Understanding and Mitigating Remote Code Execution Vulnerabilities in Cross-platform Ecosystem

Feng Xiao
Georgia Institute of Technology
USA

Zheng Yang
Georgia Institute of Technology
USA

Joey Allen
Georgia Institute of Technology
USA

Guangliang Yang
Fudan University
China

Grant Williams
Georgia Institute of Technology
USA

Wenke Lee
Georgia Institute of Technology
USA

ABSTRACT

JavaScript cross-platform frameworks are becoming increasingly popular. They help developers easily and conveniently build cross-platform applications while just needing only one JavaScript codebase. Recent security reports showed several high-profile cross-platform applications (e.g., Slack, Microsoft Teams, and Github Atom) suffered injection issues, which were often introduced by Cross-site Scripting (XSS) or embedded untrusted remote content like ads. These injections open security holes for remote web attackers, and cause serious security risks, such as allowing injected malicious code to run arbitrary local executables in victim devices (referred to as *"Xrce"* attacks). However, until now, Xrce vectors and behaviors and the root cause of Xrce were rarely studied and understood. Although the cross-platform framework developers and community responded quickly by offering multiple security features and suggestions, these mitigations were empirically proposed with unknown effectiveness.

In this paper, we conduct the first systematic study of the Xrce vulnerability class in the cross-platform ecosystem. We first build a generic model for different cross-platform applications to reduce their semantic and behavioral gaps. We use this model to (1) study Xrce by comprehensively defining its attack scenarios, surfaces, and behaviors, (2) investigate and study the state-of-the-art defenses, and verify their weakness against Xrce attacks. Our study on 640 real-world cross-platform applications shows, despite the availability of existing defenses, Xrce widely affects the cross-platform ecosystem. 75% of applications may be impacted by Xrce, including Microsoft Teams. (3) Finally, we propose XGuard, a novel defense technology to automatically mitigate all Xrce variants derived from our concluded Xrce behaviors.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

Exploit mitigation; Static analysis; Runtime Provenance Analysis

ACM Reference Format:
Feng Xiao, Zheng Yang, Joey Allen, Guangliang Yang, Grant Williams, and Wenke Lee. 2022. Understanding and Mitigating Remote Code Execution Vulnerabilities in Cross-platform Ecosystem. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3559340

1 INTRODUCTION

Cross-platform development frameworks are becoming increasingly popular. They greatly help developers easily build applications for multiple platforms (e.g., Linux and Windows) while needing only one codebase. With the wide community support and the prevalence of Node.js, JavaScript has become the main development language of the cross-platform ecosystem (i.e., developing cross-platform applications conveniently in JavaScript). JavaScript-based cross-platform frameworks (e.g., Electron [11], React Native [32], NW.js [28], and Neutralinojs [26]) show widespread use in the development of high profile cross-platform applications, such as Skype [39] and WhatsApp [51].

With the emerging popularity of JavaScript cross-platform frameworks and applications, their security attracts more and more attention. Recently, several popular cross-platform applications (e.g., Slack [40], Microsoft Teams [48], and Github Atom [16]) were discovered to be vulnerable to injection attacks, which were often introduced by Cross-Site Scripting (XSS) or embedded untrusted remote content such as ads. These injection vulnerabilities open security holes for remote web attackers (e.g., XSS attackers or untrusted content providers), allowing them to inject well-crafted malicious code into cross-platform applications. This causes serious security consequences, including running arbitrary local executables on the victim's device. Due to the novel cross-platform architecture, injection attacks in the cross-platform ecosystem differ significantly from traditional, well-studied remote code execution (RCE) problems (more discussion in §9). Hence, we refer to the RCE issues in cross-platform applications as Xrce attacks.

However, until now, there was little knowledge about Xrce's root cause, attack vectors, behaviors, and exploitation landscape. Although cross-platform framework developers responded quickly with multiple security features (e.g., Electron's [10] isolation of untrusted content from trusted code) and suggestions [15] to defend against Xrce attacks, these mitigation solutions and suggestions were empirically proposed mostly based on the security practices, observations, and findings learned from the regular web ecosystem. Their effectiveness is still unknown and unverified in the cross-platform environment.

Motivated by the above security concerns, we conduct the first systematic study of XRCE attacks and related defenses in the cross-platform ecosystem. In this work, we primarily focus on the remote code execution behaviors (rather than the malicious code injection process), as they introduce direct security impacts unique to the cross-platform ecosystem. In our study, we first intend to understand cross-platform applications' architectures and behaviors. However, we find they depend heavily on their corresponding cross-platform development frameworks. Different frameworks usually offer very different workflows and logic. Thus, we build a generic model for cross-platform applications to reduce their semantic and behavioral gaps. Then, we collect recently-reported security vulnerabilities (10 in total) related to cross-platform applications and frameworks. Using the above generic model, we analyze and understand the root causes and conclude three attack behaviors of XRCE (§2.4). Moreover, our study discovers previously-unknown attack surfaces unique to the cross-platform ecosystem (§2.3), uncovering more malicious code injection opportunities.

After understanding XRCE attacks, we carefully investigate and verify the completeness and correctness of related security mitigation solutions and suggestions. Consequently, we find existing defenses limited against XRCE. While existing defenses do cover certain XRCE exploits, they bring serious side effects: enabling certain security features requires extensive application changes since they blindly prevent both malicious and benign execution flows. More importantly, existing solutions miss an entire category of malicious execution flows. Due to such weaknesses, once an injection point exists in a cross-platform application, a remote attacker can easily bypass existing defenses to freely exploit XRCE.

Considering the weakness and limitation of existing defenses, we aim to evaluate the impact of XRCE attacks in the wild. For this purpose, we collect and analyze 640 real-world cross-platform applications. We find the aforementioned side effects lead to a surprisingly low defense adoption rate (i.e., 23%) in practice. Furthermore, we find the attack behavior left uncovered by existing defenses is in fact practical to exploit and impact many applications, including high-profile ones like Microsoft Teams. Finally, we report all findings to developers and are co-working with them. Many have confirmed our findings. So far, three CVEs have been issued.

Motivated by the insights and the weaknesses of existing defenses, we design and implement a novel XRCE defense system, XGUARD, to address XRCE problem from the root, i.e. distinguishing benign and malicious execution flows. To this end, XGUARD leverages static and dynamic program analysis techniques to perform fine-grained control- and data-flow validations against potentially risky flows. The validations follow a zero trust model to eliminate excessive implicit trust in existing cross-platform frameworks. XGUARD automatically verifies the execution flows by checking the sensitivity of the executed content and examining its legitimacy and trustworthiness. In this way, XGUARD can prevent XRCE by tracing and identifying suspicious operations, even when programming bugs or misconfigurations occur in the application.

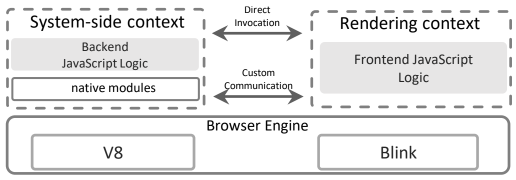


Figure 1: System diagram of a cross-platform application.

We implement the prototype by instrumenting the V8 JavaScript engine. We evaluate its effectiveness on a set of real-world cross-platform applications. The results show XGUARD successfully mitigates all 16 XRCE cases in our data set (including 13 real-world exploits and 3 synthetic attacks) with negligible false positives. Additionally, XGUARD introduces low application overhead. By evaluating XGUARD on 1.1 million LoC from real-world cross-platform applications, we find XGUARD can efficiently handle millions of flows with negligible overhead (e.g., less than 2% slow down) while effectively mitigating malicious flows from the exploitations of 16 XRCE vulnerabilities. To facilitate future research, we will release the source code of XGUARD at https://github.com/xiaofen9/XGuard.

To summarize, we make the following contributions:

- We conduct the first systematic study on XRCE attacks and their diverse attack behaviors. By investigating 640 real-world cross-platform applications, we find they are, in fact, severely impacted by XRCE, and existing defenses are limited against XRCE attacks in practice. Our findings also lead to the discoveries of novel XRCE attack surfaces and three new CVEs.
- We design and implement a novel XRCE defense system, named XGUARD, that enforces accurate and scalable fine-grained validations. XGUARD addresses two important difficulties: i) how to practically handle well-known analysis inaccuracies for dynamic languages like JavaScript to achieve fine-grained (i.e., control- and data-flow level) defenses; ii) how to enforce efficient defenses into highly complex language engines like V8.
- We comprehensively evaluate XGUARD against real-world applications (tested with a total of 1,125.55 kLOC) including Microsoft Teams and VSCode. We demonstrate that XGUARD can effectively mitigate XRCE with negligible overhead.

2 UNDERSTANDING XRCE ATTACKS

In this section, we first build a generic model for cross-platform applications (§2.1), effectively capturing a unique feature (i.e., the open runtime). Then we discuss XRCE attacks, scenarios, behaviors, and surfaces (§2.2).

