REDUCING WEB ATTACK SURFACE: MITIGATING SOCIAL ENGINEERING AND CODE INJECTION THREATS

A Dissertation
Presented to
The Academic Faculty

By

Zheng Yang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2025

REDUCING WEB ATTACK SURFACE: MITIGATING SOCIAL ENGINEERING AND CODE INJECTION THREATS

Thesis committee:

Dr. Wenke Lee, Advisor
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Brendan Saltaformaggio, Co-Advisor
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Frank Li
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Roberto Perdisci
School of Computer Science
University of Georgia

Dr. Saman Sonouz
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Cormac Herley
Senior Principle Researcher
Microsoft Research

Date approved: March 24, 2025

I dedicate this dissertation to my dearest wife, Wan Qin. Your unwavering faith, boundless love, and steadfast encouragement have been my greatest sources of strength throughout this journey. Your belief in me never wavered, even in the most challenging moments, and for that, I am endlessly grateful. This work stands as a testament to your support and devotion–without you, it would not have been possible.

ACKNOWLEDGMENTS

First, I would like to express my deepest gratitude to my advisor, Dr. Wenke Lee, for his unwavering guidance and support throughout my doctoral journey. His patience, insightful critiques, and invaluable discussions have not only shaped this dissertation but have also profoundly influenced my growth as a researcher. I am truly grateful for his mentorship, which has been instrumental in my academic and professional development.

I am also immensely thankful to my co-advisor, Dr. Brendan Saltaformaggio, whose mentorship has significantly enhanced my ability to write and present research effectively. His keen insights and constructive feedback have helped refine my ideas and make my research more compelling.

I extend my sincere appreciation to my thesis committee members, Dr. Roberto Perdisci, Dr. Saman Zonouz, Dr. Frank Li, and Dr. Cormac Herley, for their valuable input and contributions. Their diverse expertise and thorough evaluations have greatly strengthened the quality of my work. I am especially grateful to Dr. Roberto Perdisci for his inspiration, wisdom, and guidance. His deep understanding and passion for research have been a source of motivation and have profoundly influenced my academic journey.

I am also deeply indebted to my colleagues, including Dr. Joey Allen, Dr. Simon Chung, Dr. Feng Xiao, Jizhou Chen, Runze Zhang, and many others. Their collaboration, shared wisdom, and unwavering support have been indispensable in bringing this work to fruition. The countless discussions, brainstorming sessions, and shared experiences in the lab have been both intellectually enriching and personally memorable.

Finally, this journey would not have been possible without the collective support, guidance, and encouragement of these extraordinary individuals. I feel truly privileged to have worked with and learned from them. My deepest appreciation goes to each of them—not only for their contributions to this dissertation but also for their role in shaping my intellectual and personal growth.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The web ecosystem has become an essential platform for communication, business, and entertainment, yet it remains highly vulnerable to various forms of injection-based cyber threats. These include not only technical exploits like code injection attacks, but also social engineering threats that inject misleading content, invisible overlays, or malicious scripts to deceive users and compromise web applications. While existing security solutions tend to focus narrowly on either system hardening or user education, they often fall short in mitigating the increasingly sophisticated and blended attack techniques seen in the wild.

This dissertation presents a unified, multi-layered defense strategy against such injection-based threats through the design and implementation of three novel security mechanisms: TRIDENT, COINDEF, and COINDX. Each system targets a different class of attack vector and collectively contributes to a reduced attack surface for both end-users and developers.

We first introduce TRIDENT, a browser-based defense system that detects and blocks social engineering attacks distributed through low-tier ad networks. By analyzing ad script behaviors, TRIDENT identifies and mitigates deceptive content injected to ad publisher websites that lead users to social engineering websites. Then, we propose COINDEF, a customized Electron engine designed to prevent code injection attacks by enforcing execution policies. Unlike traditional XSS mitigation strategies that focus on browser isolation, COINDEF ensures that only trusted code executes within the Electron environment by validating the structural integrity of JavaScript's abstract syntax tree and execution context. Last, we present COINDX, a root cause analysis framework for code injection attacks in JavaScript applications. Given the complexity of JavaScript and its dynamic dependencies, traditional vulnerability detection methods struggle with state explosion or accuracy. COINDX addresses this by constructing a simplified program based on call stack traces and applying iterative symbolic analysis to reproduce the vulnerable state and pinpoint the root cause.

By integrating these three systems, this dissertation advances web security through a proactive and automated defense model. Rather than reacting to known signatures or relying solely on user awareness, the proposed framework reduces the attack surface by preventing both social and technical injections before they can succeed. TRIDENT limits user exposure to deceptive web content, COINDEF safeguards execution environments in hybrid applications, and COINDX provides developers with precise tools for vulnerability remediation.

Ultimately, this research underscores the importance of a holistic approach to web security–one that recognizes the convergence of social and technical vectors under the broader category of injection attacks. The solutions presented here contribute to a more resilient web ecosystem by bridging the gap between content-level, runtime, and developer-facing defenses.

CHAPTER 1
INTRODUCTION

The modern web ecosystem is a cornerstone of digital communication, commerce, and information exchange. However, its vast interconnectedness and reliance on user interactions make it a prime target for cyber threats. Among these, injection attacks—whether exploiting technical vulnerabilities or user behavior—pose a significant challenge to web security.

Injection attacks that target human behavior often rely on social engineering techniques to deceive users into revealing sensitive information, clicking on harmful links, or performing unintended actions. These include phishing, impersonation scams, and misleading pop-ups that exploit psychological triggers such as trust, urgency, or fear. As web applications become more pervasive, adversaries increasingly inject deceptive content via email, social media, or compromised websites to manipulate user behavior. Malvertising (malicious advertising) further complicates the landscape by embedding psychological lures within technically legitimate platforms.

On the technical front, code injection attacks exploit flaws in web applications to insert and execute unauthorized code. Common types include Cross-Site Scripting (XSS), SQL Injection (SQLi), and Remote Code Execution (RCE), which can be used to exfiltrate data, hijack sessions, or alter application behavior. These attacks not only compromise the integrity of web applications but also often serve as delivery mechanisms for content designed to mislead or coerce users.

The convergence of these attack vectors—psychological and technical—has led to increasingly sophisticated threats. Adversaries may use code injection to present fake login interfaces, deceptive overlays, or malicious advertisements that facilitate social engineering tactics. This hybrid threat model underscores the dual nature of modern injection attacks, where both systems and users are targets of exploitation.

1.1 Problem Statement

Prior research in web security has yielded numerous techniques for addressing specific classes of threats. On the technical side, approaches such as input sanitization, content security policies (CSP), web application firewalls (WAFs), and static/dynamic analysis tools have been developed to mitigate code injection attacks like cross-site scripting (XSS) and HTML injection. While effective in constrained contexts, these solutions often rely on precise configurations, are bypassed by obfuscation or dynamic code execution, and typically operate without awareness of the broader user interface or behavioral context in which the attack occurs.

Conversely, social engineering defenses have largely focused on user education, phishing detection, or heuristic-based classification of suspicious websites and emails. These approaches struggle with high false positive/negative rates and assume users can make security-critical decisions with limited context. Moreover, attackers increasingly employ social engineering content that mimics benign UI elements and/or misuses third-party web infrastructure (e.g., ad networks, embedded widgets) to deliver malicious payloads–effectively bridging the gap between social and technical layers of exploitation.

Despite these advances, existing defenses are disjoint. They either treat code injection as a purely syntactic problem or treat deceptive injected content as a user perception issue, overlooking the fact that many modern attacks operate at the intersection of both. For example, clickjacking and fake browser dialogs exploit UI overlays (social engineering) while enabling script execution (technical exploitation). Similarly, low-tier ad networks often deliver payloads that deceive users and trigger code execution, bypassing traditional threat models.

A key limitation in existing solutions is the lack of an integrated defense strategy that proactively reduces the attack surface across both user-facing and system-level vectors.

That is, *there is a gap in systematic defenses that address both social engineering injections (e.g., deceptive overlays, misleading scripts) and code injection vulnerabilities within web applications.*

This fragmentation in the defense landscape creates blind spots that attackers can exploit. To date, there is no unified framework that systematically mitigates both social engineering content and code injection attacks by treating them as related manifestations of injection-based threats. Addressing this gap is crucial for building resilient web applications in an ecosystem where trust boundaries are fluid, attacker techniques are blended, and the line between human and machine-level compromise is increasingly blurred.

1.2 Thesis Contributions

To address these challenges, this thesis proposes a multi-layered defense strategy aimed at reducing the web attack surface by hardening the critical bridge to the Internet – the browser engine. The browser is the primary interface between users and the web, allowing users to surf the Internet without installing applications locally. However, such capability also intrigues adversaries to abuse the browser's functionalities by distributing malicious content and compromising the users. Since most social engineering and code injection attacks exploit browser-based interactions, strengthening the browser's security is critical for reducing the attack surface.

In chapter 3, we first present TRIDENT, a customized browser designed to detect and block social engineering attacks that are distributed at scale through low-tier ad networks. Our research reveals that adversaries leverage these ad networks to attract a large number of users to social engineering websites for monetization. TRIDENT indirectly identifies and mitigates social engineering threats by analyzing the behavior of ad scripts.

Next, in chapter 4, we present COINDEF, a security-enhanced Electron engine designed to defend against code injection attacks. Traditional XSS attacks on websites are typically confined to the browser, limiting their impact. However, Electron's architecture fuses web and native environments, significantly amplifying the power of XSS attacks. COINDEF enforces strict execution policies by validating the structural integrity of the abstract syntax tree and the execution context, thereby preventing unauthorized code injection in a comprehensive manner.

Finally, in chapter 5, we introduce COINDX, a framework for conducting root cause analysis of code injection attacks on JavaScript applications to help developers remediate vulnerabilities. Analyzing JavaScript applications is particularly challenging due to the dynamic nature of JavaScript and its complex dependencies. COINDX takes an innovative approach by constructing a simplified program based on a call stack trace generated by security alerts. It then applies iterative symbolic analysis on this simplified program to accurately reproduce the vulnerable program state, effectively avoiding state explosion and pinpointing the root cause of the vulnerability.

Collectively, TRIDENT, COINDEF, and COINDX contribute to attack surface reduction by proactively identifying, preventing, and analyzing hybrid cyber threats. TRIDENT reduces exposure to social engineering threats by detecting malicious advertisements before users can interact with them. COINDEF mitigates the risks posed by code injection in Electron applications by restricting unauthorized code execution at the engine level. COINDX strengthens web security by providing developers with precise root cause analysis, enabling them to remediate vulnerabilities efficiently. By integrating these three technologies, this thesis aims to minimize the exploitable entry points within the web ecosystem, thereby significantly reducing the attack surface for users and developers alike.

CHAPTER 2
LITERATURE SURVEY

This chapter presents the literature survey for the risks of social engineering attacks and code injection attacks in the web ecosystem, along with the countermeasures.

2.1 Web-based Social Engineering Attacks

Miramirkhani et al. [1] analyzed how tech support scams were distributed through advertising networks. Kharraz et al. built Surveylance [2], which is specifically designed to detect survey scams. Invernizzi et al. developed EvilSeed [3], a crawler that searches the Internet to identify risky websites that install unwanted software. Vadrevu and Perdisci [4] use visual clustering and heuristics to identify SE attack campaigns at the landing page level, which is done offline and does not focus on detecting SE-ads. These have studied web-based SE attacks through malicious advertising, and they either focus on detecting specific web SE attack vectors or lack a defensive method towards their findings.

Rafique et al. [5] built a classifier to identify Free Live Streaming online services. They found those free streaming players often had overlay ads, e.g., fake play buttons or fake close ads buttons, that were particularly designed to fit the flash player element. From their measurement, there are 5-6 overlays present on the video players. Furthermore, 93% of the video players are put under the overlays, which cover 90% area of the player. On average, a click on an overlay ad has a 50% chance of taking the visiting user to a malicious website. Though this study was done in 2015 and flash player has been depreciated since then, the same strategies are still widely used by the lower tier ad network [4]. The website stream2watch.com and its sister sites reported in this study are still active nowadays and accommodate millions of visits per month [1]. This work discovers that ads injected on free

[1]Traffic data is obtained from https://www.similarweb.com/website/stream2watch.sx

streaming websites may deliver malicious content, but it does not provide any defensive methods to protect users.

Zheng et al. [6] instrumented Chromium to measure the click interceptions on the Alexa top 250K websites. A click interception occurs when, for example, a user clicks on an HTML element to open example.com, but opens suspicious.com instead. An injected third-party script can easily achieve this goal by modifying an anchor tag's `href` attribute, registering event handlers that listen to mouse clicks and use `window.open` to open a new page, or putting visual deception elements to lure a user into clicking. The authors created a crawler to click every element on a website to trigger navigation and note down the destination URL in May 2018. The result reveals that 624 websites use different techniques to intercept clicks. The total visits to those websites, according to Similar Web API, were 43.3M per day, which is almost 70k per day per website on average. This vast traffic essentially increases the chance of getting baited by regular users. This work discusses the fundamental techniques adversaries can use to intercept users' clicks. Unfortunately, it also lacks a systematic way to detect their proposed harmful intercepts. The issues they found are still not addressed. For example, the authors gave an example website that included a malicious script in Figure 1(b). This script belongs to the AdSterra ad network. This ad network created more than 500 domains to serve its ads, leading to 7,644 social engineering websites in the study of [4]. Based on our datasets, AdSterra is still distributing such malicious scripts to its publishers to mislead end users.

Sanchez-Rola et al. [7] present the first comprehensive study of user clicks' possible security and privacy implications. This study proposes a click "Contract", which means what a user click is what the user should finally see. Nonetheless, after crawling 100k websites from Alexa Top domains and domains offering free content, roughly 20% of the websites contain an invisible overlay that intercepts users' clicks; moreover, 10% of all websites redirect the user to a completely different third-party domain. In addition, about

80% of websites mislead the users by reporting incorrect `href` attribute of links. Even worse, 45% of these link point to third-party domains. Finally, they reported that 65% of those websites have fake local clicks setup, which means the websites serve elements that should not be clickable but registered with event listeners to capture users' clicks.

These studies have shown that users often reach *Social Engineering Websites* (SE-websites) by interacting with malicious ads. More specifically, attackers are inclined to leverage low-tier ad networks to inject ads into many different publisher websites and use these ads to lure users to their SE-websites so that various attacks such as lottery scams, reward scams, tech support scams, etc., can be launched. Importantly, these low-tier ad networks often do not inject traditional ads onto the page. Instead, they inject Document Object Model (DOM) elements into ad-publishing web pages and leverage different social engineering tricks to lure users into clicking these elements to trigger ad network-driven navigation to a WSEA page. For instance, the ad network may inject a transparent overlay covering the entire publisher page and listen to users' clicks on any portion of the page. We refer to these non-traditional ads that leverage various SE tricks to lure users' clicks as *Social Engineering Ads* (SE-ads).

Clickjacking. Clickjacking is a UI redressing attack that uses multiple transparent or opaque layers to trick users into clicking on third-party content to bypass the same-origin policy [8]. The underlying motivation of this attack is to steal clicks on the first-party website to achieve specific actions on the third-party website the user has logged in, for example, making a financial transaction or clicking a "like" button on a social platform. Often, these third-party contents are loaded in iframes by malicious scripts. Framebusting [9] is a good defense against clickjacking. However, it degrades the user experience on websites that require cross-origin iframes, and the implementation inconsistencies are concerning [10]. Previous works [11, 12] rely on the users to verify what they have clicked, which is not comprehensive and has usability concerns [13].

Although clicks initiate most web-based social engineering attacks, they are not traditional clickjacking attacks. However, both attacks may use the same techniques nowadays.

Ad blocking. Generic ad blockers are efficient at blocking unwanted resources. However, they suffer from incompleteness and are accessible to evade [4, 14]. More advanced ad blockers [15, 16, 17, 18] employ ML techniques to complement the generic ad blockers. Unfortunately, they are not trained to detect SE-ads and block the subsequent events triggered by interacting with those SE-ads.

Din et al. [17] proposed PERCIVAL, a deep learning model put inside the image rendering pipeline of Chromium browser to classify images that are about to be rendered on the screen. Percival first collect images from Alexa top 1k websites and then label those images by comparing their URL with the EasyList. Percival achieves an accuracy of 96.76%. This tool classifies ads by the ad images. However, the DOM elements that intercept users' clicks do not render visible images, which can evade Percival easily.

Iqbal et al. [15] created a system, ADGRAPH, to detect ads and trackers using features extracted from a page's network activities. ADGRAPH leverage two sets of features: structural features contain the information of what kind of scripts initiate what network requests, e.g., is the script from `eval`?, is it a third party script?, the connectivity of the nodes, etc.; content features which express what those network requests are like, e.g., do they contain ad keywords?, what domain party it is?, what is the length of the URL? etc. Then ADGRAPH determines whether to block a network request by examining these features. This model outperforms the existing filter lists and can correctly distinguish benign ad/tracker resources blocked by filter lists. Siby et al. developed WEBGRAPH [18] to improve the robustness of ADGRAPH by removing the content features and adding a network, storage, and shared information flows.

ADGRAPH is tailored to block traditional online ads and trackers. Those SE-ads, such as fake buttons, overlays, and event listeners, can easily evade ADGRAPH due to their NON-AD nature. Creating those elements does not require any network requests except the first one to load the script, which can evade them by tweaking its URL a little, such as removing AD keywords and shortening its length. Therefore, ADGRAPH, serving for blocking general ads, will not work perfectly for those “fake” or “malicious” ads. WEBGRAPH [18] is an improved version of ADGRAPH tracking information flow by collecting more network activities. Such improvement does not capture the features demonstrated by SE-ads.

Browser Data Provenance.

Browser data provenance represents activities when visiting a website as a directed acyclic graph (DAG) that describes information flow between DOM objects (e.g., a JavaScript function registering a listener on a button). DAG of a browsing session can provide important insights into what a script does to the page in real-time. When a sensitive event occurs, for example, a new browser tab is created, the graph allows us to backtrack which script is responsible for the new tab.

JSGRAPH [19] instruments Chromium to log interesting DOM APIs to build a graph for forensic analysis offline. MNEMOSYNE [20] builds a provenance graph by leveraging the existing APIs in Chrome Devtool Protocol (CDP). PAGEGRAPH [16], as the successor of ADGRAPH [15], instruments the browser and expose its API through CDP, which sends a completed page graph when the web page emits `unload` event. These tools are efficient. Nonetheless, they are either for offline use or lack essential information concerning relationships between tabs.

2.2 Code Injection Attacks In The Web Ecosystem

Web Applications. Web applications are prone to Cross-Site Scripting (XSS) attacks, including *Stored*, *Reflected*, and *DOM-based XSS*. Stored XSS persists on the server and executes whenever users access the affected page. Reflected XSS occurs when input is immediately reflected in the response, while DOM-based XSS exploits client-side JavaScript to modify the DOM and execute scripts. XSS can lead to session hijacking, data theft, and unauthorized actions. Vulnerabilities arise due to improper input validation, unsafe functions like innerHTML, and lack of a content security policy (CSP). Third-party scripts and misconfigurations increase the risk. These inputs usually come from user input fields (e.g., comments, search bars), query parameters, and dynamic content rendering. Client-side JavaScript manipulation and external scripts expand the attack surface.

Node.js Applications. Node.js applications are vulnerable to *Command Injection* and *Prototype Pollution*, which is the goal of code injection in the native environment. Command injection allows attackers to execute system commands through unsanitized input in functions like `child_process.exec()`. Prototype pollution enables malicious manipulation of object prototypes, affecting application behavior globally. Vulnerabilities occur due to unsafe command execution, dynamic JavaScript properties, and reliance on third-party libraries. Poor input validation and outdated packages increase exposure to these risks. Attack surfaces include API endpoints that execute system commands, functions interacting with the filesystem, and JSON payloads in HTTP requests. Prototype pollution often targets libraries that extend or merge objects, like *lodash*.

Electron Applications. Electron apps combine Web and Node.js environments, exposing them to *Remote Code Execution* (RCE), *XSS*, and *Insecure IPC*. RCE can occur when `nodeIntegration` is enabled (e.g., through *open-redirect* to open a new window), allowing malicious scripts to execute Node.js commands. XSS in Electron escalates into RCE, while insecure IPC communication can lead to privilege escalation. Vulnerabilities arise from misconfigurations like enabling `nodeIntegration`, loading untrusted sources with `loadURL()`, and failing to use `contextIsolation`. Improper validation of IPC messages further increases the attack surface. Attack surfaces include untrusted web content, input fields in web views, and poorly implemented IPC channels. When `nodeIntegration` is enabled, injected scripts gain full access to Node.js, leading to system compromise. Recent studies highlight the security risks in applications built on the Electron framework. Xiao et al. [21] showed how shared contexts in Electron could escalate XSS attacks to severe RCE incidents. They developed XGUARD to prevent RCE, but it addresses only the symptoms, not the root cause of code injection attacks. Jin et al. [22] studied vulnerabilities in the UI components of Electron applications, proposing DOMTYPING to enforce DOM integrity, which effectively prevents code injection through DOM modifications but doesn't address dynamic code execution or *open-redirect* issues. Ali et al. introduced INSPECTRON [23], designed to identify misconfigurations in Electron applications. However, even with proper configurations, attackers can exploit Electron vulnerabilities [24] to gain privileged access.

Code Injection Mitigation. Security researchers have extensively studied code injection vulnerabilities in the web ecosystem over the past decade. Prior solutions, such as dynamic taint analysis [25, 26, 27, 28], achieve effective mitigation by instrumenting the browser engine to track data flows. However, these approaches often incur significant runtime overhead and are challenging to maintain due to the frequency of browser updates. Other efforts [29, 30], attempt to replace the dangerous `eval` with "safe" `eval` by modifying the way to invoke `eval`. While useful, this approach does not extend well to the Electron framework due to its broad attack surface and lacks tamper-resistance as they are in the same privilege layer as attackers for Electron applications. Additionally, whitelist-based solutions [31, 32, 33] enforce AST integrity to prevent web-based code injection, but they enforce policies before scripts are parsed, incurring high runtime overhead. These methods also lack the execution context needed to counter mimicry attacks in Electron applications, limiting their effectiveness in complex cross-environment applications.

Program Analysis Techniques for JavaScript. Various program analysis techniques have been developed to analyze JavaScript programs and uncover vulnerabilities. Symbolic analysis-based solutions [34, 35, 36, 37] are commonly employed to detect *prototype pollution* vulnerabilities by examining possible object inheritance issues and property changes. Dynamic analysis-based methods [38, 19, 20, 39, 40] focus on instrumenting the browser engine to gather runtime traces of JavaScript code, providing insights into potential vulnerabilities through behavior tracking. Although these techniques are effective for analyzing JavaScript behavior in traditional web environments, they cannot be directly applied to Electron applications for defending against code injection, due to the unique attack surfaces and shared contexts present in these hybrid applications.

Root Cause Analysis for JavaScript. RCA has been widely explored for binary programs but remains underdeveloped for JavaScript applications. Existing RCA frameworks such as ARCUS [41] and BunkerBuster [42] leverage symbolic execution and constraint solving to trace exploit origins in compiled binaries. However, these techniques are not applicable to JavaScript due to its semantic gap. For JavaScript applications, prior research has primarily focused on execution monitoring rather than RCA. PMForce [43] systematically analyzes postMessage handlers to detect security violations in client-side JavaScript. SilentSpring [44] explores prototype pollution vulnerabilities in Node.js applications by analyzing object dependency graphs. While these works contribute to understanding JavaScript security, they do not perform iterative symbolic analysis on reconstructed execution traces as COINDX does.

Symbolic Execution for JavaScript. Symbolic execution has been successfully applied to detect JavaScript vulnerabilities, but existing tools are not designed for root cause analysis. ExpoSE [45] is a symbolic execution engine for JavaScript that efficiently explores execution paths by symbolizing input-dependent variables. NodeMedic [46] extends symbolic execution to analyze security flaws in the Node.js ecosystem. However, these approaches are limited to bug detection and do not attempt to reconstruct execution traces for RCA.

COINDX builds upon symbolic execution by introducing iterative symbolic analysis, allowing it to refine and resolve symbolized undefined functions and variables. Unlike previous work, CoInDx operates on simplified subprograms reconstructed from call stack traces, reducing state explosion and enabling more precise vulnerability diagnosis.

JavaScript Event Tracing and Instrumentation. Event tracing and instrumentation have been widely used to monitor JavaScript execution for various purposes, such as performance profiling and debugging. Chrome DevTools [47] provides a built-in tracing tool to capture and analyze JavaScript execution events. Synode proxies `eval` and `exec` functions to trace untrusted input flows in Node.js applications. CoInDx leverages event tracing to capture execution events and reconstruct call stack traces for RCA.

CHAPTER 3
TRIDENT: TOWARDS DETECTING AND MITIGATING WEB-BASED SOCIAL ENGINEERING ATTACKS

3.1 Introduction

Social Engineering (SE) has become an ever more sophisticated and common attack method [48]. Recent surveys report that 84% of hackers leverage *Web-based Social Engineering Attacks* (WSEAs) in the cyber kill chain with a high success rate [49, 50, 51]. Moreover, 64% of companies have experienced web-based attacks, and 62% have seen phishing and WSEAs [52]. Attackers also target regular Internet users. The Federal Trade Commission received 2.8 million fraud reports from consumers in 2021 in the United States, which led to a $5.8 billion financial loss [53]. The top 3 fraud categories – impostor scams (e.g., romance scams and tech support scams), online shopping scams, and reward and prize scams (e.g., survey scams) – are commonly seen on the Internet [1, 2, 4, 14]. These scams account for $2.3 billion of losses, which almost doubled from 2020.

To mitigate the impact of WSEAs, researchers have been studying and developing countermeasures. For example, Miramirkhani et al. analyzed tech support scams [1], Kharraz et al. built Surveylance [2], which is specifically designed to detect survey scams, and Invernizzi et al. developed EvilSeed [3], a crawler that searches the Internet to identify risky websites that install unwanted software. However, these previous works only focus on specific SE attack vectors. Because of the diversity of WSEAs that users can encounter [48], there is an urgent need for new and more effective in-browser defense systems that can accurately detect generic WSEAs.

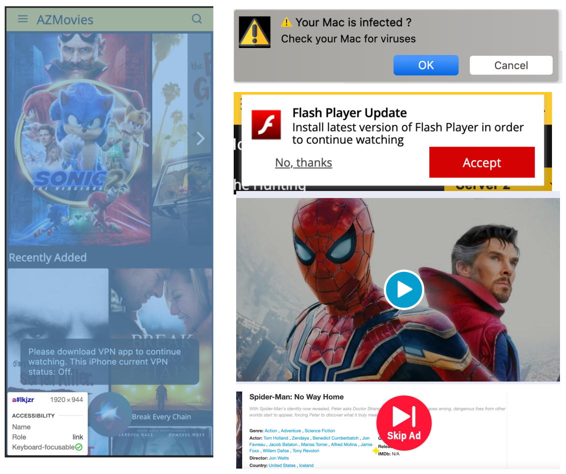
This chapter proposes a new defense system that aims to detect and block generic

WSEAs in real-time, while the user is browsing the web.

The main challenge we face is that *directly* detecting malicious web pages related to WSEAs is extremely difficult due to the large variety of social engineering tactics attackers can employ and the freedom they have in building malicious content. Therefore, in this work, we investigate how to *indirectly* detect and block WSEAs at their inception before the user interacts with the related scam content.

Recent works have shown that users often reach *Social Engineering Websites* (SE-websites) by interacting with malicious ads [5, 6, 54, 2, 1, 55, 56, 4]. More specifically, attackers are inclined to leverage low-tier ad networks to inject ads into many different publisher websites and use these ads to lure users to their SE-websites so that various attacks such as lottery scams, reward scams, tech support scams, etc., can be launched. Importantly, these low-tier ad networks often do not inject traditional ads onto the page. Instead, they inject DOM elements into ad-publishing web pages and leverage different social engineering tricks to lure users into clicking these elements to trigger ad network-driven navigation to a WSEA page. For instance, the ad network may inject a transparent overlay covering the entire publisher page and listen to users' clicks on any portion of the page. We refer to these non-traditional ads that leverage various SE tricks to lure users' clicks as *Social Engineering Ads* (SE-ads).

As mentioned above, SE-ads are non-traditional ads. They are often invisible, malicious ads that, when interacted with, navigate the browser to a landing page containing SE attacks. A previous study [6] reported that attackers often leverage two types of techniques (registering click event listeners and injecting invisible links shown in Figure 3.1a) to deploy invisible malicious ads to steal users' clicks. In addition, SE-ads also appear as misleading in-page components, such as an in-page push notification or fake ”Skip Ads” or ”Play” buttons, as illustrated in Figure 3.1b, to induce users to interact with them. Given these features, we can see that SE-ads are not traditional ads, although we still refer to them as ads because they are injected into a publisher page by ad



Please download VPN app to continue watching. This iPhone current VPN status: Off.

Your Mac is infected ?
Check your Mac for viruses

Flash Player Update
Install latest version of Flash Player in order to continue watching

(a) (b)

Figure 3.1: Example SE-ads: (a) An invisible link covering the whole viewport to force users to click; (b) Deceptive elements (fake notification, ”Play” and ”Skip Ad” buttons) to lure users into interacting with them.

networks. Therefore, rather than attempting to detect WSEAs directly by analyzing their contents and/or URLs related to the WSEAs, we focus on detecting their leading causes, namely SE-ads.

