to find the mapping between IP- and port-POIs. Hence, for each identified POI, we extract all valid representation of IPs and ports that are found in the trace data. Then, we search exhaustively for valid mappings using known good IP-Port combinations (see Appendix B for pseudocode).

Crawling Primitive. The crawling primitive used in PinPuppet mainly aims to: (i) contact a specified "crawl peer" p_c using a *getL*, and (ii) extract peers shared in the corresponding *retL* message – information which is otherwise not immediately visible to an analyst. For this purpose, the router is configured to redirect all outgoing traffic destined to p_o, the first peer contacted after rolling back, i.e., restoring VM snapshot, the Puppet to the intended p_c. For all replies sent by p_c, the Router changes the source to p_o. Therefore, the Puppet will assume it is talking to p_o even though it is communicating with p_c. To prevent peers $p_x \neq p_c$ from interfering with this step, all traffic except between the Puppet and p_c is blocked by the router. This ensures that the extracted peers are only based on the immediate *getL-retL* exchange between the Puppet and p_c. The sample is now left running for a short time period T_{crawl} (configured using the Analysis Package) after which the trace data is downloaded. Then, using the IP- and port-POIs, the Puppeteer extracts IPs and ports from the trace data. For every IP- and port-POI which are mapped together in the POI mapping, the extracted IPs and ports are merged to obtain the endpoints of the shared peers. The resulting address list is returned of the crawling primitive.

5 EVALUATION

Our utilization of POI beacons accelerates reverse engineering by simplifying and enhancing the process, resulting in improved usability and efficiency. However, we focus on malware in this paper to show that our approach also works with software that generally takes countermeasures against analysis.

We evaluated our generic POI discovery tool (see Section 4.1) using two of the recently infamous ransomewares: Locky [18] and Wannacry [21]. We proceeded as described in Section 4.2. By using the contents of some of the encrypted files as DIOIs, we were able to find the instructions of both malware which were responsible for encrypting files. We also verified this using manual reverse engineering (see Appendix F). Nevertheless, an extensive quantitative evaluation would not be feasible for a larger set of general malware due to the lack of ground truth, i.e., without reverse engineering every malware. In contrast, for P2P botnets we are able to construct a controlled evaluation environment providing us with the required ground truth. Hence, we decided to rely upon the evaluation of PinPuppet which leverages POI beacons to conduct automatic P2P botnet monitoring to demonstrate the feasibility and potential of our approach. As PinPuppet utilizes identified POIs to extract shared peers from a P2P bot, we are thus able to analyze whether our POI identification methodology works in identifying relevant POIs. In the following, we first describe the experimental setup and then present our findings.

5.1 Experimental Setup

As the main goal is to perform automated P2P botnet monitoring, it is critical to have an environment that provides ground truth information as well as the ability to repeat the experiments. Hence, evaluation with real-world botnets would not be possible due to the presence of churn and lack of a ground truth. Therefore, we decided to deploy an isolated network testbed to bootstrap the P2P botnets into forming an overlay within this network and evaluate PinPuppet on them. In the following, we first elaborate our methodology in bootstraping the botnets. Then, we discuss the research questions that we are interested to answer in our evaluation.

5.1.1 Evaluation Environment. We picked four well-known real-world unstructured P2P botnets to evaluate PinPuppet's performance: 1) ZeroAccess [13], 2) Sality [11], 3) Nugache [28], and 4) Kelihos [15]. These four botnets were selected as they have wrecked havoc in their heyday and Sality is even still active till date. we bootstrapped botnets – using real-world, unmodified malware samples – locally, i.e., isolated from the Internet. For the identification of POIs, this has no negative impact because the bot will not realize that it is only talking to a local botnet. Thus it behaves the same and uses the same instructions as if it were talking to a real botnet. We performed this bootstrapping process for four Windows botnets: ZeroAccess [13, 22], Sality [13], Nugache [28] , and Kelihos [14]. The hashsums of the corresponding samples used, are available in Appendix A.

Our approach for making the botnet available in the testbed setting is similar to the work of *Calvet et al.* for creating an in-the-lab Waledac botnet [4]. In their approach, they exploited a vulnerability found in the botnet's communication protocol to reset and manipulate the neighbor list of their emulated bots to establish connections with other emulated bots. Hence, their approach is not generalizable to other botnets unless a similar vulnerability exists in other botnets. In contrast, our generic approach leverages regular MM-mechanism features of P2P botnets to establish connectivity with other bots in the testbed.

Due to limited space constraints, we only briefly explain the concepts of our 'local botnet'[8]. We bootstrap our local botnet with $n + m$ peers (denoted by the set P_{local}) as following:

[8] A complete setup and code for the setup will be made available in a public repository for the camera ready version

(1) Create n-VMs with IPs from the sample's bootstrap list and execute the sample on the VMs. These peers are called the "local bootstrap peers".
(2) Create m-VMs with arbitrary public IP addresses. These peers will contact the local bootstrap peers, i.e., the initial n-VMs, and join the botnet overlay. Note that this is a normal behavior similar to a newly infected machine that inherits the neighbor list of the infecting node as its own bootstrap list.

For all botnets except Nugache, we utilized $n = 1$ local bootstrap peers and $m = 40$ other peers. Due to the nature of the Nugache's MM mechanism which has a relatively small-sized neighbor list, we adapted to $n = 8$ and $m = 40$. Further information on the configuration of the four local botnets can be found in Appendix C and our public repository. From this point onwards, we use live snapshots to save the state of all the VMs within the local botnet. This allows us to restore the state later on when running PinPuppet (see Section 4.3.3).

5.1.2 Research Questions. In the following, we briefly introduce our four research questions and elaborate the corresponding experimental setup.

RQ1 – Slowdown: To what extent is the malware affected in terms of a delay in execution of instructions due to the overhead from the PinPuppet instrumentation? Instrumenting a binary means adding overhead in terms of additional instructions that needs to be executed during runtime, in our case Intel pin and our PinPuppet pintool. If the instrumentation introduces a significant delay to the regular execution of malware instructions, a sample's behavior may be altered, for instance, due to transmission timeouts. Therefore, prior to exploring the utility of POIs, we conduct an investigation to determine the feasibility, taking into account the associated overhead. To measure the impact of the introduced overhead across the different samples, we measure the time each sample sends out its first (t_0), 20^{th} (t_1), and 40^{th} (t_2) message during execution for both scenarios, i.e., with and without instrumentation. For the analysis we then compare the values $T_0 := t_0 - 0$, $T_1 := t_1 - t_0$, and $T_2 := t_2 - t_1$. Each sample was evaluated five times for each measurement.

RQ2 – Identification: How effective is our identification method in identifying high-quality POIs? Our crawling primitive heavily depends upon the availability of high-quality POIs, i.e., high confidence score (Section 3.4). To answer this research question, we analyze the confidence scores of all POIs and their distribution according to bins that are called confidence-classes. Confidence class 0 corresponds to the interval $[0.0; 0.1]$, class 1 to the interval $(0.1; 0.2]$, etc..

RQ3 – POI confidence score Quality: How well does the confidence score predict the quality of extracted IPs for IP-POIs? The POI confidence score metric needs to be a good estimator of the quality of an IP-POIs. On the one hand, if the confidence score were overestimating the quality, we would extract spurious IPs. On the other hand, if the confidence score were underestimating the quality, we would disregard good POIs and potentially miss out extracting some IPs.

To quantify the quality of the results extracted by POIs, we further introduce the *correctness* metric. To calculate this metric, we refer to the set of all peers extracted by a POI as $P_{\text{extracted}}$. The sets $P_{\text{bootstrap}}$ and P_{local} refer to the set of all bootstrap peers and all peers in the local botnet, respectively. Note that in our local botnet, $P_{\text{bootstrap}} \cap P_{\text{local}} \neq \emptyset$. A peer is considered extracted correctly if and only if it is in the set $P_{\text{bootstrap}} \cup P_{\text{local}}$. The correctness for a POI is then defined as:

$$correctness = \frac{|P_{\text{extracted}} \cap (P_{\text{bootstrap}} \cup P_{\text{local}})|}{|P_{\text{extracted}}|} \tag{3}$$

RQ4 – Monitoring: How effective is PinPuppet in leveraging POIs to monitor the local botnet? Verification of the ability of PinPuppet to crawl a botnet would also indirectly validate the methodology for POI identification and prove their usefulness as beacons. For PinPuppet, among others, the crawling primitive extracts (new) peers when it receives shared peers in the response messages for the *getL* messages, i.e., a *retL* message, from bots within the local botnet. With this information, PinPuppet can identify new peers and subsequently request or crawl all newly discovered peers.

For the evaluation, we used the crawling primitive to crawl 100 random peers from P_{local} (peers can be crawled multiple times). Then, we classified all extracted peers $P_{\text{extracted}}$ according to CORRECT, WRONG, and BOOTSTRAP. Extracted peers are classified as CORRECT if they were part of the local botnet CORRECT := $P_{\text{extracted}} \cap P_{\text{local}}$. Meanwhile, if the extracted peers are not part of the local botnet but was present in the bootstrap list, they are considered BOOTSTRAP := $P_{\text{extracted}} \cap (P_{\text{bootstrap}} - P_{\text{local}})$. Finally, if the extracted peers are not part of the local botnet nor the bootstrap list, the extracted peers are classified as WRONG := $P_{\text{extracted}} - P_{\text{bootstrap}} - P_{\text{local}}$.

5.2 Results

In the following, we present our evaluation results based on the research questions outlined in Section 5.1.2.

RQ1 – Slowdown. From our analysis, the additional delays introduced for T_1 and T_2 with and without PinPuppet did not have any adverse effects towards the MM-related communication activities of the malware. The complete result of our analysis is presented in Appendix E. The largest (average) difference between instrumented and uninstrumented execution, i.e, without intel pin, was recorded for Kelihos with a 6.5% (12.95 seconds) increase for T_1. This timing delay for T_1 is rather expected and evident for all botnets, as they mostly wait after t_0, e.g., for network responses, incoming requests, or by idling. In addition, our filtering mechanisms (see Section 4.3.3) further limit the number of times an instruction is traced and thus reduces the additional overhead introduced in the analysis. Throughout the evaluation, we have used a limit of 1000, i.e., each instruction was traced at most 1000 times. This value is a good starting point as the bootstrap lists of our botnets had less than 1000 peers, thus we expect relevant instructions to be executed less than 1000 times in a single MM-cycle. However, the limit value is not that important as long as it is large enough to capture enough instruction executions such that it paints an accurate picture of the data processed by the instructions.

RQ2 – POI confidence score Effectiveness. The distribution of POIs identified by PinPuppet for each botnet in the evaluation set is depicted in Figure 3. Due to the fact that the confidence score is

not used for filtering the port-POIs (cf. Section 4.3.3), the plots only contain the data for IP-POIs.

Based on our analysis single instruction-POIs, i.e., the types "Register" and "Memory", were identified for all botnets. Moreover, there are at least a few POIs that have a high confidence score, i.e., a confidence class of eight or nine. In addition, our analysis revealed that multiple instruction-POIs were identified only for Kelihos and Nugache. Upon further investigation using the identified beacons for Nugache, we discovered a code region that is responsible to store peers in the registry using ASCII representation. As for the two Kelihos' multiple instruction POIs with confidence class of 0, the were caused by the access of instruction to the *memcpy* function.

Compared to single instruction-POIs, very few multiple instruction-POIs with a high confidence score were found. The concept and definition of confidence scores are not robust for multiple instruction-POIs compared to single instruction-POIs (cf. Section 4.3.3). Since PinPuppet heavily depends on high-quality POIs to automatically crawl the P2P botnets, multiple instruction-POIs are not considered to avoid introducing poor quality POIs which would adversely affect the effectiveness of PinPuppet. Regardless, multiple instruction-POIs are still very useful in a generic setting as it is the only option to discover long DIOIs, i.e., fitting more than a single instruction (cf. Section 3.4).

RQ3 – Identification. The detailed analysis of the data for all botnets and POIs is presented as a scatter plot in Figure 4. Note that markers are merged and their size are increased when there are multiple POIs having the identical confidence score and *correctness*. The red-dashed line is the identity line ($x = y$) and the green-dotted vertical line indicates our confidence score threshold of 0.8 which was determined through a parameter study (see Appendix D). The threshold is set < 1.0 as it is not always possible to identify all peers within the local botnet via crawling.

The correctness, as defined in Section 5.1.2, can only be calculated when P_{local} is known. With a real-world botnet P_{local} is often unknown, thus the correctness cannot be calculated. The goal for the POI confidence score is to estimate the correctness without overestimating it. For example, if a POI with a confidence score of 0.9 has a correctness of 0.2, this would mean that a lot of spurious peers were extracted. For our analysis in Figure 4, this means that the POIs should be ideally above the red line. This is mostly the case with a few exceptions that are below the line. Based on our filtering based on the confidence score threshold, i.e., ≥ 0.8, all the POIs are on or above the red line, i.e., not overestimated. Finally, as explained in Section 4.3.3, underestimation is also possible as D may not necessarily be complete. In our analysis, these are the markers way above the red line.

RQ4 – Monitoring. For each botnet, we compared the classification results with and without applying the confidence score threshold as depicted in Table 1. The results are split up between "Without confidence score threshold," i.e., by using all identified POIs, and "With confidence score threshold", i.e., removing results from POIs with a confidence score below 0.8. In addition, we also show the average confidence score used by the POIs to extract the results of each category. For simplicity, the term "correct" is used in the following if a peer falls into the CORRECT category. This applies analogously to wrong peers and bootstrap peers.

First, we observed that the confidence score threshold helps to effectively decrease |WRONG|. As the size of WRONG can adversely affect the crawler's performance, e.g., random data interpreted as IP addresses, we further investigated the nature of the peers in WRONG after applying the confidence score threshold. We compared the sets WRONG from each of the 100 crawl iterations with each other (from here on, WRONG_i refers to the set WRONG obtained in the i-th iteration) by calculating the average Jaccard index for each pair $(\text{WRONG}_i, \text{WRONG}_j)$ where $i < j$. For Nugache and Kelihos, the resulting similarity scores are 1.0, i.e., the set WRONG is consistent across all iterations and thus would become negligible with an increasing number of iterations (repeating peers can be ignored when crawling). We speculate that the large size of $|\text{WRONG}| = 32$ for Kelihos stems from an undiscovered peer list, which are contacted only in certain circumstances. This speculation is supported by the fact that WRONG always contain the same 32 peers and 26 IPs are reported to be associated with Kelihos [9].

For Sality, the similarity Score is 0.008. Throughout the analysis, except for three WRONG_i sets which interacted with a data not in D, the rest contained no peers, i.e., empty. If the empty WRONG_i sets were excluded from the similarity score calculation, this would result in a high similarity score of 0.802. Hence, they would not adversely impact any monitoring performed by PinPuppet.

Second, our findings validated that PinPuppet's crawling primitive can extract correct peers with $|\text{CORRECT}| > 0$ for each crawl cycle. However, the maximum number of correct peers that can be extracted per crawl cycle is dependent on the MM-protocol used by the botnet [13], e.g., limited neighbor list replies. Our initial expectation was that the average POI confidence score for CORRECT and BOOTSTRAP would be greater than WRONG after applying the threshold. This was the case except for Kelihos and Sality which had an equal confidence score with the other two categories. For Kelihos, we attribute this to the incomplete D, i.e., undiscovered peer list. Meanwhile, for Sality, the similar high score of 1.0 is mainly due to the fact that a particular non-IP data was extracted using the POI three times during the entire evaluation. Hence, when considering all non-empty unique elements WRONG, unfortunately, the average confidence score is considered to be 1.0 as well.

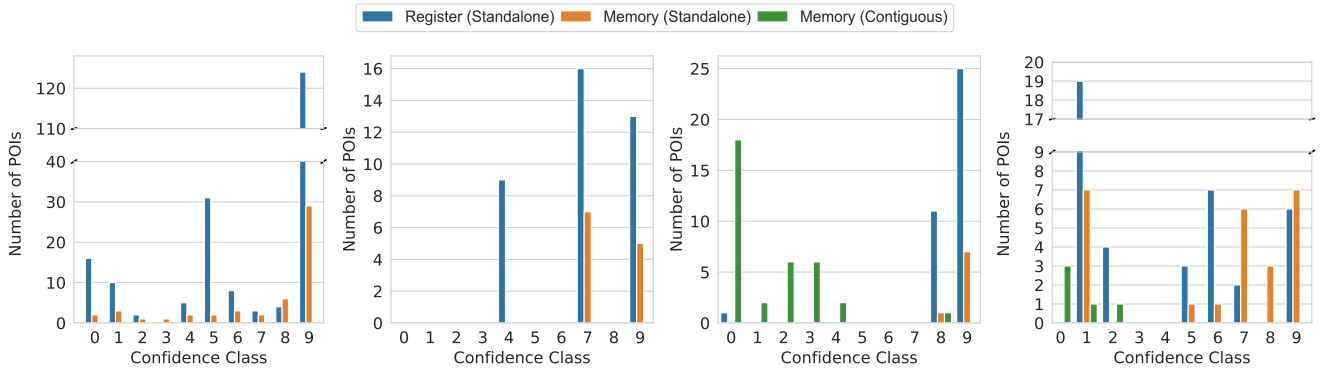
Two additional interesting observation are that for Sality and Kelihos, a big part of the bootstrap list is extracted each crawl cycle, and that Kelihos extracts almost all local peers each crawl cycle ($|P_{\text{local}}| = 41$ compared to $|\text{CORRECT}| = 40.79$). For Sality $|P_{\text{bootstrap}}| = 740$ but due to the fact that $|P_{\text{bootstrap}} \cap P_{\text{local}}| = 1$, we get that $|\text{BOOTSTRAP}| = 739$ is the maximum amount possible for this metric.

Based on the results outlined above, we can conclude that extracting shared peers from instrumenting a malware sample is feasible idea and suitable using PinPuppet. Hence, one can explore using the crawling primitive to develop a full-fledged crawler for live deployment of automated botnet monitoring.

6 DISCUSSION OF ADVERSARIAL ATTACKS

It is only natural to expect malware authors to attempt to make POIs less informative to hinder analysis using our proposal. Typical obfuscation techniques such as adding dead codes, modifying control

[9] https://pastebin.com/9euC4K9N



(a) ZeroAccess (b) Sality (c) Nugache (d) Kelihos

Figure 3: The distribution of confidence scores of POIs identified for each botnet.

		Without confidence score threshold		With confidence score threshold (≥ 0.8)	
		ø ($n = 100$)	Avg. confidence score	ø ($n = 100$)	Avg. confidence score
ZeroAccess	$\lvert P_{\text{extracted}}\rvert$	91.89 ($\sigma \approx 8.62$)	0.44 ($n = 100$ $\sigma \approx 0.01$)	17.00 ($\sigma \approx 0.14$)	0.94 ($n = 100$ $\sigma \approx 0.01$)
	|CORRECT|	15.91 ($\sigma \approx 0.51$)	0.48 ($n = 100$ $\sigma \approx 0.05$)	1.00 ($\sigma \approx 0.14$)	0.85 ($n = 99$ $\sigma \approx 0.01$)
	|BOOTSTRAP|	16.00 ($\sigma \approx 0.00$)	0.62 ($n = 100$ $\sigma \approx 0.01$)	16.00 ($\sigma \approx 0.00$)	0.96 ($n = 100$ $\sigma \approx 0.01$)
	|WRONG|	59.98 ($\sigma \approx 8.39$)	0.20 ($n = 100$ $\sigma \approx 0.01$)	0.00 ($\sigma \approx 0.00$)	n/a
Sality	$\lvert P_{\text{extracted}}\rvert$	742.09 ($\sigma \approx 1.11$)	1.00 ($n = 100$ $\sigma \approx 0.00$)	741.12 ($\sigma \approx 1.07$)	1.00 ($n = 100$ $\sigma \approx 0.00$)
	|CORRECT|	1.94 ($\sigma \approx 0.24$)	0.83 ($n = 100$ $\sigma \approx 0.03$)	1.94 ($\sigma \approx 0.24$)	1.00 ($n = 100$ $\sigma \approx 0.00$)
	|BOOTSTRAP|	739.00 ($\sigma \approx 0.00$)	1.00 ($n = 100$ $\sigma \approx 0.00$)	739.00 ($\sigma \approx 0.00$)	1.00 ($n = 100$ $\sigma \approx 0.00$)
	|WRONG|	1.15 ($\sigma \approx 1.05$)	0.29 ($n = 97$ $\sigma \approx 0.02$)	0.18 ($\sigma \approx 1.03$)	1.00 ($n = 3$ $\sigma \approx 0.00$)
Nugache	$\lvert P_{\text{extracted}}\rvert$	14.09 ($\sigma \approx 0.72$)	0.98 ($n = 100$ $\sigma \approx 0.00$)	13.09 ($\sigma \approx 0.72$)	0.99 ($n = 100$ $\sigma \approx 0.00$)
	|CORRECT|	3.45 ($\sigma \approx 0.80$)	0.98 ($n = 100$ $\sigma \approx 0.01$)	3.45 ($\sigma \approx 0.80$)	0.99 ($n = 100$ $\sigma \approx 0.00$)
	|BOOTSTRAP|	8.64 ($\sigma \approx 0.71$)	0.99 ($n = 100$ $\sigma \approx 0.00$)	8.64 ($\sigma \approx 0.71$)	0.99 ($n = 100$ $\sigma \approx 0.00$)
	|WRONG|	2.00 ($\sigma \approx 0.00$)	0.48 ($n = 100$ $\sigma \approx 0.00$)	1.00 ($\sigma \approx 0.00$)	0.92 ($n = 100$ $\sigma \approx 0.00$)
Kelihos	$\lvert P_{\text{extracted}}\rvert$	301.98 ($\sigma \approx 12.89$)	0.63 ($n = 100$ $\sigma \approx 0.06$)	235.79 ($\sigma \approx 1.49$)	0.85 ($n = 100$ $\sigma \approx 0.00$)
	|CORRECT|	40.79 ($\sigma \approx 1.49$)	0.46 ($n = 100$ $\sigma \approx 0.12$)	40.79 ($\sigma \approx 1.49$)	0.85 ($n = 100$ $\sigma \approx 0.00$)
	|BOOTSTRAP|	163.00 ($\sigma \approx 0.00$)	0.76 ($n = 100$ $\sigma \approx 0.03$)	163.00 ($\sigma \approx 0.00$)	0.85 ($n = 100$ $\sigma \approx 0.00$)
	|WRONG|	98.19 ($\sigma \approx 12.41$)	0.72 ($n = 100$ $\sigma \approx 0.02$)	32.00 ($\sigma \approx 0.00$)	0.85 ($n = 100$ $\sigma \approx 0.00$)

Table 1: Analysis of the average number of peers extracted for each crawl iteration classified by category.

flow, etc. [32] do not affect POIs as there will still be instructions which process data using a particular context. Furthermore, we successfully applied our approach to pack all our malware samples using UPX [24] or a custom packer. Despite encountering various packing mechanisms, none were able to bypass our technique.