2.1 Understanding Cross-Platform Ecosystem

As discussed in §1, we build a generic application model to reduce the semantic gap between different applications. As cross-platform application architectures and logic often rely on their corresponding development frameworks, we conduct our investigation on multiple popular cross-platform application frameworks, including Electron, NW.js, and Neutralinojs. Note that our investigation mainly targets desktop frameworks because desktop applications (on Windows or Linux) often accept diverse inputs (such as local files), which suggests they may suffer more from XRCE.

Consequently, we find current cross-platform development frameworks share one unique feature: they all run JavaScript in two interacting contexts (a chromium-based [19] and a Node.js-based [71] environment). This dual-context design originates from cross-platform applications' goal to render user interfaces like traditional browser environments while also performing system-level functionalities (e.g., accessing file system) like native applications.

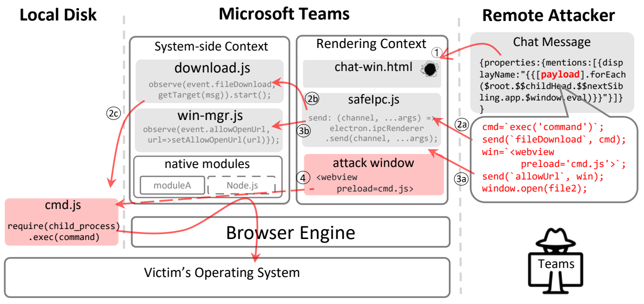
As depicted in Figure 1, in such a dual-context model, a cross-platform application often includes two separate sets of programming logic: rendering and system-side. The rendering logic is responsible for rendering user interfaces and handling user inputs; the system-side logic communicates with the rendering logic and performs low-level system operations (e.g., accessing file access and network I/O). The system-side context is privileged and manages several native modules, through which JavaScript code can access low-level system resources. While rendering and system-side contexts are separated, they can interact with each other in the following two channels: (i) Direct invocation channel: the system-side context can directly expose its internal methods (e.g., native module APIs) to the rendering context, and allow the rendering context to access the functionalities inside the system context. (ii) Custom communication channel: Another popular practice is for developers to implement their own communication methods, such as event-based inter-process communication (IPC), allowing the rendering context to send inter-context events to the system-side context and further trigger its internal functionalities.

While different frameworks have distinct implementations, sharing this dual-context model lets them deliver many intrinsically similar features. From the development perspective, different frameworks share many third-party libraries. For example, all three frameworks can use the popular front-end library React [34] in their the rendering context [14, 31, 33]. On the security side, the frameworks share similar defensive behaviors. For instance, both Electron and NW.js have a defense feature to control the cross-context access behavior (i.e., Electron's `node-integration` [12] vs NW.js's `node-frame` [29]). For more discussion see §3.2.1.

2.2 XRCE Attacks

To study XRCE, we search bug bounty programs [17, 18], CVE databases [7, 42] and related industry studies [45, 46]. After filtering out older reports (over three years old), we collected 10 cases containing a diverse set of attack behaviors. As summarized in Table 1, these XRCE vulnerabilities affect many high-profile applications (e.g., Microsoft Teams and Slack), and introduce serious security implications, such as arbitrary OS-level command execution.

Upon our generic application model, we perform an in-depth study on these vulnerabilities, including understanding diverse attack behaviors, reproducing serious attack consequences, and analyzing the root causes. Due to the intrinsic cross-framework similarities within the generic model, our study is applicable to different types of frameworks. We develop our study in the following way: First, we detail an example of real-world XRCE vulnerability. Second, we explore feasible attack scenarios. Finally, by studying real-world XRCE cases, we conclude three typical attack behaviors and discover two novel attack surfaces for XRCE.



```
observe(event.fileDownload,
  getTarget(msg)).start();
```

```
observe(event.allowOpenUrl,
  url=>setAllowOpenUrl(url)});
```

```
send: (channel, ...args) =>
    electron.ipcRenderer
    .send(channel, ...args);
```

```
<webview
    preload=cmd.js>
```

```
require(child_process)
  .exec(command)
```

```
{properties:{mentions:[{disp
layName:"{{[payload].forEach
($root.$$childHead.$$nextSib
ling.app.$window.eval)}}"}]}
}
```

```
cmd=`exec('command')`;
send(`fileDownload`, cmd);
win=`<webview
        preload='cmd.js'>`;
send(`allowUrl`, win);
window.open(file2);
```

Figure 2: A real-world XRCE exploit in Microsoft Teams.

2.2.1 Real-World XRCE Attack Example. In this section, we use a vulnerability [25] found in Microsoft Teams to demonstrate an end-to-end XRCE exploit. This example shows how insecure cross-context interactions can lead to RCE.

Microsoft Teams is a popular business communication tool with almost 145,000,000 daily active users (up to Nov. 2021). Figure 2 shows a simplified diagram of the Teams attack. First, a remote attacker installs Teams on his own device. Then, the attacker sends a well-crafted malicious chat message to the victim user. When the victim attempts to read new messages, the malicious message enters the rendering context. At this point, the message content is saved in the message pool and not allowed to execute. To successfully launch an XRCE attack, the attacker exploits an AngularJS expression injection flaw [2] in the rendering context. In this way, the payload (red text) embedded in the malicious message can be executed under the rendering context (as indicated by ①).

However, this is still not enough: to achieve RCE on the victim's machine, the attacker needs to perform cross-context interactions with sensitive methods in the privileged system-side context. In particular, this involves three steps. First, as the cross-context path (2a)-(2b) shows, the malicious code in the rendering context fires an IPC event 'fileDownload' in his payload (i.e., first `send()` in the red text dialog), which uses a cross-context communication channel to invoke a system-side method (i.e., `observe()` in `download.js` in Figure 2). As a result, the local disk downloads a remote malicious JavaScript file. The malicious rendering process code then aims to execute this freshly retrieved file. However, to access a local file, the attacker needs to open a new window, which is limited by the developers' security configurations. Thus, as indicated by (3a)-(3b), the malicious code issues another cross-context event 'allowUrl' to call a similar event-handling method (`observe()` in `win-mgr.js`) as in (2a)-(2b). By doing so, the attacker can manipulate the previously mentioned security configuration. Lastly, the malicious code opens the new window, where the attacker loads a bootstrapping script (`cmd.js`) to the system-side context (as indicated by ④). To achieve the final attack effect, the malicious bootstrapping script invokes the sensitive API `exec()` from the Node.js runtime to achieve arbitrary code execution in the victim's OS.

2.2.2 XRCE Definition. In this section, we formally define several important terms and describe their properties. We then define more technical aspects of XRCE.

Injection Point: An injection point is where adversaries inject malicious payloads into the cross-platform application and run them. Any application functionality in the rendering context may be abused to inject malicious code if it accepts/loads external input. Note that, while injection points are entries of XRCE, in this work finding injection points is not our research goal. Instead, we mainly focus on understanding XRCE attacks, vetting the related mitigation, and proposing a novel injection-agnostic defense.

Cross-context flow: Given a cross-platform application, if a function in one context interacts with the other context (e.g., accessing variables across contexts), it constitutes a cross-context flow. For a typical XRCE attack, the remote attacker needs to first inject malicious code into the rendering context (through an existing injection point) and then perform cross-context flows to reach the privileged system-side context (where the attacker can access privileged primitives like Node.js APIs).

Threat Model: In the XRCE attack scenario, we consider cross-platform applications to be benign, and attackers to be remote web attackers or third-party untrusted content providers. Attackers aim to exploit injection vulnerabilities in victim cross-platform applications and launch XRCE attacks. In particular, remote attackers inject malicious content into the exposed rendering context, run the malicious payload, launch malicious cross-context flows, and finally abuse the functionalities inside the privileged system context.

Thus, to successfully perform XRCE attacks, we assume the victim application contains at least one code-level injection point. We find this security assumption feasible and practical for the following reasons: (i) Injection vulnerabilities are prevalent in web code and ranked third in the 2021 OWASP top ten vulnerabilities list [1]. According to statistics from 2019 [43], XSS, one of the most widely used injection methods, had 3,490 cases (17% of all reported vulnerabilities) reported in a single year. (ii) Our study finds that the open runtime exposes novel attack surfaces, allowing previously unexploitable programming logic to be reached and abused by XRCE. Cross-platform applications usually load/accept diverse content from multiple channels, increasing the risk of exploitation.

2.3 Finding New Attack Surfaces

Since cross-platform applications render web scripts similarly to traditional web applications, it is naturally believed they share the same attack surface. For example, attackers will target a vulnerable front-end script (in the rendering domain) to inject and run malicious code (similar to a typical web exploit). However, our study reveals that cross-platform applications actually expose a larger attack surface, with an entirely new and unique extended portion.