Although most SE-ads come from ad networks, existing ad-blocking tools are not effective in detecting SE-ads for two major reasons. First, the ads are not generally visible, so ad-blocking tools such as Percival [17], which block ads through the image rendering pipeline, cannot detect them. Second, the ad networks that distribute these SE-ads are extremely motivated to evade ad blockers [4]. For example, in Table 3.7a and Table 3.7b, we show that neither commercial ad blocker [57] nor the most recent state-of-the-art ML-based ad-blocker [15] is effective against SE-ads.

To address the challenge of detecting SE-ads to mitigate WSEAs, we propose TRIDENT – a novel system that detects SE-ads in real-time and blocks the subsequent web-based social engineering attacks. To this end, TRIDENT develops an in-memory graph representation of a web page and its activities, for example, registering event listeners to intercept clicks, manipulating Document Object Model (DOM) to inject deceptive elements shown in Figure 3.1, which we call the *Web Action History Graph*

(WAHG). During a user's browsing session, TRIDENT uses the WAHG to protect users from potential SE attacks that are launched through SE-ads in real-time. Specifically, during a user's browsing session, TRIDENT vets each navigation event to determine if it is initiated by a SE-ad. When TRIDENT detects the navigation is related to a SE-ad, it redirects the user to an interstitial page to make the user aware of the danger ahead.

To extensively evaluate TRIDENT, we crawled over 100K websites obtained from October 2021 to January 2022, which allowed us to collect 258,008 unique JavaScript files and their running contexts, including over 1,479 SE-ads. In our experimental evaluations, we found that TRIDENT can detect SE-ads with an accuracy of 92.63% with a precision of 90.63%, and a recall of 96.28%, outperforming prior work [15] by more than 10%.

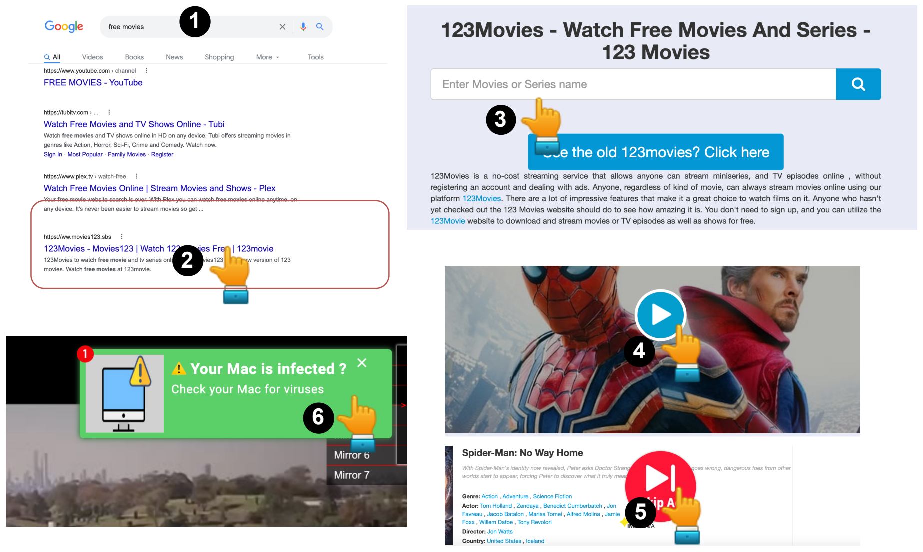
3.2 A Motivating Example & Challenges

In this section, we present a real-world example of SE-ads hosted on a high-ranking search result from Google Search and discuss the limitations of prior, generic ad-blocking work.

3.2.1 A Motivating Example

In this section, we introduce a real-world motivating example that demonstrates exactly how victims arrive on SE-websites by interacting with the SE-ads.Figure 3.2 gives a clear description of how an ad network manipulates the users to interact with SE-ads by including JavaScript (JavaScript) code into a content-sharing website, also known as an ad publisher.

Google Search Result Leads to SE Attacks. The attack begins on the popular Google search engine where the victim, Alice, completes a Google search for the phrase, "free movies". Despite Google Search being one of the most highly-respected search engines, it still struggles to filter out websites that include malicious content from the top results of the search. For instance, at the time of writing, Google Search returns an illegal movie sharing website (*ww.movies123.sbs*) in the *top 4 results* for the query "free movies". As a result, Alice is unfortunately supplied with a mixture of benign and malicious search results. As


Google
free movies
1
All
Videos
Books
News
Shopping
More
Tools
2
123Movies - Watch Free Movies And Series - 123 Movies
Enter Movies or Series name
3
See the old 123movies? Click here
4
5
Skip Ads
Your Mac is infected ?
Check your Mac for viruses
6
Mirror 6
Mirror 7

https://www.youtube.com › channel
FREE MOVIES - YouTube

https://tubitv.com › ...
Watch Free Movies and TV Shows Online - Tubi
Watch **free movies** and TV shows online in HD on any device. Tubi offers streaming movies in genres like Action, Horror, Sci-Fi, Crime and Comedy. Watch now.
Sign In · Most Popular · Family Movies · Register

https://www.plex.tv › watch-free
Watch Free Movies Online | Stream Movies and Shows - Plex
Your **free movie** website search is over. With Plex you can watch **free movies** online anytime, on any device. It's never been easier to stream movies so get ...

https://ww.movies123.sbs
123Movies - Movies123 | Watch 123Movies Free | 123movie
123Movies to watch **free movie** and tv series online ... Movies123 ... new version of 123 movies. Watch **free movies** at 123movie.

123Movies is a no-cost streaming service that allows anyone can stream miniseries, and TV episodes online , without registering an account and dealing with ads. Anyone, regardless of kind of movie, can always stream movies online using our platform 123Movies. There are a lot of impressive features that make it a great choice to watch films on it. Anyone who hasn't yet checked out the 123 Movies website should do to see how amazing it is. You don't need to sign up, and you can utilize the 123Movie website to download and stream movies or TV episodes as well as shows for free.

Spider-Man: No Way Home

With Spider-Man's identity now revealed, Peter asks Doctor Strange ... goes wrong, dangerous foes from other worlds start to appear, forcing Peter to discover what it truly mean...

Genre: Action , Adventure , Science Fiction
Actor: Tom Holland , Zendaya , Benedict Cumberbatch , Jon Favreau , Jacob Batalon , Marisa Tomei , Alfred Molina , Jamie Foxx , Willem Dafoe , Tony Revolori
Director: Jon Watts
Country: United States , Iceland

Figure 3.2: Motivating Example: ① Alice typed "free movies" in Google Search. ② Alice clicked on the fourth result. ③ Alice wanted to search for "Spider-Man: No Way Home", so she clicked the input box to start typing. ④ Alice found the movie and clicked the play button. ⑤ Alice was annoyed by the ads and clicked "Skip Ads". ⑥ The movie finally played, and an in-page notification ward Alice that viruses had infected her Mac. ③, ④, ⑤, and ⑥ are SE-ads. They are invisible elements, fake buttons, or in-page push notifications.

this is one of the top results, many users may click on the link to *ww.movies123.sbs*.

At first glance, this website appears innocuous while also providing a diverse selection of popular, well-known movies. However, under the hood, *ww.movies123.sbs* includes scripts obtained from low-tier ad networks that have one goal in mind: to trick visitors into clicking on the SE-ads these scripts so they can make money from their malicious activity. Looking at Figure 3.2, several mouse event listeners, registered on `#document`, intercept Alice's click on the search box at step ③. In fact, any click on the page triggers the listeners, which dynamically determines what page to open for Alice. Due to these click interceptions, Alice is obligated to interact with SE-ads when trying to search for a movie to watch. Before Alice can type the movie name, the SE-ad opens up a new tab, which asks Alice to install "Rainbow Blocker" which is a known AdWare [58]. When Alice arrives at the spider-man movie, she clicks on the play button at step ④ and "Skip Ads" at ⑤. Unfortunately, the SE-ads are attempting to trick Alice into downloading browser extensions, which claim to be necessary to watch the movie. However, after further manual analysis of their code, we found that these extensions were trackers and AdWare, which track users and harm their digital privacy. After seven clicks, Alice could watch the movie after closing all the opened tabs. While Alice is watching, an in-page notification pops up to warn Alice that her Mac is infected. Alice becomes nervous and clicks on the banner to download software to clean her mac at step ⑥. This software was confirmed to be an AdWare by VirusTotal [59].

Low-tier Ad Networks Are Popular but Hijack Clicks. Ad publishers are inclined to cooperate with low-tier ad networks, which pay more than high-profile advertising platforms [60]. For example, AdSterra pays USD $17.55 for a click [61], which is 10x more than what Google ads pay.

These low-tier ad networks, therefore, are strongly motivated to elicit clicks by the fact that advertisers pay more for clicks than impressions [62]. To achieve the goal of harvesting as many clicks as possible, these low-tier ad networks may use SE tricks. As described by the reverse engineered ad scripts in Figure 3.3b in the Appendix, they inject in-line scripts to insert a transparent layer and register a mouse event listener. The visitor is then forced to trigger the event listener to open a new window and load ads. This approach is extremely different from what the high-profile ad networks do, and does not follow the general standards [63, 64, 65, 66]. In contrast, looking at the pseudo code from Google ads in Figure 3.3a, the ad publisher prepares a container for the ad script to inject an iframe that can isolate the ad's contents such that it cannot directly access the first party's contents. However, the standard ad publishing process is not likely to get more clicks.

Therefore, these content-sharing websites, as ad publishers, prefer low-tier ad networks even though these ad networks may use SE tricks to get more clicks. Thus, the low-tier ad networks can transfer a fraction of their high revenue obtained from advertisers to those ad publishers. The advertisers are satisfied by having more ads exposed to users which would turn into a higher conversion rate. This business model undoubtedly is intriguing to attackers and provides them with opportunities to spread malicious content (e.g. unwanted software, WSEAs, etc.).

3.3 Design

3.3.1 Overview

In this section, we introduce TRIDENT, a novel real-time detection system for identifying *Social Engineering Ads* (SE-ads) and blocking navigation to potential *Social Engineering Websites* (SE-websites). At a high-level, TRIDENT takes advantage of two intuitions: (1) SE-ads use tricks (e.g., click-jacking and social engineering) to lure users into interacting with strategically placed DOM elements and triggering unwanted browser navigation; and (2) SE-ads often navigate the user to malicious websites that host social engineering attacks (e.g., tech support scams, malicious downloads, etc.). Therefore, to detect SE-ads and block the subsequent events, TRIDENT monitors the user's browsing session and vets each navigation to determine if it may be related to a SE-ad. More specifically, during this vetting process, TRIDENT extracts features related to how this navigation was initiated and passes these features to its classification module. Finally, if TRIDENT determines this navigation is SE-ad related, TRIDENT presents an interstitial page to warn the user of the danger ahead.

While prior approaches [2, 1, 3] focus on specific SE attack vectors, TRIDENT takes a more generic approach that relies on the causality of how users end up in SE-websites. Namely, TRIDENT detects WSEAs by detecting the anomalous techniques, which intercept users' clicks by all means, routinely used by SE-ads which often lead to websites that host SE attacks. TRIDENT achieves this by leveraging the design illustrated in Figure 3.4. First, TRIDENT instrument Chromium by extending the Chrome DevTools Protocol framework (CDP) [47] with a new agent, Social-Engineering agent (`SEAgent`). While a user is visiting a website, the `SEAgent` collects JavaScript actions (e.g. event listener registrations, DOM modifications) and sends them to a background daemon. The background daemon builds an in-memory graph representation of the web page and its activities, which we call *Web Action History Graph* (WAHG). While TRIDENT builds and updates the WAHG, it also extracts *property features*, *action features*, and *consequence features* about the page's JavaScript code from the graph. These features describe how these scripts are included, what contexts the scripts are running in, and what the scripts do on a web page, respectively. These features are then passed to TRIDENT's classification module, which classifies the navigation as related to SE-ads or benign.

In the remainder of this section we first give an example of the Web Action History Graph of the motivating example in subsection 3.3.2, and then explain how TRIDENT instruments the Chromium browser to collect JavaScript actions in subsection 3.3.3. Next, we discuss how the WAHG is constructed while the user is browsing a website in subsection 3.3.4, and the feature extraction along with it in subsection 3.3.5. Finally, we introduce the classifier in subsection 3.3.6.

3.3.2 Web Action History Graph

The *Web Action History Graph* (WAHG) is a graph-based representation of a web page. Nodes in the graph represent web objects (e.g., window, resource, DOM node, etc.) and edges represent causal relationships between objects. For example, when a script inserts a new DOM element into the DOM tree, an edge from the script to the element will be connected into the WAHG. We formally define all graph objects and relationships between the objects in Table 3.1.

To demonstrate the WAHG's capability to represent SE-ads, we provide an example WAHG of the suspicious publishing page, "www.movies123.sbs", that Alice navigates to in the motivating example (③) in Figure 3.5. Additionally, to improve the clarity, the example only contains the portions of the WAHG related to two SE-ad attacks on the page. The first SE-ad is launched by an inline script on "ww.movies123.sbs" and is represented by the set of nodes connected by the blue edges. The inline script initiates the deployment of

Table 3.1: WAHG Objects, Relationships, and Key attributes.

Object Type	Attributes
Frame	security_origin, url, is_page
Window	url
Resource	url, type
Script	url, is_isolated, frame_owner
Function	url, is_eval_or_new_function, location
DOM Node	tag_name, is_inserted_by_js
HTML Parser	frame_owner

(a) Graph Objects. The unique ID for each object is omitted.

Relationship	Example
Attached	Frame → Frame
Compiled by	Script → Frame
Created	Script/Function → Frame
Add event listener	Script/Function → Function
Listen to events	Function → DOM Node
Add callback function	Script/Function → Function
Navigated	Frame → Frame
Opened	Frame → Window
Load	Window → Frame
Respond	Parser/Script/Function → Resource
Response	Resource → Parser/Script/Function

(b) Relationship between objects.

the SE-ad by scheduling a delayed callback to be executed using `setTimeout`. When this callback is executed, the callback adds a new mouse event listener onto the `#document` element which consequently covers the whole viewport. When Alice clicks on the input box to search for a movie, the click is effectively hijacked. The mouse event listener on `#document` is fired and redirects Alice to the malicious "Rainbox Blocker" website. The second SE-ad attack is shown by the dashed yellow path, which is initiated by the same inline script, but with a different deployment technique. More specifically, the previously mentioned inline script injects a third-party ad script that also uses `setTimeout` to create an iframe and insert it onto the page. If Alice clicks on the "Skip Ad" button, which is rendered in the iframe, it would cause Alice to download a malicious Chrome Extension.

This example demonstrates the fine-grained details related to a web page that is embedded into the WAHG.

3.3.3 Social-Engineering Agent

The Social-Engineering Agent (`SEAgent`) module resides within the browser. During a user's browsing session the `SEAgent` collects the necessary events to construct the WAHG. To minimize our footprint in the browser, we implemented the `SEAgent` on top of the Chrome DevTool's Protocol framework (CDP) [47].

CDP implements several "domains" where each domain has a set of APIs and events related to a particular aspect of a web application (e.g., `DOM`, `Network`, or `Debugging`). Internally, each domain relies on a backend "Inspector Agent" that encapsulates the necessary instrumentation to support the domain. At first glance, it may appear that the existing domains may be enough to build the WAHG. For example, the `DOM` domain provides information for DOM modifications, and the `DOMDebug` domain exposes an API to collect current event listeners. Unfortunately, we found that existing domains were not capable of supporting real-time information collection, which requires an event to be emitted immediately when the hooked function is called other than calling APIs to collect information proactively. For example, the `DOMDebugger.getEventListeners` API collects current event listeners on the DOM at the query time. We have to call this API frequently to capture every registered and removed listener, which is cumbersome and risky that a malicious listener may be removed when we call this API. Moreover, the `Debugger` domain does not implement event hooks for JavaScript executing stack, which is essential for JavaScript action attribution discussed in subsection 3.3.4.

To address this problem, we implement the `SEAgent`, which implements four types of hooks to collect JavaScript actions for constructing the WAHG in real-time. All the hooks are implemented to emit events. Whenever the hooked API is called, it emits an event immediately. For example, the instrumentation in event listener registration collects the

Table 3.2: Instrument Hooks to Construct WAHG.

Hooks	Description	Locations
DOM	Record DOM actitivties including adding and remove DOM nodes and modifiying nodes' attributes and style. Attribute the operation to a JS function if applicable.	Node creation, insertion, and removal
		Node attributes modification
		Node style update
Page	Record frame activities including iframe attachment and detachment, frame navigation, and opening new tabs. Attribute the operation to a JS function if applicable.	iframe attach and detach
		Frame navigation
		Opening new windows
Network	Record network activities including what resources are requested and who are responsible for the requests	Network requests
		Network responses
Script	Record JavaScript activities including what scripts are compiled and executed, what user callbacks are added, and what event listeners are registered.	Script compilation, execution
		Function invocation
		Add user callbacks
		Add event listeners

`event_target`, `event`, and the listener `function` whenever a script or function calls `addEventListner` to meet the real-time requirement for feature collection. The details of the hooks are listed in Table 3.2.

3.3.4 WAHG Construction

In this section, we discuss in detail how TRIDENT uses the event logs collected by the `SEAgent` to progressively construct the WAHG in real-time.

TRIDENT parses every event and translates the results into nodes and edges. There are two important attribution steps that TRIDENT performs: JavaScript attribution, which associates DOM events to a responsible JavaScript file, and navigation initiator attribution, which determines which script requests the navigation such that TRIDENT only needs to inspect paths to this script node on the WAHG instead of inspecting all the script nodes. We discuss in detail how both tasks are completed in the remainder of this section.

JavaScript Attribution. TRIDENT needs to attribute all DOM events to the accountable script. To do so, for each interaction and event we attribute the event to the current executing JavaScript function. For instance, when the executing script "../..7d94.js" inserts an event listener onto the page, we connect the script to the listener in Figure 3.6a. This approach addresses most cases for finding the responsible JavaScript file, However,the two global JavaScript functions, `eval` and `Function`, pose challenges when we are trying to attribute events to the correct functions or scripts. For example, when an external script loads, it invokes `eval` to evaluate a JavaScript code snippet. This process requires compiling the snippet and generates a new script object, but this snippet will not have a valid source (or URL). In these cases, we assign any events caused by the snippet to its caller's URL. We use the same approach for the `Function` API as it works similarly to `eval`.

Navigation Initiator. There are two types of navigation initiators: a script or a user's action (e.g., clicking a link, typing in the address bar). Finding the initiator to a navigation event helps us reduce the analysis space by avoiding analyzing all scripts, because attackers have to navigate the users to the websites under their control. Figure 3.6a presents a JavaScript function initiator, the click listener, which opens a new page. By analyzing the WAHG, TRIDENT can locate the responsible script that may lead to a SE-websites. Obviously, not all navigation events are initiated by JavaScript code directly. In Figure 3.6b, an anchor tag is inserted by a timer callback function. When the user clicks on the link, it opens a new window. Based solely on the information on this path, TRIDENT cannot determine what code is responsible for the navigation. To handle these cases, TRIDENT learns what `href` attribute is assigned to or updated for all the anchor nodes. It connects the JavaScript function that modified or updated the anchor node to the new window by matching their URLs.

3.3.5 Feature Extraction

In this section, we discuss the features used to learn the characteristics of malicious scripts and benign scripts.

Table 3.3: Feature Groups Used by TRIDENT.

Property Features
execution context (first party or third party frame)
script type (inline, remote file, `eval`, or `function`)
owner (first party or third party)
requestor (HTML parser or another script)
requestor's properties
Actions Features
register event listeners (`event_type`, `event_target`)
add timer callbacks (`setTimeout`, `setInterval`)
insert DOM nodes (`node_type`)
open new windows (`url`, `target`)
initiate same-tab navigation (`url`)
attach iframe (`src`)
modify DOM node attributes (attributes)
send network requests (`resoure_type`, `url`)
Consequence Features
of redirect hops
of unique domains
redirect type (JS-driven, response-header-driven)

Feature Descriptions

TRIDENT's features are divided into three groups – *property*, *action*, and *consequence* features – as shown in Table 3.3. The first group introduces what type the script is and where it comes from; the second group describes what the script does on the web page; and the last group contains redirect information. We leverage our domain experts' intuitions on web development and experiences from previous studies [6, 54] to choose these features to describe what happens before and after navigation.

Property Features. *Property* features target the properties of a script including how the script is included in a web page, who owns the script, and the context it is running in. TRIDENT determines the property features when a script is compiled and executed. If the script is inserted to the web page by another script, TRIDENT adds the requestor's properties, too. First-party scripts are usually included by the website operator, which implies they are mostly trusted, whereas third-party scripts (e.g., ad scripts from ad networks) are unverified and should not be trusted. As described in the motivating example in subsection 3.2.1, the legitimate ad scripts follow the FTC rules [64] to inject ads, for example, by isolating their ad contents inside an iframe. In contrast, SE-ad scripts are strongly motivated to elicit user's click by all means. Therefore, TRIDENT uses this feature group to learn whether a suspicious action should be trusted or not.

Action Features. *Action* features represent the behaviors exhibited by a script on the web page. These actions primarily are related to click hijacking including registering event listeners, adding large hyperlinks, and injecting visually deceptive elements. Each action becomes an edge in the WAHG. TRIDENT extracts the features from both the node's and the edge's properties. For instance, the *register event listeners* feature considers the `event_type` of the edge and `event_target` of the target node. More specifically, a JavaScript function registers an event listener that listens to mouse events on a specific DOM element. This DOM element is the `event_target`. TRIDENT checks whether this DOM element is a JavaScript inserted DOM Node or a built-in large element (e.g. `#document`, `body`). For actions involved in network requests such as *open new windows*, *attach iframe*, *initiate same-tab navigation*, and *send network requests*, TRIDENT examines the URL to determine where the resources are from. This feature group helps TRIDENT learn to separate malicious activities from benign ones. For example, appending an transparent hyperlink covering the whole viewport is more suspicious than adding a visible iframe to load benign ads.

Consequence Features. *Consequence* features describe what happens after the navigation. We extract the URLs in the redirect chain and collect the number of unique domains. TRIDENT also checks whether the redirect is initiated by JS or a HTTP response header. We consider the redirect chain between the first page and the eventual landing page because the window directly opened by clicking an ad usually is not the eventual landing page [67, 4]. This is from the intuition that ad networks need to determine what ad to present by collecting the user's cookies before making a decision. Unlike clicking on ads, clicking on a link to an article usually directly opens the article without any redirects because the website knows where the user is heading. Therefore, redirects between the opening action and the final landing action are good indicators of ads. This is useful for TRIDENT when determine whether a newly opened tab is for ads or not.

To conclude, TRIDENT's main goal is to detect navigation made by clicking benign ads, links, or SE-ads. Simply put, benign ads follow FTC rules which create iframes that do not intercept users' clicks; genuine links usually do not need to redirect the users multiple times; and SE-ads steal users' clicks by all means and redirect the users to SE-websites.

3.3.6 Blocking SE-ads related Navigation

The final portion of TRIDENT is its classification module. When a navigation is about to occur, the extracted features discussed in subsection 3.3.5 are passed to the classification module, which will classify the navigation as SE-ad-related or benign. If the navigation is determined to be SE-ad-related, TRIDENT will block the navigation to prevent the user from being directed to the social-engineering attack. Internally, TRIDENT uses a random forest [68] classifier for classification. We configure the random forest as an ensemble of 100 decision trees with each decision tree using $\sqrt{N}$ features, where N is the total number of features.

During visiting, the `SEAgent` send events to the post-processing daemon continuously, which build the WAHG. When a navigation request is scheduled, the daemon extracts the features except the consequence features and sends them to the classifier. When the navigation is about to commit, the daemon receives the updated consequence features and reruns the classifier before the landing page commits. When the classifier classifies a navigation request as malicious, the `SEAgent` inserts an interstitial warning page to make the user aware of the dangers ahead. Note that we use one single model rather than two models, trained with and without consequences features, because the performance difference is minimum as shown in subsection 3.3.5.

3.4 Evaluation

This section discusses the extensive experimental evaluations we completed for TRIDENT and compares TRIDENT with the state-of-the-art tools. Our evaluations address the following research questions:

RQ1: How accurately can TRIDENT detect navigation initiated by SE-ads?

RQ2: Are the features used by TRIDENT understandable and robust?

RQ3: How well does TRIDENT perform compared with the state-of-the-art tools?

RQ4: What is the runtime performance overhead for `SEAgent`?

3.4.1 Experiment Setup

In this section, we discuss the websites used in our evaluation and how we simulated user actions to trigger SE-ads and navigation to SE-websites for data collection.

Data Source. Our data collection process relied on *publicwww.com* (P.W.) [69], a popular source code search engine, to collect scripts that may deploy SE-ads. We obtained over 100,000 ad publisher websites by searching JavaScript code snippets on P.W. by following the approaches used in the study [4]. These JavaScript code snippets were obtained by analyzing websites, which were open-sourced in that study, and websites we encountered by searching for free content-sharing websites, which prefer to include low-tier ad networks as suggested by the prior research [5].

Table 3.4: Navigation Events Made by Scripts in The Training Dataset.

Class Label	New-Tab Nav.	Same-Tab Nav.
Malicious	1,358	121
Benign	5,726	250,803

Crawler Design. Unlike prior works [15, 16, 18] that only crawl the Internet by loading the home page, this work requires a crawler to interact with as many SE-ads as possible. We built the crawler on top of Puppeteer [70] to simulate users' interactions with the web pages, and developed a clicking strategy conducive to triggering navigation events. First, we collect anchor elements that point to a different origin and place them in an anchor node pool. Additionally, we collect elements with mouse listeners in a mouse event pool. Because large elements have a higher chance of being clicked, we sort the DOM nodes in descending order of the element's bounding box size to prioritize the elements that are most likely to capture a real user's clicks. Our crawler clicks the elements in these pools one-by-one. If a click triggers navigation, the crawler takes a screenshot of the navigated page.

We deployed the crawlers into 20 docker containers simulating users' interactions with websites from October 2021 to January 2022 to collect training data and in March 2022 to collect data for examining TRIDENT's robustness. We use these two datasets to evaluate the accuracy, investigate TRIDENT's false positives and false negatives, and compare with the state-of-the-art tool, which we will discuss in detail in the following sections. We will refer to the first dataset as the training dataset and the second as the testing dataset.

3.4.2 Ground Truth & Dataset Cleaning

This section first introduces the techniques we used to collect ground truth for the datasets and then discusses our approaches to cleaning and balancing the datasets.

Ground Truth. Prior works [15, 16] rely on EasyList and EasyPrivacy [71] as the ground truth to label ads related URLs. Unfortunately, these lists focus on generic ads. Using these lists as the ground truth would make TRIDENT target generic ads, rather than SE-ads, which is not our goal. To identify the ground truth in our datasets, we developed a semi-automated approach as the following.

- **Landing page screenshots clustering.** During crawling, when a new tab is open, or cross-origin navigation occurs, the crawler will take a screenshot of it. Following the methodology in [4], we cluster the screenshots and manually review each cluster to identify whether a landing page is a SEW. If it is, we identify the script responsible for initiating this navigation and label it malicious.

- **Categorical BlockList, Google Safe Browsing, and VirusTotal.** We choose three services for identifying whether a website is malicious or not, a categorical BlockList [72] on Github, which is popular in the community and is updated frequently, Google Safe Browsing (GSB) [73], and VirusTotal (VT) [59]. We consider a URL malicious if it falls in the buckets of Malware, Scam, Abuse, Phishing, and Fraud in the BlockList, is determined unsafe by GSB, or is flagged out by at least one of the engines in VirusTotal. Then, we feed all landing page URLs, and the URLs in the redirect chain to these three services to label them automatically. If a page's URL is labeled malicious, we find the script which initiated the navigation to this page and label it malicious.

We aggregate the results from the two components. A script is considered malicious when either of the two components says it is malicious.

Datasets Cleaning. We found that the datasets were heavily imbalanced after labeling. There were two problems in the datasets: (1) the data was heavily imbalanced between classes, and (2) the data was imbalanced within the negative class (e.g., more scripts for rendering first-party contents than scripts for injecting third-party ads). This is expected because benign scripts are ubiquitous. Training TRIDENT directly on this imbalanced dataset would undoubtedly produce a poor model. There are generally two strategies to overcome the imbalanced dataset problem: (1) over-sample the minor (malicious/positive) class, or (2) under-sample the majority (benign/negative) class. To address our problems, we decided to under-sample the negative class as recommended by the state-of-the-art techniques [74, 75] to reduce the false-negative rate as our goal is to detect SE-ads as accurate as possible.

Additionally, we removed "silent" scripts that do not invoke any DOM APIs of our interest and under-sampled the same number of positive class from the negative class, which addressed the first problem. To address the second problem, we analyzed the distribution of the features and found that benign scripts tended to navigate the users in the same tab. In contrast, the malicious scripts preferred to open new windows, as shown in Table 3.4. Randomly sampling from the benign class would yield a large portion of same-tab navigation entries, making a performant classifier. However, this classifier would not generalize to websites that open windows in new tabs, which are data points near the classification border. Therefore, we need to choose more samples near this border, in this case, more entries in the new-tab navigation from the benign class. After running this experiment, we chose 50% from new-tab entries and 50% from the same-tab ones. We will explain why we choose this ratio in subsubsection 3.4.3.