Nevertheless, future malware that leverage code virtualization techniques built upon VM technologies would definitely pose a problem [17]. Such malware will no longer exhibit useful POIs, as they only execute virtual instructions that would appear random and misleading to our approach.

In Section 4.1, we assumed all possible anti-Pin countermeasures to be circumvented. This is required due to Pin not being specifically designed for malware analysis. Again we did not need to modify our malware samples in any way, hinting that concrete anti-pin mechanisms are not that common. However, there are other DBI frameworks similar to Pin with added anti-instrumentation hardenings, e.g., PEMU [33] and PANDA [7]. For instance, Pin operates from within the same memory space as the process being instrumented, PEMU instruments a process running within a VM from outside of the VM. Thus PEMU does not introduce memory artifacts that could be detected by the instrumented process. However, even hardened frameworks are not fully transparent to the application being instrumented. Considering the lack of development of PEMU at the time of writing, we opted for Pin which had better development, documentation, and support.

An adversary could also employ mechanisms that are specifically targeting our POIs identification process. Firstly, the deliberate introduction of irrelevant POIs within the binary. A malware could be designed to additionally process legitimate botnet data at dead ends in control flows. While this action does not change the behavior

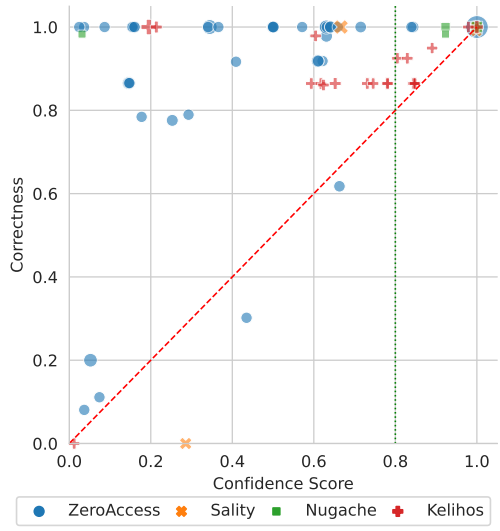


Figure 4: The correctness of POIs.

of the malware itself, the additional instructions would however be marked as POI by our methodology. As such, manual reverse engineering could again become more labor intensive, i.e., due to irrelevant beacons. Nevertheless, despite the introduction of the irrelevant POIs, the relevant POIs would still remain a subset of all identified POIs. As such, automated analyses similar to PinPuppet would not be affected by this attack.

Secondly, an adversary could try to actively manipulate the POI confidence score. For instance, a P2P bot could always include a subset of random IPs and ports within a valid *retL* message that would eventually ignored by the recipient bot. Since these "peers" are never contacted, they would also not appear in D. Hence, the POI confidence score for would be artificially lowered, making it harder to compare the quality of POI. This could cause PinPuppet to extract erroneous peers thus significantly hindering its crawling capabilities.

7 CONCLUSION AND FUTURE WORK

Malware reverse engineering is a tedious and labor-intensive task that can provide valuable insights to detect, monitor, or remediate malware. In this paper, we proposed a method to introduce POI as beacons to aid reverse engineers in analyzing malware. POIs are instructions that are found to interact with user-specifed data (DIOIs) that allows an analyst to focus, i.e., via the beacons, on the main goals of dissecting the malware. The identification of the beacons can be influenced by the analyst by providing goal-relevant data, e.g., filecontent – to discover encryption related routines, within the proposed methodology. To demonstrate the feasibility and the potentials offered by POIs, we analyze two ransomware and successfully identify their encryption routines. Additionally we have developed a prototype called PinPuppet that is able to automatically crawl four well-known P2P botnets. Moreover, our POI scoring mechanism succesfully managed to reduce the number of falsely extracted peers in our crawling. As future work, we would like to investigate combining our approach with dynamic taint tracking to enhance POI identification.

ACKNOWLEDGMENTS

This work has been partly funded by Universiti Sains Malaysia's Short Term Grant [304/PNAV/6315576].

REFERENCES

[1] Golam Sarwar Babil, Olivier Mehani, Roksana Boreli, and Mohamed-Ali Kaafar. 2013. On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. In *2013 SECRYPT*. 1–8.
[2] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd ACSAC*. 189–200.
[3] Thomas Barabosch, Adrian Dombeck, Khaled Yakdan, and Elmar Gerhards-Padilla. 2015. BotWatcher. In *Research in Attacks, Intrusions, and Defenses*. Springer International Publishing, Cham, 565–587.
[4] Joan Calvet, Carlton R. Davis, Jose M. Fernandez, Jean-Yves Marion, Pier-Luc St-Onge, Wadie Guizani, Pierre-Marc Bureau, and Anil Somayaji. 2010. The case for in-the-lab botnet experimentation: creating and taking down a 3000-node botnet. In *Proceedings of the 26th ACSAC*. 141–150.
[5] Lorenzo Cavallaro, P. Saxena, and R. C. Sekar. 2007. Anti-Taint-Analysis : Practical Evasion Techniques Against Information Flow Based Malware Defense.
[6] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 196–206.
[7] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. 1–11.
[8] Julien Duchene, Colas Le Guernic, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. 2018. State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques* 14, 1 (2018), 53–68.
[9] Kevin P Dyer, Scott E Coull, and Thomas Shrimpton. 2015. Marionette: A programmable network traffic obfuscation system. In *24th USENIX Security Symposium (USENIX Security 15)*. 367–382.
[10] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.
[11] Nicolas Falliere. 2011. Sality: Story of a peer-to-peer viral network. *Technical Report, Symantec Corporation* 32 (2011).
[12] Claudio Guarnieri. 2010. Cuckoo Cuckoo Sandbox. cuckoosandbox.org. Accessed: 2020-04-11.
[13] Shankar Karuppayah. 2018. *Advanced Monitoring in P2P Botnets: A Dual Perspective*. Springer.
[14] Max Kerkers, José Jair Santanna, and Anna Sperotto. 2014. Characterisation of the kelihos. b botnet. In *IFIP AIMS*. Springer, 79–91.
[15] Max Kerkers, José Jair Santanna, and Anna Sperotto. 2014. Characterisation of the Kelihos.B Botnet. In *Monitoring and Securing Virtualized Networks and Services*, Anna Sperotto, Guillaume Doyen, Steven Latré, Marinos Charalambides, and Burkhard Stiller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91.
[16] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. 2010. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *2010 IEEE Security and Privacy*. IEEE, 29–44.
[17] Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2018. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. *Computers & Security* 74 (2018), 202–220. https://doi.org/10.1016/j.cose.2018.01.008
[18] Malwarebytes Labs. [n. d.]. Locky: Ransom.Locky | Malwarebytes Labs | Detections. https://blog.malwarebytes.com/detections/ransom-locky/. Accessed: 2022-01-31.
[19] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*.
[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.
[21] Savita Mohurle and Manisha Patil. 2017. A brief study of wannacry threat: Ransomware attack 2017. *International Journal of Advanced Research in Computer Science* 8, 5 (2017), 1938–1940.
[22] Alan Neville and Ross Gibb. 2013. ZeroAccess Indepth. *Symantec Security Response* (2013).

[23] James Newsome, David Brumley, Jason Franklin, and Dawn Song. 2006. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the 13th ACM conference on Computer and communications security*. 311–321.
[24] Markus F.X.J. Oberhumer, László Molnár, and John F. Reiser. [n. d.]. UPX: The Ultimate Packer for eXecutables - Homepage. https://upx.github.io/
[25] Tonya Riley. 2020. The Cybersecurity 202: Global losses from cybercrime skyrocketed to nearly $1 trillion in 2020, new report finds. washingtonpost.com/politics/2020/12/07/cybersecurity-202-global-losses-cybercrime-skyrocketed-nearly-1-trillion-2020/. Accessed: 2022-01-31.
[26] Edward Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). *Proceedings - IEEE Security and Privacy*, 317–331. https://doi.org/10.1109/SP.2010.26
[27] August See, Leon Fritz, and Mathias Fischer. 2022. Polymorphic Protocols at the Example of Mitigating Web Bots. In *European Symposium on Research in Computer Security*. Springer, 106–124.
[28] S. Stover, D. Dittrich, John Hernandez, and S. Dietrich. 2007. Analysis of the Storm and Nugache Trojans: P2P Is Here. *login Usenix Mag.* 32 (2007).
[29] Mingshen Sun, Tao Wei, and John CS Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 331–342.
[30] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. 2020. An observational investigation of reverse {Engineers'} processes. In *29th USENIX Security Symposium (USENIX Security 20)*. 1875–1892.
[31] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. 2009. ReFormat: Automatic reverse engineering of encrypted messages. In *European Symposium on Research in Computer Security*. Springer, 200–215.
[32] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. *Proceedings - 2010 International Conference on Broadband, Wireless Computing Communication and Applications, BWCCA 2010*, 297–300.
[33] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. 2015. PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*. Association for Computing Machinery, New York, NY, USA, 147–160.

A HASHES OF MALWARE SAMPLES USED FOR THE EVALUATION

Table 2 lists the MD-5 hashes of the samples used in our evaluation.

Botnet	MD5
ZeroAccess	`ea039a854d20d7734c5add48f1a51c34`
Sality	`d35cf3c2335666ac0be74f93c5f5172f`
Nugache	`0c859cfad2fa154f007042a1dca8d75b`
Kelihos	`9b68b45afa269ba1b0c01749fa4b942f`
Wannacry	`84c82835a5d21bbcf75a61706d8ab549`
Locky	`d9e1e9cf9bc5322fa3de1f4cb731a624`

Table 2: Hashes of the samples used.

B MATCHING IP-PORT POIS

Algorithm 2 illustrates the algorithm we used to map IP- and port-POIs. This allows us to extract the exact IP and port combination used to crawl a peer. This algorithm is not needed when a botnet uses a fixed port for its MM-protocol (e.g., ZeroAccess).

Algorithm 2: Matching IP- and port-POIs.

Input: IPPois, PortPois, $P_{\text{bootstrap}} \cup P_{\text{socket}}$
Data: PoiMapping<IPPoi, List<PortPoi>>: poiMapping
Result: The mapping between IP- and port-POIs.

```
// extractedIPs and extractedPorts correspond
// to the data processed at the respective POI.
for ipPoi, extractedIPs in IPPois do
    for portPoi, extractedPorts in PortPois do
        matches←true
        for ip, port in zip(extractedIps, extractedPorts) do
            if (ip, port) ∉ P_bootstrap ∪ P_socket then
                matches←false
        if matches then
            poiMapping [ipPoi].append(portPoi)
return poiMapping
```

C AUXILARY INFORMATION ABOUT THE LOCAL BOTNETS

The Nugache local botnet requires eight local bootstrap peers, as every peer has at most 15 active connections [28]. Thus, only having one local bootstrap peer is not enough. Not all peers who join the overlay receive new local peers.

Sality does not use a fixed port and, instead, uses ports which depend on the computer name [13]. To make our Sality local botnet more realistic, all peers in the local botnet have unique computer names. In addition, we have added NAT rules to set the correct port for MM-traffic being sent to the local bootstrap peer.

D CONFIDENCE SCORE THRESHOLD PARAMETER STUDY

The POI confidence score threshold needs to be carefully chosen to find the balance between a) wrong peers being extracted and b)

	Without PinPuppet		
	T_0	T_1	T_2
ZeroAccess	12.21 ($\sigma \approx 0.24$)	20.01 ($\sigma \approx 0.00$)	20.00 ($\sigma \approx 0.00$)
Sality	335.58 ($\sigma \approx 0.56$)	32.57 ($\sigma \approx 0.00$)	43.71 ($\sigma \approx 0.00$)
Nugache	0.61 ($\sigma \approx 0.32$)	405.60 ($\sigma \approx 0.00$)	405.60 ($\sigma \approx 0.00$)
Kelihos	0.78 ($\sigma \approx 0.02$)	200.30 ($\sigma \approx 0.00$)	600.29 ($\sigma \approx 0.00$)
	With PinPuppet		
	T_0	T_1	T_2
ZeroAccess	14.81 ($\sigma \approx 0.03$)	19.97 ($\sigma \approx 0.01$)	20.00 ($\sigma \approx 0.00$)
Sality	689.36 ($\sigma \approx 82.87$)	32.51 ($\sigma \approx 0.10$)	43.44 ($\sigma \approx 0.15$)
Nugache	58.67 ($\sigma \approx 32.80$)	406.75 ($\sigma \approx 0.08$)	406.27 ($\sigma \approx 0.09$)
Kelihos	0.81 ($\sigma \approx 0.02$)	213.25 ($\sigma \approx 0.31$)	601.19 ($\sigma \approx 0.03$)

Table 3: The overhead measurements for each botnet in seconds ($n = 5$).

how many peers are extracted. We have analyzed these metrics in Figure 5 and have determined 0.8 as the threshold (marked with a gray-vertical line). 0.8 strikes a good balance between the number of wrong extracted peers, i.e., the distance between the upper and the lower lines, and the total number of extracted peers. 0.9, for example, would not have been suitable as then, Kelihos only extracts very few peers.

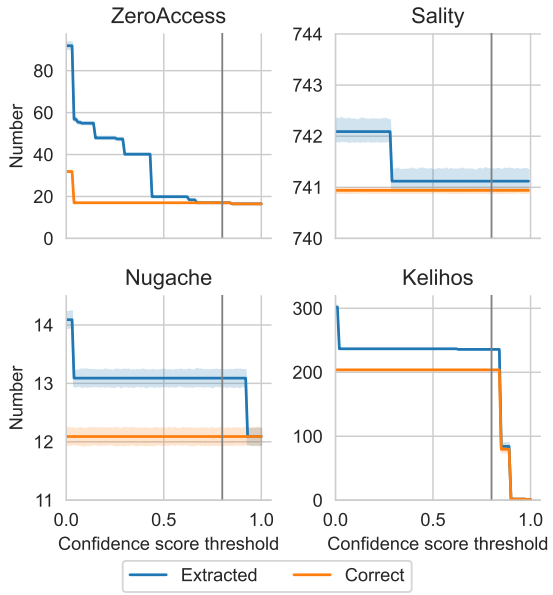


Figure 5: Peer extraction analysis

E OVERHEAD MEASUREMENT RESULTS

The complete measurement data from the analysis of the overhead introduced by using Pin to trace a sample is provided in Table 3. For each botnet, the measurement was repeated 5 times. The evaluation in Section 5.2 is based on this data.

F IDA AND GHIDRA PLUGINS FOR IMPORTING AND DISPLAYING POIS APPLIED TO RANSOMWARE

After having identified POIs using the process from Figure 1, a reverse engineer can use the binary where the POIs were identified, the generated list of POIs, and our IDA or Ghidra plugin for further analysis. Note that the base address in IDA or Ghidra must be the same as when extracting the POIs.

For the ransomware Locky and Wannacry it went as follows. We use a VM with snapshot capability. In the VM an API monitor is running, that monitors file reads and writes. Then the ransomware is executed and our Pintool attached, step ② of Figure 1. After the ransomware wrote (encrypted) some files, appending its file extension (.osiris, .wnncry), the ransomware process is suspended. The tracefiles as well as the encrypted files are copied to another machine. By reverting the VM the unencrypted files can be restored and can also be copied. The encrypted or cleartext file content bytes can be used as $d \in D$ (Step ③). We used 4-16 bytes of the encrypted file and of the unencrypted file. Using the algorithms of Section 3.3 POIs were automatically identified (Step ④). We filter the POIs that were found system modules (Step ④). The memory regions where the remaining POIs were located were then extracted as a memory dump, one region each for both ransomware. Both regions were not listed as modules, but contained executable code. Below, the Ghidra plugin shows the results for Locky and the IDA plugin shows the results for Wannacry. The shown POIs were validated using dynamic analysis, i.e, whether file content is really accessed at these points.

For Ghidra, the plugin is implemented in Python. After enabling the Plugin in the script manager, it can be activated in the toolbar. One is presented with a dialog box where one needs to select the POIs JavaScript Object Notation (JSON) file. Afterwards, the POIs show up as bookmarks in the disassembled code. Figure 6 shows the Locky ransomware POI loaded in Ghidra.

The IDA plugin is also a Python script which is read by IDA. This script can be selected via the menu File→Script File.... Afterwards, a dialog box will appear and allows the selection of a JSON

file where the exported POIs are stored. These POIs are then imported into IDA as disabled breakpoints. They are then visible as an overview in the breakpoint view and also in the disassembler view. Figure 7 shows such a view for the Wannacry ransomware. Unlike Locky, several POIs were identified. The image shows an AES encryption function where `byte_10007A3C` could be confirmed to be the S-box.

As the reverse engineer knows the data set D corresponding to the POIs, it is immediately known which data is processed at the marked code locations. Based on this information, reverse engineering efforts can be focused on relevant code regions. Based on the goals of the analyst, these beacons would significantly reduce the time take to review the malware under scrutiny.