New Attack Surface #1: Scripts in local file system. To find injection points, the known method is to search for potentially exploitable vulnerabilities within the (front-end) scripts running in the rendering context. However, we find that the open runtime unlocks new possibilities for injection point discovery. Concretely, we find that any scripts in the local file system can be targeted and attacked. For example, the attacker may target a random script in the file system (e.g., `/usr/lib/en.html`) and use an `iframe` to invoke it in the rendering context. Typically, if a script satisfies the following two conditions, it becomes a tempting target for XRCE: (i) it accepts external input (e.g., via query string); (ii) the external input is evaluated. Such scripts are not hard to find and there are many real-world use cases [3] for such scripts. Note the attacker leverages local scripts rather than malicious remote scripts because remote scripts cannot access the parent frame to create cross-context flows due to SOP restrictions (More technical details can be found in [3]).

Table 1: XRCE vulnerabilities in high-profile software. D means direct invocation abuse, B means browser state poisoning, and C means communication abuse

#ID	Product Name	Behavior			Attack Effects	Reference
		D	B	C		
1	MS Teams			✔	Execute Command	[25]
2	WhatsApp	✔			Read File	[52]
3	Slack	✔	✔		Execute Command	[41]
4	Discord		✔	✔	Execute Command	[9]
5	Wordpress Desktop	✔			Execute Command	[53]
6	Simplenote	✔			Execute Command	[38]
7	Rocket.chat[1]	✔			Execute Command	[36]
8	Rocket.chat[2]		✔	✔	Execute Command	[35]
9	Github Atom	✔			Execute Command	[3]
10	Mattermost	✔			Execute Command	[37]

New Attack Surface #2: Bundled script in other cross-platform apps. We find it also feasible to abuse bundled scripts in other cross-platform applications so long as the target application recognizes the archive format. For example, all Electron applications archive their files using a compression format called `asar`. As a result, an attacker located in one app (e.g., Slack) can leverage scripts from another app (e.g., VSCode) to inject code (back into Slack). We believe this is a very practical attack surface. Due to the prevalence of cross-platform applications, there is a high chance the victim uses multiple Electron applications and these cross-platform applications are usually installed in default paths known to attackers.

The extended attack surface makes some attack scenarios more feasible and practical. For example, to attack with the social engineering approach (i.e., distributing malicious frames via malicious ad campaigns), the attacker needs to inject malicious code into vulnerable scripts in the same origin as the victim application. With this new surface, the attacker no longer needs to limit their injection point search to the victim application itself. Instead, they may utilize out-of-app scripts to run malicious JavaScript.

More importantly, the new attack surface is harder to mitigate. Controlling the code quality of other applications on users' machines is beyond a developer's ability. Even though a developer may carefully audit their application and deploy multiple defenses, an XRCE attacker can still achieve code injection using another low-quality code base on the victim's operating system.

2.4 Exploring Diverse Attack Behaviors

XRCE requires initializing malicious cross-context flows. To fully explore XRCE, we must capture and model all potential attack behaviors that may launch malicious cross-context flows. To this end, we analyze the vulnerabilities shown in Table 1 and group existing exploits into three different attack behaviors.

Direct Invocation Channel Abusing. In this attack behavior, the malicious rendering context code launches cross-context flows by directly leveraging exposed native system-side modules. Generally, these exposed modules are divided into two categories. (i) Node.js modules: These contain many sensitive APIs (e.g., `exec()`) useful for malicious code to access and manipulate system-level resources. (ii) Cross-platform framework-specific libraries: Electron packs its built-in APIs into modules, and many of them can be abused to introduce serious attack effects. For example, `openExternal()`, from the built-in `shell` module, can execute arbitrary binaries.

Browser State Poisoning. Because the system-side and rendering contexts execute in the same browser engine (as shown in Figure 1), they share some global states such as JavaScript prototypes. The attacker can control certain system-side behaviors by modifying global variables used by the system-side context and launching a cross-context flow. For example, in Rocket.Chat, the system-side context uses RegExp.test() to perform security checks. The malicious code located in the rendering context can modify `RegExp.prototype.test()` to change the logic of the corresponding security checks, and perform XRCE attacks. Note browser state poisoning is not limited to the above-described prototype pollution. Technically speaking, browser state poisoning includes any malicious modifications to global states in the browser runtime.

Custom Communication Channel Abusing. Similar to the first attack behavior, this category abuses an existing inter-context (custom communication) channel. Since these channels often make heavy use of native APIs to serve cross-context flows, XRCE attackers find them useful in conducting attacks. For example, to implement a custom communication channel in Electron, the system-side context defines and exposes a set of APIs to the rendering context via an Electron-specified feature named preload [13]. These self-defined APIs are usually wrapper methods of sensitive native APIs (e.g., IPC sender methods) from native modules. For example, BoostNote (a popular online notebook) leverages preload to expose a wrapper method of the dangerous API `openExternal()` to the rendering context, meaning any untrusted code in the rendering context can run arbitrary binaries with the exposed wrapper. Note that this vulnerability is previously unknown. We made a responsible disclosure to the vendor and were assigned a corresponding CVE (we will discuss more new vulnerabilities in §3.3).

3 VETTING THE STATE-OF-THE-ART DEFENSES

3.1 Overview

Considering XRCE's serious security consequences, the cross-platform development community, especially Electron, responded quickly and proposed several mitigation features and suggestions [15]. In addition, the open web runtime inherits security solutions already deployed in the traditional web environment, such as CSP (Content Security Policy). However, all of these defenses are empirically proposed based on past traditional web environment security observations. Their effectiveness is unclear and unverified in the cross-platform ecosystem.

Thus, in §3.2 we conduct a broad scope investigation on the state-of-the-art defenses, targeting different framework-specific security features and traditional defenses inherited from browser security. We find existing defenses are limited against diverse XRCE attacks from both theoretical and practical aspects:

- By design, existing defense infrastructures are neither complete nor practical. They either leave certain attack behaviors uncovered or introduce serious side effects (§3.2).
- For real-world applications, existing solutions are poorly deployed. What is worse, vulnerable XRCE patterns are very common (§3.3).

Considering the weaknesses of existing defenses, we further perform an in-depth assessment of XRCE's security impact in 640 real-world cross-platform applications. We find over 75% are potentially exploitable (once injection points exist), and identify real-world evidence to support the two points listed above. More specifically, we find cross-platform application developers encounter difficulties deploying security features in practice due to the side effects (§3.3.1). It is also common for developers to use insecure (unprotected) custom communication channels, causing previously-unknown vulnerabilities (§3.3.2). Below we present the details of our analysis.

3.2 Weaknesses of Existing Defenses

In this section, we discuss the design of existing XRCE security features and their shortcomings.

3.2.1 Built-in defenses. As XRCE became more prevalent, cross-platform framework vendors evolved quickly and proposed several defenses and recommendations. For example, Electron provided 16 general suggestions (along with two major security features) [15] detailing the security practices developers should follow, while NW.js also designed the `node-frame` feature to mitigate potential XRCE attacks. However, we find several problems with these security recommendations and features: First, the guidelines provided by these security suggestions are vague and hard to follow. For instance, Electron suggests developers "limit" the usage of navigation (Suggestions 13). This offloading of security responsibility to developers is a bad idea, as developers face difficulties distinguishing what should be limited from their various code logic.

Second, some of the suggestions are not realistic. For example, Electron states developers should always keep their Electron runtime up-to-date to avoid potential security issues. However, our study finds this very difficult to follow, since many compatibility issues exist between different Electron versions. Simply updating the Electron runtime in a cross-platform application often crashes the existing logic. We find even high-profile vendors like Microsoft hardly follow this principle in their products, e.g., Microsoft Teams.

Third, the recommended security features are poorly deployed in practice. Below, we perform an in-depth analysis of the two most important security features and discuss their weaknesses.

Node Integration. This option (called `node-integration` in Electron and `node-frame` in NW.js respectively) has the potential of mitigating the first type of attack behaviors (i.e., direct invocation channel abusing). Essentially, with the `node-integration` feature disabled, the rendering context can no longer directly access the system-side context. One exception is that Electron's `preload` [13] feature allows the rendering context to access a pre-defined set of APIs in the system-side context. See more discussion in §3.3.2.

Context Isolation. This security feature (`context-isolation`) is designed (by Electron) to further isolate the rendering and the system-side contexts. Essentially, it handles the second attack behavior, i.e. browser state poisoning. In particular, when `context-isolation` is enabled, it separates global states (e.g., global variables and prototypes) shared between the systems-side and rendering contexts.

Weaknesses. Although these features seem promising to eliminate the first two attack behaviors, they also introduce serious side effects, leading to low deployment in practice. We find these features block not only attack behaviors but also legitimate behaviors: they hardly distinguish between benign and malicious cross-context flows. In particular, we find the direct invocation channel is frequently used in benign code, such as in Slack. Thus, developers must choose to either refuse/disable these defense features or accept them but expend efforts to re-implement the impacted legitimate functionalities in other ways. In our study, we find many developers even intentionally avoid using existing security protections to let their applications run normally, resulting in the unsuccessful mitigation against the first two behaviors.