3.4.3 TRIDENT Performance

To answer **RQ1**, we evaluated our model using 10-fold cross-validation on the training dataset and reported the average accuracy. Next, we discuss the disagreement between TRIDENT and the ground truth data by testing the model on 1,000 websites sampled from the testing dataset.

Table 3.5: Model Performance with Different Approaches of Under-sampling the Majority Class.

New-tab Nav.	Same-tab Nav.	Accuracy	Precision	Recall
100%	0%	87.76%	86.69%	89.31%
90%	10%	88.30%	86.09%	91.68%
50%	**50%**	**92.63%**	**90.63%**	**96.28%**
0%	100%	99.76%	99.78%	99.43%
Random		99.36%	99.14%	99.59%
No Sampling		97.69%	89.71%	76.39%

Accuracy

This section discusses how we tackled the imbalanced data problem and describes the model's performance trained with the balanced dataset.

First, we trained the model with the raw imbalanced dataset (no sampling), which gave us a good accuracy rate but bad precision and recall rates, as shown in Table 3.5. Next, to improve the performance, we used five approaches to balance the dataset. Table 3.5 presents the results. Notably, the more STN entries we sample, the better the model performs. However, it lacks generality. When we trained the model with NTN benign samples (all benign data points near the borderline), the accuracy dropped to 87.76%. Although the model has the lowest accuracy, this situation (each navigation opens a new tab) is implausible. As shown in Table 3.4, 97.77% of the navigation events happened in the same tab for the benign class. Therefore, to be conservative and include a good number of data points near the borderline from the benign class, we decided to use 50% from the NTN entries and 50% from the STN entries for the benign samples to balance the dataset.

With this training dataset, TRIDENT detects SE-ads related navigation with 92.63% accuracy, 90.63% precision, and 96.28% recall.

False Positives & False Negatives

We now analyze TRIDENT's false-positive and false-negative cases. We selected 500 random websites from the benign class and 500 from the malicious class. We then tested this dataset on the model, which yielded the accuracy of 93.01%, the precision of 93.34%, and the recall of 93.02%. From these 1,000 websites, TRIDENT reported 20 false negatives and 136 false positives. We now discuss these cases with examples.

False Positives. We noticed that 72% of the false positives had ad scripts from PopCash [76]. After revisiting these websites manually, we confirmed these scripts injected SE-ads on the page. Because the block lists we used to label the ground truth are imperfect, it is normal to miss these scripts. Fortunately, TRIDENT detected them based on their behaviors. In addition to the PopCash cases, we found another 23 websites that inject SE-ads, which were not labeled correctly. For the remaining 15 websites, seven were adult websites that injected ad scripts to redirect visitors to live cam websites; the other eight rendered regular ads, which TRIDENT classified incorrectly. Excluding the mislabeled cases, we only have 15 false positives, all interactions with regular ads. These cases convert to a 1.5% false positive rate. Moreover, they are ads, and it does not harm the users when TRIDENT blocks them.

False Negatives. After investigating the false negatives, we found 17 cases out of the total 20 websites were caused by incorrect labeling. For example, VirusTotal incorrectly flagged out websites navigated from *www.reg.ru*. Excluding those mislabeled cases, we found three false negatives: one script on a streaming website, one on an adult website, and one on a cryptocurrency scam website. The streaming (*ligastream2.blogspot.com*) and cryptocurrency scam (*lookscrypto.de*) websites had click event listeners registered to hijack user's clicks as of writing. However, the ad scripts employed the Pop Under technique, which opens a new window for the same website the user is visiting and navigates the original tab to a cross-origin website. Because opening a same-origin window is likely benign, TRIDENT did not detect the script that opened the new tab. However, the navigation made from the original tab was classified as malicious. The adult website *hentaibedta.net* embedded malicious links in its first-party content. Specifically, it included ad images that pointed to an external website (*ouo.io/QqJgfz*). This external website eventually landed the user on a malicious browser extension downloading page and two reward scam pages. The SE-ads on the adult site were injected by the first-party script and behaved as if they were the first-party content. Although TRIDENT failed to detect the script, we argue that the script was a first-party script and the website operator injected those SE-ads on purpose. Furthermore, it is extremely infrequent: only one website out of 1000 websites did this. By tweaking the features' values for this false negative, we found that if the script was from a third-party, TRIDENT could detect it. To conclude, TRIDENT can capture most malicious SE-ad activities except when the website, as the owner, injects SE-ads as the first-party content on purpose.

3.4.4 Feature Importance and Robustness

To answer **RQ2**, we assessed TRIDENT's classifier by analyzing its feature importance to confirm that the features were understandable and reflected domain experts' intuitions. Beyond explaining feature importance, we analyzed our model's robustness against concept-drift [77] and evading techniques.

Feature Importance

We select the features based on our domain knowledge, experts' intuitions, and previous studies [6, 54]. We want the features to be meaningful and understandable. To this end, we evaluated the feature group importance guided by the Leave-One-Group-Out approach proposed in [78]. We reported the results in Figure 3.7 using ROC curves. The *property* feature group has the lowest AUC score, whereas the *action* feature group has the highest score. This result is understandable that the properties of a script do not indicate its maliciousness, and what a script does reflects its objective most. Although it is 0.03% lower than the best score by using all the three feature groups, based on the discussion in subsubsection 3.4.3, the *property* feature group is helpful when a data point is near the decision boundary.

Also, the scores of training with and without consequence features only have 0.67% difference. Therefore we can use one single model at the two spots mentioned in subsection 3.3.6.

Robustness

In this section, we evaluated how well TRIDENT performs against concept-drift [77] by testing the model using the testing dataset. Next, we tested the robustness of TRIDENT's classifier by altering feature values to simulate evading TRIDENT.

Concept Drift. Machine learning models are known to lose their effectiveness over time due to the underlying changes in the data distribution used to train the model. We build TRIDENT to slow down the degradation process by focusing on the behaviors of the scripts that inject SE-ads. To this end, we evaluated TRIDENT's accuracy over time by testing it on a dataset crawled in March 2022. The results showed that TRIDENT achieves an accuracy of 90.66% with a precision of 88.18% and a recall of 91.75%. The accuracy, precision, and recall dropped by 1.97%, 2.45%, and 4.53%, respectively. From the testing dataset, we observed that 5.33% benign websites tended to navigate users with more than one redirect in March 2022, while it was 2.90% two months ago. This behavior shift made TRIDENT produce a higher false-positive rate (7.67%). The degradation could be reduced by periodically retraining the model with new data.

Evasion Simulation. We have discussed one sample that evaded TRIDENT in subsubsection 3.4.3. Given the limitation of gathering more evading samples, we simulated the evasion by altering the feature values. We generated four guidelines based on our domain experts' intuitions for feasible evading techniques: (1) include the malicious script as the first-party script; (2) put the script as inline script; (3) directly bring the user to SE-websites without redirects; and (4) behave as benign scripts while steal clicks. Based on these four guidelines, we reported the evasion rates in Table 3.6 by techniques.

First, we changed the *property* feature groups to make the scripts first-party and/or inline. This alternation yields a maximum of 5.11% evasion rate. Next, we let the attacks directly bring the users to the SE-websites. This change alone leads to a 3.62% evasion rate.

Table 3.6: Evasion Rates by Altering Key Feature Values.

Approaches	**Evasion Rate**
First-party script (Fst.Pty.)	2.13%
Inline script (Inl.)	5.11%
No redirects (NoRdr.)	3.62%
NoRdr. + Fst.Pty.	2.56%
NoRdr. + Inl. + Fst.Pty.	9.17%
Do not request external resources	1.49%
Do not add callbacks	1.49%
Do not attach iframes	1.92%
Do not modify node attributes	1.70%

When combining the techniques used for the *property* features, the evasion rate went up to 9.17%. Finally, we tested altering the *action* features, which is the most challenging part since we need to keep the attacks valid. We took a conservative approach in that we kept the features related to DOM manipulations, including event listener registrations, DOM node modifications, etc. We only updated the remaining features in this feature group and reported the result in the lower part in Table 3.6. We did not report the combination of these behaviors since the evasion rate did not increase significantly. The highest evasion rate was 1.92% by not attaching iframes on the page.

In summary, we found that the attackers can evade TRIDENT with a high rate only if they can include their malicious scripts as first-party by colluding with the website owner or compromising the web servers. However, this is unlikely because the attackers can have better choices of compromising visitors when they can access the web servers.

3.4.5 Comparison to State-of-The-Art Systems

To answer **RQ3**, we compared TRIDENT with two state-of-the-art tools: Brave Shields, the adblocking module for Brave Browser [57] from industry, and ADGRAPH [15] from academia. We first show Brave Shields is insufficient using filter-list based approach and then show ADGRAPH was not suitable for SE-ads.

Traditional Blacklist-Based Ad-Blockers

Adblock Plus is the most popular blacklist-based ad-blocker. It leverages manually maintained blacklists to deny or whitelists to allow ad or tracker traffic. Brave Browser has integrated a variety of filter lists, which are a superset of Adblock Plus's, so we setup its ad-blocking component [79] locally to see how well TRIDENT performs against traditional ad-blockers. Brave Shields takes in a script URL and a frame URL and returns a binary decision. We feed Brave Shields our the script URLs along with their corresponding running frame's URLs and analyze the disagreements between Brave Shields and the ground truth. As described in dataset cleaning section, we obtained 1,479 positive samples for the training dataset, of which Brave Shields missed 14.74%. To make a fair comparison, we tested Brave Shields on our two batches of datasets. First, we performed a 70/30 training/testing split of our training dataset, following the data balancing method we used previously, and trained a model to test the testing split. The second dataset was the testing dataset we collected in March 2022. To evaluate how well TRIDENT performs against Brave Shields, considering our labeled datasets as ground truth, we only need to focus on false negative rate, the rate of evading the detection. Table 3.7a reports that TRIDENT outperforms Brave Shields almost by 7 times.

Machine Learning Based Ad-Blockers

We focus on two related prior works on ad-blocking: ADGRAPH, the first ML-based ad-blocking tool that is based on the contents of ads and trackers [15] and WEBGRAPH, the first ML-based ad-blocking tool that is based on the action of ads and trackers[18]. In the following, we discuss why ADGRAPH and WEBGRAPH cannot solve the problem TRIDENT is trying to solve.

First, we replicated ADGRAPH by crawling *Alexa Top 10k* using the open-sourced ADGRAPH binary, labeled its data using the latest filter lists as of writing, and built the same classifier as described in the paper. The model's accuracy was around 93% for detecting generic ads, which was 5% lower than the original model. This is not surprising that ADGRAPH suffers concept-drift over time [77]. We then created the testing dataset by letting ADGRAPH crawl random P.W. 1k websites from our website seed list. The accuracy dropped to 83.25%, and the precision for detecting the malicious class dropped to 70%. This shows that ADGRAPH for generic ads does not work well for SE-ads.

Next, we sampled 1,000 websites from the training dataset and 1,000 websites from the testing dataset, respectively. We refer the two datasets as *P.W. 1k Trn.* and *P.W. 1k Tst.* for simplicity. For each batch of P.W. 1k, 500 were from websites that were known to publish SE-ads and 500 were from websites that were benign. Then, we let ADGRAPH crawl these 2,000 websites and labeled the datasets using our ground truth. Finally, we trained ADGRAPH and TRIDENT on the same training dataset and tested them on the same testing dataset. As shown in the lower part in Table 3.7b, TRIDENT outperforms ADGRAPH by over 10%. ADGRAPH trained by *P.W. 1k Trn.* performs even worse than the generic model. However, this is not an apple-to-apple comparison. The ADGRAPH for Generic and ADGRAPH for SE-ads are two different models as they are trained on different datasets which are labeled differently. The former targets generic ads while the latter targets SE-ads. Moreover, while replicating ADGRAPH, we found URLs with protocol `data:` will be considered as NON-AD in the labeling process of ADGRAPH. This implies resources using base64 encoded URL would likely escape ADGRAPH's detection because ADGRAPH can extract nothing from such URLs. This gives the adversaries opportunities to import external scripts using `"data:text/javascript,ZG9Tb21ldGhpbmcoKQ=="` which means `doSomething()` to evade ADGRAPH.

WEBGRAPH improves the robustness of ADGRAPH by removing the content features and adding network, storage, and shared information flows. Because WEBGRAPH is not open-sourced as of writing, we are not able to evaluate it with our datasets. However, we argue WEBGRAPH is not designed to capture how a script manipulates the DOM to

Table 3.7: TRIDENT Compared With State-of-the-art Solutions.

Dataset	**FNR by Brave Shields**	**FNR by TRIDENT**
First batch	15.14%	2.13%
Second batch	12%	1.49%

(a) False Negative Rates of detecting SE-ads by Brave Shields and TRIDENT. The first batch is 30% split from the training dataset. The second batch is from the testing dataset.

Model	**Accuracy**	**Precision**	**Recall**
ADGRAPH for Generic Ads	83.25%	80.12%	81.65%
ADGRAPH for SE-ads	81.51%	71.34%	75.33%
TRIDENT	95.07%	96.11%	95.49%

(b) TRIDENT outperforms both ADGRAPH models for detecting SE-ads.

lure user to social engineering websites. Hence, its performance on our datasets should be equivalent with ADGRAPH's.

3.4.6 Runtime Overhead

To answer **RQ4**, we evaluated the runtime performance of `SEAgent`, the major component that may induce overhead, including running time and memory and CPU usage.

Running Time Overhead. To quantify the impact on the user experience, we measured the page load time to evaluate the running time overhead for the Tranco top 1k websites [80]. To measure the page load time and the induced overhead, we leveraged Chromium's `TRACE_EVENT` instrumentation infrastructure for profiling [81]. We added a new trace category named `blink.seagent` and put `TRACE_EVENT0` marco to the beginning of each instrumentation hook. Then, we enable `blink.user_timing` to measure the page load time, which is defined as the time spent between the navigation request start and the load event end [82]. For each website, we loaded the page into the browser for 10 times and selected the median page-load overhead.

The distributions of the runtime overhead are shown in Figure 3.8a. The median runtime overhead is 2.13% which results in 0.02 second increase in the page load time, which are comparable to previous works [20, 19, 16]. Looking at outliers, we found the websites have more DOM modifications were more impacted by the `SEAgent`. For instance, *www.kickstarter.com* took the longest to load with 14.34% (0.33 seconds) overhead. After checking this website, we found that JavaScript inserted more than 35,000 DOM nodes and modified the attributes of them, and then removed half of them before the page was fully loaded. These outliers are rare given that the overhead for the 95% of the Tranco 1k list is less than 5.7%.

Resource Overhead. To evaluate TRIDENT's resource usage overhead, we measured the CPU and memory usage for the websites listed in the Tranco top 1k [80]. It is challenging to separately measure the precise resource consumption of TRIDENT's components, because this would require sophisticated code instrumentation to calculate how much memory is allocated and how many CPU cycles are consumed. Therefore, we leverage an alternative approach that allows us to estimate the resource usage overhead. We use the `ps` [83] command to continuously record the CPU and memory usage of the browser processes (with 100ms granularity) while visiting the home page of every website in the Tranco top 1k list ten times (i.e., 10k page loads in total), using both vanilla Chromium and TRIDENT. Every time a page is visited, we wait for the page to be fully loaded, and then wait another 10 seconds before visiting the next page.

To compare the resource usage of vanilla Chromium and TRIDENT, we summarize the results as Cumulative Distribution Function (CDF) graphs in Figure 3.8b. As seen from Figure 3.8b, TRIDENT induces negligible CPU overhead and limited memory usage overhead, which is mainly driven by TRIDENT's need to perform data serialization and buffer browser data objects that are then recorded to the TRIDENT's trace files.

Summary. With 2.13% overhead on page load and negligible CPU and memory overhead on modern devices, we believe TRIDENT is capable of being deployed onto a real production environment in a real-time setting.

3.5 Discussions

3.5.1 Limitations

This section discusses the limitations of TRIDENT in two ways. One is the runtime environment of TRIDENT which may allow adversaries to learn the existence of TRIDENT and refuse to display malicious content. The other is that the diversity of the training dataset may be limited when we crawled the Internet.

Runtime Environment. We envision TRIDENT being deployed as a browser extension with Chrome DevTools Protocol turned on as a prototype, which exposes TRIDENT's existence. Adversaries may detect TRIDENT and then cloak themselves or refuse to display content until the users turn off TRIDENT. To address this limitation, we could embed TRIDENT directly into the browser to make it invisible to those adversaries. We leave this for future work.

Ground Truth. Unlike previous works [15, 17, 18, 67, 84] which target at generic ads and trackers, TRIDENT targets at SE-ads, which are not as ubiquitous as those ads and trackers. Thereby, we rely on *publicwww.com* to collect websites that inject SE-ads. To this end, the diversity of the training dataset is limited to a small number of ad networks we have identified by reverse-engineering their ad scripts. While TRIDENT performs well based on this dataset collected from *publicwww.com*, its accuracy may drop when it encounters unseen ad network scripts. However, TRIDENT can periodically retrain its classifier on improved ground truth as the users provide feedback.

3.5.2 Ethical Considerations

In line with previous studies [4, 14, 20] that need to crawl the Internet, our crawling experiments simulated user's clicks on ad publishers, which may lead to advertisers' landing pages. Because our primary goal is to analyze the behavior of interacting with SE-ads, we argue it would not be possible without clicking on the websites to trigger

SE-ads and navigate to SE-websites. Moreover, our crawlers do not target any specific ads or ad campaigns. They randomly choose ten clickable elements and ten links. These clicks resulted in 5,726 opened windows that load benign contents. Assuming that all of these windows eventually reached the landing pages of advertisers, we found our crawler made two clicks on the ads for each advertiser on average. Considering the average CPC (cost per click) being USD $0.75 [62], the cost to each advertiser would be USD $1.5 on average. This result shows that our crawling experiment ensured minimal financial losses for legitimate advertisers while generating results that help prevent people from falling into web-based social engineering attacks.

```
<!-- ad slot on nytimes.com -->
<div
  id="dfp-ad-top" class="place-ad placed-ad" data-position="top"
  data-size-key="top" data-google-query-id="CNrG4..."
><iframe src="https://.google..."/></div>
```

```
// an inline script to
// configure ad size for the div
var adConfig = function() {...};
// a remote script:
// doubleclick/pubads_impl_*.js
var adFrame = createAdIframe(adConfig);
appendAdFrame('#dfp-ad-top', adFrame);
...
```

(a) Ad scripts from Google Ads.

```
var popunder = {
  init: function(event) {
    ...
    return setTimeout(function() {
      // open a new window to load ads
      windowOpenerNull(),
      removeTransparentLayer()
    }, 500),
    sendClickMetrics()
    ...
  },
  createTransparentLayer: function(
  ){...},
  removeTrasparentLayer: function(
  ){...},
  ...
}

// register click event listener
document.addEventListener(
  isChrome ? 'mousedown': 'click',
  handler(event) {
    removeTransparentLayer();
    ...
    popunder.init(event);
})
```

(b) In-lined ad scripts from adSterra.

Figure 3.3: Script Snippets Comparison Between Google Ads and AdSterra.

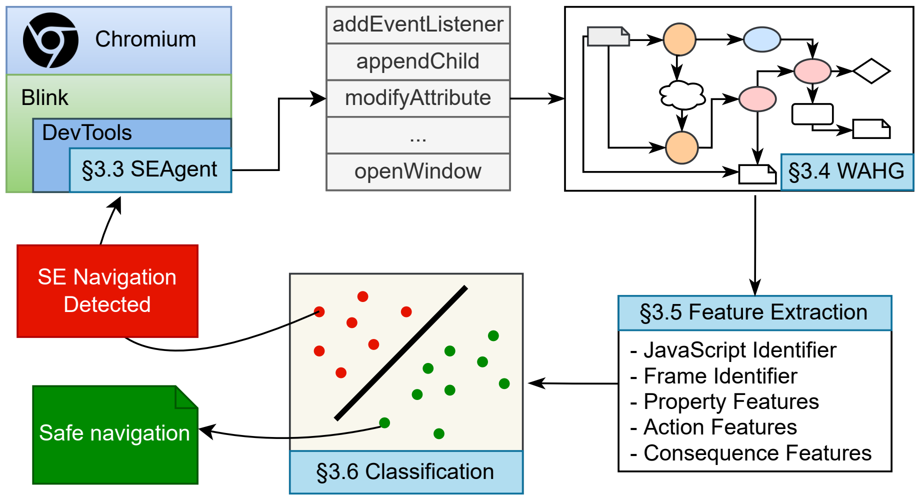


Figure 3.4: TRIDENT Design Overview.

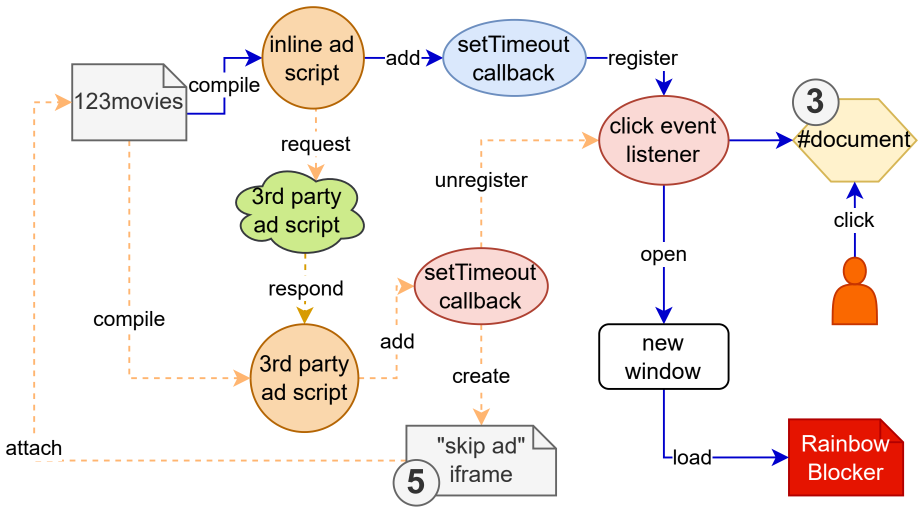
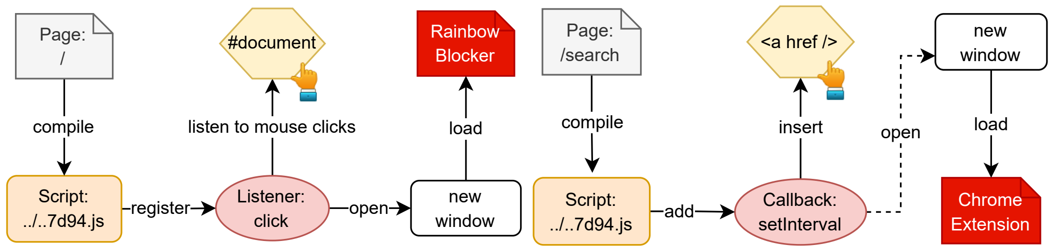


Figure 3.5: WAHG Example of The Motivating Example.



(a) Navigation Initiator as a JavaScript Function (b) Navigation Initiator as an Anchor tag

Figure 3.6: TRIDENT Attributes The Responsible JavaScript Function That Initiates The Navigation. The functions in the pink elliptical are accountable for SE-ads.

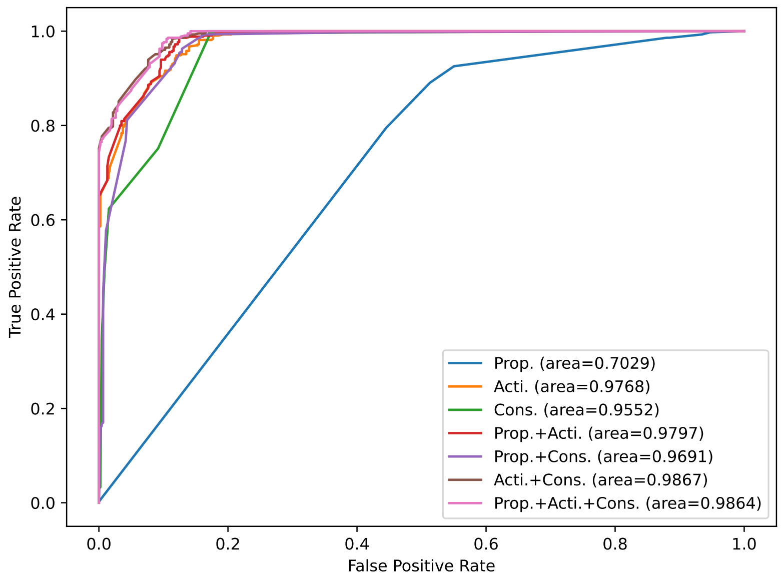
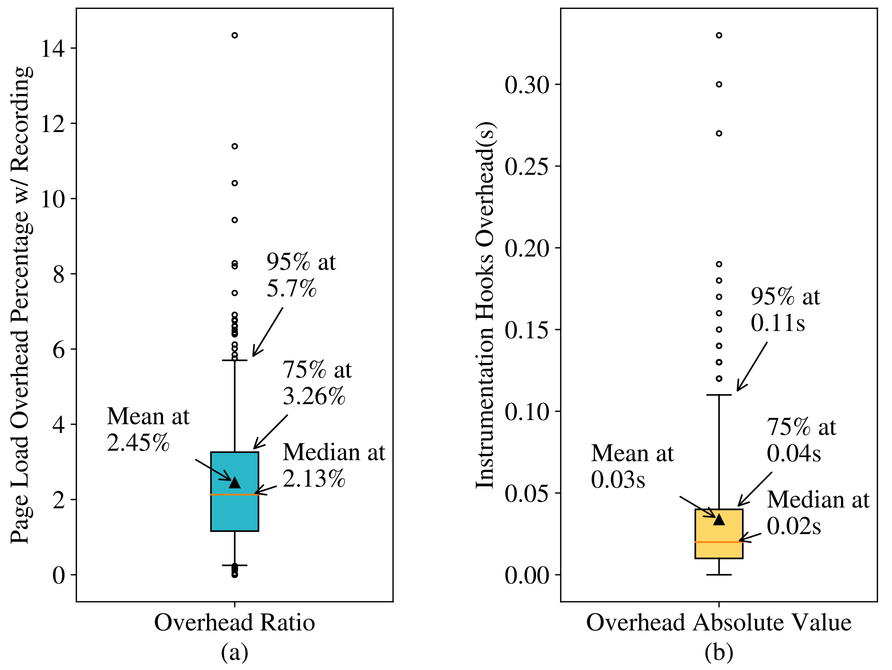
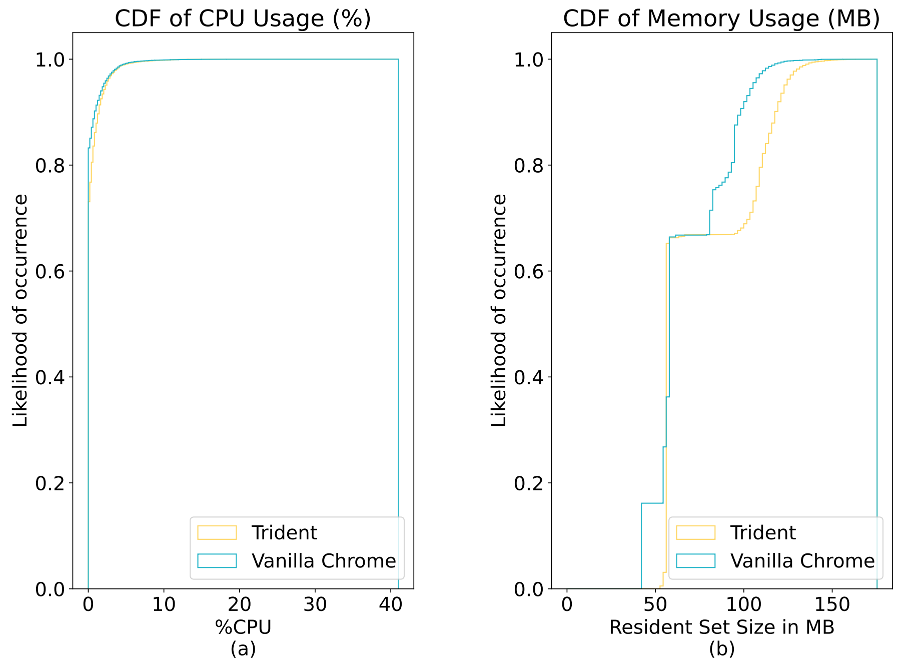


Figure 3.7: Feature Importance of Different Combinations of Feature Groups.



(a) The runtime overhead induced on the page load by TRIDENT for the Tranco 1k. (a) presents the runtime overhead increase for the page load. (b) provides the absolute time induced by TRIDENT.



(b) The runtime resource usage induced on the page load by TRIDENT for the Tranco 1k.

Figure 3.8: Runtime Performance.

CHAPTER 4
COINDEF: A COMPREHENSIVE CODE INJECTION DEFENSE FOR THE ELECTRON FRAMEWORK

4.1 Introduction

The increasing popularity of the *Electron* framework for developing cross-platform applications highlights the enduring allure of software paradigms that leverage familiar web technologies [85]. However, this novel use of web technologies outside of the sandboxed browser setting also means vulnerabilities from web applications (e.g., *cross-site scripting (XSS)*, *prototype pollution*, etc.) can affect the underlying client machine, potentially resulting in the remote execution of malicious code (RCE).