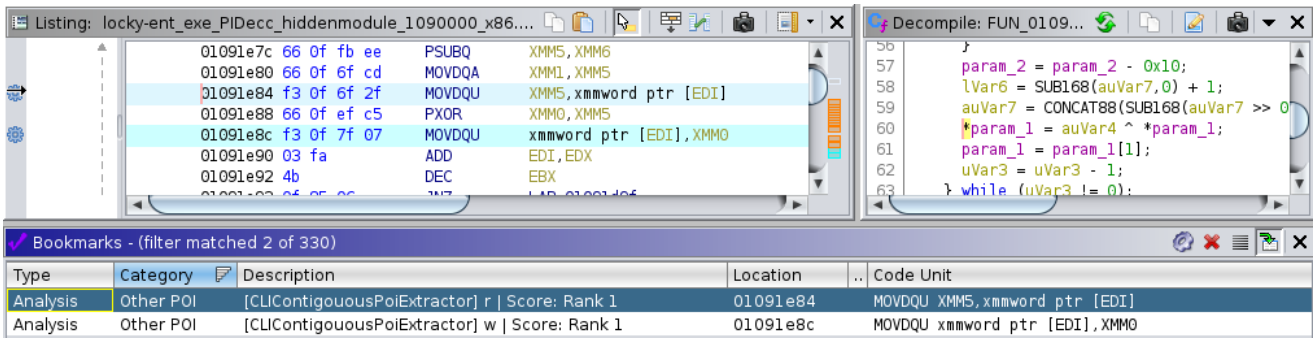


Figure 6: Ghidra plugin overview for Locky (excerpt, Base Address: `0x1090000`)

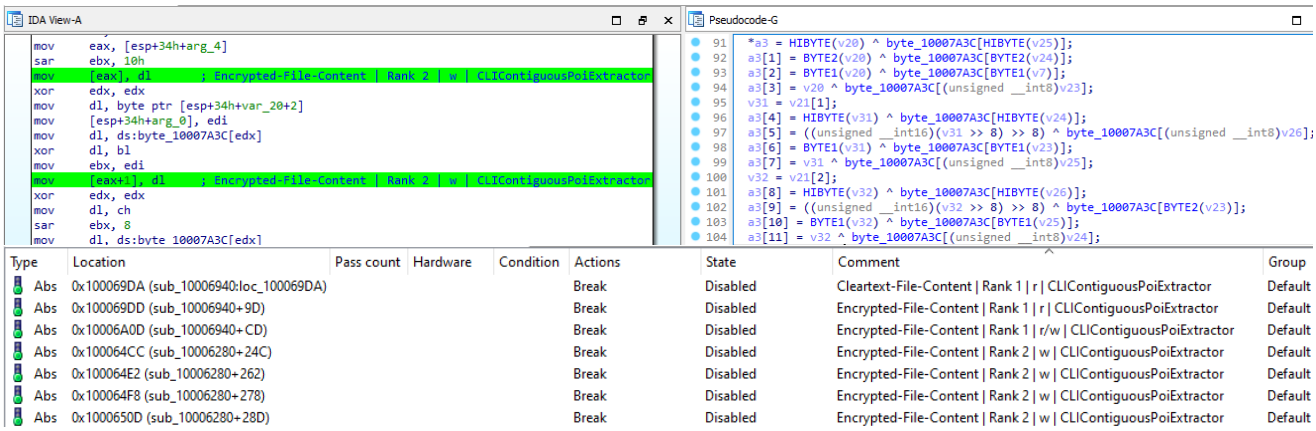


Figure 7: IDA plugin overview for Wannacry (excerpt, Base Address: `0x10000000`)

Appendix E

Polymorphic Protocols at the Example of Mitigating Web Bots

Abstract

Unwanted automation of network services by web robots (bots) increases the operation costs, and affects the satisfaction of human users, e.g., in online games or social media. Bots impact the revenue of service providers and can damage society by spreading false information. While few bots are usually not a problem, a large number is. Thus, we focus on bots that directly use a service's application protocol, as they are the most efficient and easiest to scale. Current solutions such as registration with personal data or CAPTCHAs are frustrating for users or can be easily evaded. Anti-reverse engineering and solutions for digital rights management that impede bot creation, e.g., unique client specific API keys, are only effective for the first bot. In this paper, we introduce a novel obfuscation approach that we call polymorphic protocols and that is inspired by polymorphic malware and methods to bypass censorship resistance. When using polymorphic protocols, each client of a service has an own application protocol, so that the costs of duplicating bots for an attacker significantly increases. For every bot that an attacker wants to create, he has to extract and reimplement a protocol from a valid client. We integrate our approach into an existing ecosystem and implement it exemplarily for Protobuf and Java. Our results indicate that the overhead for service providers and users is low, depending on the deployment and chosen protocol configuration. At the same time, our polymorphic protocols significantly increase the attacker costs to create multiple bots, when limited to conventional reverse engineering techniques only.

Reference

August See, L. Fritz, M. Fischer. Polymorphic Protocols at the Example of Mitigating Web Bots. European Symposium on Research in Computer Security, 2022. ©2022 Springer.

Contribution

The core idea, solution approach, and evaluation design for this publication stem from the author of this dissertation. The second author performed the implementation and evaluation. The third author contributed to the refinement of the work.

Polymorphic Protocols at the Example of Mitigating Web Bots

August See, Leon Fritz, and Mathias Fischer

Universität Hamburg, Germany
{richard.august.see,leon.fritz,mathias.fischer}@uni-hamburg.de

Abstract. Unwanted automation of network services by web robots (bots) increases the operation costs, and affects the satisfaction of human users, e.g., in online games or social media. Bots impact the revenue of service providers and can damage society by spreading false information. While few bots are usually not a problem, a large number is. Thus, we focus on bots that directly use a service's application protocol, as they are the most efficient and easiest to scale. Current solutions such as registration with personal data or CAPTCHAs are frustrating for users or can be easily evaded. Anti-reverse engineering and solutions for digital rights management that impede bot creation, e.g., unique client specific API keys, are only effective for the first bot. In this paper, we introduce a novel obfuscation approach that we call polymorphic protocols and that is inspired by polymorphic malware and methods to bypass censorship resistance. When using polymorphic protocols, each client of a service has an own application protocol, so that the costs of duplicating bots for an attacker significantly increases. For every bot that an attacker wants to create, he has to extract and reimplement a protocol from a valid client. We integrate our approach into an existing ecosystem and implement it exemplarily for Protobuf and Java. Our results indicate that the overhead for service providers and users is low, depending on the deployment and chosen protocol configuration. At the same time, our polymorphic protocols significantly increase the attacker costs to create multiple bots, when limited to conventional reverse engineering techniques only.

1 Introduction

The automated use of Internet services is an essential building block of the Internet and the web. Many services depend on each other. Examples are the embedding of a weather feed into a web page or a service that provides a price comparison by automatically querying different marketplaces. Many services provide specific interfaces for other services to allow the automation of their usage. However, there are also services without such interfaces that are intended for humans only. Automated use of a service by a program, hereafter called a bot, can affect the satisfaction of human users and can cause financial or social damage. For example, the automated use of social media can be used to spread opinions and false information, which can even influence elections [7].

Automation of a service can be done in different ways [3]. The most efficient approach is to automate API of the service. Using the API directly only requires a script. By simply executing the script multiple times, it is possible to create a large number of bots, e.g., to influence voting opinions on social media through nationwide spamming [7]. This puts service providers in a dilemma. A service needs to be easy and quick to use, as complicated requirements, e.g., for registration scare users away [14]. But registration without requiring limited resources such as phone numbers or passports makes it easier to create a bot army. The main problem is that automation can be made harder for attackers, but not completely prevented as human users must still be able to use the service without too much friction.

The threat model of this paper is as follows: The focus lies on services with a state tied to an entity, e.g., game progress or social media likes bound to an account. These services in particular suffer from bots, as there is a gain in running multiple bots. For example, it can be beneficial to use hundreds of bots to control hundreds of social media accounts. However, there is no gain in using hundreds of bots to query the static content of some blog or news portal. The attacker is not limited to a subset of reverse engineering techniques, but limited to using the application protocol to create bots.

In many cases, CAPTCHAs are the first and last defence against bots. However, they introduce user friction and are losing effectiveness as machine learning advances [24,2]. Other approaches use anti-reverse engineering techniques to oppose Man-At-The-End (MATE) attackers [1], that have access to a client application on a controlled device. Those approaches aim at making the extraction and use of the application protocol more difficult, e.g., embedding (unique) API keys in the client application or using obfuscation and anti-reverse engineering techniques [23,12]. Most of these techniques only make it difficult to create the first bot. Once a bot is created, it can be scaled again. But this is what makes API bots so threatening: the ability to quickly and inexpensively spawn large numbers of bots. Because this approach can be really harmful, this paper focuses on how such automation can be restricted.

The main contribution of this paper is an approach to combat the cost-efficient duplication of bots, which can be applied with low performance and organizational overhead. In more detail, we make the following contributions:

- We propose a method, to increase the cost of duplicating bots by assigning each client of the same service its own application protocol. We call this Polymorphic Protocols (PPs). While there are already many strong obfuscation techniques for binaries (Tigress, Thermida) [10,27] and protocol obfuscation techniques in the censorship resistance realm [11,21], we are the first to our knowledge to use obfuscation of application protocols against bots.
- We implement the approach for the widely used protocol language protobuf and the programming language Java[1]. It is easily applied to existing protocols and just requires the existing protobuf file as input. Everything else

[1] Avaiable open source at `https://github.com/UHH-ISS/polymorphic-protocols`

is generated automatically so that it can even be used in a CI/CD pipeline without the need for a developer.
- We evaluate the technical performance overhead of the approach. We also discuss the organizational overhead for developers and the additional effort for attackers to duplicate bots.

Note that polymorphic protocols are an obfuscation technique to make the **scaling** of bots more difficult and **not** to prevent the creation of bots. The approach gains from being used along-side existing anti-reverse engineering mechanisms that impede code extraction (slicing), e.g., anti symbolic execution, virtualization, or just in time compilation [4,10]. Legitimate bots and interoperability across different services is still possible, e.g., by providing special API keys after thorough verification.

The rest of the paper is structured as follows. Section 2 discusses other approaches that make it harder to create bots. Section 3 explains how polymorphic protocols can be created and applied. Section 4 describes the implementation, evaluates and discusses the results. Finally, Section 5 concludes the paper.

2 Related Work

Regarding fighting bots there are mainly three different classes of approaches. Proofs of being human, detecting unusual behavior and anti-analysis.

Proofs of being human. The main technique to identify humans are CAPTCHAs [19,20], but also personal information often requested during the registration. Such information could be, from easy to harder to provide, email address, phone number, an image of the user holding an identity card. While requiring an identity card for registration would likely solve the problem of unwanted automation, it is a hard and privacy-unfriendly requirement. Related techniques try to detect human presence through hardware interaction. As an example, Not-a-Bot [13] uses TPMs to add a tag to each network request sent some time before a mouse or keyboard interaction.

Detecting unusual behaviour. These approaches try to detect bots so that countermeasures can be taken. Well known CAPTCHA providers like [19,20] use these approaches to reduce the disturbance for human users. Unfortunately, the machine learning models are not public. A major criticism is that users then have to send their data (browser fingerprinting [8]) and behaviour (mouse movement, website traversal [19]) to third parties to assess whether or not a bot is present. Another problem arises when a user cannot be classified as a human or a bot. Then again approaches of the class *proofs of being human* like CAPTCHAs are needed which have their own weaknesses[2,24].

Anti-analysis against MATE attackers. Approaches in this class try to increase the cost of creating bots. It includes anti-reverse engineering and obfuscation techniques, e.g., detecting whether a debugger is attached, binary packing or even dummy code [12,23]. For network services, a common anti-analysis technique is to authenticate the used application protocol using API

keys prior to the use of TLS [2]. To automate a service that uses this technique, the application protocol and the relevant keys need to be extracted [15,16]. The keys themselves can also be protected by anti-reverse engineering techniques.

Tigress [10], VMProtect [25] and Themida [27] are examples for advanced software protection systems. They are applied to the source code of an application and create a protected executable, using different anti-reverse engineering and obfuscation techniques. However, even those advanced protection systems do not obfuscate the protocol. If someone can reverse engineer the protocol, despite the used protection system, the bot can again be scaled very easily. The main problem of anti-reverse engineering is that it is difficult to quantify how effective a technique is against an arbitrary attacker [1,26,4]. When a device is fully controlled by an attacker, all techniques can only increase the cost for an attacker to reverse engineer an application, but they cannot prevent it.

Polymorphic protocols are an obfuscation approach and thus related to protocol obfuscation techniques. Most of the approaches mentioned in the literature are in the area of censorship resistance [21,11]. The objective is to bypass censorship and traffic inspection by cloaking specific traffic as usual traffic. Thus, censored services can be accessed despite lacking encryption and techniques like deep-packet inspection. All those approaches have the main focus on cloaking traffic and consider performance and ease of use in other applications in the second place. Also, they assume a different attacker, namely an attacker that has only access to the traffic. Thus existing approaches cannot be directly used as PP to combat bots.

3 Polymorphic Protocols

The core idea of limiting the cost-efficient duplication of API bots are polymorphic protocols. Every client for a service communicates with the service via its own application protocol. This protocol can be seen as an identifier to distinguish between clients. When a bot is simply duplicated, all duplicates would share the same protocol. This way duplicates can be detected and excluded from further communication with the service. This effectively increases the cost to create multiple different bots. To create an API bot army, bot creators can either manually reverse engineer the application protocol from a different client for each bot that is created which is laborious. Or they can find a way to automate the reverse engineering and bot creation, but this is difficult on a technical level [18,22]. In contrast, API keys can be much easier extracted, e.g., using function hooks, and do not force the bot creators to modify their code for each new bot.

3.1 Basic Approach

An overview of the basic approach is given by Figure 1. The *protocol specification* is the base protocol, used by the service. It specifies the format of messages as well

[2] `https://github.com/see-aestas/SINoALICE-API`
`https://github.com/see-aestas/JodelApi`

as semantic information, e.g., if a message is time-critical or the dependencies between messages (order of transmission). The *client identifier* is an identifier for a client. Each client has a different identifier. Since our approach focuses on services that hold a state over a client (cf. Section 1) this is already given, because the service needs some way to match some state to a client. The *secret seed* is exactly what it is named after.

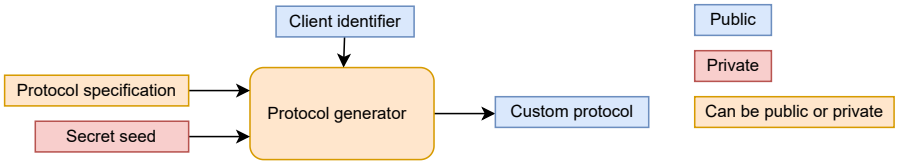


Fig. 1. Polymorphic Protocol Generation Overview

The *protocol generator* gets the *protocol specification* and the *client identifier* as well as the *secret seed* as an input. It outputs a new *custom protocol* that is different from the base protocol but keeps all mandatory semantic dependencies. Message formats, orders, encodings and other features are (pseudo)randomized transformed. There are three main classes of so-called transformations. Figure 2 shows these as well as some example sub-classes.

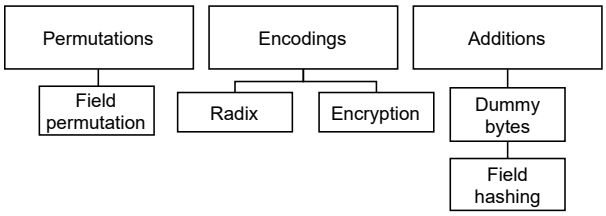


Fig. 2. Possible transformation classes

The classes are: Permutations that are length-preserving and permute message content, encodings that modify (parts of) a message, and additions that add more information to a message. These so-called transformation classes are a collection of different transformations under one label. This abstraction is needed to compare the performance of different transformations. A class can also contain sub-classes. For example, the class *dummy bytes* includes all transformations that append some extra bytes to a message. The class *field hashing* includes all transformations that hash all fields and then append the hash to the message. This class is a subclass of *dummy bytes* as a hash can be considered as a special way to generate dummy bytes. Here it is important to note that the transformations should ideally be hard to understand during reverse engineering. For

example, a custom hash function to append bytes is harder to reverse engineer than directly appending some bytes to a message. Other transformations shown here are the *field permutation* where the order of data fields of a message is permuted, *radix* encoding where a message is encoded to some other base and *encryption* where a message is encrypted using some cipher. A developer can always add more classes. The transformations are used to deterministically generate a *custom protocol*. The *custom protocol* is later integrated into the client application. Each client has its own client identifier and thus custom protocol to communicate with the same service. Only the *secret seed* must be private. Its function is to prevent an adversary to generate valid *custom protocols* using the *protocol generator* for a service that uses an unknown, secret seed.

3.2 Formal Model

A protocol $\mathcal{P}$ is defined as a set of message format specifications F and their semantics S: $\mathcal{P} = \{F, S\}$, $F = \{f_1, f_2, ..., f_n\}$ where f is a format specification for some protocol message, e.g., $f = \{$*field1:int, field2:string-base64, ..., fieldx:Object*$\}$. A generator G deterministically generates a new, custom protocol based on the base protocol $\mathcal{P}$, client ID and the secret seed as inputs. $G(\mathcal{P}, client, seed) \mapsto \mathcal{P}'$. The generator relies on transformations of message format specifications $T_x(N, S) \mapsto N'$, with $N \subseteq F$ where x denotes a transformation class. Note that multiple message format specifications can be transformed together. A transformation class is the assignment of different transformations to a more generic class (cf. Figure 2). N' are then the new custom message format specifications and the actual transformations that operate on the real messages. For simplicity, $T_x(m) \mapsto m'$, where m is some actual message from the set of all valid protocol messages M, is used for transformations on the messages. Transformations can be applied on single message format specifications or across multiple ones, respecting their dependencies and can also be chained.

Transformations are required to be easily computable and unambiguous invertible, as an encoded message must be easily decoded. Transformation classes are used to compare the performance across different schemes of transformations. In this context, we are mainly assessing the impact on the resulting protocol. Assume a developer builds the transformation class T_x. Now one would like to estimate the effects on a protocol beforehand. How much additional data is transmitted using the resulting protocol on average (ΔT_x)? How expensive is the calculation for the client and server? How different are the protocol messages ($\varnothing T_x$)? The calculation of the metrics is described in the following.

The difference between two transformations is denoted to as $|T_{xi}(N, S) - T_{xj}(N, S)|$ and $|T_x|$ to the number of transformation in a transformation class. The difference between two transformations is calculated using the normalized compression distance (NCD) [17]. The metrics can be calculated numerically. Let D_{1x} be a set of uniform random sampled indices of transformations from a transformation class x. Let D_{2x} be a a set of pairs (i, j) where $i \neq j$ and $\nexists (i', j') \in D_{2x} \mid i = j' \wedge j = i'$. The indices i, j are uniform random sampled

indices of transformations from a transformation class x. Let $M_s \subseteq M$, e.g., uniform random sampled.

The average distance Δ of transformed protocol messages of T_x. A low value indicates that a valid message might be easy to guess for an attacker.

$$\Delta T_x = \sum_{m \in M_s} \sum_{(i,j) \in D_{2x}} \frac{|T_{xi}(m) - T_{xj}(m)|}{|M_s| * |D_{2x}|} \tag{1}$$

The average compressed message length difference $\varnothing$ between original and message transformed by T_x.

$$\varnothing T_x = \sum_{m \in M_s} \sum_{i \in D_{1x}} \frac{|C(T_{xi}(m)) - C(m)|}{|M_s| * |D_{1x}|} \tag{2}$$

The uniqueness of transformations in a transformation class δ. Uniqueness is closely related to collisions. A collision is when two different transformations of the same transformation class transform a message into the same new message. A low value indicates that an attacker might be able to successfully replay monitored messages.

$$\delta T_x = \frac{|X|}{|D_{1x}| * |M_s|}, X = \{T_{xi}(m) \mid i \in D_{1x} \mid m \in M_s\} \tag{3}$$

3.3 Transforming Protocols

Unconditional Protocol Transformations Unconditional transformations include every transformation that can be done without knowing the semantic S of messages. Those transformations do not require knowledge of the protocol.

Permutation : The protocol message is permuted.
Dummy bytes : The protocol message is appended with random bytes.
Hash : A hash is appended to the protocol message (SHA1 or SHA256 or MD5)
Radix : The protocol message is converted to another random base (2-255).

Especially using cryptographic routines as part of creating the protocol hardens this approach against input reverse engineering approaches for encrypted protocols like [28] since there is no clear boundary where the protocol message is constructed and where it is encrypted.

Conditional Protocol Transformations For transformations in this class, semantic information about the protocol must be available. This requires a formal protocol specification which should also contain semantic information. A distinction needs to be made between transformations that are applied to a single message (S) and those that are applied to multiple messages (M).

Delay : Delay a non-critical message (S).

Swap : Swap the sequence of messages (M).
Split : Split messages (S).
Merge : Merge messages (M).
Custom logic : Process messages already in some way on client side, e.g., calculation of certain data (SM).

Transformations on multiple messages are more difficult to use and maintain for a developer as they may introduce side effects. In addition, semantic changes in an application must be correctly transferred to the protocol. Thus, conditional transformations are considered less safe than unconditional transformations. The possible conditional transformations depend on the application and the semantics of messages. While some transformations make it harder to recover the protocol, e.g., Swap and all unconditional transformations, others make it difficult to associate actions with sent network messages, e.g., the delay transformation.

Randomizing Protocols The next step is to select transformations for a custom protocol and chain them together. For each message format specification in a protocol, first possible conditional and then unconditional transformations are selected. In the end, multiple transformations are applied to each format specification. How many is up to the developer and also depends on the format specification and semantics.