More importantly, existing defenses leave the widely used custom communication channel totally unprotected. Since this type of channel is highly customizable and different developers may use distinct implementations, frameworks face difficulties in understanding their semantics and enforcing generic protection.

3.2.2 Other related defenses. Besides built-in solutions, some defenses inherited from the traditional web scenarios may also handle Xrce. In particular, since XSS is a major injection point of Xrce, XSS mitigation solutions are one of the most-related existing defenses, but still lack the ability to prevent Xrce. For example, Content Security Policy (CSP) primarily mitigates XSS. However, existing work shows it is hard to deploy CSP correctly [74] and CSP can be bypassed unexpectedly. More importantly, existing XSS defenses [59, 78, 79] cannot fundamentally mitigate Xrce by differentiating malicious cross-context flows from benign flows.

3.3 Assessment of Xrce impact

To understand how the shift of execution environment impacts existing defenses, we collect 640 open-source real-world applications and perform two experiments. In the first experiment (§3.3.1), we focus on the impacts of the first two attack behaviors, (i.e., whether the cross-platform ecosystem properly deploys corresponding defenses to mitigate risk). In the second experiment, we study the third attack behavior (custom communication channel abusing). Since no existing defenses cover this behavior, we perform a bug-hunting style searching to determine how widely this behavior can impact the current ecosystem in §3.3.2.

3.3.1 Finding #1: The first two attack behaviors are not properly mitigated. As previously mentioned, Electron proposed built-in features to mitigate the first two attack behaviors (i.e., direct invocation channel abusing and browser state poisoning). However, considering the serious side effects (i.e., blockage of benign uses), we wonder if developers are using these features.

To study this, we exhaustively collect 640 Electron applications and analyze how developers use built-in defenses. Specifically, we first use an existing syntax analysis [45] to extract configuration semantics from each application. We then measure how many applications run without fully deployed built-in defense features. Surprisingly, we find poor deployment of built-in defenses. Table 2 shows our results. From the first two rows of the table, we see that 384 (60%) and 491 (76.7%) of applications do not deploy the two major built-in defenses (`node-integration` and `context-isolation`).

Table 2: The defense deployment measurement result.

Type	Reported cases
Unprotected direct invocation channel	384
Unprotected browser states	491
Unsafe custom communication	27

To understand why Electron developers fail to follow security practices, we randomly select 30 applications from the reported cases and study their application logic. We find two potential explanations:

(I) Application usage scenarios: We find that application usage purposes largely affect developers' security configuration decisions (in a negative direction). The more OS-related operations (possibly requiring more cross-context flows) an application needs, the more likely developers give up using the defenses. In fact, over half the studied applications (16/30) involve intense OS-related operations.

(II) Lack of attention to security-related features: Our investigation finds 15 of 30 applications do not explicitly specify a valid configuration value for `context-isolation`, letting the application run in the default unprotected state (this feature is disabled by default in earlier version Electron runtime). This implies some developers do not pay enough attention to security configurations.

3.3.2 Finding #2: The third attack behavior is prevalent with no corresponding defenses. No existing solutions address the third attack behavior (custom communication), to the best of our knowledge. This naturally raises two more concerns: (i) how many unsafe communication patterns exist; (ii) how prevalent they are in the wild.

To answer the first question, we studied a set of applications by searching for unsafe custom communication implementations according to the following criteria: a safe communication interface should validate and restrict the security of the communication data. Since the receiver side (in the system-side context) of the channel often invokes sensitive native APIs, an unrestricted communication method may cause serious attack effects. According to the restriction levels we observed in Electron developers' communication implementations, we identified three types of risky patterns. As shown in Table 3, in the first pattern, developers leverage Electron's `preload` feature to expose a particular native API's reference such as an IPC sender method to the rendering context (achieving the same effect as the direct invocation channel, but with a customized set of APIs). Attackers can easily abuse such direct exposure since it includes no restriction on such communication. The second pattern uses a wrapper method to encapsulate the native API before it is "preloaded" to the rendering context as the communication method. However, we find many wrapper methods simply "forward" their arguments to the encapsulated API, meaning the rendering logic can still invoke the native API with any input data. As a result, it introduces the same risks as the first pattern. The third pattern also uses wrapper methods but does enforce certain (incomplete) restrictions on the communication data. For example, the system-side context may export a wrapper IPC method restricting the event type (e.g., `IPC.sender("HARDCODED-EVENT", arg)`). Since the attacker still controls the event content (e.g., `arg`) they may still abuse it to introduce attack effects.

To answer the second question, we measure the prevalence of the three patterns against our data set. To this end, we develop an in-house unsafe custom communication detection analysis using

Table 3: Three types of unsafe communication patterns.

Pattern Type	Examples	Reported
Raw API Object	Directly expose sensitive APIs such as IPC sender	17
Unrestricted wrapper method	Expose a communication wrapper without validating communication data	7
Partially restricted wrapper method	Expose a wrapper with partially manipulable communication data	3

the static JavaScript analysis framework 'WALA' [47]. More specifically, our analysis first encodes the three vulnerable patterns as sinks through an intraprocedural data-flow analysis. For instance, for the first pattern, the analysis pinpoints statements assigning sensitive native APIs to global variables. Then, the analysis scans the system-side logic to detect any native APIs exposed via unsafe patterns. More specifically, the analysis uses API objects imported from native modules as the source and then applies a static data-flow analysis to track the object propagation. Once the analysis finds an API object flows into any sinks, it reports its sink location.

The third row of Table 2 shows our analysis result. In total, our analysis reports 27 potential issues from the 640 applications (a further breakdown can be found in Table 3). By investigating the reported cases, we surprisingly find the most dangerous pattern (raw API object) to be the most common (17 out of 27). In particular, we notice the latest Microsoft Teams release still uses this pattern to implement their communication API, even though it has already been abused once [25]. We further inspect these cases to check if they can be exploited (assuming an injection point exists). We find 5 exploitable cases with serious attack effects (e.g., RCE). For example, in a widely-used online editor, the developer-defined IPC functionalities can be abused to download and execute arbitrary binaries. We have made responsible disclosures of the 5 discovered issues. At the time of writing, 3 out of 5 vendors confirmed the risk of the unsafe pattern, and 3 CVEs were assigned [4–6].

3.4 Key Insights

As discussed above, cross-context flows play a key role in launching XRCE attacks. While our findings confirm the weaknesses of existing defenses, we discover two important insights about cross-context flows. First, cross-context flows are shared by both malicious exploits and benign use cases. Second, a practical defense solution should not blindly break all flows like existing defenses (e.g., `node-integration`). These principles motivate us to design a fine-grained validation against malicious cross-context flows.

4 XGUARD

4.1 Design Goals

As seen in §3.2, cross-platform framework developers face many difficulties preventing XRCE attacks. We require our novel defense approach to meet the following criteria:

- **Fine-grained validation**: To defeat diverse XRCE behaviors, the new defense should dynamically monitor cross-context flows and perform fine-grained and precise validation.
- **High usability**: The new defense should be lightweight without affecting user experience. In other words, the protection overhead should be reasonable (if not negligible). We desire a low false-positive rate.
- **Transparent deployment**: To address the low adoption problem, the proposed system should be compatible with existing flows and the deployment process should not require developers to possess expert knowledge about the defense system. Additionally, the new system should not break existing defenses.

4.2 Design Overview

Following these design goals, we design and implement a novel generic defense solution, called XGUARD, to address the XRCE problem from the root, i.e., precisely distinguishing between benign and malicious execution flows. A key observation behind XGUARD's approach is that released JavaScript code bases of cross-platform applications are often trustworthy since they reflect developers' intentions. In particular, the original JavaScript code contains benign cross-context execution flows allowed by developers. Thus, when an external input enters the rendering context and performs different cross-context operations from the original cross-context flows, the unexpected operations are suspicious. In addition to starting new flows, the injected malicious code can abuse the original benign cross-context channels. Hence, XGUARD needs to conduct fine-grained analysis, deeply understand the whole picture of cross-context behaviors, and perform precise security enforcement.

To address these problems, we design XGUARD as a two-fold approach. In the first phase, given a cross-platform application, XGUARD analyzes its released JavaScript code to generate restrictive profiles for legitimate cross-context flows. One key challenge faced in this step is how to identify contained cross-context flows. To solve this, XGUARD applies static program analysis, suitable for learning all cross-context flows with its high coverage. Using static analysis, XGUARD collects necessary context information (e.g., call graphs), locates related APIs, and identifies their call sites as benign creators of cross-context flows.

In the second phase, when the target application runs, XGUARD performs real-time security enforcement. XGUARD dynamically monitors all cross-context flows, learns the real initiators of cross-context behaviors, and performs fine-grained validations. However, several challenges must be overcome when addressing this task. First, as discussed above, benign cross-context flows may be abused by untrusted code. Addressing this, XGUARD performs fine-grained integrity validation against each flow, by employing control- and data-flow tracking to collect necessary program state information and understand cross-context behaviors. However, this type of dynamic analysis is often heavy, introducing high overheads and bad user experiences. To avoid this, we design a lightweight analysis solution in the popular JavaScript engine V8.