One such attack with real-world consequences is the Water Labuu campaign discovered in October 2022 [86], which spreads malicious messages through Meiqia, an Electron-based chatting application used by over 400,000 companies for customer service. A simple click on the malicious message injects malicious code into the app and leads to the exploitation of CVE-2021-21220 [87], stealing more than 300,000 US dollars worth of cryptocurrency. Unfortunately, Meiqia is not the only one that is vulnerable. Similar campaigns can be launched against other high-profile apps like *Slack*, *Discord*, *MSTeams*, potentially targeting any subsequently discovered RCE [88, 89, 90, 91, 92, 93, 94] to execute malicious code of the attackers' choice on million's of machines and devices.

Despite extensive research into defenses against XSS for web applications [25, 26, 27, 32, 28] and RCE for native environments like NodeJS [29, 30, 40], little has been done to address the root cause of such RCE attacks–*code injection*–within Electron applications. These prior solutions are effective within their respective runtime environments (web or native), but they are not equipped to handle the dual environments nature of Electron applications, which fuse web and native environments. This fusion introduces unique vulnerabilities and a broader attack surface [24], rendering existing defenses ineffective for protecting both environments simultaneously. Although recent studies have examined the attack surfaces in Electron applications and proposed mitigation [21, 22, 23], none have addressed the underlying code injection issues driving these security risks.

In our pursuit of a comprehensive and practical defense, we investigated the underlying cause of code injection attacks in Electron applications. Our research reveals two insights: 1) A code injection attack will ultimately change the semantics of the original code, reflected by a structurally different abstract syntax tree (AST), *resulting in a modified or new AST structure*. 2) The JavaScript engine of Electron is the *choke point* to provide comprehensive protection against code injection *for the web and native environments simultaneously*. Consequently, we turned our attention to preserving AST structural integrity in the JavaScript interpreter so that we can prevent code injection by enforcing the AST structural integrity with contextual information.

To this end, we propose COINDEF, a comprehensive Code Injection Defense for the Electron Framework. COINDEF works in the JavaScript interpreter (i.e., V8) of Electron applications to build AST profiles and enforce them at runtime. COINDEF works in two phases. In the learning phase, COINDEF identifies all JavaScript code contained in the application to be protected and generates the expected AST profiles of all the code with the execution context using both static profiling and dynamic profiling. In the enforcing phase, COINDEF validates every JavaScript code that is being interpreted by comparing the observed AST against those extracted in the learning phase. Furthermore, COINDEF leverages predefined security policies (*security-first* or *usability-first*) at runtime to accommodate unseen AST profiles that are not profiled in the learning phase. As working in the choke point in the execution pipeline for all code, COINDEF enables comprehensive mitigation against maliciously injected code. COINDEF also incurs negligible runtime overhead by taking a free ride (precomputing AST profiles) when the parser is parsing the

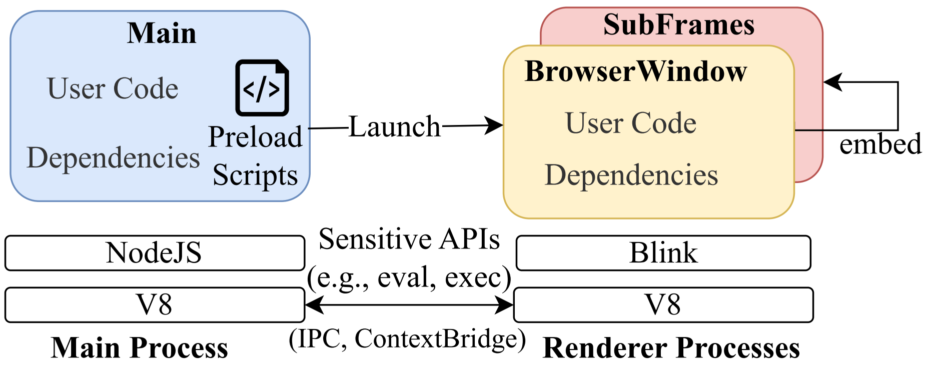


Figure 4.1: The Process Models of Electron.

source code. As such, code only needs to be validated once in its lifetime, incurring no runtime overhead during user interactions.

We evaluated CoInDef on 20 representative real-world Electron applications with code injection and RCE vulnerabilities, listed in Table 4.2. The applications we tested include widely used ones like *Slack* and *MSTeams*, as well as popular open-source projects such as *boostnote* and *joplin*. We first collected the AST profiles as the expected behavior model for each application using the approach discussed in subsection 4.3.4. After completing the learning phase, we proceeded to carry out 20 exploits using payloads previously reported as successful. The evaluation results show that CoInDef effectively blocked all the exploits. Furthermore, CoInDef only incurred, on average, a 3.96% *one time* overhead at startup time (subsection 4.4.2) and negligible (Table 4.5) over the remaining lifetime of the Electron application. Additionally, we conducted a comparison with state-of-the-art solutions for Electron applications (subsection 4.4.3) to highlight the comprehensive protection CoInDef provides and discussed where prior solutions failed.

4.2 Background & Challenges

Process Models. Figure 4.1 presents the architectural overview of the Electron framework. Conceptually, the Electron framework consists of two processes. The main process assumes responsibility for native API access with the support of NodeJS. The renderer process focuses on rendering the UI and managing user interactions backed up by Blink, Chromium's renderer engine. Upon initializing the main process, Electron's main process creates a `BrowserWindow` object. This window object launches an HTML page as the UI that runs in the renderer process. Notably, the renderer process relies on Blink and thus inherits the security mechanisms of Chromium, including site isolation, which prevents direct communication between different web frames [95]. To establish communication between the main and the renderer processes, Electron introduces preload scripts, which are bounded to the renderer process and provide `BrowserWindow` objects with access to sensitive APIs (e.g., `shell, ipcRenderer`) defined in those preload scripts through Inter-process Communication (IPC). Additionally, Electron offers developers the `NodeIntegration` option to directly integrate NodeJS's context into the renderer processes, facilitating fast development.

SubFrames. Chatting applications like *Discord* provide "In-App View" features to allow their users to directly view external resources (e.g., watch a YouTube video) in the main application window. However, these applications do not load arbitrary external content. Instead, they have clearly defined the Content Security Policy (CSP) to only load trusted resources and render them in the isolated `iframe`, which we annotated as `SubFrames` in Figure 4.1. The JavaScript code running in the `SubFrames` is isolated from the main application to prevent it from tampering with the main application. Since these resources are only presented at runtime, they are unknown to static analysis tools and may not be there when being dynamically profiled.

The shared context between the main and the renderer processes provides UI access to the native OS resources. Meanwhile, it opens security holes for Electron applications: *code injection in the UI can springboard into the main process and achieve RCE* by either directly invoking those APIs through shared context enabled by misconfigured `contextIsolation` [89] or crafting a series of sophisticated payload to abuse the correctly exposed APIs [91]. `contextIsolation` is the critical security feature

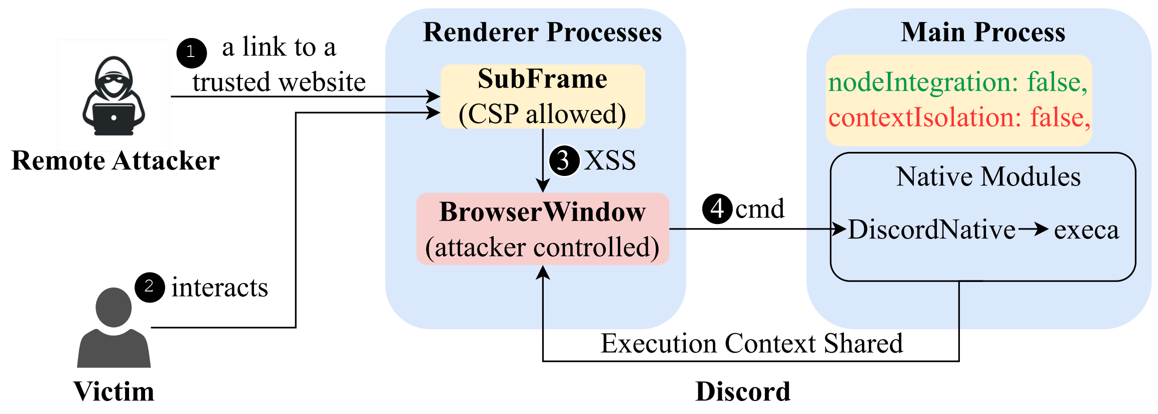


Figure 4.2: A Real-world Motivating Example.

Electron creates to ensure the JavaScript code running in the UI's process cannot arbitrarily access the powerful NodeJS context running in the main process. With context isolation enabled, Electron creates `contextBridge` to expose functions connected to native OS resources to the UI for desktop experience. Unfortunately, these security suggestions are not well followed by application developers [21, 23]. Even worse, attackers have found ways to bypass these restrictions in some scenarios [96, 97].

In the rest of the paper, we use `MainProcess` to represent the main process, `BrowserWindow` to represent the top frame in the renderer process, and `SubFrames` to represent non-top frames in the renderer process.

4.2.1 A Motivating Example

As shown in Figure 4.2, a real attack scenario [89] begins by sending a website link to a potential victim in step ❶. Although *Discord* has defined CSP to prevent code injection, it cannot guarantee all its partners follow the same security practices. As a result, there is still a possibility for trusted partners with insufficient security practices to be exploited and serve as an open gate for the entire system.

In this example, one of the trusted partner websites, *sketchfab.com*, has a cross-site scripting (XSS) vulnerability when rendering annotations for 3D models. The attacker takes advantage of this feature and sends the victim a well-crafted 3D model containing malicious JavaScript code. Then, the XSS bug gets triggered as the victim interacts with the 3D model isolated in a `SubFrame` in step ❷ using the "In-App View" feature. Specifically, the XSS payload successfully navigates the parent frame (the top window or the content of the `BrowserWindow`) to an attacker-controlled webpage in step ❸ by exploiting CVE-2020-15174 [98], which is called a `open-redirect` attack. Since this attacker-controlled webpage is now in the context of the top window, it gains access to *Discord*'s `BrowserWindow` context, escaping from the isolated `SubFrame`. Meanwhile, due to the inadequate setting (`contextIsolation: false`) of isolation between the main and the renderer processes, the UI can access the sensitive NodeJS APIs (e.g., `fs`, `child_process`, etc.) even in the renderer process. This misconfiguration gives the attacker further opportunities to compromise the main process. Consequently, the attacker overwrites two JavaScript's built-in methods (i.e., *prototype pollution* attacks) to successfully invoke a privileged NodeJS module `DiscordNative` and achieves remote code execution through `execa` in step ❹.

4.2.2 Challenges

If the victim is using the web version of *Discord*, the attack would stop in step ❸ because the user has been navigated away from *discord.com*. There would be no subsequent attacks at all. Unfortunately, in the setting of *Discord*'s Electron application, because of the shared context (APIs) between the main and the renderer processes, the initial code injection attack evolves into a more severe RCE, causing more harm to the victim. If we just focus on defending against step ❸, attackers can always find alternatives to trigger step ❹, rendering the single-point defense ineffective. Based on this observation and the requirements of users, we conclude three significant challenges to overcome to mitigate code injection attacks in Electron applications:

C2. Negligible Runtime Overhead. The protective system must operate with minimal runtime overhead to avoid any noticeable lag or slowdown that could lead users to disable the defense.

C3. Tamper-Proofing. To ensure robust security, the protective system should be tamper-resistant, meaning it must be deployed within a privileged layer that is inaccessible to remote attackers, thereby preventing any unauthorized modifications.

4.3 Design

4.3.1 Threat Model & Assumptions

We design COINDEF to prevent code injection attacks for Electron applications in a comprehensive and fast manner for Electron applications in the production environment. It defends against remote attackers attempting to inject malicious payloads into Electron applications and safeguards users who unknowingly copy and paste such payloads. These attacks exploit vulnerabilities in Electron applications and their dependencies, allowing remote attackers to control the victim's device or steal their digital assets. Therefore, applications that deliberately accept and execute arbitrary user inputs are out of scope. Importantly, in the context of Electron applications, the application, its dependencies, and vendors are not intentionally malicious but rather vulnerable. Therefore, issues related to the software supply chain attacks fall outside our scope. Moreover, COINDEF does not require the code of Electron applications to be human-readable. In other words, COINDEF takes the code released in the production build as is, which is usually minified, bundled, or obfuscated.

4.3.2 Design Overview

Figure 4.3 illustrates the high-level work flow of COINDEF. COINDEF takes in as an Electron application in the learning phase to construct its AST profiles for enforcement. In the enforcing phase, COINDEF validates the AST profiles generated at runtime against the learned ones. Unlearned AST profiles encountered at runtime are handled according to predefined security policies (detailed in subsubsection 34) based on the protection mode:

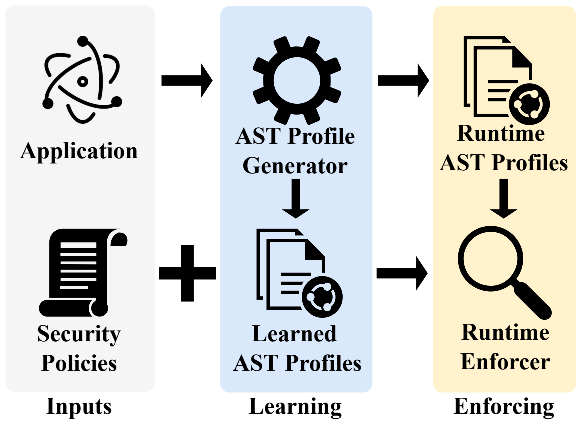


Figure 4.3: CoInDef Design Overview.

security-first or *usability-first*. To overcome the challenges outlined in subsection 4.2.2, CoInDef operates in the language interpreter, a central place where all JavaScript code (i.e., both web and native) must pass through. This placement not only ensures comprehensive protection (**C1**) but also is tamper-proof (**C3**) from remote attackers since CoInDef has higher privileges than the JavaScript code for residing in the interpreter. Furthermore, CoInDef takes advantage of the existing JavaScript code parsing process to get an almost free ride for constructing AST profiles, incurring negligible runtime overhead (**C2**).

4.3.3 AST Profile

A good AST profile should only block maliciously injected code while allowing legitimate ones. In the context of code injection, an AST profile in CoInDef is an abstract representation of the source code that is either existing legitimate static code or dynamically generated with legitimate inputs from remote sources (e.g., user inputs, network responses) given its running context. The naive code-signing method can guarantee the legitimacy of the static code but cannot accommodate the dynamically generated code that can change the signature frequently. To achieve this goal, CoInDef builds AST profiles by constructing context-aware AST structural signatures.

Table 4.1: Data Nodes for AST Structural Signature.

Node Type	**Parent Type**	**Data Type**
literal	any	string or number
name	property	string or number
key	property	string or number
value	property	object or array literal, string, or number
variable proxy	not (call or assign or new)	string

AST Structural Signature

To generate such AST structures for any code, whether hard-coded (i.e., static) or dynamically generated (i.e., dynamic), CoInDef extracts each AST node's type, value, and position when the interpreter parses the source code. To allow varying legitimate user inputs for dynamic code, CoInDef replaces the concrete values with placeholders for certain *data* nodes. Table 4.1 lists all types of data nodes in an AST including the value of a *literal*, *name*, or *key* node which can only be a string or a number; the value of a *value* node under a property parent can be an object literal, an array literal, a string, or a number. By design, none of these nodes should introduce function definitions or invocations in an AST. If controlled by an attacker, a `variable proxy` (i.e., the variable's identifier) can be pointed to a function or an object. Therefore, CoInDef only allows the value to change when the variable proxy node is not under a `call`, `new`, or `assign` expression.

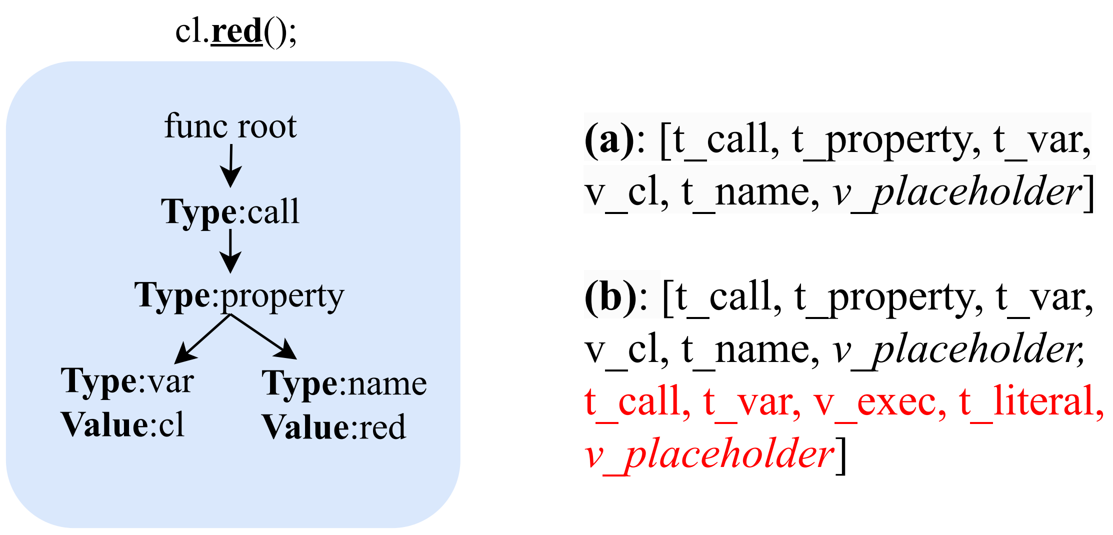
Such an AST structural signature prevents the (malicious) inputs from defining a new function, overwriting the prototype functions, calling existing functions, or executing code directly, which simply searching for function-related operations cannot achieve. For instance, given JavaScript code that is vulnerable to code injection attacks: ``eval(`cl.${color}()`)``, the dynamic input is `color`. The expected legitimate `color` can be a literal (e.g., `red`). When `eval` executes, an AST is generated as shown in Figure 4.4a. However, attackers can change the value of `color` from `red()` to `red();exec('cmd')//`, resulting in arbitrary code execution and a different AST structure as highlighted in Figure 4.4b. By checking the code's AST structural signature in Figure 4.4b with the baseline in Figure 4.4a, COINDEF can detect and reject the malicious code, `exec('cmd')`. Since COINDEF does not consider node's values for `property.name`, any legitimate, expected inputs to `color` (e.g., `yellow`, `green`) can pass the validation.

Detecting only function invocations in the AST is insufficient because injected code can execute directly without invoking existing functions. To address this, our AST structural signature captures both function-related operations and direct code execution paths. This comprehensive approach ensures that even non-invoked, standalone code injections are detected, enhancing the robustness of our protection by accounting for all possible injection vectors within the AST structure.

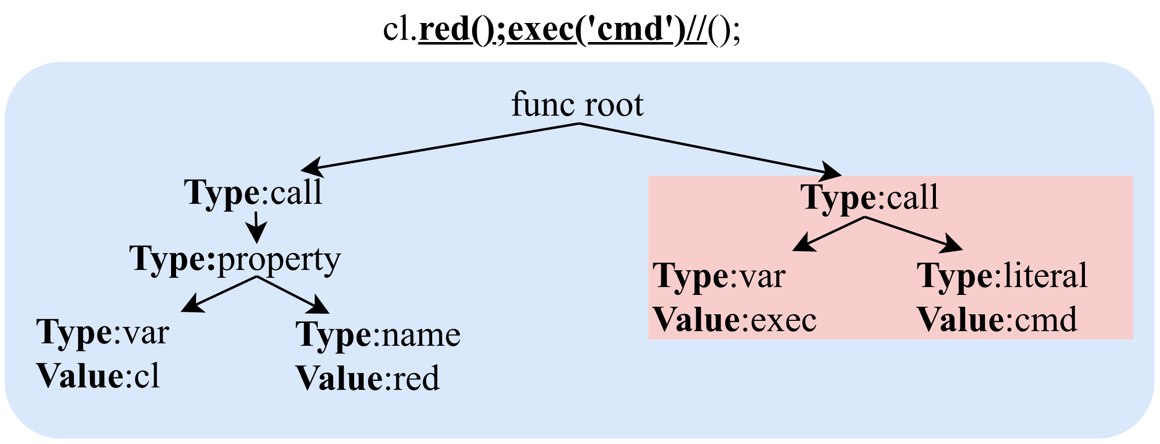
Execution Context Annotation

The execution context is essential for Electron applications to distinguish the privileges of web and native code and generate finer-grained AST profiles for mitigating mimicry attacks [100]. For example, suppose the attackers can leverage Electron's vulnerabilities (e.g., CVE-2022-29247 [101]) to access the native environment through the web layer. In that case, they can directly import NodeJS libraries as what has been defined in the native code to pass the AST structural signature validation. Based on the least-privilege principle, COINDEF isolates the AST profiles based on their running processes (i.e., `MainProcess`, `BrowserWindow`, and `SubFrames`) and annotates each AST profile with the process context and code context, including the caller information and callsite location if it is an `eval`-like call, exemplified in Figure 4.5. To facilitate fast enforcement, COINDEF also divides the AST profiles into two categories, static and dynamic, per process.

Static Profile. COINDEF considers an AST profile static when the interpreter interprets a static script file and functions defined in the script file. For instance, when Electron renders an HTML page, the HTML parser invokes V8 to compile code defined in


cl.red();
func root
Type:call
Type:property
Type:var
Value:cl
Type:name
Value:red
(a): [t_call, t_property, t_var, v_cl, t_name, v_placeholder]
(b): [t_call, t_property, t_var, v_cl, t_name, v_placeholder, t_call, t_var, v_exec, t_literal, v_placeholder]

(a) Legitimate Code.



(b) Malicious Code.

Figure 4.4: AST Structural Signatures of Legitimate and Malicious Code. An example taken from `eval(cl.`${color}()`)` [99] shows that code injection alters the AST structural signature.

`<script src=''A.js''>`. Currently, the requester is the HTML parser; therefore, static profiles have no execution context other than the running process. Due to the lazy compilation policy[1], not all JavaScript code in a script file is interpreted immediately. That is, some functions, if not invoked immediately, are only parsed and compiled when other code invokes them. Specifically, while parsing the source code of a script, the parser will skip some functions that are not immediately invoked and remember their names and scopes. When those skipped functions are invoked, the parser will parse the function body

[1]https://v8.dev/blog/preparser

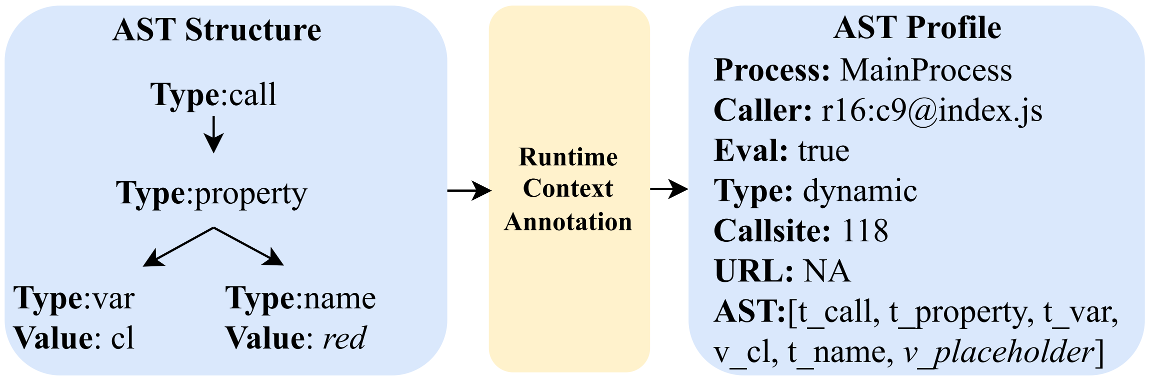


Figure 4.5: An AST Profile Example.

to generate bytecode for them. At this time, the requester is a JavaScript function. However, these functions are defined in the static script file and are not changeable. Therefore, COINDEF still considers AST profiles of such lazily compiled functions static after checking their original script with local files.

Dynamic Profile. COINDEF considers an AST profile dynamic when the code comes from dynamic code generation APIs (e.g., `EventHandler, eval, document.write`), obtained from the code injection sinks defined in CodeQL [102]. Beyond the code injection sinks, COINDEF also considers the scenario where one script includes or imports another. For example, when script A includes script B by appending another `<script>` tag or runs `eval`-like or `importScript` APIs to execute code dynamically, COINDEF considers the AST profile of Script B or the dynamically evaluated code as a dynamic profile. Note that dynamically generated code also complies with the lazy compilation policy. COINDEF uses the same method to assign types to functions defined in dynamically generated code.

Figure 4.5 shows that an `eval`-like API is invoked at the 118th character of a caller function defined at row 16, column 9 in the file "index.js." Since this code is dynamically generated, the profile type is "dynamic" and URL is marked as "N/A." Because `property.name` is a data node, the value is marked as a placeholder.

4.3.4 AST Profile Collection

Modeling an Electron application can be done either statically or dynamically. However, the accuracy of static analysis decreases significantly on bundled, minified, or obfuscated JavaScript code, which is often the code format of Electron applications for production release. Moreover, static analysis cannot provide an accurate execution context in the enforcing stage. For example, without running the code, there is no way to be sure what process it will be running in. Therefore, we opted for a hybrid learning approach to build AST profiles of an application for COINDEF. Static profiling guarantees the completeness of the application, while dynamic profiling complements it by annotating the contextual information and recording any dynamic code execution at runtime.

COINDEF collects AST profiles at the function level to preserve AST structural integrity within the execution context. This choice aligns with the JavaScript interpreter's compilation process, where functions serve as the fundamental units for execution. In line with this design, COINDEF enforces policies by letting the interpreter return a `noop` function when an AST profile is determined invalid; otherwise, it returns the original function object. With this function-level granularity, COINDEF learns static profiles P_S through static profiling (S) and dynamic profiles P_D through dynamic profiling (D) for an Electron application (A), which together is denoted as $P_A = P_S \cup P_D$.

Static Profiling. COINDEF customizes D8[2], a shell interface to V8, to exclusively invoke V8's parser and generate AST structural signatures. This customization disables the lazy compilation described in subsubsection 4.3.3, forcing the parser to build the AST structural signature for every JavaScript function defined in the local files. This enables static profiling to provide COINDEF with a comprehensive model of the application. Specifically, COINDEF obtains P_S as:

$$P_S = \{p_{i,j} = G(i,j) | \forall F_i \in A, \forall f_j \in F_i\}$$

[2]https://v8.dev/docs/d8

where P_S is the set of all AST structural signatures collected from every JavaScript function in every file of A and G is the AST structural signature generation procedure. F_i is a JavaScript file contained in A's installation package, and f_j is a JavaScript function defined in F_i. i is the URL (e.g., `file://path`) of a JavaScript file and j is the location (i.e., row and column) of a JavaScript function. During dynamic profiling, $p_{i,j}$ is annotated with runtime context based on i. It is important to note that P_S is a complete set of all JavaScript functions defined in A.

Dynamic Profiling. CoInDef comprehensively exercises each application to supplement the static profiles (P_S) using a semi-automated approach that simulates user interactions, similar to other state-of-the-art techniques [21, 103, 104]. This approach begins with a crawler that systematically interacts with the application's UI, clicking on buttons and menus and typing text. To improve the performance of this approach by covering more complex features, we supplement it with manual exercising based on the application user manuals If available, we also employ end-to-end test cases to simulate user interactions and trigger features, ensuring we cover all relevant functionalities. Given a dynamic code trigger action T in such a dynamic profiling procedure, CoInDef obtains:

$$P_D = \{p_{i,j} = G(i,j) | C_j \in F_i, F_i \in A, T(i,j) \text{ is triggered}\}$$

where G is the AST structural signature generation procedure and C_j denotes dynamic code execution APIs within the JavaScript file F_i, which are discovered during static profiling.

4.3.5 Runtime Enforcement

In the enforcing phase, CoInDef validates AST profiles generated at runtime ($R = R_D + R_S$, where R_D and R_S are dynamic and static AST profiles generated at runtime) against those learned in the learning phase ($P = P_D + P_S$). Any unlearned profiles, defined as

$U = R - P$, are handled according to predefined security policies. Specifically,

$$U = (R_D + R_S) - (P_D + P_S) = (R_D - P_D) + (R_S - P_S)$$

Since P_S is a complete set of JavaScript functions in A, $R_S - P_S = \emptyset$, concluding $U = R_D - P_D$. This unlearned set (positives) U includes true positives caused by attacks, denoted as T, and false positives caused by unlearned features, denoted as F. The goal of the enforcement is to block T and accommodate F with *best-effort* under the premise of *security-first* following security policies.

As outlined in algorithm 1, the enforcer takes in as the current runtime AST profile r and the learned AST profiles P. If r is a static profile, CoInDef simply checks it against P_S by its URL and callsite (line 2-line 6). When r is a dynamic profile, CoInDef tries to find a match in P_D based on its URL, caller, and callsite. If CoInDef can find the learned AST profiles for the given execution context, it lets r proceed as long as $R.ast$ is validated (line 7-line 11). Otherwise, CoInDef applies the predefined security policies on r to determine its security impact and proceed accordingly (line 12-line 33).