Algorithm 1: Selecting transformations

input: $\mathcal{P}$ - the base protocol $\mathcal{P} = \{F, S\}$
input: *prg* - PRG initialized using client ID and secret seed
input: T_c - set of conditional transformations
input: T_u - set of unconditional transformations
Result: $\mathcal{P}'$ transformed custom protocol

1 $F' = \{\}$, $S' = S$;
2 **for** $f \in F$ **do**
3 T'_c, T'_u = getAllowedTransformations(T_c, T_u, f, S');
4 $t_c, t_u = prg.choice(T'_c, T'_u)$;
5 $f' = t_c$(f) `// conditional transformation`
6 F'.add(t_u(f')) `// unconditional transformation`
7 S'.add(f, t_c) `// necessary for transforming multiple` f
8 $P' = \{F', S'\}$

One possibility is to use a pseudo-random generator to select transformations, as displayed in Algorithm 1. The basic idea is that for each message format specification $f \in F$ a random, allowed unconditional and conditional transformation is applied. If no transformation is allowed, f is not transformed. If a transformation of T_c is a transformation for multiple messages (multiple f), the transformation is applied to all of them. This can be handled using the semantics

and identifying the allowed transformations in Line 3 and 7. How the allowed transformations are selected depends on how the semantics are implemented. One possibility to assign semantics via annotations of message format specifications (@delay, @not$[T_x, T_y, ...]$). The implementation and which semantics have to be modeled depends heavily on the application.

While it is possible to use the algorithm without assigning semantics to format specifications, it can cost performance. For example, for video streaming or downloading files, a relative increase of the transmitted data might not be desired. This can be addressed by excluding transformations that increase the data length, or by excluding this message completely from any transformation.

3.4 Using Polymorphic Protocols

Our approach enables the generation of custom protocols. How this is used is up to the service.

Deployment Strategies We highlight two deployments.

Full-Polymorphic Each client has a different custom protocol.
Time-Polymorphic Each client has the same custom protocol, the protocol is changed after a certain time.

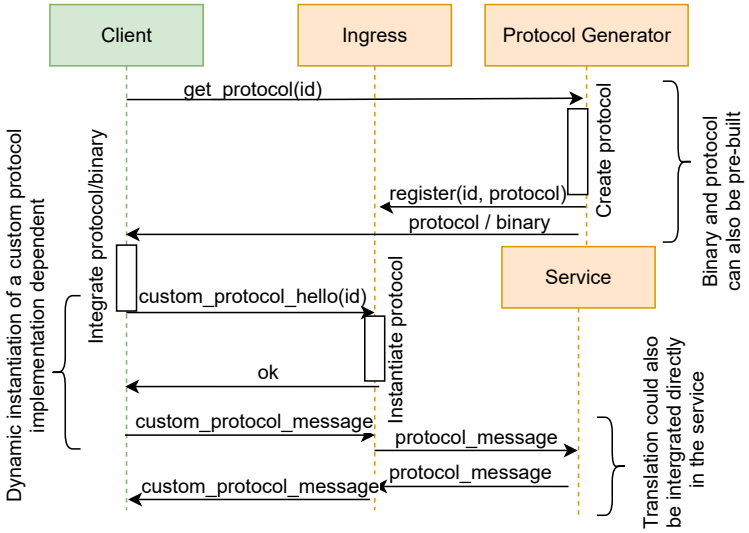


Fig. 3. Overview of Possible Polymorphic Protocol Communication Usage

While the approach in this paper tries to keep the complexity, performance loss, developer and user friction as low as possible, they still exist (Section 4).

Thus, PPs are best used for applications that are already heavily affected by bots, e,g, social networks or games and resource-intensive to emulate (mobile or desktop applications). Figure 3 shows how dynamically instantiating and using a PP, e.g., for Full-Polymorphic deployment, could be implemented. The client starts with the capabilities to request a custom protocol. It sends its client ID to the protocol generator. The protocol generator generates a custom protocol, registers it at the ingress server and sends it to the client. Depending on the use case the custom protocol is sent back as a standalone binary or a binary fused with the main application. When the client wants to communicate with the server it first sends a custom protocol hello. The custom protocol is looked up at the ingress, server and instantiated. When a client receives the message that the protocol is found and instantiated it can begin sending custom protocol messages to the ingress. The ingress acts as a proxy and translates the custom protocol messages back to protocol messages that can be understood by the service. Note that the procedure in Figure 3 is meant as an example and there are many more ways to use and deploy PPs.

Reducing Deployment Costs Polymorphic protocols do not necessarily have to be dynamically loaded by the client and not every client must use a custom protocol by default. A practical approach for mobile applications is to use PP only on clients that are running on rooted devices or emulators. Devices running unmodified systems must pass an integrity check, e.g, play integrity[3] and share one custom protocol. This check also ensures that a bot cannot simply use the base protocol as it would need to pass those integrity checks. It is also possible to assign a new protocol to a client after a certain time or number of protocol messages. Another strategy is to give a set of devices the same custom protocol, e.g., based on region, OS, IP address, update version or time. It can also be coupled with bot detection systems, to control the protocol change cycle for users. Slow for legitimate users, and fast for abusive users. Thus, the cost of using a custom protocol can be reduced, as well as the user friction, as legitimate users have fewer forced updates due to protocol changes.

4 Evaluation

In this section we are summarizing the evaluation results of our approach. For that, we have evaluated the costs for service providers to use the approach and the effort for attackers that want to build multiple bots for services protected by a polymorphic protocol. We answer the following research questions:

RQ1 : What is the overhead of different transformations?
RQ2 : How does the approach compare to using unique and client-specific API keys to hinder an attacker to create multiple bots for a service?
RQ3 : What is the technical and organizational overhead for a service that wants to use polymorphic protocols?

[3] `https://developer.android.com/google/play/integrity/overview`

4.1 Implementation

Our objective is to minimize the effort of using our approach. Instead of creating our own (unrestricted) protocol description language, we integrate it into an existing ecosystem. For this, we have chosen to use *Google protobuf* as the description language and extended their *Java* bindings to be compatible with our approach. Porting the implementation to other programming or protocol languages is possible. Currently, the implementation is not feature complete to *protobuf v3* [4]. We support the basic features, i.e., messages, all primitive (scalar) data types, and nesting (inheritance). Other features may work, but are not fully tested. Thus, if some service already uses *protobuf* and *Java*, creating a polymorphic protocol is effortless (cf. Appendix A).

As there is support for protobuf and java on all major systems, our approach can be used on all these systems without adjustments. Our implementation[1] only needs the proto file (protocol specification) as input and generates the new custom protocol, i.e., another proto file and necessary code wrappers. The wrappers are used by both the client and the server. To use the protocol, the generated wrapper files only need to be integrated into the respective project by replacing the old files and setting the package name.

4.2 Performance Evaluation

Using PPs comes with a performance overhead, which we evaluate in this section. We evaluate the transformation classes first individually and then in their combination with our protocol implementation. Everything is run on a Windows 11 computer I7 7700K CPU with 16GB DDr4 2660 Mhz RAM.

RQ1: Transformation Performance The metrics (average distance between two transformed protocol messages Δ_x, average compressed message length increase in bytes $\varnothing_x$, uniqueness of transformations δ_x) from Section 3.2 are used to analyze the performance of the transformation classes. The number of dummy bytes is limited to four and the radix transformation to the bases 2-255. Furthermore, we also measure the time for doing the transformation. Transformations are applied directly to bytes. To not assume some specific protocol, we sample 100,000 protocol messages as random bytes with the length derived from a normal distribution $\mathcal{N}(100, 25)$. Each sampled message is transformed up to 1,000 times per transformation class.

The results (Table 1) indicate that not every transformation class is the same and classes must be selected according to use cases. The permutation transformation does not increase the message length (by definition). Dummy bytes affect the message length but are way faster to calculate. The hash transformation affects the message length but is also fast to calculate. Since it is also about forcing an attacker to invest as much work as possible to extract the different transformations, efficient but complex transformations are wanted. While an evaluation

[4] `https://developers.google.com/protocol-buffers/docs/proto3`

of transformation classes gives some insight into the performance, it does not assess the performance and usage of the classes in a system.

Transformation class	Δ_x	$\varnothing_x$	δ_x	time
Permutation of message bytes	0.924	0	1	413.23 (s per 1 mil)
Dummy bytes (4)	0.086	4	1	1.07 (s per 1 mil)
Hash	0.261	26.6	1	2.91 (s per 1 mil)
Radix (2-255)	0.88	12.89	1	160.85 (s per 1 mil)

Table 1. Transformation class properties

RQ3: Protocol Performance While RQ1 evaluates the individual performance of single transformations this does not allow to assess the performance of the complete protocol. The performance, namely processing time, additional program size, build time and resource utilization, is compared to the base protocol. Four different protocols are considered. The notation $P_{\mathit{messages.fields}}$, e.g., $P_{10,15}$ is a protocol with 10 different messages, each message has 15 fields. In the following P denotes the base protocol and $G(P)$ the transformed protocol generated from P. There are a total of 15 different (scalar) value types in protobuf. For our evaluation we use all types equally often within one message. Thus, each message always contains a multiple of 15 fields.

Build Overhead Table 2 shows the results of the build overhead. A build includes the transformation of the protocol, the creation of wrapper classes, and the compilation of the (new) protocol using the protobuf compiler.

	$\lvert G(P_{1,15})$	$\lvert G(P_{100,15})$	$\lvert G(P_{1,150})$	$\lvert G(P_{100,150})$
Build Time	+3%	+72%	+9%	+128%
Memory	-14%	+18%	+33%	+94%
Protocol Size	+138%	+144%	+202%	+295%

Table 2. Mean program build properties: Difference of polymorphic compared to the original protocol in percent (N=100)

For PP, the size of the necessary wrapper classes is included in the protocol size. As expected the build time for the PP is higher than for the base protocol. However, the number of protocol messages seems to have a higher impact on the build time and protocol size for PP than the number of fields. The protocol size can be explained by the fact that for each message a separate additional java wrapper is created for the transformations. This also affects the build time.

The peak memory utilization is harder to interpret as it is mostly dependent on the protobuf compiler. The more and the larger the messages are, the more

memory is needed to create the protocol. That the build process for $G(P_{1,15})$ uses less memory than for $P_{1,15}$ could be confirmed by multiple measurements. A reason for this could not be found.

Client and Server Performance Table 3 shows the performance overhead of using PP. Note that this is the overhead while processing messages. Thus, handling multiple messages sequentially *does not* multiply the load. A division in client and server is not necessary as PP can be implemented as a proxy that translates a list of protocol messages (cf. Section 3.4). The performance for handling connections is then that of the chosen reverse proxy. We divide the performance in sending and receiving messages. In doing so performance assessments are possible for servers that primarily receive or send more data. Send Time includes setting each field and serializing the message, Receive Time includes deserializing and accessing all fields. Time on the wire is not measured. The data size is the size of the serialized message. We choose random values for fixed-length fields and 100 random bytes for dynamic fields, e.g., strings. We only consider single messages since the time for multiple messages can be upscaled. We also do not include dummy fields, as they just increase the message size to the set value.

	$\lvert G(P_{1,15})$	$\lvert G(P_{1,75})$	$\lvert G(P_{1,150})$	$\lvert G(P_{1,225})$
Send Time	+118%	+79%	+117%	+180%
Send Memory	+130%	+178%	+150%	+187%
Send CPU Load	+85%	+71%	+83%	+66%
Receive Time	+41%	+98%	+116%	+125%
Receive Memory	+46%	+12%	+31%	+85%
Receive CPU Load	+85%	+71%	+83%	+66%
Data size	+23%	+19%	+24%	+27%

Table 3. Mean protocol performance without dummy fields. Difference of polymorphic protocol compared to the original protocol in percent (N=100)

The results show that PP are 2.09 times slower (around 1.5 ms) than the base protocol and cause approx 76% more CPU and 102% more memory load (average of send and receive performance). Thus, a service needs to spend approx two times more resources to handle the same amount of clients. This can be improved using optimizations described in Section 3.4.

The reason for this is that access to a field of a PP passes through two wrapper classes. First, the wrapper created by our implementation is called, which performs the transformation and additionally calls the wrapper created by protobuf. For the base protocol, the wrapper created by protobuf is directly used. Even though the PP are two times slower, this is in the area of single milliseconds and should be negligible for non real-time applications. When using PP around 1.2 times more data is used. Note that certain messages or fields, e.g., real-time messages or large byte arrays that are used to stream video data, can be excluded from being transformed, thus saving data and computation time.

4.3 Security Discussion

This section discusses the additional cost of an attacker against polymorphic protocols as well as the organizational effort for service providers to use them.

RQ2: Attacker Cost Estimating the costs for an arbitrary Man-At-The-End attacker is difficult and cannot be calculated accurately [1,26]. Many existing techniques that assess how much an obfuscation costs an attacker [4,5] cannot be applied to our scenario because they are targeted to a specific subset of reverse engineering techniques. In contrast, our approach is about increasing the cost of duplicating bots for an attacker and not targeted against any particular reverse engineering technique. Testing our approach against automated protocol reverse engineering approaches was also not possible, as many approaches focus on unencrypted protocols[28] and only a few implementations are available [5].

To estimate the effort required by an attacker, we compare our approach to the method currently used in practice, i.e., using API keys to encrypt and authenticate protocols. We consider the Full-Polymorphic and Time-Polymorphic deployment described in Section 3.4 as well as two different attackers.

Restricted-MATE Normal Man-At-The-End [1] attacker who has access to the binary. The attacker is restricted and cannot automatically extract API keys or the custom protocols (R-MATE).

Unrestricted-MATE MATE attacker, without any restriction and limitation of reverse engineering techniques (U-MATE).

The attacker wants to setup multiple bots for the service. The U-MATE attacker loses when it needs to put in manual work, as this already impedes the creation of a bot army. We divide the bot creation into the reverse engineering and bot writing phases.

R-MATE This attacker is probably most common since automatic extraction of data (API keys) and code (polymorphic protocol) is difficult on an engineering level, especially for encrypted protocols [28,18,22].

Using the Time-Polymorphic deployment the attacker needs to extract the protocol or API key within the period. After that, the bot has to be written within this period. Then the bot can be effectively duplicated and the attacker can exploit the service. After some time the attacker must repeat all the steps. Since the custom API key does not change the protocol, the attacker only has to replace the key in the existing bot program. Using a PP, the entire protocol of the bot needs to be replaced and all new transformations implemented. This indicates that the development time for PP is larger than for API keys. The argumentation for the Full-Polymorphic deployment strategy is analogues. Thus for this limited attacker PP increase the cost of duplicating bots. The analysis of the next attacker argues that since PP can include variable API keys, PP are at least as hard to extract as API keys alone.

[5] `https://github.com/techge/PRE-list`

U-MATE This attacker needs to automatically extract protocols from a given binary. Whether this extraction happens for each update (Time-Polymorphic) or multiple client binaries (Full-Polymorphic) does not matter. Thus, the two deployment strategies are equivalent for the attacker. This results in comparison on how much effort it takes to automatically extract an API key and how much effort it takes to extract the communication protocol. A PP can also contain encryption transformations and cryptographic keys. These keys can be hidden and protected just like the API keys. Thus, the extraction of API keys can be seen as a sub-problem of extracting a communication protocol and requires at least as much effort as extracting API keys only.

The most effective attacks on PP are code reuse and slicing techniques, where certain parts of a binary are reused or extracted, e.g, the communication protocol. The challenge here is to isolate and extract *minimal and executable* code responsible for transforming network messages. In the simplest case, the attacker just executes the whole application and uses dynamic binary instrumentation techniques, e.g., code injection and hooks [6], to directly call the desired functions. However, running the whole application is very resource-intensive and thus not suitable for creating a large number of bots. Extracting minimal and executable code is still an open research field and not completely reliable [26,9]. Note that defending against code reuse and slicing attacks is out of scope for our approach. There is a lot of research [9,4] in this regard, including commercial [25,27] and public tools [10], especially in the areas of digital rights management and anti-reverse engineering. Those countermeasurements, e.g., function virtualisation and just in time compiling, can be used together with our approach.

By using these techniques to hinder the attacker from using automated techniques and tools, the attacker remains with the capabilities of the *R-Mate attacker* and the approach can increase the effort to create a bot army. This shows that PPs slow down an attacker more than API keys.

RQ3: Organisational cost The appropriate deployment depends heavily on the needed security. Furthermore, optimizations discussed in Section 3.4 should be used to decrease the operational and organizational cost.

Full-Polymorphic In this setting, the remote backend must provide a matching endpoint for each custom protocol. A simple but resource-intensive way is to keep them up all in parallel. Another option is to spin endpoints up dynamically, however this requires development effort (cf. Figure 3). Due to the dynamic nature of custom protocols they cannot be served as a static file from CDNs, e.g., from the AppStore or PlayStore. A solution is to create a base application that dynamically loads and integrates protocols. This base application can be served again from CDNs (cf Section 3.4). While the protocol is then dynamically loaded, a large part of the application, i.e, all static assets like models, images, and videos can still be served from CDNs.

Time-Polymorphic This deployment strategy introduces almost no organizational cost. CDNs can be used as usual and only one endpoint needs to be provided. On the downside, the approach allows an attacker to create a bot

army for a certain time (cf. Section 4.3). Different to the Full-Polymorphic deployment, this approach burdens the user by requiring regular updates. This burden can be reduced by using approaches from Section 3.4.

The results show that the costs for a defender are not large but not marginal either. For consumer devices and applications that exchange only a few and rather small messages, the technical overhead should be well manageable. Especially since time critical messages or messages that transmit large amounts of data do not have to be transformed. The organizational effort for Time-Polymorphic deployments is minimal but impedes an attacker less (cf. Section 4.3). For Full-Polymorphic deployments, however, there is some organizational effort involved. This mode is more suitable for applications that suffer heavily from bots, otherwise, it does not justify the effort.

4.4 Limitations

First, while PPs make it harder to create bots for a service in comparison to API keys, it is not accurately determined *how much* harder it is for an arbitrary attacker. This is something the approach shares with other obfuscation techniques. Similar literature addresses this by limiting attackers to subsets of techniques, e.g., only static analysis. These attacker models however do not fit our objective. Determining the hardness could be approached in an empirical study, but would require access to a lot of reverse engineers. This limitation is compensated by the benefit that while the increased effort for an attacker cannot be accurately determined, the overhead for using PPs in their simplest form is low and can be easily be tested for a service. PPs are nonetheless an additional overhead and should only be used for services that are already having problems with bots.

Second, PPs are affected by slicing and code reuse approaches where an attacker can use the existing code of a client to build a bot. Therefore, PPs should be used together with anti-reverse engineering approaches that impede slicing and code reuse, e.g., [9].

5 Conclusion

While previous work focused on making it harder to build a first bot, we present an approach that fights the scaling of bots by forcing a bot creator to extract not just client API keys but the whole application protocol for each bot that is created. We evaluate the resources needed to build and use polymorphic protocols. Without optimization, polymorphic protocols transmit approx. 1.2 times more data compared to the base protocol and are 2.09 times slower (around 1.5 ms). Depending on the deployment the organizational effort required to integrate our approach into existing services is low as it can be used as a drop-in replacement. Considering MATE attackers polymorphic protocols introduce more cost for an attacker compared to using API keys alone. In the future, we would like to investigate the extent to which polymorphic protocols make it harder for an attacker to create bots, compared to API keys. This shall be evaluated as part of a CTF or as a study.