Second, in some cases, when an unexpected (not detected in the first phase) cross-context flow arrives, we find we still cannot directly reject it. Static analysis is not always precise or perfect. The call graph may miss some edges, hiding some legitimate cross-context flows in the first phase. To mitigate this, we design a lightweight constrained execution environment to accommodate these unexpected cross-context flows.

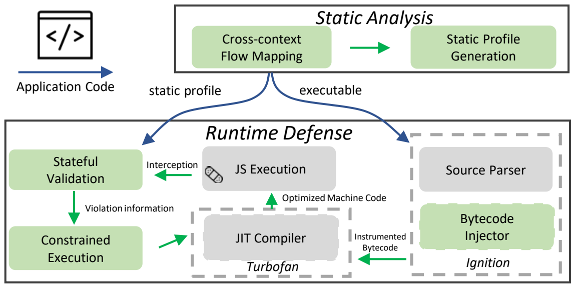


Figure 3: XGuard Overview

5 STATIC PROFILE GENERATION

In this section, we introduce the first phase of XGuard: applying static analysis on the target application to collect necessary information for runtime enforcement. First, we detail XGuard's approach of comprehensively mapping our defined cross-context flow to the actual application code. Then, we introduce how XGuard generates restrictive profiles to guard the identified (legitimate) flows from abuse. In particular, we discuss how XGuard handles inaccuracies in JavaScript program analysis and achieve the goal of extracting sufficient information needed for runtime enforcement.

Mapping cross-context flows to application logic. We use static analysis on the released code to pinpoint benign cross-context flows launching from the rendering context. We observe certain sensitive APIs provided by either Node.js modules or Electron modules often initialize cross-context flows. For example, to send an IPC message across the context border, the rendering context code needs to invoke an API called `ipcRenderer.send()`. From our observation, we first find all native modules available in Electron and Node.js, and add their sensitive APIs (e.g., IPC sender methods) to our list (called the target API list for the remainder of this paper). Currently, the list contains 42 items: 36 Node.js APIs like `child_process.exec()`, 4 Electron framework-specific APIs (e.g., `ipcRenderer.send()`, and 2 native JavaScript APIs. While the target API list may not be complete, we can easily expand it over time.

Locating Mapped cross-context flows in the rendering context. With the target API list, XGuard searches the rendering context code for all call sites that invoke sensitive APIs. To accurately locate all call sites, XGuard builds a comprehensive searching method integrating two different static analysis techniques: lightweight lexical-based and fine-grained taint-based call site searching. The former parses the application code to get its Abstract Syntax Tree (AST). By traversing the AST, XGuard identifies the call site with the same lexical function name as in the target API list. This method is powered by an observation that lexical names usually stay constant between the native modules and the rendering context (i.e., developers usually do not change the name of native API when invoking them from the rendering context). The second technique performs a def-use analysis to check if any API on the target list is imported from the native modules. If so, it marks the imported variable as the source and performs static taint tracking. Once the program calls the tainted method, the analysis reports an invocation site. After identifying all call sites, XGuard uniquely identifies them as a call site set <`F, LOC`> where `F` is the API name and `LOC` is the code location information including file name and line number.

Generating Restrictive Profile. After recognizing all cross-context flow call sites, a straightforward defense enforces a whitelist-based profile (similar to AppArmor [24]), only allowing the identified call sites to launch cross-context flows. However, this will not comprehensively mitigate Xrce because adversaries can still invoke and abuse existing call sites (if the target API is encapsulated in a function). To prevent such reuse attacks, an ideal approach performs end-to-end control-flow verification (i.e., validating the entire call chain of an existing call site). Nevertheless, generating restrictive profiles for such end-to-end verification is challenging: Although static analysis's completeness makes it a good candidate for profile generation, JavaScript dynamic features (e.g., dynamic property access) make it hard to extract accurate control-flow information (especially to identify the entire call chain). For XGuard, such inaccuracies will lead to false positives.

Hence, instead of solely relying on end-to-end control-flow verification, we design a novel multi-aspect (control- and data-flow) integrity enforcement to address the above limitation. More specifically, we use the observation that a real Xrce exploit not only changes control flow but also alters data flow (i.e., parameters of a sensitive API). Hence, when the control-flow integrity cannot be fully verified (due to JavaScript static analysis inaccuracies), XGuard launches a data-flow analysis to reason and verify the "provenance" (often a function) that creates the input parameters to the sensitive API. In this way, our system effectively reduces the assumption of the restrictive profile's accuracy. The multi-aspect enforcement works as follows: For profile generation, XGuard conservatively performs a static analysis to extract call chain information for each call site set <`F, LOC`>. To represent the legitimate invocation relationship between the sensitive call site and its callers (and the callers' caller), XGuard uses an adjacency list-like data structure where each vertex stores location information in a set structure similar to the call site set. The minimal static profile, in the runtime enforcement phase, will be combined with an accurate but lightweight onsite data-flow analysis (see §6 for more details) to achieve the desired control- and data-flow integrity.

6 RUNTIME ENFORCEMENT

After assembling the restrictive profiles, XGuard performs multi-aspect dynamic runtime security enforcement at the control- and data-flow level on all cross-context flows (as discussed in §5). In particular, when facing a cross-context execution flow, XGuard first conducts provenance analysis to understand its whole behavior, especially about who initiates the flow. To do so, XGuard designs novel lightweight control- and data-flow analyses. Then, it uses this control- and data-flow information combined with the first phase's static analysis results, to perform stateful security validation and enforcement in a resilient execution environment.

6.1 Runtime Provenance Analysis

6.1.1 Intercepting Cross-Context Flows. Before enforcing the restrictive profile on a cross-context flow at runtime, XGuard needs to first pause the application execution and intercept the cross-context flows from the rendering context (including scripts with the `preload` option). To achieve this, we customize V8 by injecting a hook at the beginning of related sensitive API calls. In particular, V8 includes two phases: code translation (i.e., converting JavaScript to low-level bytecode) and bytecode execution. We find in the translation phase, the V8 method `BytecodeGenerator::GenerateBytecodeBody` translates a function body's code, which is a good place for doing instrumentation. Thus, XGuard customizes this method to achieve the goal of identifying the target sensitive APIs and injecting runtime hooks on their entries.

6.1.2 Collecting Runtime Information. With the instrumentation hooks in place, XGuard collects cross-context flow runtime information. It uses this information later to verify the integrity of flows. First, to collect control-flow information, XGuard leverages V8's `v8::internal::JavaScriptFrameIterator` to traverse the call stack and learn the call path. XGuard also handles potential incomplete call stacks: For async calls, XGuard collects call chains from `v8_inspector::m_asyncTaskStacks`, which is maintained by V8 to map async calls with the call sites that create them. Moreover, V8's "current microtask" [20] allows us to track common async patterns (e.g., promise chains) with negligible overhead (i.e., zero-cost async stack traces [50]). For events, XGuard enforces validation on both the event listener and event sender side to ensure neither malicious event listeners nor malicious events can be injected. Taking Electron's event-based IPC communication as an example, XGuard intercepts any invocation to `IPC.sender()` and backwardly trace runtime information from it.

As mentioned in §5, XGuard performs a lightweight data-flow analysis if the control-flow integrity cannot be fully verified. This data-flow analysis determines the 'provenance' (often a function) that creates the input parameters to the related sensitive APIs. The provenance analysis helps XGuard verify the data-flow integrity. However, this is not a trivial task. Data-flow tracking is widely used in securing applications, but we find existing techniques [56, 68] heavyweight and unsuitable for a real-time defense against Xrce attacks. Thus, we need to design and develop a new lightweight tracking technique, requiring us to make trade-offs between low overhead and high precision.

XGuard introspects the context and scope of each function on the call stack, as variables' ownership information (i.e., who creates the variable) can be inferred from them. Thus, XGuard backwardly traverses the call stack provided by the control-flow analysis, retrieves the scope for each function, and further checks if this scope includes a reference to the input parameters of the sensitive API being called. More specifically, for obtaining context and scope, we find V8 creates a "context stack" in parallel to the call stack when it calls a JavaScript function [49]. This context stack maintains a `v8::internal::Context` object for each function on the call stack. The Context object includes a reference to a `v8::internal::ScopeInfo` object, which maintains information about the scope of each function. So XGuard can walk this stack and call `ScopeIterator::VisitScope` to introspect the scope for each function on the call stack. Note XGuard chooses to backtrack literal-type variables such as strings and integers rather than all data types (e.g., nested objects) since it is enough to mitigate Xrce (all methods on our target API list accept literal values as parameters).