Security Policies for Unlearned AST Profiles

We define the security policies for CoInDef based on Electron's existing security model and best practices. These policies focus on assessing the data flow for imported dynamic code, the execution context in which this code is intended to run, and the security origins of remote resources. Additionally, CoInDef employs two working modes: *security-first* and *usability-first* to balance the security and usability.

First, CoInDef performs a data flow analysis on r along its call stack trace to determine whether its source code consists only of locally defined variables. For instance, a network response is considered a remote variable, as it is not defined within the local scope, whereas a hard-coded string is defined locally. Based on this analysis, CoInDef assigns r a scope designation of `local` or `remote` (line 12). If r is confirmed to have a `local` scope, meaning the source code is concatenated with hard-coded strings intended by developers, COINDEF permits r to proceed regardless of the running process (line 15-line 17).

Next, COINDEF checks the security origin of r, assigning it one of three values: `same`, `cross`, or `empty` (line 13), where `empty` indicates that the dynamic code is generated through `eval` or `Function`. Simultaneously, COINDEF loads allowed security origins as defined in the application, obtained through the hybrid profiling process (line 14). COINDEF then examines the process in which r is intended to run to determine its privilege level. If r is in the `MainProcess` process, COINDEF rejects it based on a zero-trust security policy (line 18-line 20). If r is in the `BrowserWindow` process and its security origin differs from that of `BrowserWindow`, COINDEF also rejects it. If the security origin matches, COINDEF consults a user-defined property (i.e., `SameOriginInBrowser`) to decide whether to permit same-origin content in `BrowserWindow`, defaulting to `false` (line 21-line 26). For `SubFrames`, COINDEF denies r if the user has disabled `SubFrames`, which is disabled by default; otherwise, COINDEF allows r only if its security origin matches one of the allowed security origins (line 27-line 33).

4.3.6 Security Analysis

Now, we formalize the security analysis as follows. **Definitions.**

$\mathcal{T}_{model}$: an AST representation of the program execution learned in the profiling phase.

$\mathcal{T}_{runtime}$: an AST resulting from runtime input (e.g., user input, dynamic code) in the enforcing phase.

$\mathcal{V}$: the set of $\mathcal{T}_{model}$ learned from the whole program, and $\mathcal{T}_{model} \in \mathcal{V}$.

$\mathcal{C}$: the execution context of a $\mathcal{T}_{model}$ at a given program point, and $\mathcal{T}_{model} \in \mathcal{V}_{\mathcal{C}}, \mathcal{V}_{\mathcal{C}} \subseteq \mathcal{V}$.

$\mathsf{DataNode} \subset \mathcal{T}$: the set of leaf nodes representing data values (e.g., string literal) expected to vary at runtime.

$\mathsf{ExecNode} \subset \mathcal{T}$: the set of internal nodes representing executable logic (e.g., call expression).

$\mathsf{IntermediateNode} \subset \mathcal{T}$: the set of internal nodes representing expressions and statements (e.g., block statement).

$\mathsf{Children}(n)$: each node $n \in \mathsf{IntermediateNode}$ has a sequence of children forming a subtree, denoted by $\mathsf{Children}(n) = [n_1, n_2, \ldots, n_k]$.

AST Structural Signature Enforcement. CoInDef enforces the AST structural signature by the function:

$$\mathsf{Match}(\mathcal{T}_{model}, \mathcal{T}_{runtime}) = \bigwedge_{i=1}^{k} \mathsf{match}(m_i, r_i)$$

where CoInDef requires that $\forall m_i \in \mathcal{T}_{model}, r_i \in \mathcal{T}_{runtime}$, $\mathsf{match}(m_i, r_i) = \texttt{true}$. We denote the type of a node $n \in \mathcal{T}$ as:

$$\mathsf{type}(n) = \begin{cases} \mathsf{data}\ if\ n \in \mathsf{DataNodes} \\ \mathsf{exec}\ if\ n \in \mathsf{ExecNodes} \\ \mathsf{intermediate}\ otherwise \end{cases}$$

$\forall m_i \in \mathcal{T}_{model}, r_i \in \mathcal{T}_{runtime}$, the value of $\mathsf{match}(m_i, r_i)$ function is set to the following conditions:

1. Type mismatch is disallowed:

$$\mathsf{type}(m_i) \neq \mathsf{type}(r_i) \Rightarrow \mathsf{match} = \texttt{false}$$

2. Data nodes must match in type:

$$\begin{cases} \mathsf{type}(m_i) = \mathsf{data} \\ \mathsf{type}(r_i) = \mathsf{data} \end{cases} \Rightarrow \mathsf{match} = \texttt{true}$$

3. Exec nodes must match exactly:

$$\begin{cases} \mathsf{type}(m_i) = \mathsf{exec} \\ \mathsf{type}(r_i) = \mathsf{exec} \end{cases} \Rightarrow \mathsf{match} = (m_i == r_i)$$

4. Intermediate nodes must match recursively on their children:

$$\begin{cases} \mathsf{type}(m_i) = \mathsf{intermediate} \\ \mathsf{children}(m_i) = [m_{i_1}, m_{i_2}, \ldots, m_{i_k}] \\ \mathsf{type}(r_i) = \mathsf{intermediate} \\ \mathsf{children}(r_i) = [r_{i_1}, r_{i_2}, \ldots, r_{i_k}] \end{cases}$$

$$\Rightarrow \mathsf{match}(m_i, r_i) = \bigwedge_{j=1}^{k} \mathsf{match}(m_{i_j}, r_{i_j})$$

Execution Context Enforcement. CoInDef enforces that any runtime AST must conform to the allowed structure for its execution context:

$$\forall\, \mathcal{T}_{runtime}, \exists\, \mathcal{T}_{model} \in \mathcal{V}_{\mathcal{C}}$$

$$\text{such that } \mathsf{Match}(\mathcal{T}_{model}, \mathcal{T}_{runtime}) = \texttt{true} \text{ at Context } \mathcal{C}.$$

Now, consider a code injection gadget on the DOM, `element.appendChild(user_inputs)`. Let $\mathcal{U}$ be the `user_inputs`, If CoInDef observes only non-script DOM nodes from $\mathcal{U}$ during the learning phase, CoInDef obtains $\mathcal{V}_{\mathcal{C}} = \emptyset$, where $\mathcal{C}$ is the execution context of this API call. Then, CoInDef will block any injected scripts since $\neg(\exists\, \mathcal{U}_{model} \in \mathcal{V}_{\mathcal{C}})$. If $\mathcal{U}$ contains script elements in the learning phase, then $\mathcal{U}_{model} \in \mathcal{V}_{\mathcal{C}}$. Enforcement then permits only structurally consistent ASTs with variability restricted to data leaf nodes:

$$\exists\, \mathcal{U}_{model} \in \mathcal{V}_{\mathcal{C}} \quad \wedge \quad \mathsf{Match}(\mathcal{U}_{model}, \mathcal{U}_{runtime})$$

Thus, the attacker's freedom is limited to modifying literal values (e.g., strings, constants) without the ability to define, modify, or invoke arbitrary logic. Any injected payload that deviates from the learned script structure will be blocked since:

$$\mathsf{Match}(\mathcal{U}_{model}, \mathcal{U}_{runtime}) = \texttt{false} \text{ at Context } \mathcal{C}.$$

For cases where input validation cannot be definitively resolved–due to incomplete learning, ambiguous context, or mixed data types–CoInDef defers to a predefined set of security policies (**??**). These policies combine static and dynamic analyses, such as local data flow inspection and runtime execution context profiling, to maintain security guarantees while supporting usability.

4.3.7 Implementation

As illustrated in Figure 4.6, CoInDef integrates instrumentation hooks into V8's interpreter to gather AST node information, modeling both static and dynamic code behaviors.

When code is processed, the language parser tokenizes it and constructs two objects: `script_info` and `parser_info`. These objects contain metadata about the script, including the AST, source location, text range, and the origin of the code (static or dynamic). Before bytecode generation begins, CoInDef collects contextual information by examining the JavaScript stack frames provided by V8, selecting the top frame as the

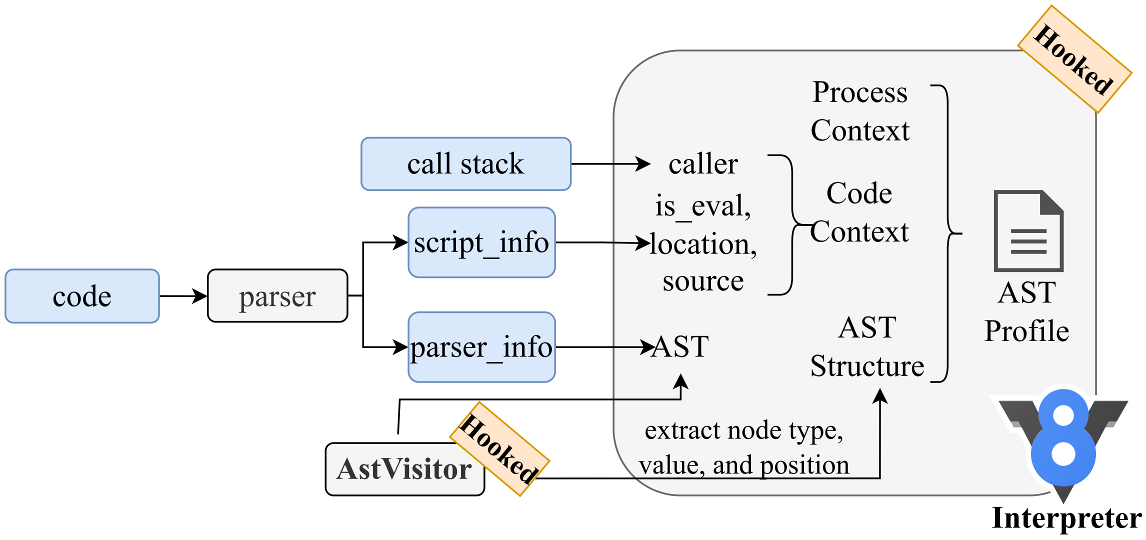


Figure 4.6: AST Profile Hooks Implementation.

caller. It extracts the dynamic code's location if sourced from `eval`-like APIs to establish the code context. During bytecode generation, CoInDef hooks into the base `AstVisitor` functions to extract node type, value, and position within the AST. Note that CoInDef does not profile internal JavaScript functions from NodeJS and Electron, as these are loaded before the user program and are immutable. To annotate processes, CoInDef utilizes hooks in Electron's web frame delegates. V8's `Isolate` represents isolated instances of the V8 engine, with the main process running NodeJS's JavaScript context and browser windows running standard JavaScript with additional Electron APIs. By hooking into Electron, CoInDef identifies the process creating the `Isolate` and annotates the process context accordingly.

The modifications for AST profile generation and enforcement are minimal, comprising fewer than 150 lines of code in V8 and an additional 500+ lines in two self-contained C++ files for logging and validation.

4.4 Evaluation

Algorithm 1: AST Profile Enforcement.

```
Data: Current Runtime AST Profile $r$, Learned AST Profile $P$, Security Policies $S$
Result: Whether to generate the function object
1  begin
       // r is exemplified in Figure 4.5.
2      if $r.type$ == static then
3          $P_S \leftarrow$ a nested hash map keyed by url, callsite for $r.process$;
4          $p \leftarrow P_S.find(r.url, r.callsite)$;
5          return $p$.validate($r.ast$);
6      end
7      $P_D \leftarrow$ a nested hash map keyed by url, caller, callsite for $r.process$ ;
8      $ps \leftarrow P_D.find(r.url, r.caller, r.callsite)$;
9      if $ps \neq \emptyset$ and $ps$.validate($r.ast$) then
10         return true ;
11     end
       // security policy for other cases.
       // local or remote dataflow scope of r.
12     scope $\leftarrow$ localDataFlow($r$) ;
       // empty-, cross- or same-origin
13     origin $\leftarrow$ originCheck($r.url$) ;
       // obtained from hybrid profiling.
14     $allowedOrigins \leftarrow S.allowedOrigins$ ;
15     if scope == local then
16         return true;
17     end
18     if $r.process$ == MainProcess then
19         return false;
20     end
21     if $r.process$ == BrowserWindow then
22         if origin != same then
23             return false;
24         end
           // S.SameOriginInBrowser is false by default.
25         return $S.SameOriginInBrowser$;
26     end
27     if $r.process$ == SubFrames then
           // S.SubFrame is false by default.
28         if not $S.SubFrame$ then
29             return false;
30         end
31         $fOrigin \leftarrow$ frameOrigin($r.caller$);
32         return $fOrigin \in allowedOrigins$;
33     end
34 end
```

Table 4.2: A Diverse Set of Applications Vulnerable to Code Injection and RCE Attacks.

#	Application	Vulnerable Version	Electron Version	Line of Code	GitHub Stars Reference	Injection Description	Attack Vectors
1	MSTeams	v1.4.00.4855	v8.5.5	187,295	N/A [91]	a message to achieve template injection attacks	TI, S-XSS, RCE
2	Slack	v4.3.2	v7.1.9	153 (minified)	N/A [88]	an embedded frame opening a malicious webpage	OR, RCE
3	Discord	v0.0.14	v11.4.2	10,932	N/A [89]	a embedded frame opening a malicious webpage	OR, PP, RCE
4	VSCode	v1.63.1	v11.2.1	3,114 (minified)	147k [105]	a local file exploiting markdown preview	MP, RCE
5	GraSSHopper	v1.1.7	v12.0.6	71 (minified)	N/A [22]	a text rendered as HTML in a popup window	D-XSS, RCE
6	ARDM	v1.4.9	v11.4.9	11,271	25.8k [22]	a text rendered as HTML	S-XSS, RCE
7	Joplin	v2.9.12	v19.0.10	172,199	36.2k [106]	the language indicator for the markdown code format	MP, RCE
8	Boostnote	v0.22.0	v12.0.14	103,203	20.6k [22]	code rendered as HTML for the markdown code format	MP, RCE
9	Altair-graphql	v4.0.11	v14.0.1	1,804	4.7k [22]	query description rendered as HTML	S-XSS, RCE
10	Appium-desktop	v1.22.0	v7.1.2	133 (minified)	4.5k [107]	incoming http request reflected as HTML	R-XSS, RCE
11	Simplenote	v2.9.0	v9.1.0	20,615	4.4k [94]	markdown file not being properly sanitized	MP, RCE
12	BlockBench	v3.9.3	v13.1.2	49,280	2.1k [22]	a filename rendered as HTML	D-XSS, RCE
13	electron-crud	v2.8.0	v10.0.0	1,168	1.5k [22]	database records rendered as HTML	S-XSS, RCE
14	arc-electron	v16.0.1	v13.1.1	9,971	1.3k [93]	HTTP header rendered as HTML	S-XSS, RCE
15	vmd	v1.34.0	v3.0.9	1,976	1.2k [108]	markdown file not being properly sanitized	MP, RCE
16	antares-sql	v0.5.6	v14.0.1	256,525	1.1k [22]	database table names rendered as HTML	S-XSS, RCE
17	Markdownify	v1.4.1	v7.2.4	10,337	868 [109]	markdown file not being properly sanitized	MP, RCE
18	Poddycast	v0.8.0	v11.2.1	2,395	160 [110]	bookmark rendered as HTML	S-XSS, RCE
19	OhHai Browser	v3.4	v8.2.5	2,736	52 [111]	bookmark rendered as HTML	S-XSS, RCE
20	Jukeboks	v2.2.2	v11.2.3	1,360	23 [22]	filename rendered as HTML	D-XSS, RCE

Attack Vectors – TI: Template Injection, MP: Markdown Preview, OR: Open-Redirect.
Attack Vectors – PP: Prototype Pollution, R-XSS: Reflected XSS, D-XSS: DOM-Based XSS, S-XSS: Stored XSS.
N/A in GitHub Stars means they are not open-sourced.

To evaluate CoInDef 's effectiveness and practicability in protecting users from code injection attacks and whether it effectively addresses the challenges presented in subsection 4.2.2, we address the following research questions:

RQ1: How effectively can CoInDef protect users from code injection attacks and the subsequent RCE?

RQ2: How do FP and FN impact user experience?

RQ3: What is the runtime overhead?

RQ4: How comprehensive is CoInDef compared with the SOTA tools?

To answer these research questions, we evaluate CoInDef on 20 diverse applications that are vulnerable to code injection and RCE, using their real-world exploits. These applications are selected for their broad representation across software categories, scales, types of vulnerabilities, and use cases. We source them from the GitHub Advisory Database [112], articles, and technical blogs [91, 89, 88, 105, 92].

4.4.1 Effectiveness

To address **RQ1** and **RQ2**, we evaluate CoInDef on 20 applications with diverse code injection vulnerabilities and report on its effectiveness in blocking all RCE exploit attempts while only incurring non-intrusive false positives.

Diverse Code Injection Vulnerabilities

As described in Table 4.2, the code injection points of the vulnerabilities generally fall into three categories: messages, remote resources, and markdown files, covering the prevalent code injection attack vectors for Electron applications, including *template injection*, *markdown preview*, *open-redirect*, *prototype pollution*, and *all types of XSS*. Among these attack vectors, *open-redirect* is particularly threatening in Electron applications and is not handled in prior work [22, 21], because it opens a new website and completely takes over the `BrowserWindow`, resulting in legitimate call chain through

IPC to the `MainProcess`. For example, collaboration applications like *MSTeams*, *Slack*, and *Discord* have injection points within the messaging feature. In the case of *MSTeams*, an attacker can exploit a vulnerability in rendering the display name of a mentioned user in a group chat to launch a code injection attack on AngularJS's template engine. For *Slack*, the injection point is a malicious file uploaded to *file.slack.com* and then sent to a victim via messaging. Similarly, for *Discord*, attackers can inject malicious code through messages. These two attacks leverage malicious iframes and a vulnerability of Electron to launch an *open-redirect* attack to load attacker-controlled content in the `BrowserWindow`, which pollutes the prototype of certain built-in functions to achieve RCE. Some applications load remote resources and injection points can be found within these resources. For example, in *electron-crud*, attackers can inject JavaScript code into the records of the connected database to launch a Stored XSS. The malicious records allow attackers to execute arbitrary code. Productivity applications like *VSCode* have injection points within markdown files. These files may contain well-crafted HTML payloads that are not properly sanitized. When users interact with these markdown files, the injected code executes, leading to RCE.

Experiment Setup

The primary objective of evaluating COINDEF is to assess its effectiveness in blocking code injection attempts while allowing legitimate user inputs. To ensure realistic testing, we use the vulnerable versions of each application listed in Table 4.2, ensuring that all code injection attacks constitute novel (i.e., 0-day) attacks for each application and COINDEF is agnostic to these attack vectors.

Note that all the attack payloads have been confirmed to work and are modified without causing harm to the end host but triggering the logs that indicate it is an attack. Specifically, for applications exploited by malicious messages, we used an MITM proxy to inject malicious payload into the messages. For example, to exploit *MSTeams*, we first logged in *MSTeams* with CoInDef enabled as a potential victim. Then, we used another account to send a message to the potential victim. The message was hijacked and injected with a malicious payload. Then, we observed whether the malicious payload was executed to log the compromise indicator for each step in the attack chain. We injected code into the remote content for applications that exploit them. For example, to exploit *electron-crud*, we created a MySQL database and inserted a malicious record. Then, we use *electron-crud* to read the malicious record to compromise the application. For applications exploited by markdown previews, we crafted malicious markdown files and opened the files with those applications.

Learning Phase

Using the methods outlined in subsection 4.3.4, we collect AST profiles for all 20 applications. Table 4.3 presents the size of these AST profiles alongside the human effort required for their collection, measured in hours. The AST profiles are organized by process (i.e., `MainProcess`, `BrowserWindow`, and `SubFrames`) and type (i.e., `static` or `dynamic`). Notably, only three collaborative applications (i.e., *MSTeams*, *Slack*, and *Discord*) and *VSCode* exhibit dynamic code in `SubFrames`. This is because collaborative applications include "In-App View" features, while *VSCode* executes extensions in isolated environments. Our results show that 18 out of the 20 applications generate and run code dynamically, with more complex applications producing and executing greater amounts of dynamic code. The applications also run more dynamic code within the `BrowserWindow` process compared to `MainProcess`, which is expected since users interact primarily through the UI components, triggering dynamic code generation in `BrowserWindow`. Notably, achieving a converged state of learned AST profiles required approximately 4 hours for complex applications and about 30 minutes for simpler ones, as shown in the last column of Table 4.3. In the converged state, the number of AST profiles no longer increases, indicating that our exercising strategies have fully

Table 4.3: AST Profiles Collected in the Learning Phase.

Application	Main Process		Browser Window		Sub-Frames		Human Hours
	St.	Dyn.	St.	Dyn.	St.	Dyn.	
MSTeams	3,034	67	16,591	127	0	32	4
Slack	4,115	37	15,682	198	0	56	4
Discord	1,491	3	24,619	649	0	82	4
VSCode	2,808	135	21,806	310	0	9	2 (0.1*)
GraSSHopper	831	42	5,798	189	0	0	1
ARDM	1,311	75	4,486	3	0	0	1
Joplin	873	73	5,734	53	0	0	2 (0.1*)
Boostnote	710	24	9,226	1	0	0	1
Altair-graphql	1,439	1	10,515	13	0	0	1
Appium-desktop	2,243	11	7,315	53	0	0	1
SimpleNote	1,337	2	4,234	13	0	0	1 (0.1*)
BlockBench	3,378	24	24,773	67	0	0	1
electron-crud	1,372	0	11,893	1	0	0	1
arc-electron	1,033	0	11,449	11	0	0	1
vmd	773	0	8,453	34	0	0	0.5
antares-sql	1,379	2	5,551	125	0	0	1
Markdownify	732	0	1,142	1	0	0	1
Poddycast	1,091	0	1,864	1	0	0	0.5
OhHai Browser	963	0	2,435	0	0	0	0.5
Jukeboks	1,268	0	912	0	0	0	0.5

$*$ time cost with automated end-to-end testing. St.: Static. Dyn.: Dynamic.

covered the necessary code paths.

Enforcing Phase

When learning is complete, we enter the enforcing mode for each application and use them as our daily drivers for a month, during which we attack each application. We evaluate CoInDef based on the two security modes defined in CoInDef. The default mode, *security-first*, disables `SubFrames` from loading cross-origin resources (i.e., `S.SubFrame = false`) and rejects unknown same-origin content in `BrowserWindow` (i.e., `S.SameOriginInBrowser = false`), as outlined in algorithm 1. The alternative mode, *usability-first*, permits both cross-origin resource loading in `SubFrames` and same-origin content in `BrowserWindow` to prioritize

Table 4.4: Effectiveness of CoInDef under Enforcement.

Application	Code Cov.*	# of Attacks	Security-first		Usability-first		Overhead % (ms)
			FP	FN	FP	FN	
MSTeams	77.68%	3	0	0	0	0	6.07% (39)
Slack	84.71%	5	2	0	0	1	9.31% (19)
Discord	78.36%	6	10	0	0	1	6.79% (701)
VSCode	86.92%	4	5	0	0	0	1.83% (4)
GraSSHopper	88.12%	3	0	0	0	0	4.18% (12)
ARDM	82.57%	2	0	0	0	0	4.11% (16)
Joplin	79.28%	6	0	0	0	0	5.70% (63)
Boostnote	95.51%	3	0	0	0	0	2.71% (13)
Altair-graphql	75.58%	5	0	0	0	0	4.64% (18)
Appium-desktop	79.27%	3	0	0	0	0	4.49% (80)
SimpleNote	92.04%	3	0	0	0	0	2.97% (13)
BlockBench	78.61%	2	0	0	0	0	5.67% (48)
electron-crud	79.31%	7	0	0	0	0	2.51% (9)
arc-electron	93.27%	5	0	0	0	0	1.72% (11)
vmd	91.58%	3	0	0	0	0	3.55% (13)
antares-sql	85.49%	8	0	0	0	0	3.87% (9)
Markdownify	92.31%	4	0	0	0	0	2.90% (12)
Poddycast	100%	2	0	0	0	0	1.04% (2)
OhHai Browser	100%	2	0	0	0	0	2.35% (7)
Jukeboks	100%	3	0	0	0	0	2.71% (11)
Mean	87.03%	3.95	0.85	0	0	0.10	3.96%

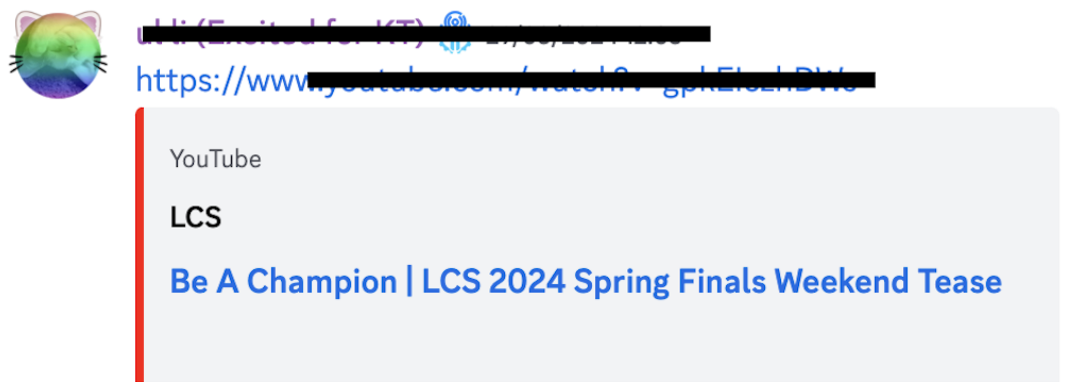
*: code coverage during enforcement testing.

flexibility and usability.

Code Coverage During Enforcement Testing. To fairly evaluate the defensive mechanism for CoInDef, we adopt function-level code coverage on first-party code, as defined by V8 [113]. This choice is motivated by the well-documented bloat of JavaScript dependencies, where developers may import a single function from a large module, resulting in low overall code coverage if third-party code is included. Moreover, we focus on function-level rather than block-level code coverage, as CoInDef enforces the integrity of the AST profile for JavaScript functions, not their individual blocks. Once a function passes validation, it is allowed to execute in full. We use Acorn [114], a popular JavaScript parser, to statically count the number of functions in the first-party code as our ground truth. During the testing phase, we log the number of functions invoked to calculate the test coverage. As shown in Table 4.4, the applications were exercised thoroughly. For simple and small applications (three out of 20), achieving 100% code coverage was straightforward. For medium size applications (five out of 20), we achieved over 90% code coverage. For more complex applications (12 out of 20), such as *Slack* and *VSCode*, we achieved over 75% code coverage, indicating robust test coverage across varying application complexities.

Protection Result. As illustrated in Table 4.4, we executed a total of 79 attacks across 20 applications, with each application facing between two and eight targeted attacks. Each individual attack focused on a specific code injection point. However, in some cases, reaching RCE required a series of attacks in a kill chain, as illustrated in subsection 4.2.1. For instance, six attacks launched against *Discord* formed two separate attack sessions, each involving three interdependent attacks (i.e., *open-redirect*, *prototype pollution*, and *RCE*) that combined to achieve RCE. In the *security-first* mode, COINDEF successfully blocked all code injection attempts for all 20 applications with few false positives. In *usability-first* mode, COINDEF allowed two code injections to bypass protection in two applications but still prevented the final RCE with zero false positives. This result suggests that for applications primarily used locally without loading remote content (e.g., *GraSSHopper*, *SimpleNote*, *Markdownify*), COINDEF reliably prevents code injection attacks under both modes. For applications (e.g., *Slack*, *Discord*) with more interactive features, such as "In-App View" functionality, COINDEF may miss one attack targeting `SubFrames` but it nonetheless delivers robust protection by securing critical processes in the `BrowserWindow` and `MainProcess` contexts, blocking any code injections into these processes and subsequent RCE attacks (**RQ1**).

False Positives. In our evaluation, we observed 17 false positives in the security-first mode across 3 of the 20 applications tested, while the usability-first mode produced zero false positives. These 17 cases fall into two categories: 1) unlearned features provided by remote


[redacted]
https://www[redacted]

YouTube

LCS

Be A Champion | LCS 2024 Spring Finals Weekend Tease

Figure 4.7: The False Positive Example Observed on Discord. Although the "In-App View" video player is disabled, the user can still click the link to open the video using a browser.

content and 2) unlearned updates. Specifically, 12 false positives resulted from "In-App View" features in *Slack* and *Discord*, such as video playback or file previews within isolated `SubFrames`. In security-first mode, scripts running within `SubFrames` are disabled, blocking this content, as illustrated in Figure 4.7. Although this leads to an empty video window, users can still click the link to open it in an external browser. The remaining five false positives occurred in *VSCode* due to extension updates that introduced new script files. We suggest that users can temporarily enable learning mode for these extensions to gather the necessary AST profiles for these updates.

False Negatives. In the usability-first mode, we observed two false negatives, while the security-first mode produced zero false negatives. Both false negatives stemmed from exploits targeting "In-App View" features. Specifically, these attacks injected code into trusted `SubFrames`, attempting to escalate privileges by controlling a `BrowserWindow`. Although the injected code in `SubFrames` succeeded in accessing a `BrowserWindow` object, CoInDef blocked further code execution in the `BrowserWindow` process due to its strict security enforcement upon the `BrowserWindow` process. Therefore, CoInDef prevented the RCE even though the attack succeeded in the initial stage in `SubFrames`.

The FP and FN analysis suggest that the impact on the user experience is minimal

(**RQ2**). If the users want to enjoy the "In-App View" features, they can opt-in for the *usability-first* mode, in which the security in the main application is still guaranteed.