References

1. Akhunzada, A., Sookhak, M., Anuar, N.B., Gani, A., Ahmed, E., Shiraz, M., Furnell, S., Hayat, A., Khan, M.K.: Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. Journal of Network and Computer Applications **48**, 44–57 (2015)
2. Alqahtani, F.H., Alsulaiman, F.A.: Is image-based captcha secure against attacks based on machine learning? an experimental study. Computers & Security **88**, 101635 (2020)
3. Amin Azad, B., Starov, O., Laperdrix, P., Nikiforakis, N.: Web runner 2049: Evaluating third-party anti-bot services. In: Proceedings of the 17th DIMVA (2020)
4. Banescu, S., Collberg, C., Ganesh, V., Newsham, Z., Pretschner, A.: Code obfuscation against symbolic execution attacks. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. pp. 189–200 (2016)
5. Banescu, S., Collberg, C., Pretschner, A.: Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In: 26th USENIX Security 17. pp. 661–678 (2017)
6. Berdajs, J., Bosnić, Z.: Extending applications using an advanced approach to dll injection and api hooking. Software: Practice and Experience **40**(7), 567–584 (2010)
7. Bessi, A., Ferrara, E.: Social bots distort the 2016 us presidential election online discussion. First monday **21**(11-7) (2016)
8. Boda, K., Földes, Á.M., Gulyás, G.G., Imre, S.: User tracking on the web via cross-browser fingerprinting. In: Nordic conference on secure it systems. pp. 31–46. Springer (2011)
9. Cheng, X., Lin, Y., Gao, D., Jia, C.: Dynopvm: Vm-based software obfuscation with dynamic opcode mapping. In: International Conference on Applied Cryptography and Network Security. pp. 155–174. Springer (2019)
10. Collberg, C.: the tigress c obfuscator (2001), `https://tigress.wtf/about.html`
11. Dyer, K.P., Coull, S.E., Shrimpton, T.: Marionette: A programmable network traffic obfuscation system. In: 24th USENIX Security 15. pp. 367–382 (2015)
12. Gagnon, M.N., Taylor, S., Ghosh, A.K.: Software protection through anti-debugging. IEEE Security & Privacy **5**(3), 82–84 (2007)
13. Gummadi, R., Balakrishnan, H., Maniatis, P., Ratnasamy, S.: Not-a-bot: Improving service availability in the face of botnet attacks. In: NSDI. pp. 307–320 (2009)
14. Heath, N.: Expedia on how one extra data field can cost $12m. `https://www.zdnet.com/article/expedia-on-how-one-extra-data-field-can-cost-12m/` (2010), accessed: 2021-10-18
15. Karuppayah, S., Fischer, M., Rossow, C., Mühlhäuser, M.: On advanced monitoring in resilient and unstructured p2p botnets. In: 2014 IEEE International Conference on Communications (ICC). pp. 871–877. IEEE (2014)
16. Karuppayah, S., Roos, S., Rossow, C., Mühlhäuser, M., Fischer, M.: Zeus milker: circumventing the p2p zeus neighbor list restriction mechanism. In: 2015 IEEE 35th International Conference on Distributed Computing Systems. pp. 619–629. IEEE (2015)
17. Li, M., Chen, X., Li, X., Ma, B., Vitányi, P.M.: The similarity metric. IEEE transactions on Information Theory **50**(12), 3250–3264 (2004)
18. Liu, M., Jia, C., Liu, L., Wang, Z.: Extracting sent message formats from executables using backward slicing. In: 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies. pp. 377–384. IEEE (2013)

19. Liu, W.: Introducing recaptcha v3: the new way to stop bots. https://developers.google.com/search/blog/2018/10/introducing-recaptcha-v3-new-way-to (2018), accessed: 2021-05-20
20. Machines, I.: Stop more bots. start protecting user privacy. https://www.hcaptcha.com/ (2018), accessed: 2021-05-20
21. Mohajeri Moghaddam, H., Li, B., Derakhshani, M., Goldberg, I.: Skypemorph: Protocol obfuscation for tor bridges. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 97–108 (2012)
22. Narayan, J., Shukla, S.K., Clancy, T.C.: A survey of automatic protocol reverse engineering tools. CSUR **48**(3), 1–26 (2015)
23. Roundy, K.A., Miller, B.P.: Binary-code obfuscations in prevalent packer tools. ACM Computing Surveys (CSUR) **46**(1), 1–32 (2013)
24. Sivakorn, S., Polakis, I., Keromytis, A.D.: I am robot:(deep) learning to break semantic image captchas. In: 2016 IEEE EuroS&P. pp. 388–403. IEEE (2016)
25. Software, V.: VMProtect Software Protection (2021), https://vmpsoft.com/
26. Talukder, M., Islam, S., Falcarin, P.: Analysis of obfuscated code with program slicing. In: 2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security). pp. 1–7. IEEE (2019)
27. Technologies, O.: Oreans Technologies : Software Security Defined. (2022), https://www.oreans.com/Themida.php, accessed 2021-12-07
28. Wang, Z., Jiang, X., Cui, W., Wang, X., Grace, M.: Reformat: Automatic reverse engineering of encrypted messages. In: European Symposium on Research in Computer Security. pp. 200–215. Springer (2009)

A Generating a custom protocol

An example is given in the following. Consider a game with multiple messages. All messages are defined in *GameMessages.proto*. One message is StatusInformation. A part of the message definition is shown in Listing 1.1.

```
message StatusInformation {
  string name = 1;
  int32 playerNumber = 2;
  int32 plays = 3;
  ...
}
```

Listing 1.1. Untransformed protobuf excerpt

The protobuf file is then transformed by applying our implementation of Algorithm 1. The result is another protobuf file, including some transformations (Listing 1.2). The whole message is encrypted using AES with a random key. Some message types have changed and some variables have been split into two variables.

```
message StatusInformation { // Encrypted using AES
  string plays_p1 = 1;     /* before: int32 | p1 */
  int32 playerNumber = 2; /* unmodified */
  bytes name = 3;          /* before: string */
  bytes plays_p2 = 4;      /* before: int32 | p2 */
```

```
  ...
}
```

Listing 1.2. Transformed protobuf excerpt

The normal protobuf compiler uses the transformed protobuf file *GameMessages.proto* and generates methods to read, write and serialize the fields and messages in a file called *GameMessages.java*. Our implementation automatically adds classes that apply the transformations on top of the code generated from the protobuf compiler. If the supported feature set of protobuf has already been used previously the generated classes of our implementation are a drop-in replacement. This process is the same for the client as well as for the server of an application.

Appendix F

Encrypted Endpoints: Defending Online Services from Illegitimate Bot Automation

Abstract

Automated usage of web services by programs, known as bots, poses risks such as data scraping, spam, and cyber attacks. For instance, X suffers from millions of bot accounts typically controlled by relatively fewer adversarial organizations to create fake likes and comments. The most widely used solution to distinguish humans from bots (CAPTCHA) is perishing due to advances in machine learning. Obfuscation techniques in binaries, applications, or websites are designed to impede the creation of bots but fail to prevent their scalability. Bypassing these measures often requires only a one-time effort. We propose encrypted endpoints as a novel strategy to combat the scalability of web bots, particularly in scenarios where bots leverage multiple accounts. For that we assign unique endpoints (URLs) to each user account, thereby restricting bot applicability across different accounts and necessitating the extraction of account-specific endpoints per bot instance. Our approach is applicable to a wide range of services utilizing endpoints, including desktop and mobile applications, web applications, and even static or HTML-only websites. We implemented our approach directly within a backend framework and observed that the latency overhead is less than 0.1ms per request, which constitutes less than 1% of the total request time. Our solution, developed as simple middleware, can be easily integrated in existing projects with low effort. Additionally, we have extended our approach to the Jinja2 template engine, thereby supporting encrypted endpoints for websites out of the box. Our analysis indicates that our approach not only effectively protects against simple bots but also, when coupled with obfuscation techniques, further impedes bot creation.

Reference

August See, K. Röbert, M. Fischer. Encrypted Endpoints: Defending Online Services from Illegitimate Bot Automation. International Symposium on Research in Attacks, Intrusions and Defenses, 2024.

Contribution

The solution approach and full implementation and evaluation for this publication originated from the author of this dissertation. The second author is credited with the core idea. The second and third authors contributed to the refinement of the work.

Encrypted Endpoints: Defending Online Services from Illegitimate Bot Automation

August See
Universität Hamburg
Hamburg, Germany
richard.august.see@uni-hamburg.de

Kevin Röbert
Universität Hamburg
Hamburg, Germany
kevin.roebert@uni-hamburg.de

Mathias Fischer
Universität Hamburg
Hamburg, Germany
mathias.fischer@uni-hamburg.de

ABSTRACT

Automated usage of web services by programs, known as bots, poses risks such as data scraping, spam, and cyber attacks. For instance, X suffers from millions of bot accounts typically controlled by relatively fewer adversarial organizations to create fake likes and comments. The most widely used solution to distinguish humans from bots (CAPTCHA) is perishing due to advances in machine learning. Obfuscation techniques in binaries, applications, or websites are designed to impede the creation of bots but fail to prevent their scalability. Bypassing these measures often requires only a one-time effort. We propose *encrypted endpoints* as a novel strategy to combat the scalability of web bots, particularly in scenarios where bots leverage multiple accounts. For that we assign unique endpoints (URLs) to each user account, thereby restricting bot applicability across different accounts and necessitating the extraction of account-specific endpoints per bot instance. Our approach is applicable to a wide range of services utilizing endpoints, including desktop and mobile applications, web applications, and even static or HTML-only websites. We implemented our approach directly within a backend framework and observed that the latency overhead is less than 0.1ms per request, which constitutes less than 1% of the total request time. Our solution, developed as simple middleware, can be easily integrated in existing projects with low effort. Additionally, we have extended our approach to the Jinja2 template engine, thereby supporting encrypted endpoints for websites out of the box. Our analysis indicates that our approach not only effectively protects against simple bots but also, when coupled with obfuscation techniques, further impedes bot creation.

CCS CONCEPTS

• **Security and privacy → Web application security**; *Software reverse engineering*.

KEYWORDS

web bots, obfuscation, endpoints

ACM Reference Format:
August See, Kevin Röbert, and Mathias Fischer. 2024. Encrypted Endpoints: Defending Online Services from Illegitimate Bot Automation. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30-October 02, 2024, Padua, Italy.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3678890.3678918

RAID 2024, September 30-October 02, 2024, Padua, Italy

ACM ISBN 979-8-4007-0959-3/24/09
https://doi.org/10.1145/3678890.3678918

1 INTRODUCTION

The automated use of Internet services is an integral part of the Internet and the Web, especially as web services increasingly depend on each other. This ranges from retrieving resources from CDNs to price comparison portals to unauthorized automation, e.g., by bots in social media. In the following, the unauthorized automated use of a service by scripts or computer programs is called a *bot*. In addition, we focus on bots that gain an advantage by creating many accounts.

While automation is a drive of our Internet, some services should not be automated, as this can lead to financial and even social damage. Bots significantly increase the load of services (bad bots 27.7% and automated traffic in total 42.3% in 2021 [2]) and thus increase infrastructure costs. This applies to all services, but there are certain services where bots can do greater harm. Social media, for example, is intended for human users only. Bots that automatically create or control multiple accounts can be used on a large scale to spread false information and opinions. This does not only increase infrastructure costs but also affects the satisfaction of human users. Beyond that, bots have been successfully used in the past to influence elections [9].

Utilizing the Application Programming Interface (API) or endpoints of a service is the most efficient method for bot development, known as API/endpoint-based bots. These bots, which operate through HTTP requests from scripts or automated browsers, can be easily created using automated tools [24, 34]. These tools facilitate extracting service interaction data, simplifying session replays. Given their prevalence, our paper focuses on defenses against these bots, as discussed in Section 2.

The main problem of bots is something all bot types have in common. Once written, they are easy to duplicate and thus to scale. For example, a X bot that automatically likes everything with a specific tag, e.g., *#conference* will work for every account. This ability to scale is what makes bots so dangerous. While one bot likely will not have much negative effect, an army of bots will. Using the endpoint directly is most natural and scales better than automation via the user interface, so we focus on how such bots can be restricted.

More and more companies are using anti-bot solutions, such as CAPTCHA or obfuscation approaches.CAPTCHAs are the most popular defense against bots. They use problems that are easy for humans to solve but difficult for computers. However, the number of problems that fall into this category is decreasing with the advancement of machine learning [7, 35]. Additionally, CAPTCHAs

negatively impact the user experience and waste time, even when modern risk assessment approaches do not prompt every user with a CAPTCHA challenge [1]. As a result, obfuscation approaches are used to hinder the creation of bots by making it more difficult for bots to extract information [40], such as URLs, email addresses, or information about the availability of specific products, e.g., graphic cards. However, many bots do not need to extract information, e.g., those that replay sessions or that use the API directly. For example, for spamming comments a bot needs to know the endpoint location (URL) and what data is accepted. Existing obfuscation approaches leave this aspect out because it changes the way the web works. While, our proposed obfuscation approach addresses this gap.

Our main contribution are unique encrypted endpoints so that bots cannot be scaled easily anymore. In more detail, we make the following contributions:

- We introduce encrypted endpoints to hinder the scaling of bots. This is achieved by assigning unique endpoints to each account in a service. Thus, every bot is only valid and usable for a specific account. We call this encrypted endpoints. Existing obfuscation techniques for Binaries [12] or for HTML and JavaScript [14, 40, 44], obfuscate the "application" itself but not the endpoints. Thus, they offer no protection against bots that use the endpoints directly, e.g., using python requests or which replay previously recorded sessions [24, 34]. Our approach and code obfuscation work in tandem, complementing each other. While our approach safeguards against session replays and API bots, code obfuscation adds an extra layer of complexity, rendering the extraction of client-specific encrypted endpoints more challenging. By altering the endpoints for each client periodically, the effort required by an attacker can be further increased. It is worth noting that while our approach was initially designed for accounts, it is also applicable to services without a login feature.
- We purpose methods to enhance the usability of our approach for both users and service providers. So, despite unique URLs, those URLs can still be shared between users without impacting the efficacy against bots. Further, service providers do not have to predetermine all of their endpoints to utilize our approach.
- We implement the approach as a middleware for FastAPI and the Jinja2 templating engine. The source can be found here[1] as well as simple demonstration in form of a video.
- We evaluate our approach's performance and organizational overhead and discuss the additional effort required for attackers to scale bots.
- As an added benefit, our approach can protect against directory traversal attacks and attacks that rely on guessing or injecting data into the path or parameters, as our method renders URLs non-guessable and user-specific.

Note that the primary objective is to increase the effort to scale bots by preventing a bot to be used on a different account, and not the creation itself. While our approach alone achieves this, it **heavily** benefits from being used together code obfuscation to make the extraction of client-specific encrypted endpoints more challenging (cf. Section 5.1). Legitimate bots, security testing, and interoperability across different services are still possible, e.g., by providing special API keys after thorough verification.

The remainder of this paper is structured as follows. Section 2 introduces our threat model, specifying the types of bots we aim to protect against, and describes the attacker model used throughout this paper. Section 3 outlines the requirements for bot defense and reviews related work in the field. Section 4 describes our approach to encrypted endpoints and discusses potential optimizations to restore functionalities such as link sharing. Section 5 presents our evaluation of the proposed approach and discusses its effectiveness in defending against the specified attacker model. It also explores the limitations of our approach. Section 6 doutlines the requirements for code obfuscation techniques that can be integrated with our method. Finally, Section 7 concludes the paper.

2 THREAT AND ATTACKER MODEL

In our threat model a service is accessed by bots. The service either cannot or chooses not to depend on more robust user authentication methods, such as phone verification or presenting personal identification documents. This decision is grounded in realism, as authentication processes that introduce friction tend to deter users from engaging with the service [18]. The service has already implemented account-bound rate limiting, ensuring that actions like purchases and upvotes are constrained within certain limits per account. This setup necessitates a logical reason for bot creators to create multiple bots, each controlling distinct accounts. This scenario is common across various types of websites, including those in social media, e-commerce, and gaming. However, for websites that solely provide information, our approach is not applicable.

For example, on most social media platform, content visibility is influenced by upvotes. Each account is permitted to vote only once, making it advantageous for bot operators to control multiple accounts for the purpose of artificially boosting the visibility of specific content through coordinated upvoting.

2.1 Considered Bots

Our defense mechanisms against bots, specifically targets API-based bots. Our rationale is that API-based bots are prevalently used, easily scalable, and consequently represent a substantial threat. Delving further into the matter:

API-based bots are simpler to develop and maintain because they interact directly with a service's endpoints, circumventing the complexities associated with graphical user interfaces (GUIs). In contrast to GUI elements that are prone to frequent changes, API endpoints, particularly those that are versioned (for example, "/v1/user"), provide a stable interface for automation. This stability diminishes the necessity for continual updates in contrast to bots operating on the UI. However, with current technological advancements, UI-based bots are expected to become more adaptable shortly.

Furthermore, API-based bots have the advantage of automated tools that facilitate the creation of bots by extracting endpoints and data from service interactions. These tools enable straightforward session replays [24, 34], making API/endpoint-based bots a more viable option for our research, which emphasizes efficient and stable automated interactions.

[1] https://github.com/8mas/encrypted-endpoints

API-based bots also require significantly less computational resources compared to their UI-based counterparts, which need to render graphical interfaces and are tightly bound to the underlying program and UI. For each instance of a UI-based bot, a separate interface instance is required, leading to escalated resource consumption and higher scaling costs. This contrast is especially pronounced in contexts like gaming and mobile applications, where UI-based bots are resource-intensive, and scaling becomes more costly.

2.2 Attacker Models

We define two attacker profiles:

Endpoints Only (EO) Attacker. : This EO attacker strategy focuses solely on utilizing service endpoints to develop bots. By recording network traffic, attackers can use tools such as *mitmproxy2swagger* and *charles-extractor* [24, 34] to extract endpoints and data formats from the session, facilitating session replay and bot creation with minimal technical expertise required. This approach bypasses the need to parse desktop binaries, mobile applications (APKs, IPAs), or web elements (HTML, CSS, JavaScript), offering a streamlined method to create efficient bots.

Endpoints and Data Parsing (EP) Attacker. : Contrary to the first attacker model, this EP attacker involves utilizing service endpoints and processing and parsing available data. In the context of mobile and desktop programs, this includes binaries and apps (APKs, IPAs), which are inherently difficult to *parse* and often require reverse-engineering using disassembly and debugging tools. In the context of the web, this encompasses server responses in formats such as HTML, XML, and JavaScript. This method is typically adopted for services that lack a comprehensive API. Direct interaction with specific endpoints to obtain structured data, such as JSON (e.g., "/v1/product/id" providing product prices and availability), is generally more straightforward and reliable than parsing semi-structured data like HTML. The latter, often relying on tools such as XPath or CSS selectors, is more susceptible to errors and complications due to potential changes in the data's structure or format [26].

3 REQUIREMENTS AND RELATED WORK

3.1 Requirements for Bot Defense Approaches

We have identified several requirements for bot defense approaches:

- *Transparent to user*: The approach should not disrupt the user experience, thus avoiding time wastage through mechanisms such as captchas [1] and preventing loss of revenue due to additional hurdles in website navigation [18].
- *Zero data collection*: No data should be collected to differentiate between humans and bots, thereby ensuring privacy. This includes overt measures like requiring an ID to use a service, as well as subtler techniques that collect data such as IP addresses and mouse movements [27].
- *Seamlessly integrable*: The approach should be seamlessly integrable into existing services and codebases with minimal effort.
- *Small performance overhead*: The approach should have minimal impact on the performance of the service, ensuring it remains usable.
- *UI and domain agnostic*: The approach should be UI-agnostic to allow versatile application across different user interfaces. Additionally, it should be domain-agnostic, suitable for use in various contexts such as desktop applications, mobile apps, or HTML-only websites.
- *Resistance against attackers (EO, EP)*: The approach should effectively counter the described attacker models (EO and EP).
- *Open source*: Ideally, the approach should be open source to ensure developers can adapt it as needed.

3.2 Related Work

We divide the related work into three categories. Proving Humanity, Anomaly Detection, and Anti-Analysis. The approach we take in this paper falls into the Anti-Analysis category. A high-level overview of how our approach compares to related work is given in Table 1.

Requirements	CAPTCHA [27, 28]	Brewer et al. [11]	See et al. [32]	Code obfuscation. [12, 36, 39]	User specific API keys	Facebook Anti-Tracking[3]	Our System
Transparent to users		●	●	●	●	●	●
Zero data collection		●	●	●	●	●	●
Seamlessly integrable	●	●		◐	◐		●
Small performance overhead	●	●	◐	◐	●	●	●
UI and domain agnostic	◐	◐	◐	◐	◐		●
Resistance against attacker (EA)	●		●		●		●
Resistance against attacker (EDA)	●		◐[2]		◐[2]		◐[2]
Open source		●	●	◐[3]	◐[4]		●

◐ partially fulfilled ● fulfilled

Table 1: Related bot defending techniques.

Proving Humanity. Approaches in this category aim to prove a user's humanity directly, for example, through possession or human knowledge. The strongest proof that a user is a human would be requiring an official government-issued ID. While this would certainly make it hard to scale bots, it is also a privacy-unfriendly requirement. Even if users did not care about privacy, it would disrupt the user experience, directly impacting companies' revenues [18]. A very well-known, and at present, the standard way to prove humanity is CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) [27, 28]. CAPTCHA relies on problems that are hard to solve for computers and easy for humans, such as reading distorted characters or selecting specific images.

[2]Only when coupled with code obfuscation approaches.
[3]Some implementations like Tigress are available.
[4]This is more a generic technique than an implementation.

This approach, however, has some serve limitations and is becoming less and less effective. First, CAPTCHA breaks the user experience. Second, the time wasted to prove humanity is 500 years each day, alone for CAPTCHA issued by Cloudflare [1]. To deal with this, there is research to decrease the number of issued challenges to users, e.g., privacy pass [13]. In addition, for each correctly solved CAPTCHA, the users receive some tokens. These tokens do not leak information about the user and can be used to bypass CAPTCHA. However, the main problem of CAPTCHA is that the number of problems that are hard to solve for computers and easy to solve for humans is decreasing because of the advancements in machine learning [19, 25, 35].

To address the problems of CAPTCHA, there are developments to attest humanity through cryptographic routines, Trusted Plattform Modules (TPMs), and Trusted Execution Environments (TEEs). The idea is that by possessing a rare resource, e.g., security keys [43], Iphones [20], etc. CAPTCHA no longer has to be solved. Here it must be ensured that no information about the actual user is leaked, but also that a simple (virtual) transfer of the resource is not possible. These approaches [20, 43] are compatible with the pricacy pass [13] protocol and are designed to avoid displaying CAPTCHA altogether.