Algorithm 1 State Validation Algorithm

```
Require:
    $P$ = the profile generated in the static analysis
    $m$ = the running program instance
 1: FSM ← buildFSM($P$)
 2: $\varphi$ ← getExecFlow($m$)
 3: if validCallsite(FSM, $\varphi$) = False then
 4:     exec($m$, NONE)
 5: end if
 6: while endOfCall(FSM, $\varphi$) = False do
 7:     $\varphi_{callee}$ ← $\varphi$
 8:     $\varphi$ ← getCaller($\varphi$)
 9:     if validTran(FSM, $\varphi_{callee}$, $\varphi$) = False then
10:         if validDataFlow(FSM, $\varphi$) = True then
11:             exec($m$, FULL)
12:         else
13:             exec($m$, SANDBOX)
14:         end if
15:     end if
16: end while
17: exec($m$, FULL)
```

6.2 Conducting Security Enforcement

In this section, we show how XGuard performs control- and data-flow integrity validation based on the static profiles.

6.2.1 Stateful Validation. To validate the integrity, XGuard uses a stateful checking mechanism. Specifically, when the application starts, XGuard parses the static calling profile of each cross-context flow (i.e., call chains ending at the identified call sites) into a finite-state machine (FSM). In the FSM, states represent function definitions and transitions define legitimate calling relationships between them. With the FSM, XGuard can efficiently perform comparisons in cases where multiple potential paths to a target API method exist or the static analysis cannot build the full call path (e.g., loops).

Algorithm 1 summarizes how the validation works. The algorithm takes as inputs: the calling profile `P` and the running program `m`. The algorithm first builds an FSM according to `P`, and checks if the FSM contains an entry state for the current intercepted flow. If it finds a valid entry, XGuard performs a control-flow analysis as described in §6.1.2 to get dynamic call chain information. With the collected control-flow information, XGuard traverses the constructed FSM and ends when either it finds a control-flow state inconsistency or visits the entire FSM. For state inconsistency cases (line 10), XGuard launches a data-flow analysis to verify whether existing (legitimate) nodes in the profile created the parameters of the current sensitive API. Note that this design allows XGuard to reduce the frequency of the relatively slow data-flow checks. This optimization does not impact the defense effectiveness because XGuard still ensures either no ability for reuse attacks (the full call chain is validated) or attackers cannot introduce meaningful attack effects (only legitimate callers can affect sensitive API parameters).

Once verification ends, XGuard dispatches the intercepted execution to the corresponding execution modes. XGuard has three separate modes: full execution, sandboxed execution, and no execution. In the next section, we introduce our constrained execution environment and how XGuard runs different cross-context flows with different state mismatches.

6.2.2 Violation Handling with Constrained Execution. As shown in Algorithm 1, XGuard's stateful validation algorithm executes cross-context flows of different states mismatched with distinct modes such as `NONE`(line 4), `FULL` (line 11) and `SANDBOX` (line 13). We achieve this with our novel constrained execution environment. The rationale behind such an environment is that we design XGuard for high usability and fault tolerance. In other words, XGuard wants to minimize the impact of false alarms introduced by pragmatic limitations (e.g., inaccurate static analysis) and tries its best to securely execute as many cross-context flows as possible. With this goal in mind, XGuard takes the following violation handling strategy: For cross-context flows with call paths fully consistent with the static profile (line 17) or with data created by trusted parameter-creators (line 11), XGuard dispatches them to the full execution mode, allowing all methods. For suspicious flows, i.e., flows partially consistent with the static profile and containing data from previously-unknown callers (line 13), our engine executes them in the sandboxed mode, allowing only methods on a filtered target API list. Finally, if XGuard faces a completely new flow (not containing a recognized caller end-to-end), the engine directly denies its execution to prevent damage to the hosting system.

Below we discuss more implementation: The execution may fall into three categories. In the full execution mode, the executed code is flagged as benign and runs normally. In the no-execution mode, XGuard deems the code suspicious and thus directly blocks its execution. In the sandboxed execution mode, cross-context flows are allowed to launch, but their later execution is limited (as shown in Algorithm 1). For the design and implementation of the sandboxed mode, creating a 'completely-isolated' environment would perfectly defend against malicious flows. However, we find such an implementation solution overly heavy and resulting in high overhead. This is unsuitable for cross-context flow handling in a real-time defense system, which prefers low overhead and a good user experience. Instead, we design a lightweight solution by shrinking the system access capabilities of suspicious cross-context flows: We create a filtered API list based on our target API list and only allow methods in this list in sandboxed execution. For our prototype, we prepare the filtering list by selecting methods that cannot access the hosting OS so that we constrain all cross-context flows to the application. To prevent environment escape, modifying the mode status and the API list needs extra privileges, not allowed to application-level code. In our evaluation, we verify such a lightweight solution accommodates the cross-context flows not identified in the first phase (by static analysis).

7 EVALUATION

We evaluate XGuard's ability to mitigate Xrce on a set of popular Electron applications. Our series of experiments measure XGuard from both defense effectiveness and performance perspectives.

7.1 Dataset

To evaluate XGuard, we collect a set of real-world open-source Electron applications from the Electron App store (including the applications in Table 1). Since different applications require distinct set-up procedures (e.g., compilation and account registration) and some are even professional software (e.g., neural network visualizers) with learning curves, enumerating all functionalities of all 640 applications exceeds the potential manual effort. Hence, we choose to exhaustively exercise the application logic of a smaller set. In total, we perform our evaluation on 20 real-world applications and one trivial vulnerable application we constructed (to increase attack behavior coverage). By evaluating XGuard against 1.1 million LoC from real-world applications in 8 different categories like Productivity and Finance (categorized from the Electron official website [10]), our data set represents a wide selection of the ecosystem.

Table 4: Exploits mitigated by XGuard. Note we patch certain applications to reproduce the full exploits. The references for 9-13 are pending on the vulnerability confirmations.

Exploit#	Application Name	Reference	Behaviors	Blocked
1	MS Teams	HIGH-PROFILE[25]	C	✓
2	Slack	HIGH-PROFILE[41]	D + B	✓
3	Discord	HIGH-PROFILE[9]	B + C	✓
4	Simplenote	HIGH-PROFILE[38]	D	✓
5	Rocket.chat^1	HIGH-PROFILE[36]	D	✓
6	Rocket.chat^2	HIGH-PROFILE[35]	B + C	✓ w/CI
7	Github Atom	HIGH-PROFILE[3]	D	✓
8	Mattermost	HIGH-PROFILE[37]	D	✓
9	Zulip	RANDOM	D	✓
10	mini-diary	RANDOM	D	✓
11	freelook	RANDOM	D	✓
12	webtorrent	RANDOM	C	✓
13	eagluet	RANDOM	D	✓
14	Trivial^1	PATTERN 1 of Table 3	C	✓
15	Trivial^2	PATTERN 2 of Table 3	C	✓
16	Trivial^3	PATTERN 3 of Table 3	C	✓

7.2 Overview

To evaluate XGuard's ability to mitigate Xrce attacks, we design two experiments. First, we evaluate XGuard's effectiveness against real attacks. For this purpose, we prepare 16 Xrce exploits against 13 vulnerable applications (12 from the real-world data set and one self-built). The evaluation results show XGuard successfully protects all of them from Xrce and seamlessly integrates with existing defenses. Second, we study XGuard's usability when handling benign operations. To this end, we test the 21 applications (20 real-world plus our one trivial application) with exhaustive benign operations. We find XGuard performs precise validation, achieving a low false-positive rate and incurring only a few milliseconds of overhead. We discuss more details below.

7.3 Attack Mitigation

7.3.1 Xrce Exploit Mitigation. This experiment targets our 13 vulnerable applications. Among the 13, we include 7 high-profile applications from Table 1 and the other 5 vulnerable applications flagged in our empirical study (§3.3.1). Since the third attack behavior (custom communication abuse) is under-studied, to increase the exploit coverage, we built a trivial application to fully explore the three unsafe communication patterns in Table 3. To evaluate XGuard's defense capability, we collect and generate 16 Xrce exploits that launch cross-context flows with different attack behaviors.

Table 4 presents the mitigation results. From "Behaviors" we can observe our exploits contain diverse attack behaviors (D is direct invocation abuse; B is browser state poisoning, and C is custom communication abuse). By enabling XGuard on the corresponding vulnerable applications, we successfully mitigate all exploits. Note that for exploit #6, Electron's built-in context isolation feature integrates to handle the prototype pollution component of the exploit. This suggests XGuard is compatible with existing defenses.

7.3.2 Validation Behaviors. Besides the end-to-end mitigation experiment, we perform an in-depth analysis on XGuard's internal behaviors when performing benign operations, seeking to provide insights into how XGuard poses restrictions on cross-context flows.