4.4.2 Runtime Overhead

To answer **RQ3**, we conducted two runtime performance evaluation experiments to assess the runtime overhead introduced by COINDEF to applications. Both experiments were performed on Ubuntu 22.04, equipped with an Intel CPU E5-2680v3 operating at 2.50GHz and 64 GB DDR4 memory running at 2133 MHz.

Page Load. In the first experiment, we measured the overhead caused by COINDEF during the page loading process, which is in line with prior work [15, 20, 39, 19, 16, 21]. We initiated a timer when the `navigationStart` event was triggered and concluded the measurement when the `loadEventEnd` event was fired on the initial web page. This time difference represented the page load time. As some applications load remote websites as their first page (e.g., *Slack* and *Discord*), we ran the application once before testing to warm up the network cache, thus reducing the impact of network latency. For more complex applications like *VSCode*, which encompass multiple pages/frames, we only measured the overhead for the main frame (e.g., the editor window for VSCode).

To calculate the overhead for page load, we executed the application ten times with and without COINDEF enabled, selecting the median time cost. As indicated in the last column of Table 4.4, COINDEF introduced an average overhead of only 3.96%, with the highest overhead of 9.31% observed for *Slack* and the lowest of 1.04% for *Poddycast*. Although the overhead for *Slack* appears relatively high, the additional time contributed by COINDEF is merely 19 milliseconds. Notably, COINDEF exhibits higher overhead for applications that require network connections, such as *MSTeams* and *Discord*. For instance, *Discord* performs update checks before loading the first page, and our testing environment's first page comprises rich content with 57 script files sourced from *discord.com*. These 57 script files triggered validation for over 20,000 AST profiles.

Table 4.5: Runtime Overhead During User Interaction.

Time (ms)	**Runs**	**Min**	**Median**	**Mean**	**Max**
Baseline	10	111.9	116.9	116.63	119.9
CoInDef	10	113.9	116.9	117.02	120.1
Overhead		1.78%	0.00%	0.33%	0.17%

Interaction. Consistent with previous work [22], we utilized the Speedometer 2.0 benchmark suite [115] to measure the overhead for user interactions. The Speedometer 2.0 benchmark suite comprises 17 uniquely implemented *TodoMVC* applications. The benchmark simulates user actions of adding, completing, and removing items from a to-do list. In this process, the benchmark sequentially executes the 17 *TodoMVC* applications. For each application, the test begins by adding 100 items one at a time, with each item containing some content. Subsequently, the test iteratively marks each item as complete. Finally, the benchmark concludes by removing all the items individually. We repeated this process ten times, with and without CoInDef. As shown in Table 4.5, the time cost for running all 17 *TodoMVC* applications ranges from 111.9 milliseconds to 119.9 milliseconds for the baseline version of Electron, while it is between 113.9 milliseconds and 120.1 milliseconds. The interactive overhead is as small as 0 and as large as 1.78%. By design, CoInDef is not supposed to introduce runtime overhead for user interaction because CoInDef does not hook in function calls. Therefore, we applied all the data samples to measure their confidence intervals and came to the conclusion that these two sample sets come from the same distribution, which indicates that CoInDef does not introduce any runtime overhead during user interactions. Note that in scenarios where the application loads remote scripts, there will be minimal overhead as we measured for app startup (shown in Table 4.4). However, compared with the network latency during user interaction, such milliseconds overhead is negligible.

Storage. The storage overhead grows linearly (i.e., O(n) where n is the number of functions defined in the app). The largest one (Discord) is around 2 MB on disk and around 10 MB

Table 4.6: COINDEF Compared With the State-of-The-Art.

Attack Vectors	# of Attacks	SYNODE [29]		DOMTYPING [22]		XGUARD [21]		COINDEF*	
		FP	FN	FP	FN	FP	FN	FP	FN
Discord									
OR	1	0	1	0	1	0	1	1	0
PP	1	0	1	0	1	0	1	0	0
RCE	1	0	1	0	1	0	0	0	0
jukebox									
D-XSS	1	0	1	0	0	0	1	0	0
RCE	1	0	1	0	0	0	0	0	0
AllinOne†									
D-XSS$_D$	2	0	2	1	0	0	2	0	0
D-XSS$_E$	2	1	0	0	2	0	2	0	0
S-XSS$_D$	2	0	2	1	0	0	2	0	0
S-XSS$_E$	2	1	0	0	2	0	2	0	0
OR	2	0	2	0	2	0	2	1	0
PP	1	0	1	0	1	0	1	0	0
RCE	10	0	6	0	6	2	0	0	0
Total CI	14	2	10	2	9	0	14	2	0
Total RCE	12	0	8	0	7	2	0	0	0

CI-Code Injection, OR-Open Redirect, PP-Prototype Pollution, D-: DOM-based, S-:Stored, XSS$_D$: XSS via DOM Manipulation, XSS$_E$: XSS via dynamic code execution.
†: 14 code injection attacks leading to 12 RCE. PP does not lead to RCE directly.
$*$: in the *security-first* mode

in memory, indicating this overhead is negligible.

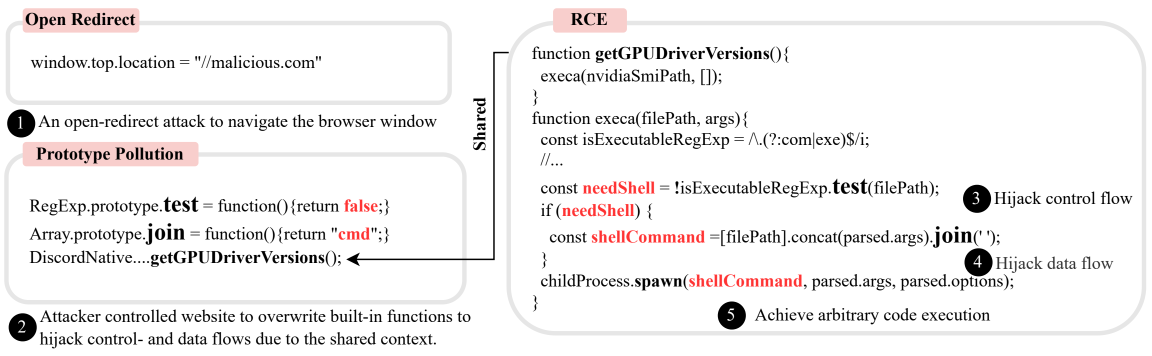
4.4.3 Comparison To State-of-The-Art

To answer **RQ4**, we compare COINDEF with the most relevant state-of-the-art solutions that either directly protect Electron applications or can be extended to do so. Based on our research, we identified three SOTA solutions: SYNODE, XGUARD, and DOMTYPING. While SYNODE was originally designed to protect NPM modules only, it can be extended to address RCE threats in Electron applications. XGUARD and DOMTYPING are specifically designed for Electron applications.

Experiment Setup. For a representative comparison, we selected three applications: *Discord*, a complex application discussed in subsection 4.2.1; *Jukeboks*, a simpler application; and a custom-built application (i.e., *AllinOne*) that incorporates a range of attack vectors, including XSS, OR, PP, and RCE. Specifically, we derived code injection attacks based on the CVEs identified in our evaluation dataset and the XSS cheat sheet from OWASP [116] into *AllinOne*. These attacks include DOM-based XSS via DOM manipulation (D-XSS$_D$) and dynamic code execution using `eval` (D-XSS$_E$), Stored XSS through DOM manipulation (S-XSS$_D$) and dynamic code execution using `eval` (S-XSS$_E$), open redirect (OR) from a `SubFrame`, prototype pollution (PP), and the final RCE with each code injection attack leads to except for PP. In total, the *AllinOne* contains 11 code injection points leading to 10 RCE attacks. Notably, we set COINDEF to the default *security-first* mode for this comparison.

Result. We present the comparison result in Table 4.6. SYNODE failed to block 10 code injection attacks, eight of which ultimately resulted in RCE. This is because SYNODE cannot cover code injection through DOM modifications including *open-redirect*. Moreover, SYNODE produces two false positives for dynamic code execution through `eval` due to the inaccuracy of static analysis on implicit data flow for string concatenation. DOMTYPING failed to prevent nine code injection attacks, leading to seven cases of RCE. this is because DOMTYPING does not handle *open-redirect* and dynamic code execution. Furthermore, DOMTYPING reported two false positives for legitimate DOM modifications. Although XGUARD successfully prevented all RCE attacks, it failed to block all 14 code injection attempts, meaning that attacks could progress to critical stages before being mitigated. Meanwhile, XGUARD reported two false positives due to the incomplete call chain modeled based on minified JavaScript code. In contrast, COINDEF provided complete protection, successfully preventing all code injection and RCE attacks across all scenarios. This underscores COINDEF 's comprehensive coverage and robust defensive capabilities compared to existing solutions.

Case Study. To better understand the comprehensive protection COINDEF offers, we elaborate the experiment process with a case study for *Discord* in Figure 4.8. The attacker


Open Redirect
1 An open-redirect attack to navigate the browser window
Prototype Pollution
2 Attacker controlled website to overwrite built-in functions to hijack control- and data flows due to the shared context.
Shared
RCE
3 Hijack control flow
4 Hijack data flow
5 Achieve arbitrary code execution

```
window.top.location = "//malicious.com"
```

```
RegExp.prototype.test = function(){return false;}
Array.prototype.join = function(){return "cmd";}
DiscordNative....getGPUDriverVersions();
```

```
function getGPUDriverVersions(){
  execa(nvidiaSmiPath, []);
}
function execa(filePath, args){
  const isExecutableRegExp = /\.(?:com|exe)$/i;
  //...
  const needShell = !isExecutableRegExp.test(filePath);
  if (needShell) {
    const shellCommand =[filePath].concat(parsed.args).join(' ');
  }
  childProcess.spawn(shellCommand, parsed.args, parsed.options);
}
```

Figure 4.8: The Detailed Attack Chain on Discord.

first injected JavaScript code onto the DOM of the vulnerable website. The injected code launched a *DOM-based XSS* attack to initiate the *open-redirect* attack. When compiling the code for navigating the browser window of *Discord*, CoInDef disabled any code running in `SubFrames`, and thus rejected the compilation under the *security-first* mode. Subsequently, the interpreter returned a `noop` function, and the *open-redirect* attack was prevented in step ❶. What if the user was under *usability-first* mode and the attacker reached ❷? To show that CoInDef can still provide protection and prevent the attack from evolving into a remote code execution attack, we turned on the *usability-first* mode. When compiling the malicious script inside the malicious website, CoInDef detected that the malicious JavaScript code was going to run in the context of `BrowserWindow`. Then, CoInDef pulled out the AST profiles built for `BrowserWindow` and found no entries for the malicious script. Subsequently, CoInDef rejected the compilation and the interpreter returned a `noop` function. If the attacker was smarter and could craft the malicious code to pass the validation and get executed in step ❷. To simulate this effect, we modified the code and let the attack proceed to this step. However, in order to invoke the privileged module (`DiscordNative` shared by the `MainProcess`) to launch RCE, the attacker must overwrite two methods for two built-in JavaScript objects: `test` for `RegExp` to hijack the control flow in step ❸ and `join` for `Array` to hijack the data flow in step ❺. No matter how sophisticated the attacking code became when being executed, it had to create two functions, which created two new AST profiles. These two new dynamic profiles were never found in the AST profiles for `BrowserWindow` given its running context. Thus, CoInDef blocked the *prototype pollution* and subsequently prevented the RCE.

Throughout these steps, it is clear that DomTyping and Synode failed to handle any of them because this attack chain does not require DOM modifications and the *prototype pollution* attack hijacks both control and data flow to feed an arbitrary command into `spawn`. Even if we extended Synode to cover more injection APIs (e.g., `spawn`), it failed to generate AST templates for this lib as it is designed to execute arbitrary commands. Although XGuard caught the final RCE by blocking an unauthorized call chain to `spawn`, the attackers can change strategies by stealing the cookies of *Discord* or launching social engineering attacks when redirecting the `BrowserWindow` to a phishing website.

Robustness. CoInDef demonstrates its robustness by successfully blocking all attacks, among which are 20 cited attacks and 59 variants trying to evade CoInDef. These attacks include DOM manipulation, dynamic execution, and other evasion techniques built on top of our domain expertise. To achieve this robustness, CoInDef limits attackers' ability to inject foreign code through: execution context integrity enforcement (subsubsection 4.3.3) and AST structural integrity enforcement (subsubsection 4.3.3) as formalized in subsection 4.3.6. First, CoInDef tracks execution context integrity, ensuring that even structurally valid but behaviorally malicious scripts are detected. Attackers cannot arbitrarily construct a legitimate AST for a given injection point, as the execution context restricts possible AST modifications. Then, CoInDef only allows data nodes to change to accommodate legitimate user inputs. Any injected code inevitably modifies a data node into a subtree, which CoInDef blocks. This enforcement applies across DOM-based injections, template literals, and `eval`-like APIs, preventing attackers from rewriting execution logic.

Deployment. CoInDef's executable identifies the required Electron version and downloads a corresponding instrumented Electron version, leaving the application code intact. This convenient integration offers improved code injection security without modifications or source code access. To generate AST profiles, the application developers can leverage the existing UI testing cases to provide comprehensive AST profiles. As for IT administrators and home users, they can leverage automated crawlers and manual exercises to build AST profiles that are customized for their use patterns, in line with prior work [31, 32, 22].

Portability. The modifications CoInDef has made for AST profile generation and enforcement are minimal, numbering fewer than 150 lines in the V8's code base and two self-contained files for logging and validation, accounting for 500+ lines of C++ code. These files containing CoInDef's hooks remain almost consistent across various Electron versions evaluated for CoInDef, simplifying porting and reducing maintenance and compatibility efforts.

4.5 Discussions

Code Coverage. As observed in subsection 4.4.1, CoInDef will incur false positives in the security-first mode due to unlearned AST profiles from remote resources. It is well understood that no single security system can promise full code coverage during the dynamic profiling of extensive programs, meaning some code may not be profiled. Using the hybrid profiling approach discussed in subsection 4.3.4, the AST profiles we collected support 20 applications with a testing code coverage of 87.03% on average, with only incurring a few false positives. We consider further increasing the code coverage for CoInDef as an orthogonal problem. CoInDef can benefit from the recent research in debloating and fuzzing web applications [104, 103, 117, 118] for expanded coverage.

Direct Command Injection. CoInDef does not offer direct protection against command injection (e.g., malicious data directly injected into `exec`). However, data injection through code injection attempts is prevented by CoInDef. To improve CoInDef further, we could add NodeJS process handler protection and generate shell command profiles for inputs sent to related APIs (e.g., `exec`, `spawn`). We leave this as future work.

CHAPTER 5
COINDX: CODE INJECTION DIAGNOSIS FOR JAVASCRIPT VIA ITERATIVE SYMBOLIC ANALYSIS ON SUBPROGRAMS

5.1 Introduction

JavaScript drives a vast ecosystem, from client-side user interfaces to backend services and cross-platform applications. Its dynamic nature and flexibility have propelled its widespread adoption, but these same qualities also make it a prime target for malicious exploitation. Code injection attacks, including cross-site scripting (XSS) [119, 43, 28, 120] and remote code execution (RCE) [44, 29, 121, 21, 122], have plagued JavaScript applications for decades [123, 124], exposing sensitive user data, enabling unauthorized system access, and undermining application integrity. The persistent prevalence of these attacks highlights a critical need for robust defenses to secure the JavaScript ecosystem.

Existing runtime enforcement solutions [29, 21, 22, 31, 32] aim to safeguard applications by monitoring abnormal execution patterns, enforcing code integrity, and blocking exploits in real-time. While these approaches have proven effective at mitigating attack *symptoms*, they fall short of addressing the underlying *root cause*–the vulnerabilities that enable the attacks in the first place. By targeting anomalous behaviors (e.g., suspicious function calls [21] or unexpected payloads [29]) rather than the injection points that generate them, these defenses leave JavaScript applications susceptible to novel exploitation paths and recurring vulnerabilities.

Root Cause Analysis (RCA) offers a promising pathway to address this gap by identifying the origins of code injection vulnerabilities, enabling developers to eliminate systemic weaknesses. While there has been prior research [41, 42] in RCA for binary programs, applying these methods to JavaScript is impossible due to the significant *semantic gap* between system- and web-level semantics. Specifically, attacks on JavaScript apps do not rely on binary exploitation. Even if an Intrusion Detection System (IDS) detects malicious behaviors stemming from a code injection attack through a web application, RCA for binary cannot discover the RCE or XSS being the root cause because these attacks do not exploit the language engine (the binary component), which is designed to execute any code passed to its interpreter. Therefore, we need a solution for the JavaScript ecosystem.

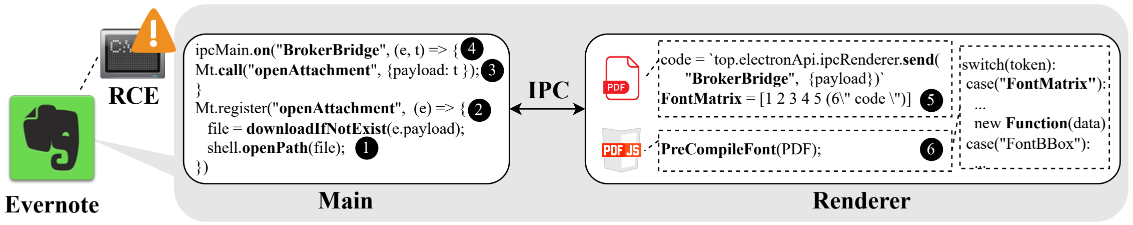
To close this semantic gap, we propose CoInDx, a novel RCA framework for code injection attacks on JavaScript applications. The primary goal of CoInDx is to accurately pinpoint the code injection point and understand how the vulnerability is exploited given an external alert signal. Unfortunately, it is rarely easy to diagnose what leads to a code injection attack for JavaScript applications. The complicated data flows due to the asynchronous nature of JavaScript make it difficult to trace the vulnerabilities to the origin (i.e., injection point). The dynamically generated code at runtime can obscure the source of vulnerabilities. Moreover, the vast and interconnected JavaScript package ecosystem makes isolating root causes difficult.

To address these challenges, we researched the monitoring mechanisms of those state-of-the-art solutions and found a key insight – What if we could compose a simpler program that preserves the exploitable program state based on the call stack trace when the monitoring system alerts? Thus, the vulnerability analysis space can be reduced significantly without concerning state explosion. The call stack trace can effectively help construct the control and data flows, circumventing the challenge of static analysis on JavaScript [21, 29]. The call stack frame also provides source code if there is dynamically generated code in the call chain, addressing the obscurity of the source of vulnerabilities.

Following this insight, we build CoInDx with four components: Stack Trace Recovery (subsection 5.3.2), Program Composer (subsection 5.3.3), Iterative Symbolic Executor (subsection 5.3.4), and Root Cause Analyzer (subsection 5.3.5. To enable effective RCA, the developers publish their JavaScript applications with COINDX's instrumentation code for tracking event registration, triggering, and handling. When a code injection monitoring system issues an alert, COINDX dumps the call stack trace, which ends with the exploiting function call (e.g., `eval`). Then, the program composer (subsection 5.3.3) consumes the call stack trace and produces a simplified program, where the closure variables and dependent functions are marked as symbols. Next, the symbolic executor (subsection 5.3.4) performs dynamic taint analysis by executing the program iteratively to produce execution traces that lead to the exploit. Finally, the execution traces are passed to the root cause analyzer (subsection 5.3.5) to identify the vulnerability type and code injection location.

To systematically measure COINDX's accuracy, we evaluated it on SecBench.js [125], a benchmark of vulnerable JavaScript modules. COINDX achieved zero false positives and zero false negatives, meaning it correctly identified and analyzed all tested vulnerabilities. The benchmark dataset includes a diverse range of attack patterns, including direct injection, context-sensitive injection, and indirect injection scenarios, demonstrating COINDX 's robustness in different exploitation models. To assess COINDX's effectiveness in diagnosing real-world exploits, we tested it on 12 exploits targeting publicly disclosed vulnerabilities. These exploits span across 11 different JavaScript applications, covering a variety of execution environments, including client-side web applications, native Node.js applications, and Electron-based desktop applications. The vulnerabilities examined include DOM-based injections, dynamic script executions, and IPC-based injections in Electron. COINDX successfully found all the root causes correctly. The runtime incurred by COINDX for tracking event registration, triggering, and handling is negligible with 1.24% and for web applications, 2.18% for Node.js applications, and 3.71% for Electron applications. The memory and storage overhead of storing these logs are less than 10 MB on average.

5.2 Background & Challenges



```
ipcMain.on("BrokerBridge", (e, t) => { 4
Mt.call("openAttachment", {payload: t }); 3
}
Mt.register("openAttachment", (e) => { 2
  file = downloadIfNotExist(e.payload);
  shell.openPath(file); 1
})
```

```
code = `top.electronApi.ipcRenderer.send(
    "BrokerBridge", {payload})`
FontMatrix = [1 2 3 4 5 (6\" code \")] 5
```

```
PreCompileFont(PDF); 6
```

```
switch(token):
 case("FontMatrix"):
   ...
   new Function(data)
 case("FontBBox"):
   ...
```

Figure 5.1: A Real-World Example of Exploiting CVE-2024-4367 [126]. The code is simplified for brevity.

5.2.1 Code Injection in JavaScript Apps

Web Applications. Web applications are prone to Cross-Site Scripting (XSS) attacks, including *Stored*, *Reflected*, and *DOM-based XSS*. Stored XSS persists on the server and executes whenever users access the affected page. Reflected XSS occurs when input is immediately reflected in the response, while DOM-based XSS exploits client-side JavaScript to modify the DOM and execute scripts. XSS can lead to session hijacking, data theft, and unauthorized actions. Vulnerabilities arise due to improper input validation, unsafe functions like innerHTML, and lack of a content security policy (CSP). Third-party scripts and misconfigurations increase the risk. These inputs usually come from user input fields (e.g., comments, search bars), query parameters, and dynamic content rendering. Client-side JavaScript manipulation and external scripts expand the attack surface.

Node.js Applications. Node.js applications are vulnerable to *Command Injection* and *Prototype Pollution*, the goal of code injection in the native environment. Command injection allows attackers to execute system commands through unsanitized input in functions like `child_process.exec()`. Prototype pollution enables malicious manipulation of object prototypes, affecting application behavior globally. Vulnerabilities occur due to unsafe command execution, dynamic JavaScript properties, and reliance on third-party libraries. Poor input validation and outdated packages increase exposure to these risks. Attack surfaces include API endpoints that execute system commands, functions interacting with the filesystem, and JSON payloads in HTTP requests. Prototype pollution often targets libraries that extend or merge objects, like *lodash*.

Electron Applications. Electron apps combine Web and Node.js environments, exposing them to *Remote Code Execution* (RCE), *XSS*, and *Insecure IPC*. RCE can occur when `nodeIntegration` is enabled (e.g., through *open-redirect* to open a new window), allowing malicious scripts to execute Node.js commands. XSS in Electron escalates into RCE, while insecure IPC communication can lead to privilege escalation. Vulnerabilities arise from misconfigurations like enabling `nodeIntegration`, failing to use `contextIsolation`, and loading untrusted sources with `loadURL().` Improper validation of IPC messages further increases the attack surface. Attack surfaces include untrusted web content, input fields in web views, and poorly implemented IPC channels. When `nodeIntegration` is enabled, injected scripts gain full access to Node.js, leading to system compromise.

5.2.2 A Real-World Example

The root cause analysis in CoInDx begins when an end-host *runtime* monitor flags a running Electron-based application for executing some abnormal code. Monitoring systems that can detect such events generally fall into two categories: system-level monitoring [127, 128] and application-level monitoring [21, 29]. However, CoInDx does not rely on how or why the application was flagged. The only prerequisite is identifying the flagged application. It is crucial to note that these monitoring systems detect attacks at the symptom level–when the attack manifests–rather than at its onset or root cause. Consequently, the information provided to CoInDx indicates only where the attack was detected, not how it began.

We illustrate the high-level concept of CoInDx using a real-world example: exploiting CVE-2024-4367 [126] on Evernote, an Electron-based application with over 200 million active users [129]. This example is particularly relevant because Electron applications combine characteristics of both web applications and Node.js applications. Successfully investigating attacks on Electron apps provides valuable insights that can be extended to both Web and Node.js applications.

CVE-2024-4367 is a code injection vulnerability in *PDF.js*, a library Mozilla maintains for rendering PDF comments in browsers [130]. This vulnerability allows attackers to inject malicious JavaScript into the font matrix defined in a PDF document.

As illustrated in Figure 5.1, the monitoring system detects and flags an RCE attack from Evernote. COINDX begins its analysis by capturing a stack trace at the point where a shell command is spawned in step ➊. However, since the shell spawning function resides within a callback function, the stack trace only includes two functions–steps ➋ $\rightarrow$ ➊. The source of the callback remains unclear at this point.

To reconstruct the execution flow, COINDX bridges event listeners and emitters, connecting ➍ $\rightarrow$ ➌ to ➋. Despite this progress, process isolation causes a break in the event chain. To overcome this challenge, COINDX monitors IPC through the context bridge shared between the main and renderer processes. This enables COINDX to connect ➏ $\rightarrow$ ➎ to ➍, ultimately reconstructing a complete call chain from ➏ to ➊.

Next, COINDX extracts each function in the reconstructed call chain and performs symbolic state reconstruction. The goal is to verify whether the call chain is vulnerable and exploitable by simulating the attack within a smaller, self-contained program. To achieve this, COINDX symbolizes two types of variables: 1) function arguments to the top caller – these inputs are likely attacker-controlled and critical for determining exploitability; 2) closure variables – these variables and external functions not defined within the call stack are symbolized as symbolic procedures to assist symbolic execution. Using these symbolized variables, COINDX constructs a simplified program and executes it in a concolic execution environment. When the `file` argument in step ➊ becomes symbolic and includes symbolic expressions (i.e., the PDF buffer) derived from step ➏, confirming it as a vulnerability.

Next, CoInDx focuses on localizing the root cause by analyzing the symbolic execution trace. This involves a backward trace analysis starting from the symbolic variable `file` in step ❶, tracing back to the point where an alternative state was forked.

In this example, the critical conditional branch occurs inside the function `PreCompileFont` in step ❻. This function extracts font information from the PDF's metadata and parses the header using a switch-case table. When processing the `FontMatrix` field, it constructs a new `Function` object to execute the payload embedded in the PDF–revealing the root cause of the attack.

As output, CoInDx provides 1) a reconstructed execution trace at the branch level, 2) the branch that leads to the initial code injection attack, and 3) a recommended fix for the vulnerability.

5.2.3 Challenges

Achieving this root cause analysis requires reconstructing the vulnerable execution trace, which poses significant challenges for JavaScript applications due to their highly dynamic nature:

C1. Lack of event tracing support. JavaScript does not provide built-in mechanisms to construct the call stack trace for event activities. As a result, connecting ❸ to ❷ within the same process and ❺ to ❹ across processes is impossible using existing infrastructure. This lack of support also extends to web applications for *client-side-redirect* attacks (i.e., a malicious script navigates the user to the attacker-controled website) and Node.js apps that leverage `node:events`.

C2. Dynamic code generation. The code executed at step ❺ is generated dynamically. During symbolic state reconstruction, it becomes unclear where and what to invoke in its caller function at step ❻, leading to an incomplete composed program. This challenge applies to all JavaScript apps, as RCA aims to find the injection point.

C3. Root cause location. JavaScript production releases often bundle multiple packages with code that is minified or obfuscated. This significantly complicates isolating and identifying the root cause of vulnerabilities.

5.2.4 Assumptions

We consider code injection attacks against JavaScript applications and assume that the underlying JavaScript engine is trustworthy. To analyze the root cause of such a code injection attack, CoInDx relies on an external code injection detector, such as Synode [29] or XGuard [21], to produce code injection signal so that CoInDx can collect the call stack trace from the triggering point. Since CoInDx reconstructs the vulnerable state of the program by composing a new, simpler program, it requires access to the source code. However, CoInDx does not require the source code to be human-readable. In other words, CoInDx takes the code of production release as is, which is often minified, bundled, or obfuscated.

5.3 Design

To investigate the root cause of code injection attacks in JavaScript applications (e.g., the real-world example discussed in subsection 5.2.2), we propose CoInDx. CoInDx systematically addresses the key challenges in root cause analysis by tracking event registrations and executions to reconstruct an equivalent but simpler vulnerable symbolic state as the original program. CoInDx operates in four key stages:

Event Activity Tracking. It monitors and records event registration, triggering, and handling to recover the full call stack trace.

Program Recomposition. It constructs a simplified program containing only the functions involved in the recovered trace, mitigating state explosion for symbolic analysis.

Iterative Symbolic Analysis. It iteratively performs symbolic execution to reconstruct the vulnerable symbolic state.