Anomaly Detection. These approaches try to detect bots through their characteristics, e.g., user-agent or display size and behavior. CAPTCHA providers use these approaches for an initial risk assessment to decrease the number of CAPTCHA presented to users [27, 28]. A harder or no CAPTCHA is presented depending on the calculated score. However, characteristics such as IPs, user agent, resolution, and cookies are needed to calculate the risk. This information can fingerprint the system and track users across multiple websites. In addition, mouse movements and keystrokes allow the user to be identified, not just the browser or system [5, 33]. This is a major criticism of current CAPTCHA systems.

Brewer et al. [11] introduce another way that resembles the idea of honeypots. Every link on a web page is surrounded by fake links that are not visible to the user. A bot that tries to crawl or scrape the website will likely visit a fake link and is thus exposed. In contrast to other anomaly detection approaches, this is very privacy-friendly and stands out positively due to low false positives.

Last, anomaly detection approaches for bot detection suffer from one common problem. If a bot behaves exactly like a human user (characteristics and behavior), distinguishing between humans and bots is no longer possible. However, forcing a bot to behave like a human is a great achievement since it increases the cost of creating undetected bots. Bot writers then need to consider the characteristics and behaviors of humans on a website, and the written bots are not as effective as they could be. However, these advanced bots will not be detectable through anomaly detection alone.

Anti-Analysis. These approaches aim to complicate the creation of bots and increase the costs associated with extracting information, thereby aligning with the objectives of this paper. We describe the requirements for code obfuscation used in conjunction with our approach in Section 6.

The primary challenge in anti-analysis engineering lies in quantifying the efficacy of a technique against an undefined attacker [6, 8, 38]. When an attacker fully controls a device, all anti-analysis techniques can merely elevate the cost for an attacker to reverse engineer an application without the possibility of completely preventing it.

In the context of web bots, obfuscation techniques serve to complicate the extraction of essential information required for bot creation, such as accepted protocols, endpoints, or API keys. Numerous methods exist both for data extraction from a program and for its obfuscation.

However, more than obfuscating the location of elements is required. For many bots, e.g., spambots, only the endpoint and the data format are needed. Consider Listing 1 website that displays a URL with GET parameters and a form. Even if the location is obfuscated (cf. listing 2), the endpoint (`example.com/api`, `example.com/api/user`) and the parameters (`param1`, `name1`) are always the same. Thus, once known, e.g., through recording the traffic, they can be used indefinitely, and the bot can be scaled easily.

Listing 1: Unobfuscated HTML

```
1 <a id="link1"
      href="example.com/api?param1=hello">Link1</a>
2 <form action="example.com/api/user" method="POST">
3     <input name="name1" id="input1" value="world"/>
4 </form>
```

Listing 2: Obfuscated HTML (IDs, XPath)

```
1 <form action="example.com/api/user" method="POST">
2     <input name="name1" id="random2" value="world"/>
3 </form>
4 <a id="rand1"
      href="example.com/api?param1=hello">Link1</a>
```

There are many approaches that address code obfuscation in the web context. These include academic papers, free-to-use tools [10, 23, 45], and commercial software [21, 22, 29]. Some commercial solutions offer comprehensive packages that handle all three aspects at once [21, 29]. While most approaches claim minimal overhead, they often do not evaluate it.

It is ideal to have an approach where the resource cost of creating the obfuscation is low, and ideally, each client receives its own obfuscated version of the website. While some approaches are non-deterministic, the majority are deterministic, necessitating an additional randomization step. However, websites can be pre-obfuscated, and the obfuscated versions can then be distributed to clients. Most tools are user-friendly and do not require extra configuration for the frontend. Typically, users need only to give their final HTML, JS, or CSS files as input, which are then obfuscated.

A paper analyzing the top 10K Alexa websites found that less than 0.4% use obfuscation on JavaScript [30] and 68.8% use minimization. Another paper analyzing the top 100K Alexa websites found that 0.67% of scripts are obfuscated, but 38% are minimized. While these percentages are not high, they indicate that code obfuscation is employed on some of the most popular websites.

Vikram et al. [40] address this. They build a tool, NOMAD, to defend against web bots without breaking the website for human users. The tool randomizes the Name and ID parameters of HTML form elements for each session. Thus, forcing the bot to extract the correct names and IDs for each session. However, this is limited to only forms and does not apply to (GET) parameters.

Wang et al. [41] introduce WebRanz, a novel mechanism for circumventing ad-blockers by employing randomization techniques to mutate HTML elements and their attributes without affecting the visual appearance or functionality of web pages. This approach invalidates the pre-defined patterns utilized by ad-blockers, thereby enabling content publishers to deliver advertisements effectively. WebRanz also provides a defense against web bots that manipulate DOM objects using similar pattern-matching techniques. The authors evaluate the system on 221 Alexa top web pages and eight bot scripts, demonstrating that WebRanz successfully evades ad-blockers and mitigates the impact of bot scripts with minimal overhead. It is a promising candidate to use in conjunction with our approach to counter the *Endpoint and Data Parsing Attacker*.

See et al. [32] follow a similar path. Their approach assigns a new application protocol for every user. Thus all protocol messages are unique, and bots cannot be scaled. The main problem of this work is that it is not easily usable for HTTP, and to be lightweight needs much fine-tuning.

The examples provided thus far have predominantly pertained to the web context. However, the same principles apply to binary obfuscation techniques. Examples of sophisticated software protection systems include Tigress [12], VMProtect [36], and Themida [39]. These systems are directly applied to the source code of an application, yielding a protected executable that utilizes a range of anti-reverse engineering and obfuscation techniques, thus complicating the extraction of information such as endpoints from compiled binary. However, despite the sophistication of these systems, they struggle to effectively obfuscate endpoints. This limitation arises because data transmission to these endpoints can be observed, for example, by utilizing a system-wide proxy. Consequently, bots can be scaled with relative ease once again.

User-specific API keys that are located in the client application and CSRF tokens might be used to impede bot creation, but original serve distinct purposes. For example, API keys may be used for identifying and authenticating a user across sessions, providing a persistent form of security, whereas CSRF tokens are designed to protect against cross-site request forgery attacks by ensuring that each request to a server is accompanied by a unique token, verifying the request's legitimacy. These mechanisms can be considered predecessors or foundational elements for our technique aimed at impeding the scaling of bots. Essentially, our method applies the concept of a CSRF token across a broader domain, compelling bot creators to extract it to make server-accepted requests. API keys in applications can offer similar functionalities by requireing an attacker to first reverse engineer the API key to authenticate custom requests. Our design, in contrast, is more versatile than CSRF tokens or API keys. It does not require an execution mechanism (like necessary for API keys to authenticate data), nor is it limited to HTTP. Further, it can operate statelessly. The only necessity is an endpoint identifier, which could include, but is not limited to, URLs. Our methodology and API keys are complementary, not exclusive. Integrating them can further obscure not only the endpoint but also the transmitted data, thus forming a comprehensive defense against direct and indirect attacks.

Facebook is using encrypted URLs to combat URL stripping [3]. Every parameter of a URL is encrypted and signed. Thus, it is no longer possible to drop tracking parameters without invalidating the whole link. As this technique is most similar to our approach, we provide a side by side comparison in Table 2.

Feature	Facebook's Implementation	Our Implementation
Goal	Prevent addons like Clear URL from dropping certain tracking parameters, as they break the link.	Prevent scaling of bots by requiring them to extract URLs from the source or binary.
Method	Encrypt and sign parameters using (likely) a general key, not client-specific.	Encrypt and sign both path and parameters using a client-specific key.
Security	–	Prevents techniques like directory traversing, guessing paths, local file inclusion.
Availability	–	Free and open source.

Table 2: Comparison of URL Protection Strategies

4 ENCRYPTED ENDPOINTS

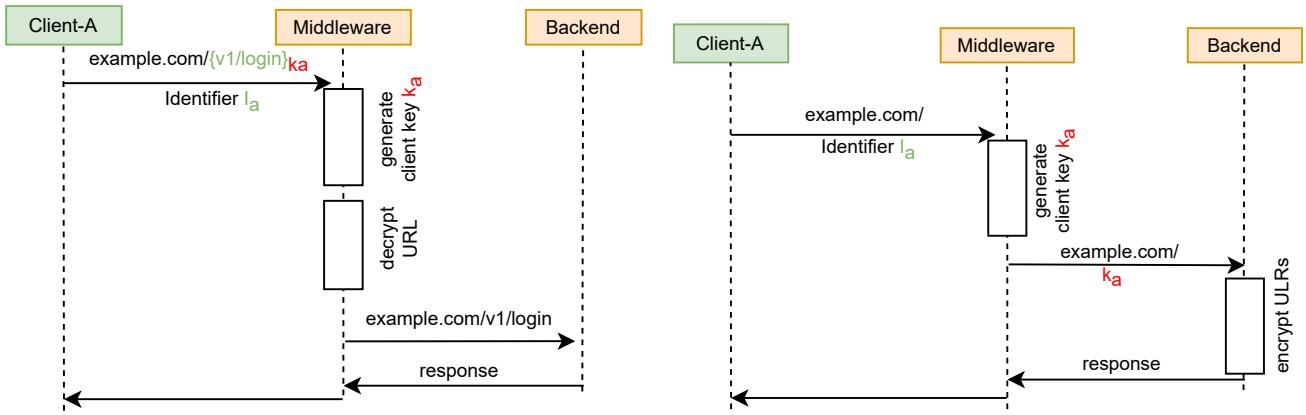
The core idea to limit the scalability of endpoint-based bots is the usage of encrypted endpoints. Every client (user), e.g., distinguished by session cookie or IP, receives a version of the same website with unique URL-Paths and parameters. Paths and parameters are encrypted, signed, and only valid for one client. If a bot is created, it cannot be duplicated easily, as the URLs are only valid for the particular client instance and, i.e., the bot. To create multiple bots, the bot creator is forced to extract the custom URLs from the website. This effectively prevents bots that rely on the simple replay of data [24, 34]. While extraction of URLs is trivial in normal cases, this can be made considerably more difficult by using obfuscation approaches from related work like [11, 14, 40]. Since URL paths and parameters are signed, any modification to them can be detected. A live demonstration of this concept has been implemented in a web application[5]. It is important to note that while the example provided is a web app, for ease of sharing and demonstration purposes, the underlying principle is equally applicable to both computer and mobile applications.

4.1 Basic Approach

Figure 1 outlines the proposed methodology. The encrypted endpoints middleware, which can be integrated with the server, receives requests from clients. The middleware generates a secure client key from the client identifier (e.g., a user ID) as detailed in Section 4.3.1. We assume that this identifier is non-forgeable.

In a typical scenario as seen in Figure 1a, the client sends a request to an encrypted endpoint, such as a URL, accompanied by its client identifier. The middleware, upon receiving this information, generates a client-specific key to decrypt the URL. If decryption is successful, the URL is forwarded to the backend server, which then retrieves the response and sends it back through the middleware

[5]https://github.com/8mas/encrypted-endpoints



(a) Encrypted Endpoint Usage Backend Only

(b) Encrypted Endpoint Usage in Context of Webpages and HTTP

Figure 1: Overview of Encrypted Endpoint Usages

to the client. It is important to note that clients are incapable of generating valid URLs independently; instead, the middleware and backend collaborate to create and distribute these URLs to the client, as illustrated in Figure 1b. In this process, the initial request from a client is directed to an unencrypted endpoint, for instance, *example.com/*. The middleware subsequently generates the client key and relays the request to the backend. The backend server then retrieves the resources and encrypts all URLs within these resources using client-specific encrypted endpoints.

The encrypted response is then transmitted back to the client, enabling the use of these encrypted endpoints as shown in Figure 1a. Navigation on the client's side proceeds normally, with additional URL requests being validated and decrypted by the middleware. For applications such as desktop and mobile apps, these endpoints can be pre-populated during compilation or installation, enhancing the security and functionality of the system.

Figure 2 illustrates the efficacy of encrypted endpoints in mitigating bot activities and preventing accidental exposure of sensitive files. In Figure 2a, Client-B attempts to access a URL specifically generated by the server for Client-A. As part of this process, Client-B sends its identifier, prompting the middleware to generate the client-specific key k_b and attempt to decrypt the accessed URL. This attempt fails due to a Message Authentication Code (MAC) mismatch, as the URL was encrypted using k_a, resulting in an error response. This mechanism ensures that bots configured for one account cannot be repurposed for another, enhancing security. However, it is important to acknowledge the unintended consequence of inhibiting link sharing, a potentially undesirable outcome. We address this issue in Section 4.3.3.

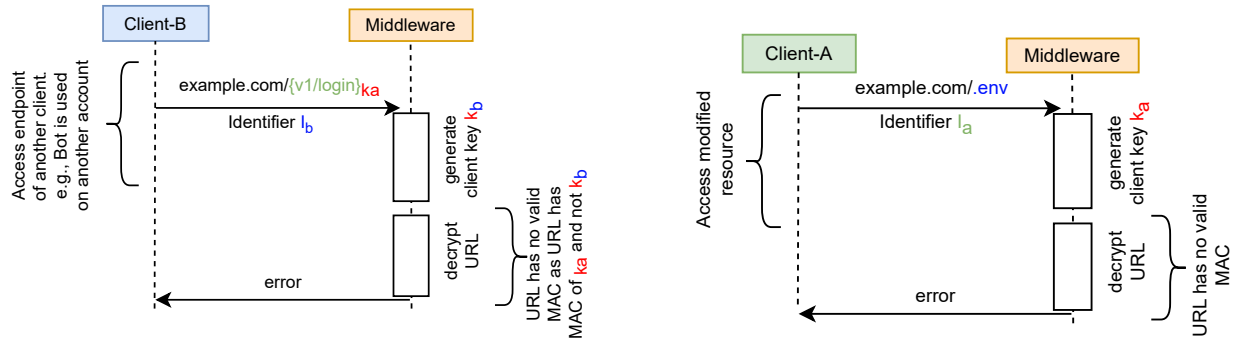
Figure 2b depicts a scenario where a client attempts to access a URL not issued by the server, such as during an attack aimed at discovering sensitive files through techniques like environment variable file scanning or directory traversal. The principle remains consistent with the previous example, where the middleware generates k_a based on the client's provided identifier. Since the accessed URL lacks a valid MAC, an error is returned, and access to the resource is denied. This feature of our approach effectively acts as a URL whitelist, allowing access only to URLs returned by the server to the client. Consequently, this strategy significantly reduces the risk of resource guessing, scanning for sensitive files, and mitigates certain attack vectors such as directory traversal and SQL injection in specific contexts (refer to Section 5.3).

This construction compels bot creators to dynamically search for the current endpoints, representing a significant step forward in impeding bots that rely on hardcoded or memorized endpoints [24, 34]. Nevertheless, extracting endpoints from sources such as HTML, binary code, software, Java, applications, or similar materials is not inherently challenging. Therefore, our approach gains a considerable advantage by incorporating obfuscation techniques that increase the difficulty of identifying specific endpoints.

It is essential to recognize that relying solely on code obfuscation is inadequate for deterring bots. Automated tools, as mentioned in [24, 34], can efficiently extract all necessary endpoints and data from a recorded session, facilitating the easy creation of session replays and bots.

4.2 Formal Model

An application, e.g., a website W contains a list of URLs, $U = \{u_1, ..., u_n\}$. We filter URLs from this list, that either cannot be encrypted, e.g., if the resource is located on a third party location or should not be encrypted, e.g., if it is a public shared URL (cf. Section 4.3.3). While normally URLs contain more, e.g., the scheme, for simplicity, our URLs only contain a path p and a set of parameters a. Any operation on a URL u_i is on the concatenated string of path and parameters, i.e., $u_i = p_i||a_i$. A client c has a unique identifier I_c. The server has a main key k_m. Using a key derivation function $KDF(k_m, I_c)$, the server generates a client key k_c. If identifiers are reused between clients, e.g., IP addresses, a nonce that is regularly changed should also be included in the KDF. The client key encrypts and authenticates each URL $u \in U$ of a website using authenticated encryption (AE) like AES-GCM, which return the encrypted message e as well as a Message-Authentication-Code (MAC) t_e. The URL line must not leak anything about the URL's semantics. Thus,



(a) Invalid URL for Client B because the URL was generated for Client A, rendering the MAC invalid.

(b) Invalid URL due to the client's direct access attempt on a URL not issued by the server.

Figure 2: Overview of Potential Errors and the Effectiveness of Encrypted Endpoints in Enhancing Security.

a MAC alone is not enough. Since URLs might be differentiable by their length, padding can be used to increase the length of URLs.

$$u' = \langle e, t_e \rangle = AE_{k_c}(u) \tag{1}$$

In this equation, u' is the newly constructed URL that replaces URL u. The encrypted path and parameters are e, and t_e is the message authentication code of e. Both are constructed using the client key k_c, only known by the middleware. The client then receives a version of the web page W' where every URL $u_i \in U$ is replaced with u'_i. In the subsequent figures, we employ the shorthand notation $\{url\}_{key}$ to denote authenticated encryption of URLs.

This construction is stateless, i.e., the middleware does not need to save any client keys as they are derived from the client identifier sent with every request. Note that encrypting URLs is necessary, and a MAC is not sufficient because URLs can be identified by their paths and parameters (cf. Section 4.3.1 and Section 5.1). A stateful construction can save CPU time but uses space as compensation, i.e., each custom URL must be stored for each client.

As long as the client identifier is not easy to share and secure cryptographic primitives are used, these constructions allow URLs to be used only by the respective client. No URLs can be created that the server did not create itself.

4.3 Using Encrypted Endpoints

This section describes how encrypted endpoints could be used productively. The choice of the client identifier, possible error handling, and possible optimizations are discussed.

4.3.1 Choosing the Client Identifier. We operate within a designated threat model, where mechanisms such as rate limiting are implemented (cf. Section 2). The client identifier, depicted in Figure 1 serves as the foundation for generating the corresponding client key, a crucial element utilized in URL encryption. To preempt duplication, the client identifier must be associated with a resource that proves challenging to replicate or that can be subject of rate limiting. For example, one can achieve this by constraining account activities and limiting actions based on User IDs, IP addresses or browser fingerprints. This safeguard is necessary since the URLs generated will be identical for a given client identifier.

There are several ways to choose client identifiers or even to combine different factors. Here, we will discuss the advantages and disadvantages of the identifiers User ID, IP address and browser fingerprint. However, combining different identifiers can improve the robustness of the system.

Session Cookie / User ID In scenarios where the service incorporates encrypted endpoints alongside a login mechanism, a user ID or session cookie is used to identify a client. This setup ensures that URLs are customized for individual accounts, thereby anchoring bots to specific accounts.
Note that although duplicating a session cookie is feasible, this offers no advantage to the attacker. The subsequent bot would be confined to the same account without any incremental capabilities. For example, it could neither cast additional upvotes on the same post on a social media platform nor purchase more products than its predecessor. Despite the potential feasibility of generating valid session cookies, perhaps through automated registration, the differentiation provided by unique user IDs mandates that bots be specifically tailored to their respective accounts.However, is its reliance on a login feature, which might not be applicable for all services. In such cases, alternative identifiers should be considered.

IP Address The IP address constitutes a practical identifier for a broad range of applications, automatically accompanying each request. To prevent IP address spoofing, this identification method relies on the exchange of multiple packets, such as those involved in a full TCP handshake, which helps deter trivial alterations of the source IP. However, the use of IP addresses as client identifiers presents challenges, such as the invalidation of all URLs upon an IP address change—occurring when a user switches from mobile data to Wi-Fi—due to the resultant alteration in the derived client key. Strategies to address these concerns are discussed in Section 4.3.3. It is crucial to enforce a per-IP rate limit to thwart the repeated initiation of a single bot from the same IP. Additionally, the coexistence of multiple devices behind

NAT, sharing an IP address, necessitates the combination of IP and port for a more effective client identifier construction.

Device/Browser Fingerprint Utilizing a device or browser fingerprint generated on the client side serves as an effective method for identifying users accessing the service through an app or browser. This technique surpasses the limitations of easily spoofable attributes, such as browser version and installed fonts, by incorporating at least one distinctive and difficult-to-replicate feature. For instance, leveraging a canvas fingerprint allows for the unique identification of devices based on their graphics processing capabilities [31]. The primary challenge with this method is the reliance on executing client-side code, such as JavaScript, to obtain the fingerprint. This necessity might restrict its applicability in environments where client-side code execution is disabled.

While numerous techniques exist for creating an identifier, the optimal solution is undoubtedly the use of a user/client/account ID, generated after login. This method is inherently unique to each client, thereby circumventing issues associated with IP-based identifiers, which may change when switching networks.