Table 5: The breakdown of XGuard's flow validation behaviors. #FI: # of cross-context flow intercepted; #FC: # of flow checked; #CS: # of control information safe; #DS: # of data information safe; #FP: # of (deduplicated) false positives)

#ID	Application Name	Basic Information		Test Coverage	Defense Effectiveness				
		Version	LOC		#FI	#FC	#CS	#DS	#FP
1	MS Teams	1.4.00.4855	187,295	50.35%	61,970	2,969	2,936	33	0
2	Mattermost	4.6.2	41,546	87.92%	240,504	3,352	3,350	2	0
3	Simplenote	2.9.0	20,615	44.17%	371,231	508	507	1	0
4	Discord	0.0.14	10,932	58.78%	9,172,424	21,092	21,090	2	0
5	Slack	4.3.2	153 (minified)	44.85%	1,552,952	10,261	10,261	0	0
6	Rocket.Chat	2.17.9	16,303	100%	566,400	5,642	5,642	0	0
7	Atom	1.21.0	52,633	67.87%	4,727,829	56,830	56,763	67	0
8	VSCode	1.56.0	654,596	50.18%	1,890,600	13,379	11,333	634	1
9	browser-base	6.0.0	18,187	91.13%	166,475	1,170	999	171	0
10	webtorrent	0.24.0	19,718	100%	474,844	3,984	3,903	81	0
11	zulip	5.6.8	6,628	100%	367,276	4,484	4,381	103	0
12	devdocs	0.0	823	59.81%	89,475	1,714	1,714	0	0
13	netron	4.9.0	82,933	100%	188,411	2,959	2,955	4	0
14	clipper	1.0.4	628	100%	14,683	668	667	1	0
15	eagluet	0.1.3	385	96.22%	1,593	59	22	0	1
16	upcount	0.8.3	3,909	100%	47,963	546	0	0	1
17	exifcleaner	3.6.0	1,305	54.54%	1,448	44	44	0	0
18	mini-diary	3.3.0	5,908	69.08%	153,482	3,372	3,342	30	0
19	headset	3.3.3	626	79.54%	153,853	461	461	0	0
20	freelook	1.0.1	427	100%	206,154	265	255	10	0

"Defense Effectiveness" of Table 5 breaks down different internal checking aspects. First, the total number of cross-context flows (`#FI`) exceeds the number of flows entering the validation process (`#FC`), suggesting not all cross-context flows are validated. This is because trusted framework-specific internal logic (e.g., code loading) also creates flows. XGuard excludes them because they are not visible (and thus not exploitable) to the unsafe rendering context.

Another important insight is how effectively our static profile directs our validation logic. In Table 5, "`#CS`" presents the number of flows consistent with our static calling profile. On average, 92.2% of flows pass the call chain validation, meaning our static approach effectively describes the calling profile of real applications. For those flows (less than 8%) not fully represented by our static profile, XGuard validates their data-flow information. By comprehensively considering both dynamic control- and data-flow states, XGuard can protect a variety of applications. That includes apps as large as Discord and VSCode (600 kLOC+) that perform various types of cross-context access (e.g., I/O event), or apps as small as mini-diary (less than 6 kLOC) that perform only a few kinds of system access.

7.4 Usability Analysis

7.4.1 False Positive Analysis. Since XGuard performs fine-grained validations against all cross-context flows, we must evaluate if it is too restrictive and introduces false positives. To this end, we test XGuard with our 20 real-world application set in two ways. First, we create an automatic exploration tool to trigger as many cross-context operations as possible. This tool is designed to randomly generate different types of events (e.g., click and navigation).

Second, we find the automatic tool insufficient and limited by many operations (e.g., submitting forms with strict formats, account registration, and log-in). To mitigate this, our test also involves significant manual effort. We aim to enumerate every feature and functionality of each application to be tested, according to its documentation (if available). For each feature and functionality, we try different inputs (2-3 types) to trigger more branches. We are particularly interested in deep programming logic consisting of multiple steps, which our automatic tool may not well test.

Code Coverage. During testing, we measure the combined coverage ("Test Coverage" in Table 5) of the two exploration methods using V8's built-in code coverage statistic [23]. Note that we only count the rendering context for "Test Coverage" while our "LOC" counts both contexts. Hence, the two metrics are not directly comparable. Our testing method effectively exercises various application logic: It achieves over 60% dynamic coverage for most applications. Note that we exclude remote web frames dynamically loaded by `win.loadUrl(URL)` from coverage measurement and flow monitoring in the evaluation since we as non-developers do not have the information about the complete code base.

Even for large applications like VSCode, we achieve over 50% coverage. We cannot achieve higher coverage because some functionality requires certain (impractical to satisfy) conditions. For example, for debugging functionality, VSCode implements multiple versions of code to serve different OS platforms. Since we implement our prototype under Linux, we mainly focus on testing the Linux section. For future work, we are considering extending the implementation to other OSes.

False Positives. Table 5 shows false positives captured by XGuard ("`#FP`"). Notably, XGuard faces only three false-positive flows (after deduplication). After further investigation, we find the same cause for each. The underlying call graph used by the static profile fails to connect certain call paths. In VSCode's false positive, for example, the call path breaks at a very early stage (i.e., the second caller) during the backtrack. With such a partial call chain profile, our data-flow validation can not be effectively performed either. As a result, XGuard falsely dispatches the flow to the non-execution mode. While the results seem promising, we acknowledge XGuard is still subject to potential false negatives due to runtime provenance inaccuracies (please see §9 for more discussion).

7.4.2 Runtime Performance. The runtime overhead is another important evaluation metric of XGuard. Note that we perform our static analysis phase offline and the analysis time is reasonable (e.g., 5 min for 600+ KLoC VSCode). Therefore, the remainder of this section focuses on the runtime performance of XGuard's dynamic enforcement module. The runtime performance experiments are

Table 6: Runtime overhead for large-size (VSCode), small-size (Netron), and worst-case (Demo).

Tested Program	Page Load		In-app Execution		
	%	Time (ms)	Flow #	μs/flow	Overall (ms)
VSCode	11.15	8.40	1615	99.57	177.23
Netron	1.76	1.60	157	114.21	22.29
Demo	<1.00	<1.00	51	336.39	17.84

performed on a standard laptop with 8 Intel Core i7-8650U CPUs running at 1.90GHz and 16GB of memory.

To evaluate whether XGuard can enforce fine-grained profiles without affecting user experience, we perform two extensive experiments investigating XGuard's performance overhead. First, we measure the *page load* overhead induced by XGuard, where the page load time is defined as the time required for the app to load its initial resources (such as JavaScript code and pictures). The page load metric is important because previous studies show that a slow page load time can lead to frustrated users [58]. Second, we measure the overhead induced by XGuard when the user interacts with the application. This is important to determine if XGuard introduces any sluggishness when the application is used.

To evaluate the page load overhead induced by XGuard, we measure the time required to load applications of different complexities, with complexity determined by the size of the app's codebase. Specifically, we select VSCode (654 KLoC), Netron (83 KLoC), and our self-built demo application (203 LoC) (the demo application exhaustively invokes all methods in XGuard' target API list to emulate the worst-case scenario of intensively invoking cross-context flows) For each app, we conduct 10 trials, both with and without XGuard's instrumentation. The results of this experiment are displayed in the *Page Load* column of Table 6. The results show that, for the Demo and Netron, the overhead induced is negligible and less than 2%. For VSCode, we see that XGuard increases the page load slightly by 11.15%. After further investigation, we attribute VSCode's higher overhead to two main factors. First, the static profile generated for VSCode is extremely large and has 5.7x more entries than Netron's. Second, XGuard only loads the static profile at process creation. This optimization allows XGuard to avoid reloading the profile every time a page reloads. Unfortunately, VSCode's design does not allow it to utilize this optimization, since it creates a new rendering context each time a page loads. These two factors lead to VSCode's slightly larger runtime increase over the others. However, since the page load overhead for VSCode is still only 8.40ms, we argue it is low enough not to significantly impact the user's experience.

Next, we determine the impact XGuard's instrumentation has on the application's performance when a user interacts with the application. For this experiment, we develop a customized version of XGuard that tracks the number of flows invoked and the execution time of XGuard's instrumentation code. Next, a user intensively interacts with the application for five minutes. The results for this experiment are provided in the *In-app Execution* column of Table 6. We see that the average required monitoring time for each monitored flow is less than 337 microseconds. This shows the negligible overhead of XGuard's instrumentation.

8 RELATED WORK

Cross-platform App Security. Recently, the security community has paid more attention to cross-platform application security. For example, Luca Carettoni briefly discussed an empirical study of Electron security [45], and reported a security concern about Electron's `preload` feature [46]. Although these reports offered insights about Xrce, we still lacked knowledge about Xrce's root cause, attack vectors, behaviors, exploitation landscape, as well as unknown (unverified) effectiveness of existing defenses. To the best of our knowledge, this is the first academic work to perform a comprehensive study of the security of the cross-platform ecosystem, and propose a generic and practical defense approach to mitigate Xrce.