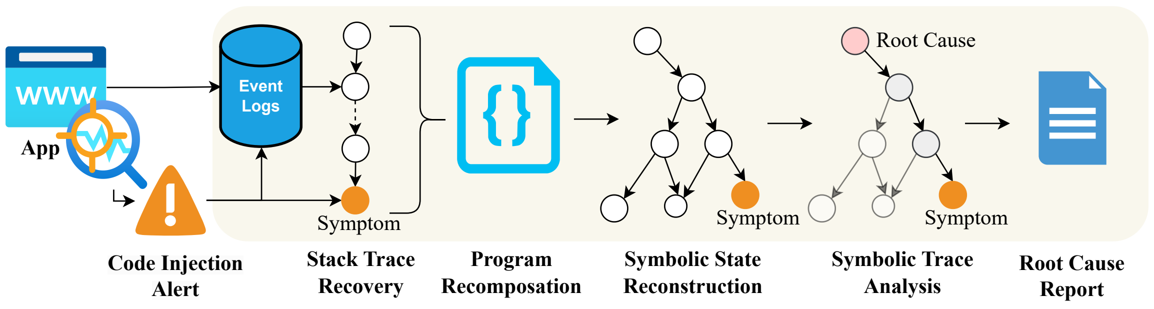


Figure 5.2: CoInDx Workflow Overview.

Root Cause Report Generation. It analyzes the execution traces to determine the root cause, pinpoint the injection location, and provide a recommended fix.

5.3.1 Design Overview

Figure 5.2 illustrates the high-level workflow of CoInDx. A JavaScript application (Web, Node.js, or Electron) executes within CoInDx's event tracking instrumentation code that monitors event activities. These events are captured in Event Logs, which facilitate the reconstruction of a complete call stack trace, addressing **C1**.

When a code injection alert is triggered, CoInDx initiates an investigation. First, it extracts the call stack trace originating from the detected symptom. Then, it identifies the top caller and checks whether it corresponds to an event handler. Next, if the top caller is an event handler, CoInDx iteratively looks for the function that triggers the event in the Event Logs to recover the complete call stack trace.

Once the call stack trace is fully reconstructed, CoInDx extracts all relevant JavaScript functions from the source code of the application. If a function is dynamically generated (e.g., `eval`), CoInDx retrieves its source code from the call stack frame when available, addressing **C2**. The extracted functions are then linked together, with closure variables and functions marked as symbolic variables.

Next, the recomposed program is fed into a symbolic execution engine, which generates symbolic execution traces iteratively. CoInDx analyzes these traces to determine whether they lead to the code injection symptom and, if so, whether they indicate a vulnerability.

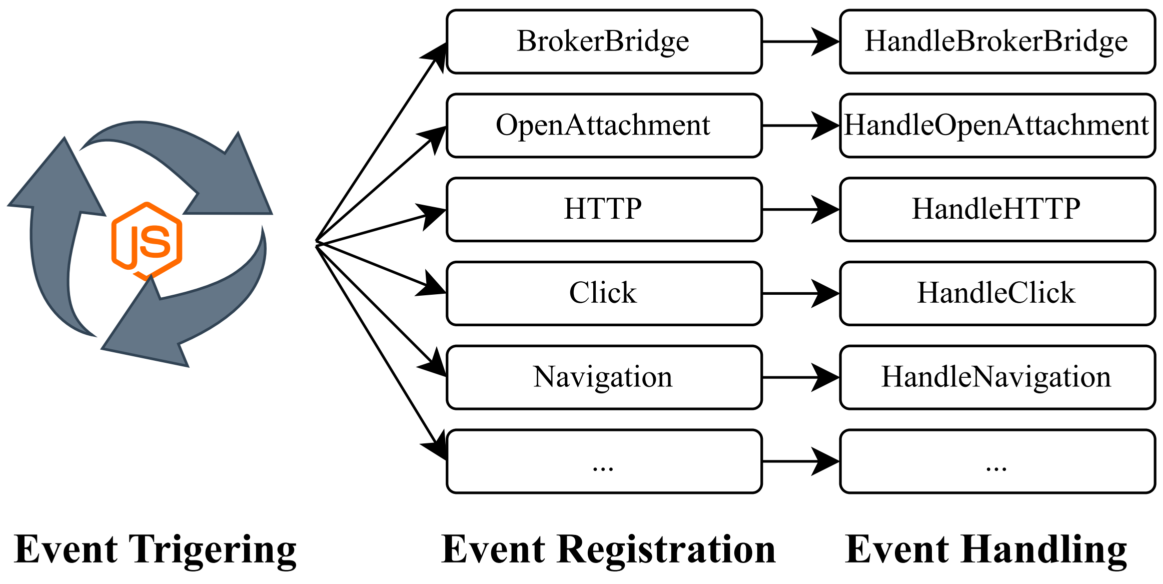


Figure 5.3: Tracking Event Activities for Stack Trace Recovery.

Finally, CoInDx produces a root cause report, which includes 1) a reconstructed execution trace at the branch level, 2) the branch that initiated the code injection attack, and 3) a recommended patch to mitigate the vulnerability.

5.3.2 Stack Trace Recovery

A fundamental challenge in analyzing code injection attacks is reconstructing the execution context that leads to the vulnerability. Traditional debugging tools provide stack traces at runtime, but they often fail to capture the complete execution history in event-driven JavaScript environments. This limitation arises because JavaScript applications, particularly in Web, Node.js, and Electron environments, heavily rely on asynchronous event handling, where execution contexts are fragmented across multiple callbacks.

Given these challenges, CoInDx is designed to *track event registration, triggering, and handling explicitly*, allowing it to reconstruct the execution history beyond what standard stack traces provide. To achieve this goal, CoInDx systematically tracks event activities with additional instrumentation as illustrated in Figure 5.3.

Event Registration Tracking. When a JavaScript function registers an event handler (e.g., via `Target.addEventListener()` on DOM objects or `EventEmitter.on()` for Node.js environment), CoInDx records the event type, associated callback, registration location in the source code, and the call stack trace.

Event Triggering Tracking. When an event is triggered (e.g., via user interaction or code from `Target.dispatchEvent()`, CoInDx logs the event type, payload, and the current call stack trace.

Event Handling Tracking. When an event handler executes, CoInDx captures the event's context, including the event type and the stack frame at the point of execution.

Call Stack Reconstruction. Upon detecting a code injection symptom, CoInDx iteratively traces back the execution path by matching handled events to their activity logs, recovering the full call stack trace.

Modern JavaScript applications frequently generate code dynamically (e.g., using `eval`-like APIs). To ensure completeness, CoInDx extracts dynamically generated functions from stack frames at execution time. Then, CoInDx maps these functions to their originating contexts when possible. Finally, CoInDx incorporates dynamically created closures and bound variables in the reconstructed execution flow.

5.3.3 Program Recomposition

Once the full stack trace is recovered, the next challenge is preparing the execution context for symbolic analysis. Directly analyzing the entire application is infeasible due to state explosion and unnecessary complexity. Instead, CoInDx extracts a *simplified, self-contained program* derived from the recovered stack trace, ensuring that it retains all essential execution semantics while remaining tractable for analysis.

The key insight behind program recomposition is that the vulnerable state/trace is determined by a small subset of the program rather than the entire codebase. By isolating only the functions involved in the recovered call stack, we eliminate irrelevant code while

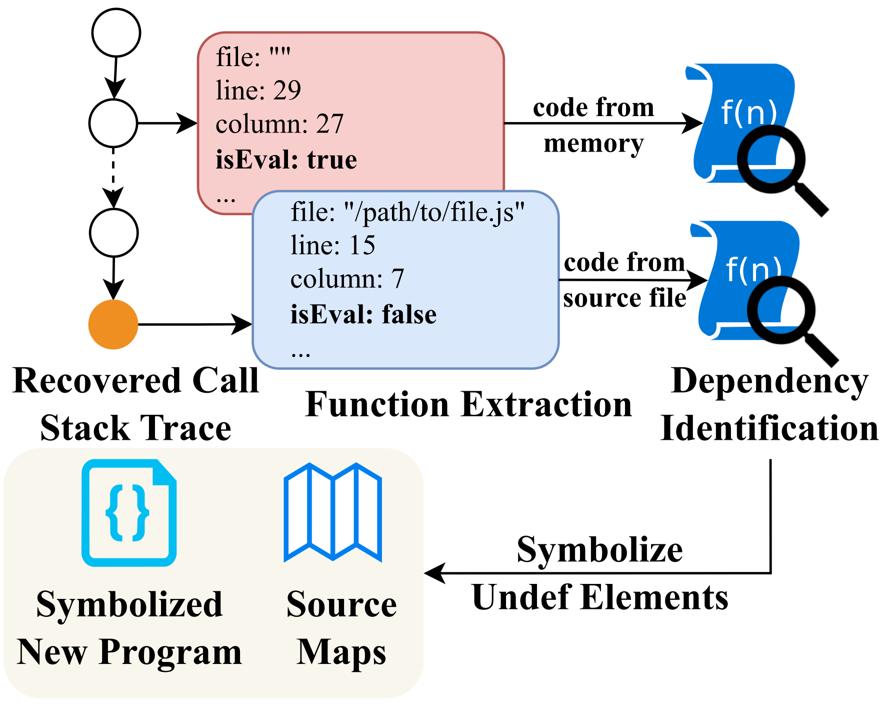


Figure 5.4: The Composer Extracts Source Code and Creates A Symbolized New Program.

preserving the necessary execution semantics. Any missing dependencies, such as external functions or variables, are replaced with symbolic values, allowing COINDX to generalize execution traces and explore additional attack paths. This approach ensures that the recomposed program remains lightweight, scalable, and expressive enough to reproduce and extend the original execution trace. To achieve this goal, COINDX executes the procedures as illustrated in Figure 5.4.

Function Extraction. COINDX extracts all JavaScript functions present in the recovered stack trace from the source code for statically defined functions and those dumped source code for dynamically generated functions discussed in subsection 5.3.2. These functions constitute the core execution context.

Dependency Identification. COINDX analyzes function dependencies, identifying required variables, closures, and external function calls, using static analysis (e.g., JavaScript linter).

Symbolic Representation of Undefined Elements. Any function, variable, or external

API that is not explicitly recovered is replaced with a symbolic value. This ensures that symbolic execution can explore multiple execution paths without being constrained by missing dependencies.

Control Flow Reconstruction. CoInDx connects the extracted functions according to their invocation relationships, ensuring that the recomposed program retains the logical execution flow.

Source Code Mapping. Since the call stack trace includes the location of each function, it is simple for CoInDx to map the extracted functions with their original source. CoInDx maintains three hash maps: one for location in the source code with its function code, one for location in the new program with its function code, and one for location in the new program with the location in the source code.

By incorporating symbolic values for missing dependencies, the recomposed program can generate execution traces that are a superset of the original execution trace. This is because there may be branches within functions. In the vulnerable path, branches were taken based on concrete inputs. To this end, the original vulnerable trace is fully preserved.

5.3.4 Iterative Symbolic Analysis

After program recomposition, the next step is to perform dynamic taint analysis on the simplified program. To achieve this goal, we choose a symbolic execution engine to facilitate the dynamic taint analysis because the recomposed program contains undefined functions and/or variables (i.e., closures). Directly executing the program will lead to a crash. Using a symbolic execution engine, on the other hand, can handle those undefined closures, preventing the execution from crashing. Since JavaScript applications vary widely in execution environments and runtime semantics, CoInDx ensures broad compatibility by leveraging the *Electron runtime* (supporting both web and Node.js) for symbolic execution. This setup enables precise tracking of symbolic values while preserving JavaScript ’s dynamic execution model.

Algorithm 2: Iterative Symbolic Analysis

```
Data: A Symbolized Program $Prog$, Source Code $Src$, A Symbolic Execution
      Trace $T$.
Result: The symbolic representation of variable to the alerting function.
1  begin
2      if not ReachSink($T$) then
3          return;
4      end
5      symbol $\leftarrow$ ExtractSymbol($T$);
6      while HasUndefFunc(symbol) do
           // The symbol contains functions defined outside
              the call stack.
7          $Prog \leftarrow$ Concretize(symbol, $Src$, $Prog$);
           // Extracting those functions from the source code
              and concretizing $Prog$.
8          $T \leftarrow$ SymbolicExecute($Prog$);
           // Re-execute the concretized $Prog$.
9          symbol $\leftarrow$ ExtractSymbol($T$);
10     end
11     return symbol;
12 end
```

CoInDx reproduces the vulnerable execution state by running the program in a real-world JavaScript environment. To achieve this goal, CoInDx performs symbolic execution on the recomposed program. However, a single pass of symbolic execution may not fully resolve the data flow, especially when functions or variables were initially symbolized due to missing definitions. Instead of assuming all symbolic values are attacker-controlled, CoInDx systematically re-evaluates the execution traces by iteratively concretizing and propagating symbolic constraints. This allows CoInDx to distinguish between genuinely attacker-controlled data flows and benign symbolic values (e.g., functions defined outside the call stack trace).

Symbolic Execution. First, CoInDx automatically creates a harness to invoke the recomposed program by marking the input arguments as symbols. Next, CoInDx creates symbolic procedures for undefined functions identified in subsection 5.3.3. Then, the symbolic execution engine explores all feasible execution paths while tracking constraints on symbolic values. Finally, after CoInDx explores all paths, CoInDx extracts the execution traces that reach the symptom function (i.e., the alerting function).

Iterative Analysis. After obtaining the symbolic execution traces, CoInDx needs to determine whether the injected code contains attacker-controlled variables. Traditional taint analysis methods often rely on predefined sources (e.g., network responses, user input) to infer attacker control. However, due to the presence of symbolized undefined functions and variables in the recomposed program, a straightforward taint analysis may yield incomplete or incorrect results. To address this, CoInDx employs *iterative symbolic analysis* to refine the data flow to validate whether an alerting function receives attacker-controlled input.

As illustrated in algorithm 2, given the symbolized program $Prog$, the source code src, and a symbolic execution trace T, the iterative symbolic analysis starts with checking whether the trace reaches the alerting function. If yes, it moves to determine whether the final symbol flowed to the function containing undefined functions. These undefined functions are symbolic functions defined in subsection 5.3.3. If the symbol has them, CoInDx retrieves the function definitions from the source code and concretizes the program $Prog$. Specifically, CoInDx replaces the symbolic procedures with concrete function definitions. Note that, it is possible that the concrete function also invokes functions not defined locally (closure functions). In this case, CoInDx will mark those closure functions as symbolic procedures again. Then, CoInDx executes the concreted $Prog$ again until the final symbol does not contain any undefined functions. Finally, this iteration process returns the symbol only containing symbolic variables and their constraints.

5.3.5 Root Cause Analysis

The final step in CoInDx is to analyze the symbolic execution trace to pinpoint the injection point and identify the attack vector that enabled the vulnerability. Given a symbolic execution trace reaching an alerting function, the goal is to determine: 1) where in the execution flow the injection occurred and 2) which data flow paths allowed attacker-controlled input to reach the injection sink. This process enables precise vulnerability attribution and facilitates automated patch recommendations.

The key insight behind this is that the injection point and attack vector can be traced by examining the data flow within the symbolic execution trace. Instead of only identifying the final function that executes malicious code, COINDX systematically traces back the symbolic constraints to determine how attacker-controlled input propagated through the execution path. This ensures that the analysis captures both the immediate injection point and the underlying weakness that allowed the exploit.

Identifying the Injection Point. The injection point usually occurs in the earliest program location where an attacker-controlled value enters the execution flow and propagates to an injection sink. To locate this point, COINDX first performs backward symbolic trace analysis, tracing execution paths in reverse from the alerting function to determine where injected data originated. Next, dependency tracking is used to analyze variable assignments, function arguments, and return values, ensuring that all symbolic values contributing to the injection sink are correctly identified. To confirm attacker influence, COINDX performs taint verification, checking whether the tracked symbolic values stem from sources such as network responses, user inputs, or other external influences. Finally, once the earliest program location introducing attacker-controlled data is found, COINDX classifies it as the injection point, associating it with the relevant control flow path.

Extracting the Attack Vector. To determine the attack vector, COINDX analyzes how the injected data traversed the program and reached the injection point. The first step in this process is control flow path analysis, where COINDX reconstructs the sequence of function calls, event handlers, and callback chains that facilitated data propagation. Next, COINDX performs symbolic constraint analysis, examining transformations applied to the injected data, such as string concatenation, sanitization attempts, or encoding operations. If modifications exist, CoInDx determines whether these transformations still permit an attack. Based on these analyses, CoInDx classifies the vulnerability into one of three categories: *direct injection*, where the attacker-controlled input reaches the sink without modification; *context-sensitive injection*, where the input undergoes transformations but remains injectable due to improper sanitization; and *indirect injection*, where the attack occurs through an intermediate function call or event-driven execution.

Generating the Root Cause Report. After identifying the injection point and attack vector, CoInDx generates a root cause report to assist developers in vulnerability remediation. The report includes the reconstructed execution trace, which highlights the key program branches leading to injection, helping developers understand how the attack unfolded. Additionally, the report specifies the injection point, providing the exact source code location where attacker-controlled data first entered the execution flow. To further clarify the attack mechanism, CoInDx details the attack vector, explaining how the injected data propagated through the program. Finally, the report provides suggested patch recommendations, including mitigation strategies such as input sanitization, secure API replacements, or event handler restrictions, with the help of current LLM models.

5.3.6 Implementation

To ensure the practicality and effectiveness of CoInDx, we carefully design its implementation to balance security, performance, and usability. This section details the implementation of stack trace recovery, program composition, and symbolic execution.

Stack Trace Recovery. To recover full execution traces in an event-driven JavaScript environment, we modify the JavaScript prototype chain and make it read-only. This design choice allows developers to integrate CoInDx into their codebase without requiring modifications to the underlying JavaScript runtime of their end users. By intercepting event registrations and executions through prototype modification, CoInDx systematically tracks event bindings and reconstructs call stacks without interfering with application logic. This approach is both secure and practical. Since JavaScript allows prototype modifications but restricts certain built-in functions from being redefined, making our modifications read-only prevents tampering while maintaining compatibility with real-world applications. Furthermore, compared to alternative approaches—such as instrumenting the low-level browser engine or Node.js native code, our method is significantly more practical for real-world adoption. While deeper instrumentation could provide stronger security guarantees by preventing adversaries from tampering with the event tracking system, it is far less practical due to the complexity of modifying and maintaining a custom JavaScript runtime for end users.

Program Composer. To generate the recomposed program for symbolic execution, COINDX leverages acron [114], a widely-used JavaScript AST parser, alongside ESLint [131], a static analysis tool, to extract and recompose relevant code from the recovered stack trace. The program composition process consists of the following steps: 1) COINDX uses Acorn to parse JavaScript source files into an AST, allowing precise function extraction and analysis. 2) Using ESLint, COINDX performs static analysis to track function dependencies, identify closure variables, and ensure the integrity of the recomposed program. 3) Any function or variable that is unresolved in the recomposed program is replaced with a symbolic representation, ensuring that symbolic execution can generalize execution traces while maintaining correctness.

Symbolic Execution Engine. For symbolic execution, COINDX integrates with ExpoSE [45], a symbolic execution engine designed for JavaScript, running within the Electron runtime. This design choice ensures that COINDX supports both web-based JavaScript applications and native JavaScript environments such as Node.js and Electron applications. The Electron-based runtime enables COINDX to execute recomposed programs in an environment that closely mirrors real-world JavaScript execution, ensuring accurate symbolic state propagation. By running within Electron, COINDX can handle browser-based execution contexts, event-driven execution models, and native API interactions, making it more versatile than approaches that rely solely on browser-specific or native-only execution models. ExpoSE provides fine-grained symbolic constraint tracking, allowing CoInDx to explore execution paths efficiently while maintaining compatibility with modern JavaScript applications.

Root Cause Report Generation. Given the vulnerable execution trace identified, the injection point, and attack vector determined, CoInDx generates a root cause report to assist developers in understanding and mitigating the vulnerability through GPT-4o-mini. This root cause report includes: 1) the reconstructed execution trace, highlighting the key program branches leading to injection, 2) the injection point, providing the exact source code location where attacker-controlled data first entered the execution flow, and 3) suggested patch recommendations, including mitigation strategies such as input sanitization, secure API replacements, or event handler restrictions.

5.4 Evaluation

To assess the effectiveness and practicality of CoInDx, we conduct a comprehensive evaluation based on real-world applications and known code injection vulnerabilities. The evaluation is designed to answer the following key research questions:

RQ1: How accurate is CoInDx at detecting code injection vulnerabilities? We evaluate whether CoInDx can correctly identify true vulnerabilities while minimizing false positives and false negatives.

RQ2: Can CoInDx locate and analyze real-world exploits? We assess whether CoInDx can successfully reconstruct execution traces and uncover vulnerabilities in widely used JavaScript applications.

RQ3: Are the reported root causes consistent with real-world advisories and security patches? We compare CoInDx 's findings with official security advisories and developer patches to validate its effectiveness in pinpointing the root cause of vulnerabilities.

RQ4: Is CoInDx practical to deploy in real-world environments in terms of runtime and storage overhead? We measure the computational and storage overhead introduced by CoInDx and evaluate its impact on application performance.

5.4.1 Accuracy on SecBench.js

To answer **RQ1**, we evaluate CoInDx 's accuracy in detecting code injection vulnerabilities using SecBench.js [125], a benchmark dataset specifically designed for assessing JavaScript security analysis tools. SecBench.js comprises 40 known code injection vulnerabilities and 101 command injection vulnerabilities found in widely used NPM modules, making it a suitable testbed for evaluating CoInDx's detection capabilities.

The choice of SecBench.js as the benchmark dataset is motivated by several factors: First, it consists of real-world vulnerabilities found in widely used JavaScript libraries and frameworks, ensuring that our evaluation reflects security risks encountered in practical deployments. Second, SecBench.js includes diverse attack patterns, covering different ways attacker-controlled input propagates to injection sinks, such as direct user input, event-driven execution, and indirect control flow dependencies. Third, it provides ground truth labels, allowing us to objectively measure CoInDx 's true positive rate (TPR) and false positive rate (FPR) against known vulnerabilities.

Experiment Setup. To set up this experiment, we need to modify the existing testing harnesses for exploiting those vulnerabilities. Specifically, we first add the instrumentation code for tracking events. Then, we make the exploits to dump the stack trace by raising a customized `Error` exception. In other words, when the exploiting payload reaches the injection sink, a stack trace can be dumped directly. Next, CoInDx takes the stack trace and event logs to identify the root cause.

The detection performance of CoInDx on the SecBench.js dataset is summarized in

Table 5.1: Performance of CoInDx on SecBench.js.

Attack Vectors	**TP**	**FP**	**FN**	**Accuracy**
Code Injection				
Direct Injection	31	0	0	100%
Context-Sensitive Injection	7	0	0	100%
Indirect Injection	2	0	0	100%
Subtotal	40	0	0	100%
Command Injection				
Direct Injection	83	0	0	100%
Context-Sensitive Injection	15	0	0	100%
Indirect Injection	4	0	0	100%
Subtotal	101	0	0	100%
Total	141	0	0	100%

Table 5.1. The dataset includes a total of 40 code injection vulnerabilities and 101 command injection vulnerabilities. To better understand CoInDx 's accuracy across different exploitation techniques, we categorize injection attacks based on the attack vectors introduced in subsection 5.3.5.

We define a true positive (TP) as a case where CoInDx correctly identifies the root cause by locating the exact injection point and execution trace. A false positive (FP) occurs when CoInDx incorrectly identifies an injection point that is not the actual root cause. A false negative (FN) is when CoInDx fails to detect the vulnerability. Since all modules in the benchmark dataset are confirmed to be exploitable, there are no true negatives (TN), meaning TN is always zero for each attack vector.

For code injection vulnerabilities, CoInDx identifies 31 cases as direct injection, 7 cases as context-sensitive injection, and 2 cases as indirect injection. Among these, CoInDx successfully pinpoints all the root causes. For code injection vulnerabilities, CoInDx identifies 83 cases as direct injection, 15 cases as context-sensitive injection, and 4 cases as indirect injection. Among these, CoInDx successfully pinpoints all the root causes.

These results highlight CoInDx 's ability to accurately detect and attribute code

Table 5.2: Performance of CoInDx on Real World Applications.

CVE #	Application	# Functions	CoInDx # Functions	Located	Analysis Time (hours)
Web					
CVE-2023-33831	FUXA	9,156	17	Yes	0.15
CVE-2023-2564	scanservjs	9,537	10	Yes	0.12
CVE-2024-27448	maildev	10,893	18	Yes	0.18
CVE-2023-49276	uptime	12,863	11	Yes	0.58
CVE-2022-0944	sqlpad	13,881	4	Yes	0.07
Node.js					
CVE-2023-42810	systeminformation	6,394	15	Yes	1.63
CVE-2024-56334	systeminformation	7,031	12	Yes	0.56
CVE-2025-24981	@nuxtjs/mdc	10,594	5	Yes	0.12
CVE-2024-4367	pdf.js	17,380	9	Yes	1.17
Electron					
CVE-2023-2479	Appium	24,666	16	Yes	1.62
CVE-2024-49362	joplin	12,613	12	Yes	0.46
CVE-2024-4367	Evernote	20,586	28	Yes	1.82

injection vulnerabilities, with most failures occurring in complex indirect injection flows where symbolic execution constraints could not be fully resolved. The high success rate across direct and context-sensitive injections indicates that CoInDx effectively reconstructs execution traces and identifies attacker-controlled inputs. However, challenges remain in handling cases where undefined functions or event-driven execution models introduce ambiguity in symbolic constraints.

5.4.2 Locating Real-World Exploits

To answer **RQ2** and **RQ3**, we test JavaScript with 11 real-world JavaScript applications with 12 real-world exploits, covering five Web, three Node.js, and three Electron applications, to evaluate whether CoInDx can locate and analyze real-world exploits.

Experiment Setup. To set up this experiment, we first host the applications in a controlled environment. We then run the applications with CoInDx 's instrumentation enabled to capture the event registration, triggering, and handling. Before mounting the attacks, we ensure that the applications are running correctly and that CoInDx is capturing the events as expected. Then, to simulate the external monitoring tools, we make the exploits to dump the stack trace by raising a customized `Error` exception as we do for the benchmark evaluation. Next, we inject known exploits into the applications to simulate real-world attacks. We then analyze the event logs generated by COINDX to identify the root cause of the vulnerabilities. Finally, we compare COINDX 's findings with official security advisories and developer patches to validate its effectiveness in locating and analyzing real-world exploits.

Table 5.2 summarizes the results of the real-world exploit analysis. For each application, we report the number of functions defined in the original application, the number of functions COINDX extracts for iterative symbolic analysis, whether COINDX successfully locates the root cause of the vulnerability, and the analysis time in hours used by COINDX to locate the root cause. Additionally, COINDX also suggests the patch to fix the vulnerability, which is compared with the official security advisories and developer patches to validate its effectiveness in pinpointing the root cause of vulnerabilities.

Web Applications. As shown in the top part of Table 5.2, COINDX successfully locates the root cause of the vulnerabilities in all five applications, demonstrating its effectiveness in analyzing client-side rendering and event-driven execution models. The analysis time ranges from 0.15 to 0.58 hours, with an average of 0.22 hours per application. The results show that COINDX can accurately identify the root cause of vulnerabilities in web applications, making it a valuable tool for securing client-side applications.

Node.js Applications. For Node.js applications, shown in the middle part in Table 5.2, COINDX successfully locates the root cause of the vulnerabilities in all five applications, demonstrating its effectiveness in analyzing server-side processing and high-throughput workloads. The analysis time ranges from 0.12 to 1.63 hours, with an average of 0.87 hours per application. The results show that COINDX can accurately identify the root cause of vulnerabilities in Node.js applications, making it a valuable tool for securing server-side applications.

Electron Applications. Illustrated in the bottom part in Table 5.2, As a result, COINDX successfully locates the root cause of the vulnerabilities in all five applications, demonstrating its effectiveness in analyzing multi-process desktop applications. The analysis time ranges from 0.46 to 1.82 hours, with an average of 1.30 hours per application. The results show that CoInDx can accurately identify the root cause of vulnerabilities in Electron applications, making it a valuable tool for securing desktop applications.

5.4.3 Overhead

To answer **RQ4**, we evaluate the runtime overhead introduced by CoInDx when deployed in real-world applications. Specifically, we measure the overhead incurred during event registration, triggering, and handling tracking, as these operations form the core of CoInDx 's runtime instrumentation. The evaluation considers applications across three major JavaScript execution environments: web, Node.js, and Electron.

For web applications, we use two state-of-the-art benchmarks designed to measure browser responsiveness. These benchmarks provide a standardized way to assess the impact of CoInDx 's instrumentation on JavaScript execution performance. For Node.js, we select two widely used applications: ESLint [131], a widely adopted static analysis tool used by almost every JavaScript developer, and Express.js [132], the most popular backend framework for JavaScript. These applications allow us to evaluate CoInDx 's overhead in both computationally intensive workloads (such as static analysis) and high-throughput server applications. For Electron, we choose VSCode [133], one of the most widely used editors, known for its large-scale Electron-based architecture and comprehensive end-to-end testing. Additionally, we use Electron-IPC-Bench, a dedicated benchmark designed to stress IPC messages in Electron applications, allowing us to measure CoInDx 's impact on messaging events efficiency. By evaluating a diverse set of real-world applications, we ensure that CoInDx 's runtime performance is assessed across different execution models, including client-side rendering, backend processing, and multi-process desktop applications.

Table 5.3: Overhead CoInDx Incurs to Benchmark Applications.