For services not requiring authentication, a combination of the non-forgeable IP address and browser fingerprint is potentially the most effective approach. This is because, even if the IP address changes, it does not necessarily disrupt the user's session.

In the subsequent sections, we will focus on addressing challenges related to URL sharing and session resumption.

4.3.2 Partial Encrypted Endpoints. Our methodology is optimally applied to endpoints that are fully known beforehand by the server. However, this is not always feasible. For instance, search fields that accept arbitrary user input and incorporate it into a parameter present a challenge. In such cases, only the parts of the URL known to the server in advance can be encrypted, while the dynamic, user-controlled portion must be appended. This approach necessitates distinguishing between encrypted and unencrypted parts, which can be achieved, for example, by using delimiters or length fields.

If the application cannot modify itself at runtime, such as HTML without JavaScript, and defense against the EP attacker is required, no static part of the URL should remain unencrypted. This could be a vector for differentiating URLs. Furthermore, URLs should be padded to equal or random lengths so that an attacker cannot differentiate them by length.

For applications that can modify themselves at runtime, and depending on the obfuscation used, this might not be a major problem, as the URL can be split into multiple locations and the parts themselves can be obfuscated as well.

When utilizing partially encrypted endpoints, developers must specify in advance which parts are always encrypted and which may remain unencrypted. Failure to do so could enable an attacker to supply unencrypted endpoints and attempt to deduce and use them.

4.3.3 Shared URLs and Session Resumption. Two issues need to be addressed for practical use: URL sharing and session resumption. URL sharing is a widespread practice on the Internet for sharing resources. An online store, for example, relies on customers being able to send product links to each other. Further, session resumption, e.g., switching between devices or networks, is crucial for a smooth user experience.

URL Sharing. With the current design, URL sharing between users is only possible if they have the same client identifier, which should be avoided. The same is true for shared URLs between services, e.g., a online shop that is not using our approach linking to a payment provider that uses encrypted URLs. If the user is already logged in, the link will break.

The service using encrypted URLs could retain certain URLs or operations (e.g., HTTP methods like POST, GET, DELETE) to be used without our approach, i.e., as normal URLs. This could apply to URLs or URL paths that need to be public, such as incoming links that are used by unknown third parties, e.g., the payment site of a payment provider. The drawback of this method is that bots could operate on those URLs normally, so it should be used only when necessary. Note that after accessing the service using the normal URL, all subsequent URLs are encrypted again. Thus, our approach is only diminished for one URL operation. This requires the service to identify URLs and operations that should be public and to make them known to the middleware.

Another approach is to allow URLs to be accessed by a different user while restricting the access to read-only, i.e., only sharing URLs that do not alter state by restricting the operations on the URLs. This requires identifying URLs and operations that do not modify state. If HTTP is used, this should be straightforward, as the method has properties like *safe*, which indicate that they should not alter state. This is defined in the HTTP RFCs[16, 17]; however, it is currently treated as a convention and may be violated by developers. Consequently, safe methods like GET cannot always be relied upon to be safe. This necessitates operators verifying whether their application adheres to the standard. If not, the first method must be used. Thus this approach as well requires the service to identify URLs and operations that should be public and to make them known to the middleware.

This makes URL sharing between users possible, it does not affect the defense against bots. Multiple bots can now access the same URL without requiring the same client identifier. However, this applies only to „read" access. Whether one or more bots have read access to the same resource is mainly inconsequential and offers no additional value for the bot creator. In edge cases, such as displaying the availability of a product, rate limiting could be circumvented. In contrast, multiple bots can benefit the bot creator for operations that alter the state, such as sending spam or purchasing products.

For sharing encrypted URLs, the approaches the middleware needs to be able to process the request, thus the URL needs to be decrypted. In a stateless construction, the client key can be appended to the URL, encrypted, and authenticated with the server key. Thus for a read-only operation, the URL can be decrypted by the middleware by decrypting the client key first.

$$\begin{aligned} u' &= \langle e, t_e, g, t_g \rangle \\ \langle g, t_g \rangle &= AE_{k_m}(k_c) \end{aligned} \tag{2}$$

New to Equation 1 are the encrypted client key g and the accompanying MAC tag t_g. Additionally, the authenticated encrypted

client key k_c is constructed using the secret key k_m, which is known only to the middleware.

In deployment, shared URLs can be identified by adding an identifier to it, e.g., between the client encrypted URL and the client key $\langle e, t_e, \textit{delimiter}, g, t_g \rangle$. It would check if it is a state-modifying request and g is not forged. Then it can decrypt g to obtain k_c. Next, it needs to verify e, t_g and can obtain the URL. In theory, a delimiter is not necessary if g, t_g are of fixed size, but that would force the backend to treat every URL first as a shared URL.

Session Resumption. This issue arises only when an identifier that is prone to change is selected, such as relying solely on the IP address. For instance, a user's IP can change when transitioning from a mobile network to a WiFi network, leading to a scenario where all current URLs become invalid. This is not conducive to a user-friendly experience.

In cases where utilizing a more stable client identifier is not feasible, one straightforward solution is to enable URL sharing. This allows users to access shareable URLs even if their identifier changes.

A stateful approach to addressing this challenge involves the use of one-time tokens, which can circumvent URL verification once. These tokens could be stored as cookies in HTTP contexts. However, it is crucial that the token is valid only for the respective user, meaning it should bypass URL verification exclusively for that user. This can be achieved by storing the client key in an encrypted and authenticated form using the server key, as outlined in a manner similar to Equation 2.

When a client initiates a request from a new IP, resulting in a changed derived client key, the token comes into play. The process begins with the verification and decryption of the token, followed by the verification and decryption of the requested URL. It is important to note that this token is inapplicable to URLs belonging to a different client, as they are authenticated with a distinct client key. Successful access to the new URL leads to the construction of all subsequent URLs with the new client key.

Employing this method for session resumption does not undermine defenses against bots, as it does not offer them any advantage. While a bot may transfer the URL to another bot, accessing the resource is an action the original bot could have executed, maintaining the integrity of the system's bot defense mechanisms.

4.4 Optimisations

4.4.1 Validity period for URLs. In the current design, URLs for a client remain the same if the client identifier does not change. This means that when a client-specific bot is created, it does not have to re-extract the URLs for that client. The client key can be made expirable to increase the effort required to maintain bots. In a stateless design, client keys k_c can be regenerated at specified intervals t, or for each application update, by incorporating a nonce into the derivation of the client key, which would always change, $KDF(M, I_c, nonce_t)$. In a distributed infrastructure, e.g., using anycast or a load balancer, to avoid syncing the nonce, one could derive the nonce using a deterministic pseudorandom function. Due to our method of session resumption (Section 4.3.3), this does not result in any loss of user experience. Furthermore, any previously shared URLs do not expire.

4.4.2 Minimizing the Overhead. Several optimizations can be made to reduce the overhead or increase the cost of maintaining bots.

Space-Time Tradeoff The current design is stateless and regenerates client keys and URLs on the fly, which might result in increased latency due to the frequent execution of CPU-intensive operations. One optimization is to introduce a state which stores client keys or URLs. Then, if a client requests a URL, the backend only needs to check whether the URL is known. No cryptographic operation is required. The URLs can be cached in a database or memory. However, storing all URLs for each client is storage-intensive. Another less extreme time-space tradeoff is to store the client key, so it does not have to be regenerated for each request. This will decrease the CPU-intensive operations and decrease the latency. We will evaluate these methods in Section 5.2.

Risk Assesment A strong resource optimization is the use of risk assessment approaches. The generation of client-specific URLs is computationally more expensive. By using risk assessment approaches [37, 42], this overhead can be reduced for the server and legitimate clients. Simple examples of such risk assessments include identifying high-volume IPs or clients that have been connected to the service for an unusually long time or checking if the client is an automated browser like Selenium or Puppeteer. This way, the URLs can be generated only for clients that are flagged as high-risk, while the rest of the clients will be served directly from the server or a Content Delivery Network (CDN).

5 EVALUATION

In this section, we summarize the results of our evaluation. The focus is to determine the approach's overhead and compare it to similar obfuscation techniques. In detail, we answer the following research questions:

RQ1: To what extent does our approach restrict the scalability of bots, and how does it perform in comparison with alternative bot detection or mitigation strategies?

RQ2: What is the computational and operational overhead associated with employing encrypted endpoints?

RQ3: Against which specific attacks do encrypted endpoints provide protection?

We implement our approach as middleware in FastAPI, ensuring seamless integration by simply incorporating the middleware into the backend. Our implementation extends beyond the backend, offering support for rendering via Jinja2. This enables the straightforward inclusion of webpages, encompassing HTML and JavaScript, directly out of the box. Furthermore, we have also implemented features for partially encrypted endpoints, URL sharing and session resumption. These functionalities utilize authenticated encryption, specifically AESSIV, to maintain security and integrity.

For transparency and accessibility, we will release our implementation open-source on GitHu[6]

5.1 RQ1: Security Discussion

In this section, we delve into the security ramifications of our proposed methodology, specifically crafted to curtail the scalability of bot operations. Unfortunately, directly quantifying the effectiveness

[6]https://github.com/8mas/encrypted-endpoints

of our strategy in thwarting bot proliferation presents a considerable challenge. This difficulty primarily stems from the inherent complexity in measuring the success rates of bots, as accurately distinguishing between legitimate and automated requests would essentially address the issue at hand. Moreover, we do not have direct access to popular web services that are frequently targeted by bot activities. Therefore, we opt for an analytical comparison, evaluating how our approach impedes the deployment of common bot creation techniques and attacker strategies, as outlined in our threat model detailed in Section 2.

Additionally, we conduct a comparative analysis between our approach and alternative bot mitigation strategies. In particular, we focus on code obfuscation due to its closely related nature.

5.1.1 Protection Against Scaling of Bots. A bot that scales is usable for other clients and accounts without being adapted to them beforehand. Exceptions are credentials or IDs, such as a username, password, or session token, which are needed to use the service.

As for the defense, we consider the following obfuscation approaches:

Code obfuscation: The client code (e.g., HTML, Binary, Android App) is obfuscated using, for example, techniques from related work [12, 41]. WWe describe the requirements for a suitable code obfuscation to be used in conjunction with our approach in Section 6.

Encrypted endpoints: The obfuscation approach described in this paper. Each request is on an encrypted URL.

Attacker: Endpoints Only. Code obfuscation cannot stop this attacker because the code is completely ignored. This attacker examines in which order to which endpoint what was sent and repeats the session. There are simple-to-use, automated tools that facilitate this [24, 34].

Encrypted endpoints stop this attacker completely. While the attacker can build a bot that works for their client, i.e., with their client identifier, the bot does not function for a different client. Furthermore, since this attacker cannot extract endpoints from the client-side application, the bot cannot be scaled. The attacker can also not predict or generate new valid URLs if a secure authenticated encryption method is used and correctly implemented.

Attacker: Endpoints and Data Parsing. Code obfuscation alone fails to offer any significant protection, as the endpoint-only attacker represents a specific subset of adversaries who can employ the same techniques for bypassing such measures. Similarly, encrypted endpoints, in isolation, do not guarantee resilience. Attackers might extract endpoints from client-side source code, such as Java classes in Android applications, by identifying the <a> tag's ID or XPath in HTML documents or using disassemblers for binary programs.

However, the combined application of code obfuscation and encrypted endpoints introduces a robust layer of protection. Encrypted endpoints specifically counteract the weak-points inherent in code obfuscation, by preventing straightforward access to the URLs by recording the traffic. Concurrently, code obfuscation complicates the attacker's ability to pinpoint and exploit individual URLs, thereby safeguarding against direct attacks on encrypted endpoints. It is crucial to ensure that endpoints do not inadvertently disclose information that could undermine this defense, for example, by adopting consistent encryption practices across URLs. Obfuscation becomes ineffective if URLs can be distinguished through their paths.

To augment this security strategy, we advocate for the use of advanced obfuscation techniques, akin to those described in [11], which can thwart brute-force attacks. Such attacks involve systematically probing all extracted URLs to identify the correct one through trial and error.

Direct attacks on encrypted endpoints, without compromising cryptographically secure methods, are implausible. Thus, attackers are compelled to extract URLs, with the complexity of this process being contingent upon the sophistication of the employed HTML obfuscation technique. Regularly updating the endpoints for each client can significantly escalate the effort and resources required for an attacker to maintain their offensive capabilities.

5.1.2 Encrypted Endpoints Compared to Other Approaches. Our approach does not claim to replace other approaches. On the contrary, it can and should be applied in addition. All following approaches are considered in the context of web bots and their scaling.

Code Obfuscation. The limitations of code obfuscation strategies have been discussed previously, both in the preceding section and in Section 3.2. These techniques enhance the difficulty of extracting specific data from client-side sources, such as from Java Classes or HTML, for instance, by employing XPath or CSS selectors (refer to Section 3.2). This increased complexity presents substantial obstacles for the development of bots that depend on information derived from the web, such as detecting product availability.

As described in Section 5.1.1, obfuscation methods fall short in mitigating the threats posed by API bots or tactics that involve replaying previously recorded sessions [24, 34]. Likewise, similar to our proposed model, they offer no defense against UI-based bots. However, encrypted endpoints effectively counter session replay attacks and escalate the difficulty for API bot operations. Although our methodology may be bypassed by parsing request responses to acquire currently valid endpoints, the integration of code obfuscation techniques significantly impedes the extraction of such information, including endpoints. This synergy underscores the advantage of combining our approach with code obfuscation methods.

User-Specific Client-Side API Keys. As discussed in Section 3.2, employing user-specific client-side API keys and CSRF tokens for user authentication and request validation marks preliminary measures against bot proliferation. Our strategy enhances these mechanisms, extending their application to comprehensively obstruct bot scalability through the mandatory retrieval and utilization of valid tokens for server requests.

CAPTCHAs. CAPTCHAs are another kind of defense. They are designed to confirm that a client is a human by their knowledge or ability to perform tasks. This means that they are an active challenge, which is counterproductive to gaining services [18], and wastes time [1]. Our approach does not hinder regular users, but it presents difficulties for those attempting to create or scale multiple bots. The difficulties arise that session replaying tools are rendered useless and force the extraction of client/account-specific encrypted

endpoints. The use of code obfuscation techniques further complicates this extraction process.

Bot Detection and Risk Assesment. These approaches detect bots or gauge a client's likelihood of being a bot, leading to allowance, blocking, or CAPTCHA challenges. They aim to block bots without overusing CAPTCHAs, avoiding user frustration and revenue loss. Our approach does not differentiate bots. A possible combination with these approaches would be to provide only clients that are likely to be bots with encrypted endpoints and leave all others without.

Facebooks Encrypted URLs. As outlined in Section 3.2, Facebook's deployment of encrypted URLs primarily aims to prevent URL stripping, rather than deterring bot activities [3]. By encrypting and signing every URL parameter, the integrity of tracking parameters is preserved, thwarting attempts to remove them without compromising the link's validity. This method, while not designed for bot mitigation, shares similarities with our approach, prompting a detailed comparison in Table 2.

5.2 RQ2: Overhead

5.2.1 Ease of Integration. Integrating the proposed approach into a backend system is straightforward, facilitating the redirection of all requests through our middleware (cf. Listing 3). This middleware verifies the legitimacy of each request, offering configuration options such as custom identifier selection and support for partial URLs. For partially encrypted URLs, a custom validator must be written to check whether a partially encrypted URL is correct. Specifically, it must ensure that the dynamic, user-controlled part is valid so that the user cannot include more than necessary, such as extra parameters or paths. Load balancing and distributed deployment should function normally and seamlessly, as long as the client ID is deterministic, and the key derivation function (KDF) used to derive the client key is consistent, given that the implementation is stateless.

```
app.add_middleware(
middleware_class=EncryptedEndpointsMiddleware,
main_key=b"some_key")
```

Listing 3: Adding EncryptedEndpointsMiddleware to FastAPI

For the frontend, support is extended to Jinja2, enabling the invocation of an encryption function within our framework. This functionality is not confined to HTML but is also applicable to JavaScript tags and files (cf. Listing 4 and Listing 5).

```
TemplateResponse("start_page.html", {"request":
    request})
```

Listing 4: Rendering a Template Response in FastAPI

```
<!-- Encrypt URL to resources -->
<script src="{{encrypt_value('/templates/scripts.js',
    request)}}"></script>
<!-- Encrypt URLs -->
<i class="logout-icon"
    onclick="location.href='{{encrypt_value('/logout/',
    request)}}'"></i>
<!-- In JavaScript -->
<script>
fetch('{{encrypt_value("/posts/", request)}}')
</script>
```

Listing 5: Using Encrypted URLs in the Frontend

For binaries and applications, multiple integration methods exist. The simplest is to compile the binary with user-specific endpoints embedded. Alternatively, it is feasible to dynamically inject the endpoints into the application at startup. In this scenario, the application contacts a server, which then provides all necessary endpoints to the client. This necessitates that this information is obfuscated as well. For example, instead of transmitting the text form of endpoints, the endpoints could be embedded in a library, or the URLs could be transmitted encrypted and integrated into the main package within a secure enclave, such as Intel SGX.

5.2.2 URL Stretch. It is crucial to be aware of the limitations that various platforms impose on the length of URLs. Although the HTTP/1.1 specification RFC 2616 [15] does not define a maximum URL length, practical constraints are enforced by web browsers, for example, a limit of 2MB in Chromium[7]. Our experiments indicate that, as of the latest version in April 2024, all major desktop browsers (Firefox, Chrome, Edge, Safari), as well as mobile browsers (Chrome, Firefox, Safari), support URLs exceeding 20,000 characters.

Several factors influence the final size (L_{final}) of encrypted URLs, depending on the implementation specifics. The following holds for our implementation. These factors include padding ($O_{padding}$) to reach the block size, which might require an additional 128 bits, and the addition of a MAC (O_{mac}), contributing another 128 bits. Our implementation incorporates separators to support partially encrypted URLs due to their dynamic nature, adding an overhead of 2 bytes ($O_{separators}$). The overhead per encrypted URL segment, denoted by n_{blocks}, typically comprises only one segment, meaning that the entire URL is encrypted. Moreover, the transformation of encrypted data using base64 encoding typically increases the size by approximately 33% as it encodes every 3 bytes into 4 bytes (O_{base64}).

Given these considerations, the formula for calculating the length in bytes of data post-encryption can be expressed as follows:

$$L_{\text{final}} = L_{\text{data}} \cdot O_{\text{base64}} + n_{\text{blocks}} \cdot (O_{\text{padding}} + O_{\text{mac}} + O_{\text{separators}})$$

$$L_{\text{final}} = L_{\text{data}} \cdot 1.33 + n_{\text{blocks}} \cdot (16 + 16 + 2)$$

Given the length expansion, current browser limits and considering typical URL lengths, the length expansion from encryption does not appear to be problematic. Should one approach these limits, exploring text compression methods could offer a viable solution.

5.2.3 Runtime Performance. Table 5presents the latency results obtained within a local network environment using a test application we developed. The application returns a simple HTML page with a configurable number of embedded links. The baseline is the web application without our approach of encrypted endpoints, utilizing AES-GCM for authenticated encryption. It is worth noting that the processor in use is equipped with AES-NI [4]. The terminologies used in the table are defined as follows. *ENC* indicates a scenario where encrypted URLs are utilized, with the client key being dynamically derived for each request, thus eliminating the requirement

[7] https://chromium.googlesource.com/chromium/src/+/HEAD/docs/security/url_display_guidelines/url_display_guidelines.md

for the server to maintain state. *S-Key* denotes a method similar to ENC, wherein the client key is cached on the server, adopting a stateful approach to enhance performance by balancing storage space against CPU time. *S-URLs* employs a strategy akin to the S-Key method, with the encrypted URLs being cached on the server to avoid the need for recalculating or re-encrypting identical URLs.

The results from the first table highlight that caching the client key or the accessed URLs on the server, thereby maintaining a stateful approach (*S-Key*), reduces latency, though the overall runtime overhead remains minimal across all methods tested. Ideally, caching client keys temporarily, along with frequently accessed (hot) URLs, should be implemented to minimize latency.

	ENC	S-Key	S-URLs
Latency Increase	+0.0311ms	+0.0199ms	+0.0016ms

Table 3: Impact on backend server response times due to the handling (decryption) of encrypted URLs, measured in milliseconds. Profiling was conducted internally, avoiding end-to-end measurements due to their millisecond-level variability, which could compromise accuracy. An average end-to-end request in a local network is estimated between 12ms and 14ms. The average latency increase was determined over 100000 requests.

Table 4 examines the performance overhead of rendering encrypted URLs using Jinja2, employing profiling to ensure accuracy over end-to-end timing due to the latter's variability at the millisecond level.

The table compares the mean communication latency for the *base* approach, which involves not encrypting URLs, against various methods employing encrypted URLs. For each request, an HTML document containing between 10 to 100 unique URLs was returned. This comparison illustrates that caching both the client key and the URLs significantly reduces latency, a strategy that should be particularly effective for URLs with high demand.

	Base	ENC	S-Key	S-URLs
10 URLs	0.033ms	+0.158ms	+0.076ms	+0.023ms
100 URLs	0.087ms	+1.470ms	+0.662ms	+0.128ms

Table 4: Performance overhead of encrypting and rendering unique URLs. Profiling was utilized instead of end-to-end timing to circumvent the latter's millisecond-level variability, which could lead to inaccuracies. The table presents the mean communication latency in milliseconds for direct and encrypted URLs. Base refers to the baseline, meaning our web application without encrypted endpoints. The average latency increase was observed over 100000 requests.

5.3 RQ3: Protection Against Attacks

The methodology outlined in this paper primarily aims to mitigate bot-related threats while also functioning as an automatic whitelist mechanism for URLs. We briefly discuss this feature, considering it as a subject for future exploration.

This approach ensures that only URLs explicitly authorized by the server are accepted, offering protection against certain types of attacks:

Directory Traversal This method prevents attackers from navigating outside the web root directory to access restricted files, thus safeguarding against unauthorized directory access attempts (e.g., *example.com/../../etc/passwd*).

Accidental Data Exposure It protects against the accidental exposure of sensitive files, such as configuration files (*example.com/.env*), by ensuring that only URLs deliberately exposed by the server are accessible.

Local File Inclusion (LFI) It obstructs attempts to include files from the server's filesystem in the output of a web application, a tactic often used to execute arbitrary code or access sensitive information.

Reflected XSS and SQL Injection Attacks It offers some protection against attacks that misuse URL parameters to inject malicious scripts (Reflected XSS) or manipulate database queries (SQL Injection), as exemplified by attempts like *example.com?item='– drop table*. This method, however, is only applicable to endpoints where the URL parameters are predetermined, such as a shopping page displaying related products or a social media/blog platform suggesting tags (e.g., *example.com?tag=outfit*). This protection mechanism is ineffective if fully user-controlled input is accepted, such as in search fields. Additionally, in cases where this method is effective, it may inadvertently restrict users who guess parameters. For instance, a user interested in exploring hardware instead of outfits cannot simply change the tag in the URL bar, even if the tag exists.

At the heart of this protective mechanism is the requirement for each URL to carry a valid MAC issued by the server. The use of a MAC that resists existential forgery under chosen-message attacks ensures the impossibility for attackers to forge valid URLs.

However, it is crucial to recognize the limitations of this security measure. For URL segments that are completely under user control, such as parameters in search fields, our approach falls short. This is because the URL cannot be entirely pre-constructed by the server; it can only be partially created, as the user's search parameter is unknown. Further, in scenarios where an attacker successfully persuades the server to serve a malicious URL, the URL would be considered valid. In the previous example with the tag system (*example.com?tag=outfit*), this prevents an attacker from merely attempting to insert malicious payloads directly into the parameter system. However, if the attacker can create a new tag containing the malicious payload, the attack would succeed. The practicality of such an exploit depends on the website's specific configurations and security measures, which may deter or entirely prevent the attacker's success.

5.4 Limitations and Privacy Implications

While encrypted endpoints limit the scalability of bots, it is hard to quantify how much harder they make it. This is mainly due to the quality of the used obfuscation. In an ideal scenario, a perfect

obfuscation is used in conjunction with our approach, which would prevent bots' scaling. However, there is no such thing as a perfect obfuscation.

Services that use encrypted endpoints and require URL sharing between users (e.g., an online shop or social media) or need to be interoperable with third-party services (e.g., a payment provider or identity provider) must identify URLs or URL paths that need to be shareable. These URLs and corresponding actions must be communicated to the middleware implementing the encrypted endpoints. In large applications, this might increase the initial cost of using encrypted endpoints.

Further, despite the introduction of unique endpoints for each account, attackers with substantial resources, be it in terms of computational power or time, could potentially overcome this hurdle. By programming bots that simulate user interactions with the UI, attackers could sidestep the account-specific endpoint requirement. While such UI-based bot creation might prove resource-intensive, especially in the case of smartphone apps requiring the use of Virtual Machines (VMs), determined attackers may still find ways to automate these processes effectively, e.g., by starting bots sequentially to save resources. Moreover, the reliance on UI interactions constrains the capabilities of these bots to the scope of functionalities exposed through the UI. This limitation could hinder attackers seeking more nuanced and intricate actions that direct controlled requests could achieve. Therefore, while the „encrypted endpoints" approach offers strong protection against simple and some advanced bots, it might fall short against adversaries who are adept at crafting UI-based automation techniques.

Our approach can be used more as defending against bots alone. It also destroys the ability to read URLs and their parameters. In today's Internet, the URL path or its parameters often reveal information about a resource. When these URLs are encrypted, this is no longer the case. Authenticating URLs alone cannot remove tracking parameters without invalidating the URL. Facebook has already started doing this [3]. Unfortunately, we do not have any technical countermeasures against this approach. The main problem with this approach, which makes it worse than tracking cookies, is that all tracking information is stored in the URL itself and thus affects shared links.

6 COMBINING ENCRYPTED ENDPOINTS WITH CODE OBFUSCATION

Code obfuscation is orthogonal to our approach but necessary to protect against advanced and realistic EP attackers. We define several requirements that a code obfuscation technique should fulfill. The issue is that even if all endpoints are encrypted, the location within the application remains unchanged. Thus, a bot creator does not need to save the endpoint but rather the position where the URL is stored. This could be a file offset, an XPath or CSS Selector in a web context, or even a line number in JavaScript.

We identified several requirements for a code obfuscation approach. The most important is the *obfuscation of the URL location*, ensuring that the URL is not easily identifiable through its position or the relationship between objects. The URL may also be split and stored in multiple locations if the context permits, such as in programming languages like JavaScript, but not in languages like HTML or CSS. Another requirement is *non-deterministic results*, meaning the code obfuscation should ideally produce different versions of the application for different users. Deterministic obfuscation methods should be combined with non-deterministic techniques. Additionally, the obfuscation should have *low overhead in runtime and build*. The resources required to build the obfuscated application should be minimal, which could be addressed by prebuilding multiple versions of the application. Lastly, *full application obfuscation* is essential, meaning the entire application containing or potentially containing URLs should be obfuscated. This is particularly important for web applications, where it is insufficient to obfuscate only the JavaScript if it inserts URLs into the HTML.

While most applications have many options due to their ability to modify themselves at runtime, standalone HTML pages without JavaScript present a challenge. HTML cannot modify itself at runtime and must therefore include all URLs in clear text. Given that the obfuscation technique can change the location of URLs as specified in our requirements, an attacker could still brute force all possible URLs. To mitigate this, some form of rate limiting or trap URLs should be included. For instance, if a bot attempts to access a fake URL, it should be blocked from further interaction.

Using our approach, the URLs do not reveal any information about themselves as they are encrypted and can be padded to equal lengths. Suitable approaches are discussed in Section 3.2.

7 CONCLUSION

Our investigation reveals that while numerous strategies primarily aim at complicating the initial creation of bots, our proposed methodology focuses on impeding their applicability across different user accounts. This objective is accomplished by assigning unique, encrypted endpoints (URLs) to each client, thereby necessitating that attackers extract fresh endpoints for every bot directly from server responses. Implemented as a middleware layer within the backend architecture, our solution facilitates seamless integration with existing systems, ensuring minimal impact on performance. An empirical evaluation of our approach indicates a modest increase in latency overhead by approximately 0.03ms - 0.0016ms for incoming requests under the most realistic operational scenarios. Alone, our method proves effective in thwarting the efforts of simple bots and automated tools. However, its efficacy is markedly enhanced when combined with targeted code obfuscation techniques, significantly curtailing the scalability of sophisticated bot operations. Looking forward, we aim to conduct a comprehensive empirical analysis of our approach, focusing on identifying and integrating code obfuscation methods specifically designed to combat bot activities. This will thereby further reinforce the security of web services against automated threats.

REFERENCES

[1] 2021. Humanity Wastes about 500 Years per Day on CAPTCHAs. It's Time to End This Madness. http://blog.cloudflare.com/introducing-cryptographic-attestation-of-personhood/
[2] 2022. 2022 Bad Bot Report | Evasive Bots Drive Online Fraud | Imperva. https://www.imperva.com/resources/resource-library/reports/bad-bot-report/
[3] 2022. Facebook Gets Round Tracking Privacy Measure by Encrypting Links. https://www.malwarebytes.com/blog/news/2022/07/facebook-gets-round-tracking-privacy-measure-by-encrypting-links
[4] 2023. Intel® Advanced Encryption Standard Instructions (AES-NI). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-

encryption-standard-instructions-aes-ni.html
[5] Alejandro Acien, Aythami Morales, John V Monaco, Ruben Vera-Rodriguez, and Julian Fierrez. 2021. TypeNet: Deep learning keystroke biometrics. *IEEE Transactions on Biometrics, Behavior, and Identity Science* 4, 1 (2021), 57–70.
[6] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. 2015. Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications* 48 (2015), 44–57.
[7] Fatmah H Alqahtani and Fawaz A Alsulaiman. 2020. Is image-based CAPTCHA secure against attacks based on machine learning? An experimental study. *Computers & Security* 88 (2020), 101635.
[8] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 189–200.
[9] Alessandro Bessi and Emilio Ferrara. 2016. Social bots distort the 2016 US Presidential election online discussion. *First monday* 21, 11-7 (2016).
[10] BinBashBanana. 2024. BinBashBanana/html-obfuscator. https://github.com/BinBashBanana/html-obfuscator original-date: 2020-04-22T00:23:10Z.
[11] Douglas Brewer, Kang Li, Laksmish Ramaswamy, and Calton Pu. 2010. A Link Obfuscation Service to Detect Webbots. In *2010 IEEE International Conference on Services Computing*. IEEE, Miami, FL, USA, 433–440. https://doi.org/10.1109/SCC.2010.89
[12] Christian Collberg. 2001. *the tigress c obfuscator*. Retrieved 2021-12-07 from https://tigress.wtf/about.html
[13] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. 2018. Privacy Pass: Bypassing Internet Challenges Anonymously. *Proc. Priv. Enhancing Technol.* 2018, 3 (2018), 164–180.
[14] Ahmed Diab and Tawfiq Barhoum. 2018. Prevent XPath and CSS Based Scrapers by Using Markup Randomizer. *Int. Arab. J. e Technol.* 5, 2 (2018), 78–87.
[15] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. 1999. *Hypertext transfer protocol–HTTP/1.1*. Technical Report.
[16] R Fielding, M Nottingham, and J Reschke. 2022. RFC 9110: HTTP Semantics.
[17] Roy Fielding and Julian Reschke. 2014. RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): semantics and content.
[18] Nick Heath. 2010. Expedia on how one extra data field can cost $12m. https://www.zdnet.com/article/expedia-on-how-one-extra-data-field-can-cost-12m/. Accessed: 2021-10-18.
[19] Md Imran Hossen, Yazhou Tu, Md Fazle Rabby, Md Nazmul Islam, Hui Cao, and Xiali Hei. 2020. An Object Detection based Solver for {Google's} Image {reCAPTCHA} v2. In *RAID 2020*. 269–284.
[20] Apple Inc. [n. d.]. Replace CAPTCHAs with Private Access Tokens - WWDC22 - Videos. https://developer.apple.com/videos/play/wwdc2022/10077/
[21] ProtWare Inc. [n. d.]. Encrypt HTML source, Javascript, ASP. Protect links & images. HTML encryption. https://www.protware.com/
[22] Jscrambler. [n. d.]. Webpage Integrity: Manage Third-party Risks. https://jscrambler.com/webpage-integrity
[23] Timofey Kachalov. [n. d.]. javascript-obfuscator/javascript-obfuscator: A powerful obfuscator for JavaScript and Node.js. https://github.com/javascript-obfuscator/javascript-obfuscator
[24] Albert Koczy. 2023. Mitmproxy2swagger. https://github.com/alufers/mitmproxy2swagger
[25] Mohinder Kumar, MK Jindal, and Munish Kumar. 2022. A systematic survey on CAPTCHA recognition: types, creation and breaking techniques. *Archives of Computational Methods in Engineering* 29, 2 (2022), 1107–1136.
[26] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2014. Reducing web test cases aging by means of robust XPath locators. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 449–454.
[27] Wei Liu. 2018. Introducing reCAPTCHA v3: the new way to stop bots. https://developers.google.com/search/blog/2018/10/introducing-recaptcha-v3-new-way-to. Accessed: 2021-05-20.
[28] Intuition Machines. 2018. Stop more bots. Start protecting user privacy. https://www.hcaptcha.com/. Accessed: 2021-05-20.
[29] Genesis Mobile. [n. d.]. JavaScript Obfuscator - Protect your JavaScript Code. https://jasob.com/
[30] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 569–580. https://doi.org/10.1109/DSN48987.2021.00065
[31] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. *Proceedings of W2SP* 2012 (2012).
[32] August See, Leon Fritz, and Mathias Fischer. 2022. Polymorphic Protocols at the Example of Mitigating Web Bots. In *European Symposium on Research in Computer Security*. Springer, 106–124.
[33] August See, Tatjana Wingarz, Matz Radloff, and Mathias Fischer. 2023. Detecting Web Bots via Mouse Dynamics and Communication Metadata. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 73–86.
[34] see-aestas. 2020. Charles-Extractor. https://github.com/see-aestas/Charles-Extractor
[35] Suphannee Sivakorn, Iasonas Polakis, and Angelos D Keromytis. 2016. I am robot:(deep) learning to break semantic image captchas. In *2016 IEEE EuroS&P*. IEEE, 388–403.
[36] VMProtect Software. 2021. *VMProtect Software Protection*. Retrieved 2021-12-07 from https://vmpsoft.com/
[37] Grażyna Suchacka, Alberto Cabri, Stefano Rovetta, and Francesco Masulli. 2021. Efficient on-the-fly Web bot detection. *Knowledge-Based Systems* 223 (2021), 107074.
[38] Mahin Talukder, Syed Islam, and Paolo Falcarin. 2019. Analysis of obfuscated code with program slicing. In *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. IEEE, 1–7.
[39] Oreans Technologies. 2022. Oreans Technologies : Software Security Defined. https://www.oreans.com/Themida.php Accessed 2021-12-07.
[40] Shardul Vikram, Chao Yang, and Guofei Gu. 2013. Nomad: Towards non-intrusive moving-target defense against web bots. In *CNS*. IEEE, 55–63.
[41] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. 2016. Webranz: web page randomization for better advertisement delivery and web-bot prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 205–216.
[42] Ang Wei, Yuxuan Zhao, and Zhongmin Cai. 2019. A deep learning approach to web bot detection using mouse behavioral biometrics. In *Biometric Recognition: 14th Chinese Conference, CCBR 2019, Zhuzhou, China, October 12–13, 2019, Proceedings 14*. Springer, 388–395.
[43] Tara Whalen, Thibault Meunier, Mrudula Kodali, Alex Davidson, Marwan Fayed, Armando Faz-Hernández, Watson Ladd, Deepak Maram, Nick Sullivan, Benedikt Christoph Wolters, et al. 2022. Let The Right One In: Attestation as a Usable {CAPTCHA} Alternative. In *Eighteenth Symposium on Usable Privacy and Security (SOUPS 2022)*. 599–612.
[44] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, 9–16.
[45] Jason Yung. 2024. json2d/obscure. https://github.com/json2d/obscure original-date: 2016-05-30T22:04:01Z.

	Direct	ENC	S-Key	S-URLs
Latency 10 URLs	13.9ms	+3.1%	+2.4%	+1%
Latency 100 URLs	14ms	+4.3%	+2.1%	+0.7%
Latency 1000 URLs	14.1ms	+5.7%	+2.1%	+0.8%

Table 5: Mean communication latency in milliseconds relative to direct server communication, expressed as a percentage increase (N=100000). Dynamic URL generation and 100 KB webpage padding were employed.

A ALTERNATIVE PROXY IMPLEMENTATION

We further explored implementing our approach through a transparent proxy rather than as middleware, eliminating the need for any modifications to the service's code. While this method offered ease of integration to any backend, it introduced significant performance drawbacks, primarily due to the additional resources required for parsing.

A.1 Performance Overhead

Our proxy-based implementation involved intercepting requests to dynamically identify URLs, generate keys, and encrypt URLs, contributing to increased latency due to the computational demands of these operations. To quantify the impact on latency, we conducted tests on three simulated websites containing 10, 100, and 1000 URLs, respectively. Each website was standardized to 100 KB in size, inclusive of URLs and randomly generated padding bytes, with 1000 URLs approximating 95 KB of the total content. This choice was informed by an analysis of 10,000 random websites from the Majestic Million list, revealing a median presence of 106 URLs per site. The latency measurements were performed on a Debian 11 system equipped with an Intel i5-8365U CPU, targeting a Gunicorn/Flask HTTP server. Latency was calculated from the initiation of the request to the full receipt of the response, without reusing TCP connections for subsequent requests.

Table 5 presents the latency findings within a local network context, employing AES-GCM for authenticated encryption, with the processor supporting AES-NI.

The results indicate that direct server communication (*Direct*) is the fastest, with latency increases for encrypted URLs (*ENC*), stored client keys (*S-Key*), and stored encrypted URLs (*S-URLs*) being attributable to the additional computational steps involved. Notably, the performance gain from storing client keys or URLs is modest, given the already low overhead from encryption.

Transitioning from a proxy to backend middleware implementation enhances efficiency, albeit at the cost of requiring code adjustments. The proxy model, despite its straightforward setup, incurs significant performance losses, especially with an increasing number of URLs, due largely to the time-intensive nature of HTML URL parsing.

A.2 Organisational Overhead

The complexity of directly implementing encrypted endpoints varies with the existing codebase. Here, the primary challenge lies in statically extracting URLs from backend responses, a process that may miss URLs dynamically generated via JavaScript.

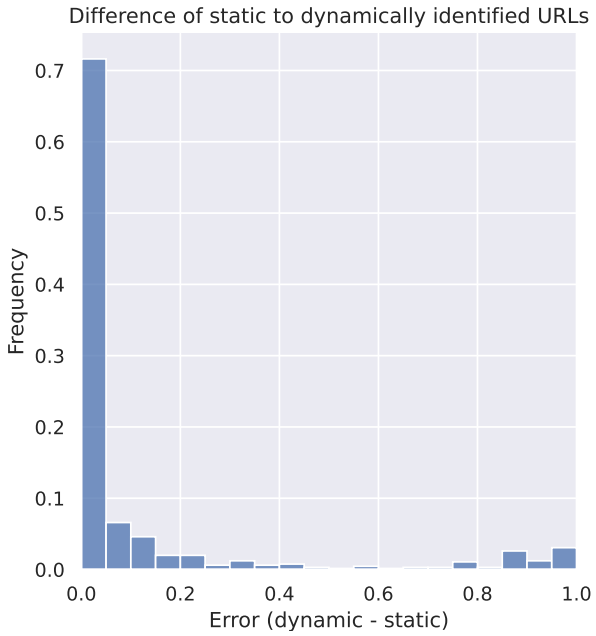


Figure 3: Overview of Possible Encrypted Endpoint Usage (N=10000)

Our analysis of 10,000 URLs from the Majestic Million list, comparing static and dynamic extraction methods, sought to quantify the potential discrepancy. While not exhaustive, this comparison sheds light on the limits of static extraction in capturing dynamically generated URLs, as illustrated in Figure 3.

This analysis reveals instances where dynamic extraction identifies URLs not found through static methods, with a mean error rate of 0.1188 and 78.22% of sites showing no discrepancy between the two. Further investigation into sites with large discrepancies highlighted asynchronous JavaScript processes as a primary factor, suggesting that encrypted endpoint deployment could benefit from mechanisms like mutation observers to capture and encrypt dynamically generated links.

Reflecting on the findings from Section 5.2, the deployment of our approach, particularly as a proxy, appears less burdensome than initially anticipated. The performance is deemed acceptable for non-time-critical applications, and the proxy model allows for experimentation without altering the existing system architecture.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Sofern im Zuge der Erstellung der vorliegenden Dissertationsschrift generative Künstliche Intelligenz (gKI) basierte elektronische Hilfsmittel verwendet wurden, versichere ich, dass meine eigene Leistung im Vordergrund stand und dass eine vollständige Dokumentation aller verwendeten Hilfsmittel gemäß der Guten wissenschaftlichen Praxis vorliegt. Ich trage die Verantwortung für eventuell durch die gKI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate.

Ort, Datum	Unterschrift