Apps	# Events Reg	# Events Trg	Runtime	Memory (MB)	Storage (MB)
Web					
Jetstream2 [134]	31	854	1.24%	7	2
Speedometer3.0 [115]	14	2,045	0.96%	5	1
Node.js					
Eslint [131]	17	651	1.65%	4	1
Express.js [132]	248	542	2.18%	26	15
Electron					
VSCode [133]	157	715	3.71%	18	10
electron-ipc-bench	20	1,000	1.71%	5	1

Experiment Setup. To set up this experiment, we ran every application in an Electron with the event tracking hooks installed. For example, we open those web benchmarks in Electron's browser window. For Node.js and Electron applications, we run the built-in test suites to measure the overhead. When running, we measure the time taken to register and consume events in the application. We also measure the memory overhead introduced by CoInDx by comparing the memory usage of the application with and without CoInDx. Additionally, we measure the storage overhead by measuring the size of the event logs generated by CoInDx on disk.

We present the results in Table 5.3. For web applications, the overhead is minimal, with a 1.24% increase in runtime. This is expected, as web applications are typically event-driven and have a low number of events compared to server applications. For Node.js applications, the overhead is also negligible, with a 2.18% increase in runtime. This is due to the lightweight nature of CoInDx 's instrumentation, which only tracks events during the execution of the application. For Electron applications, the overhead is slightly higher, with less than 3.71% increase in runtime. This is because Electron applications are more complex and have a higher number of events compared to web and Node.js applications. Overall, the results demonstrate that CoInDx introduces minimal overhead when deployed in real-world applications, making it practical for use in production environments. The memory and storage overhead of storing these logs are less than 10 MB on average.

5.4.4 Case Study: Investigating CVE-2024-4367

To illustrate how CoInDx operates in practice, we present a case study on CVE-2024-4367, a real-world code injection vulnerability affecting Evernote, an Electron-based application with over 200 million active users. This vulnerability stems from PDF.js [130], a widely used library maintained by Mozilla for rendering PDFs in browsers and desktop applications. Attackers exploit this flaw by embedding malicious JavaScript payloads within the `FontMatrix` metadata of a PDF document, which when processed, results in arbitrary code execution.

In this scenario, the payload is extracted and executed through `new Function()`, leading to RCE. A runtime monitoring system flags this event due to the abnormal execution of shell commands within the renderer process. However, traditional detection systems only capture the *symptom* of the attack–the execution of a system command-—but do not reveal how the injected payload was processed and executed. Once the external monitoring system detects the anomalous behavior, CoInDx begins its investigation. The analysis proceeds in several steps:

Stack Trace Recovery. At the moment of attack execution, CoInDx collects the stack trace leading to the execution of the shell command. Initially, this trace contains only the immediate call sequence, capturing functions within the callback function that invoked the dangerous API. However, because Electron applications use event-driven execution and IPC, the source of the vulnerability is not immediately apparent. To reconstruct the full execution flow, CoInDx bridges event listeners and emitters to link related execution paths. The event logs reveal that the vulnerable function was invoked through Electron's IPC mechanism, which is commonly used for communication between the main process and the renderer process. CoInDx traces this execution path across processes, ultimately identifying that the attack originated from a function inside `PreCompileFont()` in PDF.js.

Program Recomposition and Symbolization. After reconstructing the call sequence, CoInDx extracts all relevant JavaScript functions involved in the execution flow. The recovered functions include `PreCompileFont()`, which processes the `FontMatrix` metadata from the PDF, and `openAttachment()`, which ultimately spawns the shell command. Because some dependencies are dynamically loaded at runtime, CoInDx symbolically represents undefined variables and missing functions to generalize execution traces. By composing a simplified program that retains only the relevant execution logic, CoInDx ensures that symbolic execution remains tractable while preserving the key vulnerability conditions.

Iterative Symbolic Analysis. Using the recomposed program, CoInDx executes iterative symbolic analysis to determine whether the injection point is attacker-controlled. The symbolic execution engine marks file path arguments and metadata fields as symbolic variables and explores execution traces where these values propagate to dangerous functions. Through iterative refinement, CoInDx reveals that the injected `FontMatrix` value undergoes string manipulation and eventually reaches `new Function()`, where it is executed as JavaScript. This confirms that the attacker fully controls the payload reaching the execution sink.

Identifying the Injection Point and Attack Vector. The injection point is identified as the `FontMatrix` parser inside `PreCompileFont()`, where the PDF metadata is extracted and processed. CoInDx traces the attack vector, showing that the manipulated `FontMatrix` field bypasses any sanitization and directly reaches an execution context. The vulnerability is classified as context-sensitive injection, where the attacker's input undergoes transformations but remains exploitable.

Patch Recommendation and Outcome. Based on its analysis, CoInDx generates a root cause report that highlights: 1) The execution trace leading to the attack, including all key branches. 2) The exact injection point in `PreCompileFont()`, pinpointing where untrusted input first enters. 3) The recommended patch suggests that developers implement strict input validation for PDF metadata and prohibit the execution of dynamically generated functions to prevent such injections.

Following public disclosure, Mozilla addressed CVE-2024-4367 by introducing input validation checks and removing the use of `new Function()` for processing `FontMatrix` metadata. The patch aligns closely with CoInDx 's suggested fix, validating its effectiveness in diagnosing real-world vulnerabilities.

This case study demonstrates the effectiveness of CoInDx in diagnosing complex code injection vulnerabilities in JavaScript applications. By leveraging event tracking, program recomposition, and iterative symbolic analysis, CoInDx successfully reconstructs the execution trace, identifies the root cause, and provides actionable remediation steps. Unlike traditional runtime monitoring solutions that focus solely on attack symptoms, CoInDx enables developers to eliminate systemic weaknesses, preventing recurring exploits.

5.5 Discussion

While CoInDx demonstrates strong performance in diagnosing JavaScript code injection vulnerabilities, it has several limitations that should be considered when applying it to real-world deployments. This section discusses these limitations and potential areas for future improvement.

Handling of Complex Event-Driven Execution. JavaScript's event-driven execution model introduces challenges in reconstructing execution traces, particularly in applications that heavily rely on asynchronous event dispatching, timers, and message-passing mechanisms. While CoInDx tracks event registration, triggering, and handling to bridge execution gaps, it may fail to fully resolve non-deterministic event interleavings in multi-threaded execution contexts, such as Electron's main-renderer process communication or Node.js's asynchronous callbacks. This limitation could lead to incomplete trace reconstruction, potentially missing subtle vulnerabilities that arise due to race conditions in event execution.

Scalability of Iterative Symbolic Execution. Symbolic execution is known to suffer from state explosion, particularly when analyzing JavaScript applications with deep call stacks, dynamically generated code, or extensive control flow branching. Although COINDX mitigates this issue by recomposing a smaller subprogram from the stack trace, it may still struggle with large-scale applications where iterative symbolic analysis leads to an excessive number of execution paths. One possible enhancement is the incorporation of concolic execution (concrete + symbolic execution) to selectively concretize certain symbolic values, thereby pruning infeasible execution paths. However, this requires dumping the values of the variables at runtime, which poses privacy concerns. Future work could explore privacy-preserving concolic execution techniques to address this challenge.

Detection of Indirect Injection Attacks. COINDX performs well in identifying direct and context-sensitive injections, but it faces challenges in detecting indirect injection attacks where the exploit payload propagates through intermediate function calls, object property mutations, or external dependencies. An example of this is prototype pollution, where an attacker manipulates global object properties, leading to unintended function execution at a later stage. Because COINDX relies on execution trace reconstruction, it may not capture cases where injection points are introduced indirectly through object modifications or deferred executions. Addressing this limitation requires enhancing COINDX with data flow analysis across object hierarchies and tracking implicit control flows that introduce code execution paths not explicitly present in the stack trace.

Generalizing Root Cause Analysis to Other JavaScript Attacks. COINDX is specifically designed to diagnose code injection vulnerabilities in JavaScript applications. However, JavaScript applications face other attack vectors, such as prototype pollution, deserialization attacks, and path traversal vulnerabilities. Extending COINDX 's approach to general root cause analysis for JavaScript security could enhance its applicability across a broader range of security threats. One possible extension is integrating monitoring for other types of vulnerabilities, such as prototype pollution, by tracking object property modifications. Once detected, CoInDx could reconstruct the execution trace to identify the origins of the modification and provide insights into how it propagates through the application, which can be done asynchronous.

CHAPTER 6
CONCLUSION

In this dissertation, we have explored the challenges of injection attacks (i.e., socially engineered content and malicious code) in the modern web ecosystem, highlighting how modern web threats often span both user interface deception and technical exploitation. To address these challenges, we introduced three novel systems: TRIDENT, COINDEF, and COINDX. Together, these solutions form a unified defense framework that detects, prevents, and analyzes injection-based attacks across different layers of the web stack. TRIDENT focuses on browser-based detection of deceptive content served through lower-tier ad networks; COINDEF secures cross-platform applications by enforcing fine-grained execution policies within the Electron runtime; and COINDX provides developers with a systematic tool for identifying and understanding the root causes of code injection vulnerabilities in JavaScript applications.

By emphasizing attack surface reduction rather than reactive threat detection alone, this research advocates for a proactive security model—one that limits opportunities for exploitation before they can manifest. The proposed techniques offer complementary protections that span content filtering, execution policy enforcement, and post-mortem vulnerability analysis, thereby addressing injection threats holistically. As cyber threats continue to evolve, we anticipate that browser-integrated defenses, language-aware execution environments, and automated vulnerability analysis tools will play an increasingly vital role in safeguarding both users and developers. Ultimately, this work contributes to a more resilient web ecosystem by rethinking the boundaries between content-level, application-level, and developer-facing security.

REFERENCES

[1] N. Miramirkhani, O. Starov, and N. Nikiforakis, “Dial One for Scam: A Large-Scale Analysis of Technical Support Scams,” 2017.

[2] A. Kharraz, W. Robertson, and E. Kirda, “Surveylance: Automatically Detecting Online Survey Scams,” in *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2018-May, Institute of Electrical and Electronics Engineers Inc., Jul. 2018, pp. 70–86, ISBN: 9781538643525.

[3] L. Invernizzi, P. M. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna, “EvilSeed: A guided approach to finding malicious web pages,” in *Proceedings - IEEE Symposium on Security and Privacy*, Institute of Electrical and Electronics Engineers Inc., 2012, pp. 428–442, ISBN: 9780769546810.

[4] P. Vadrevu and R. Perdisci, “What you see is not what you get: Discovering and tracking social engineering attack campaigns,” *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, pp. 308–321, 2019.

[5] M. Zubair Rafique, T. Van Goethem, W. Joosen, C. Huygens, and N. Nikiforakis, “It’s Free for a Reason: Exploring the Ecosystem of Free Live Streaming Services,” Internet Society, May 2017.

[6] M. Zhang, W. Meng, S. Lee, B. Lee, and X. Xing, “All your clicks belong to me: Investigating click interception on the web,” in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[7] I. Sanchez-Rola, D. Balzarotti, C. Kruegel, G. Vigna, and I. Santos, “Dirty clicks: A study of the usability and security implications of click-related behaviors on the web,” in *Proceedings of The Web Conference 2020*, ser. WWW ’20, Taipei, Taiwan: Association for Computing Machinery, 2020, pp. 395–406, ISBN: 9781450370233.

[8] *Clickjacking — owasp foundation*, https://owasp.org/www-community/attacks/Clickjacking.

[9] *X-frame-options - http — mdn*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options, 2022.

[10] S. Calzavara, S. Roth, A. Rabitti, M. Backes, and B. Stock, “A tale of two headers: A formal analysis of inconsistent Click-Jacking protection on the web,” in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 683–697, ISBN: 978-1-939133-17-5.

[11] L. S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson, “Clickjacking: Attacks and defenses,” in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[12] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel, “A solution for the automated detection of clickjacking attacks,” in *Proceedings of the 5th International Symposium on Information, Computer and Communications Security, ASIACCS 2010*, 2010.

[13] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song, “Clickjacking revisited a perceptual view of UI security,” in *8th USENIX Workshop on Offensive Technologies, WOOT 2014*, 2014.

[14] K. Subramani, X. Yuan, O. Setayeshfar, P. Vadrevu, K. H. Lee, and R. Perdisci, “When Push Comes to Ads: Measuring the Rise of (Malicious) Push Advertising,” in *{IMC} ’20: {ACM} Internet Measurement Conference, Virtual Event, USA, October 27-29, 2020*, 2020, pp. 724–737, ISBN: 9781450381383.

[15] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, “AdGraph: A graph-based approach to ad and tracker blocking,” in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, Virtual Conference, May 2020, pp. 763–776.

[16] Q. Chen, P. Snyder, B. Livshits, and A. Kapravelos, “Detecting Filter List Evasion With Event-Loop-Turn Granularity JavaScript Signatures,” May 2021.

[17] Z. Ul Abi Din, P. Tigas, S. T. King, and B. Livshits, “PERCIVAL: Making in-browser perceptual ad blocking practical with deep learning,” in *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020, ISBN: 9781939133144.

[18] S. Siby, U. Iqbal, S. Englehardt, Z. Shafiq, and C. Troncoso, “WEBGRAPH: Capturing Advertising and Tracking Information Flows for Robust Blocking,” in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[19] B. Li, P. Vadrevu, K. H. Lee, and R. Perdisci, “Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions,” in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[20] J. Allen *et al.*, “Mnemosyne: An Effective and Efficient Postmortem Watering Hole Attack Investigation System,” in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Nov. 2020, pp. 787–802.

[21] F. Xiao, Z. Yang, J. Allen, G. Yang, G. Williams, and W. Lee, "Understanding and mitigating remote code execution vulnerabilities in cross-platform ecosystem," in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, US, Nov. 2022, pp. 2975–2988.

[22] Z. Jin, S. Chen, Y. Chen, H. Duan, J. Chen, and J. Wu, "A security study about electron applications and a programming methodology to tame dom functionalities," in *Proceedings of the 2023 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2023.

[23] M. M. Ali, M. Ghasemisharif, C. Kanich, and J. Polakis, "Rise of inspectron: Automated black-box auditing of cross-platform electron apps," in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[24] *Electronjs electron : Security vulnerabilities, cves*, https://www.cvedetails.com/vulnerability-list/vendor_id-17824/product_id-44696/Electronjs-Electron.html, (Accessed on 11/05/2024).

[25] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against dom-based cross-site scripting," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014, pp. 655–670.

[26] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, "From facepalm to brain bender: Exploring client-side cross-site scripting," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015, pp. 1419–1430.

[27] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis.," in *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2007.

[28] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of dom-based xss," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013, pp. 1193–1204.

[29] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node. js.," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[30] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the eval that men do," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 34–44.

[31] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, “Cspautogen: Black-box enforcement of content security policy upon real-world websites,” in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016, pp. 653–665.

[32] P. Soni, E. Budianto, and P. Saxena, “The sicilian defense: Signature-based whitelisting of web javascript,” in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015.

[33] A. Fass, M. Backes, and B. Stock, “Jstap: A static pre-filter for malicious javascript detection,” in *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, San Juan, Puerto Rico, Dec. 2019, pp. 257–269.

[34] S. Li, M. Kang, J. Hou, and Y. Cao, “Mining Node.js Vulnerabilities via Object Dependence Graph and Query,” Tech. Rep.

[35] S. Li, M. Kang, J. Hou, and Y. Cao, “Detecting node. js prototype pollution vulnerabilities via object lookup analysis,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 268–279.

[36] Z. Liu, K. An, and Y. Cao, “Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node. js template engines for malicious consequences,” in *Proceedings of the 45th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2024.

[37] M. Kang *et al.*, “Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability,” in *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2023.

[38] J. Jueckstock and A. Kapravelos, “VisibleV8: In-browser Monitoring of JavaScript in the Wild,” in *Proceedings of the Internet Measurement Conference (IMC)*, Amsterdam, Netherlands, Oct. 2019.

[39] Z. Yang, J. Allen, M. Landen, R. Perdisci, and W. Lee, “Trident: Towards detecting and mitigating web-based social engineering attacks,” in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023, pp. 1681–1698.

[40] I. Koishybayev and A. Kapravelos, “Mininode: Reducing the attack surface of node.js applications,” in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, San Sebastian, Spain, Oct. 2020.

[41] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, “{Arcus}: Symbolic root cause analysis of exploits in production systems,” in

Proceedings of the 30th USENIX Security Symposium (Security), Virtual Conference, Aug. 2021.

[42] C. Yagemann, S. P. Chung, B. Saltaformaggio, and W. Lee, “Automated bug hunting with data-driven symbolic root cause analysis,” in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Seoul, South Korea, Nov. 2021.

[43] M. Steffens and B. Stock, “PMForce : Systematically Analyzing postMessage Handlers at Scale,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 493–505, ISBN: 9781450370899.

[44] M. Shcherbakov, M. Balliu, and C.-A. Staicu, “Silent spring: Prototype pollution leads to remote code execution in node. js,” in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[45] B. Loring, D. Mitchell, and J. Kinder, “Expose: Practical symbolic execution of standalone javascript,” in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017, pp. 196–199.

[46] D. Cassel, N. Sabino, R. Martins, and L. Jia, “Nodemedic-fine: Automatic detection and exploit synthesis for node.js vulnerabilities,” in *Proceedings of the 2025 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2025.

[47] *Chrome devtools protocol*, [Online; accessed 20-January-2022], 2022.

[48] F. Salahdine and N. Kaabouch, “Social engineering attacks: A survey,” *Future Internet*, vol. 11, no. 4, p. 89, 2019.

[49] *Fully 84 percent of hackers leverage social engineering in cyber attacks*, https://www.esecurityplanet.com/threats/fully-hackers-leverage-social-engineering-in-cyber-attacks/, 2017.

[50] *The social engineering infographic - security through education*, https://www.social-engineer.org/social-engineering/social-engineering-infographic/.

[51] G. Costantino, A. La Marra, F. Martinelli, and I. Matteucci, “Candy: A social engineering attack to leak information from infotainment system,” in *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, IEEE, 2018, pp. 1–5.

[52] *15 alarming cyber security facts and stats — cybint*, https://www.cybintsolutions.com/cyber-security-facts-stats/, 2020.

[53] *New data shows ftc received 2.8 million fraud reports from consumers in 2021 — federal trade commission*, https://www.ftc.gov/news-events/news/press-releases/2022/02/new-data-shows-ftc-received-28-million-fraud-reports-consumers-2021-0, 2022.

[54] T. Yu, A. Association for Computing Machinery. Special Interest Group on Security, National Science Foundation (U.S.), Association for Computing Machinery, and ACM Digital Library., "Knowing Your Enemy: Understanding and DetectingMalicious Web Advertising," p. 1070, ISBN: 9781450316514.

[55] USENIX Association., ACM SIGMOBILE., and ACM Digital Library., "Towards Measuring and Mitigating Social Engineering Software Download Attacks," USENIX Association, 2005, p. 48, ISBN: 9781931971324.

[56] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, "The dark alleys of madison avenue: Understanding malicious advertisements," in *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, Association for Computing Machinery, Nov. 2014, pp. 373–379, ISBN: 9781450332132.

[57] *Blocking goals and policy - brave browser wiki*, [Online; accessed 20-January-2022], 2021.

[58] *Rainbow blocker adware - easy removal steps (updated)*, https://www.pcrisk.com/removal-guides/23298-rainbow-blocker-adware, 2022.

[59] *Virustotal*, [Online; accessed 20-January-2022], 2022.

[60] *How much money do websites make from advertising?* https://adsterra.com/blog/how-much-money-websites-make-from-ads/, 2020.

[61] *Best cpm rates for publishers and webmasters*, https://adsterra.com/blog/geos-with-high-cpm-rates-for-publishers/.

[62] *Google display ads cpm, cpc, & ctr benchmarks in q1 2018*, https://blog.adstage.io/google-display-ads-cpm-cpc-ctr-benchmarks-in-q1-2018, 2018.

[63] *Better ads standards - google ad manager*, https://admanager.google.com/home/resources/feature-brief-better-ads-standards/, 2018.

[64] *Advertising and marketing on the internet: Rules of the road — federal trade commission*, https://www.ftc.gov/business-guidance/resources/advertising-marketing-internet-rules-road, 2022.

[65] *What are iab standard ads? why are they important?* https://www.adpushup.com/blog/what-are-iab-standard-ads-why-are-they-important/, 2021.

[66] *Iab new ad portfolio: Advertising creative guidelines*, https://www.iab.com/guidelines/iab-new-ad-portfolio/, 2022.

[67] U. Iqbal, C. Wolfe, C. Nguyen, S. Englehardt, and D. Zubair Shafiq, "KHALEESI: Breaker of Advertising and Tracking Request Chains," Tech. Rep.

[68] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[69] *Search engine for source code - publicwww.com*, https://publicwww.com/, 2022.

[70] *Puppeteer*, [Online; accessed 20-January-2022], 2022.

[71] *Easylist*, [Online; accessed 20-January-2022], 2022.

[72] *Blocklistproject/lists: Primary block lists*, [Online; accessed 05-June-2022], 2022.

[73] *Google safe browsing — google developers*, https://developers.google.com/safe-browsing/, 2022.

[74] M. Kubat, S. Matwin, *et al.*, "Addressing the curse of imbalanced training sets: One-sided selection," in *Icml*, Citeseer, vol. 97, 1997, p. 179.

[75] H. Han, W.-Y. Wang, and B.-H. Mao, "LNCS 3644 - Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning," Tech. Rep., 2005, pp. 878–887.

[76] *Home — popcash*, https://popcash.net/, 2022.

[77] A. Tsymbal, "The problem of concept drift: Definitions and related work," *Computer Science Department, Trinity College Dublin*, vol. 106, no. 2, p. 58, 2004.

[78] Q. Au, J. Herbinger, C. Stachl, B. Bischl, and G. Casalicchio, "Grouped feature importance and combined features effect plot," *arXiv preprint arXiv:2104.11688*, 2021.

[79] *Brave/adblock-rust*, [Online; accessed 20-January-2022], 2021.

[80] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," *arXiv preprint arXiv:1806.01156*, 2018.

[81] *The trace event profiling tool (about:tracing)*, https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/, 2022.

[82] *Page load time - mdn web docs glossary: Definitions of web-related terms — mdn*, https://developer.mozilla.org/en-US/docs/Glossary/Page_load_time, 2022.

[83] "Ps(1) - linux manual page." (2022).

[84] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors," pp. 1143–1161, 2020.

[85] *Electron — npm trends*, https : / / npmtrends . com / electron, (Accessed on 08/30/2024).

[86] *How water labbu exploits electron-based applications*, https://www.trendmicro.com/en_za/research/22/j/how-water-labbu-exploits-electron-based-applications.html, (Accessed on 08/30/2024).

[87] *Cve - cve-2021-21220*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21220, (Accessed on 08/30/2024).

[88] *Slack — report #783877 - remote code execution in slack desktop apps + bonus — hackerone*, https://hackerone.com/reports/783877/, (Accessed on 09/01/2024).

[89] *Mksb(en): Discord desktop app rce*, https://mksben.l0.cm/2020/10/discord-desktop-rce.html, (Accessed on 08/30/2024).

[90] *Rce in mattermost desktop earlier than 4.2.0 - dev community*, https://dev.to/nlowe/rce-in-mattermost-desktop-earlier-than-420-5aef, (Accessed on 09/01/2024).

[91] *Oskarsve/ms-teams-rce*, https://github.com/oskarsve/ms-teams-rce/, (Accessed on 09/01/2024).

[92] *Cve-2021-28471 - security update guide - microsoft - remote development extension for visual studio code remote code execution vulnerability*, https : / / msrc . microsoft . com / update - guide / vulnerability / CVE - 2021 - 28471, (Accessed on 09/01/2024).

[93] *Execution with unnecessary privileges in arc-electron · ghsa-v3wr-67px-44xg · github advisory database*, https://github.com/advisories/GHSA-v3wr-67px-44xg, (Accessed on 09/01/2024).

[94] *Xss vulnarability in markdown mode*, https://github.com/Automattic/simplenote-electron/issues/487, (Accessed on 09/01/2024).

[95] C. Reis, A. Moshchuk, and N. Oskov, “Site isolation: Process separation for web sites within the browser,” in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[96] *Nvd - cve-2020-15096*, https://nvd.nist.gov/vuln/detail/CVE-2020-15096, (Accessed on 09/23/2023).

[97] *Nvd - cve-2020-15215*, https://nvd.nist.gov/vuln/detail/CVE-2020-15215, (Accessed on 09/23/2023).

[98] *Cve - cve-2020-15174*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-15174, (Accessed on 08/30/2024).

[99] *Security fix for arbitrary code execution - huntr.dev by huntr-helper · pull request #1 · mikeerickson/cd-messenger*, https://github.com/mikeerickson/cd-messenger/pull/1, (Accessed on 09/01/2024).

[100] D. Wagner and P. Soto, “Mimicry attacks on host-based intrusion detection systems,” in *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Oct. 2002, pp. 255–264.

[101] *Cve - cve-2022-29247*, https://nvd.nist.gov/vuln/detail/CVE-2022-29247, (Accessed on 08/30/2024).

[102] *Codeinjection*, https://codeql.github.com/codeql-standard-libraries/javascript/semmle/javascript/security/dataflow/CodeInjectionCustomizations.qll/module.CodeInjectionCustomizations$CodeInjection.html, (Accessed on 10/28/2024).

[103] R. Jahanshahi, B. A. Azad, N. Nikiforakis, and M. Egele, “Minimalist: Semi-automated debloating of {php} web applications through static analysis,” in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[104] B. A. Azad, P. Laperdrix, and N. Nikiforakis, “Less is more: Quantifying the security benefits of debloating web applications,” in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[105] *Cve-2021-43908 - security update guide - microsoft - visual studio code spoofing vulnerability*, https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-43908, (Accessed on 09/01/2024).

[106] *Joplin desktop app vulnerable to cross-site scripting · cve-2022-45598 · github advisory database*, https://github.com/advisories/GHSA-h6c2-879r-jffh, (Accessed on 09/01/2024).

[107] *Appium-desktop os command injection vulnerability · cve-2023-2479 · github advisory database*, https://github.com/advisories/GHSA-xq6j-x8pq-g3gr, (Accessed on 09/01/2024).

[108] *Cross site scripting vulnerability*, https://github.com/yoshuawuyts/vmd/issues/137, (Accessed on 09/01/2024).

[109] *Markdownify subject to remote code execution via malicious markdown file · cve-2022-41709 · github advisory database*, https://github.com/advisories/GHSA-c942-mfmp-p4fh, (Accessed on 09/01/2024).

[110] *Os command injection vulnerability found in poddycast*, https://huntr.dev/bounties/1624637557081-MrChuckomo/poddycast/, (Accessed on 09/01/2024).

[111] *Xss vulnerability · issue #23 · ohhaibrowser/browser*, https://github.com/OhHaiBrowser/Browser/issues/23, (Accessed on 09/01/2024).

[112] *Github advisory database*, https://github.com/advisories?query=type:reviewed+ecosystem:npm, (Accessed on 09/01/2024).

[113] *Javascript code coverage - v8*, https://v8.dev/blog/javascript-code-coverage, (Accessed on 09/01/2024).

[114] *Acornjs/acorn: A small, fast, javascript-based javascript parser*, https://github.com/acornjs/acorn, (Accessed on 08/29/2024), 2024.

[115] *Speedometer 2.0*, https://browserbench.org/Speedometer2.0/, (Accessed on 09/01/2024).

[116] *A03 injection - owasp top 10:2021*, https://owasp.org/Top10/A03_2021-Injection/, (Accessed on 09/01/2024).

[117] B. Amin Azad, R. Jahanshahi, C. Tsoukaladelis, M. Egele, and N. Nikiforakis, “AnimateDead: Debloating Web Applications Using Concolic Execution,” in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[118] E. Trickel *et al.*, “Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities,” in *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2023.

[119] B. Stock *et al.*, “From Facepalm to Brain Bender : Exploring Client-Side Cross-Site Scripting Categories and Subject Descriptors,” *Proceedings of the 22nd ACM*

SIGSAC Conference on Computer and Communications Security, pp. 1419–1430, 2015.

[120] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild.," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[121] F. Xiao *et al.*, "Abusing hidden properties to attack the node. js ecosystem," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Conference, Aug. 2021.

[122] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining node.js vulnerabilities via object dependence graph and query," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[123] *Owasp top ten 2017*, https://owasp.org/www-project-top-ten/2017/Top_10.html.

[124] OWASP, *Owasp top ten 2021*, https://owasp.org/Top10/, 2021.

[125] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. Staicu, "Secbench. js: An executable security benchmark suite for server-side javascript," in *International Conference on Software Engineering (ICSE)*, 2023.

[126] *Nvd - cve-2024-4367*, https://nvd.nist.gov/vuln/detail/cve-2024-4367, 2024.

[127] H. Hu *et al.*, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[128] S. Forrest, S. Hofmeyr, and A. Somayaji, "The evolution of system-call monitoring," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2008.

[129] *Evernote 2023 recap*, [Accessed 2025-02-14], 2024.

[130] *Pdf.js - home*, [Accessed 2025-02-14].

[131] *Find and fix problems in your javascript code - eslint - pluggable javascript linter*, [Accessed 2025-02-14].

[132] *Express - node.js web application framework*, [Accessed 2025-02-28].

[133] Microsoft, *Visual studio code - code editing. redefined*, [Accessed 2025-02-28], Nov. 2021.

[134] *Jetstream 2.2*, [Accessed 2025-02-28].

[135] *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Conference, Aug. 2021.

[136] *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015.

[137] *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[138] *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2023.

[139] *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[140] *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[141] *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.