Server-side JavaScript Security. Server-side programs powered by JavaScript are widely used. Researchers have found many novel risks [30, 54, 57, 67, 69, 76, 77, 81, 82] in server-side JavaScript. In particular, remote code execution attacks pose a major threat. While Xrce belongs to this class of attacks, our work differs from existing research because existing work focuses on traditional web scenarios, while our work studies a new open runtime unique to the cross-platform ecosystem.

Client-side JavaScript Security. Client-side scenarios (like browsers) have used JavaScript for over 30 years, and significant security research has been conducted. For example, researchers leverage program analysis to identify and prevent client-side attacks such as malicious scripts [55, 60–62, 65, 70, 73, 75], Cross-Site Request Forgery (CSRF) [66], XSS [59, 78, 79], and browser extension bugs [63, 72]. These relate to our work since they may also be abused as Xrce injection points. However, their defenses are orthogonal to ours: existing client-side mitigation may remove some of Xrce's injection points, but XGuard is still needed to restrict cross-context flows.

9 DISCUSSION

Portability. Derived from our generic application model and comprehensive study results, XGuard's generic defense approach can be applied to different frameworks. Furthermore, our prototype is implemented mainly through instrumenting the JavaScript engine V8. Benefiting from this architecture, the instrumentation is highly independent of framework-level code. Thus, XGuard is extensive and portable to other frameworks (in addition to Electron).

Differences with traditional web RCE attacks. Xrce differs from traditional web RCEs in both the occurring context and the attack behavior: Traditional web RCEs [44] often occur in the back-end server logic (comparable to the system-side logic in cross-platform applications). However, due to the new dual-context model, Xrce involves both back-end and front-end logic (by creating malicious cross-context interactions). Moreover, Xrce exhibits more diverse attack behaviors: Traditional RCEs usually exploit data-flow-related attack behaviors against taint-style vulnerabilities [8]. Xrce involves not only data-flow (e.g., browser state poisoning) but also control-flow (e.g., direct invocation channel abusing) attack behaviors.

Static Analysis Accuracy. XGuard leverages static call graphs to construct restrictive profiles. Our current prototype uses a robust call graph algorithm [64] with good recall: 90%+ and 98%+

Table 7: List of Potential Injection Mediums (References with * indicate they are not end-to-end exploits)

Medium Type	Description	Example Scenario	Occurrences
Software Vulnerability	Cross-site Scripting	A communication app receives a malicious message exploiting an XSS in the rendering context scripts.	[3, 9, 25, 35, 37, 38, 41, 52, 53]
	Rich Text Rendering (w/ runtime-specific flaws)	A markdown editor uses a vulnerable Electron runtime, which allows undefined behaviors of certain rich text tags (e.g., execute JS).	[21, 22]
Social Engineering	Malicious Ads Campaign	A victim application hosts a malicious third-party script (Further attack steps require SOP bypass, which is feasible through our new attack surface or other memory corruption bugs).	[27]* §2.3*

for jQuery-based and framework-less programs respectively. The accuracy rates coincide with our experiment in §7.3.2: our static profile validates 92.2% (#CS/#FC) flows. While the static profile is not 100% accurate (due to dynamic JavaScript features like `eval()`), in future work we can improve XGuard's robustness by integrating more call graph algorithms (dynamic or static).

Dynamic Information Flow Analysis. As discussed in §4 and §7, XGuard is designated to incur low overhead but has the ability to effectively validate a huge number of cross-context flows in real-time. Thus, we design a lightweight data-flow tracking approach that makes trade-offs between runtime performance and tracking accuracy, which may introduce potential false negatives for certain exploits. Nonetheless, our evaluation (§7.3.1) shows that XGuard achieves desired balance between defense effectiveness and runtime overhead. In future work, we seek to extend XGuard with the support of more propagation behaviors, like string concatenation, to further improve its tracking capability, but still maintain its runtime performance.

Prototype-related exploits. A motivated attacker can combine prototype pollution exploits when launching Xrce attacks. For example, one may poison `String.prototype.replace()` to introduce malicious values to the arguments of a legitimate sensitive call site. Depending on the call site's location, XGuard's defenses can be divided into two situations: (i) For call sites in the system-side context, XGuard incorporates existing Electron built-in solutions (`context-isolation`) to mitigate the attack. (ii) For call sites in the rendering context, we acknowledge XGuard does not handle such cases, since existing solutions [80] already provide finer-grained context partition, which can address these prototype-related attacks. For future work, we can integrate existing solutions into XGuard for higher defense coverage.

Potential Injection Mediums. To launch Xrce, an attacker need to identify at least one injection point at the target application. For clarification, we conclude potential Xrce injection mediums in Table 7. For each medium, we gather the corresponding public exploits/vulnerabilities ("Occurrences") to indicate their prevalence. XSS vulnerabilities are the most common Xrce injection medium. Note that in-the-wild exploits are not found for social engineering-related mediums (since such malicious champions are more likely to be performed without public disclosure). However, we do find evidence of this medium's technical feasibility ([27] and §2.3).

10 CONCLUSION

This paper studies Xrce in the JavaScript cross-platform ecosystem and discovers several new findings, including new attack surfaces and previously unknown vulnerabilities. Our study suggests existing defenses cannot properly address Xrce, motivating us to propose XGuard, a novel automatic Xrce defense system. XGuard performs fine-grained control- and data-flow integrity validation while incurring low overhead. As the first work studying and preventing Xrce, we hope this paper can raise awareness about cross-platform application security.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their helpful and informative feedback. Guangliang Yang is the corresponding author. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-19-1-2179, N00014-17-1-2895, and N00014-18-1-2662 and the Defense Advanced Research Projects Agency (DARPA) under contracts HR00112090031 and HR00112190087. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR or DARPA.

REFERENCES

[1] *2021 OWASP Top 10 vulnerabilities.* https://owasp.org/Top10/.
[2] *AngularJs Expression Injection Bypass.* https://sites.google.com/site/bughunteruniversity/nonvuln/angularjs-expression-sandbox-bypass.
[3] *Atom Remote Code Execution.* https://statuscode.ch/2017/11/from-markdown-to-rce-in-atom.
[4] *CVE-2021-28119: twinkle-tray arbitrary code execution through unsafe IPC.* https://nvd.nist.gov/vuln/detail/CVE-2021-28119.
[5] *CVE-2021-28154: camunda-modeler arbitrary file access through unsafe IPC.* https://nvd.nist.gov/vuln/detail/CVE-2021-28154.
[6] *CVE-2021-41392: BoostNote arbitrary code execution through unsafe IPC.* https://nvd.nist.gov/vuln/detail/CVE-2021-41392.
[7] *CVE Security Vulnerability Database.* https://cve.mitre.org/.
[8] *DEDECMS 5.7 SEARCH.PHP TYPENAME Remote Code Execution.* https://vuldb.com/?id.181400.
[9] *Discord remote code execution.* https://mksben.l0.cm/2020/10/discord-desktop-rce.html.
[10] *Electron App Store.* https://www.electronjs.org/apps.
[11] *Electron (cross-platform framework).* https://en.wikipedia.org/wiki/Electron_(software_framework).
[12] *Electron Node Integration.* https://www.electronjs.org/docs/tutorial/security.
[13] *Electron Preload Scripts.* https://www.electronjs.org/docs/latest/tutorial/process-model/#preload-scripts.
[14] *Electron React Boilerplate.* https://github.com/electron-react-boilerplate/electron-react-boilerplate.
[15] *Electron Security, Native Capabilities, and Your Responsibility.* https://www.electronjs.org/docs/latest/tutorial/security.
[16] *Github Atom.* https://atom.io/.
[17] *Hackerone Bug Bounty Program.* https://www.hackerone.com.
[18] *Huntr Bug Bounty Program.* https://huntr.dev.
[19] *Introduce the notion of a "current microtask.* https://www.chromium.org/chromium-projects/.
[20] *Introduce the notion of a "current microtask.* https://chromium-review.googlesource.com/c/v8/v8/+/1277505.
[21] *Issue 3943: Disable webview when node integration is off.* https://github.com/electron/electron/issues/3943.

[22] *Issue 4026: Prohibit nodeIntegration from being re-enabled with window.open.* https://github.com/electron/electron/issues/4026.

[23] *JavaScript code coverage.* https://v8.dev/blog/javascript-code-coverage.

[24] *Linux AppArmor.* https://apparmor.com/.

[25] *Microsoft Teams remote code execution.* https://github.com/oskarsve/ms-teams-rce/.

[26] *Neutralinojs (cross-platform framework).* https://neutralino.js.org/.

[27] *Neutron Challenge in BSides Ahmedabad CTF 2021.* https://blog.s1r1us.ninja/CTF/bsidesahm2021#h.ymq4241d2kxp.

[28] *NW.js (cross-platform framework).* https://nwjs.io.

[29] *NW.js Frames.* https://nwjs.readthedocs.io/en/nw13/References/Frames/#iframe.

[30] *Prototype pollution attacks in NodeJS applications.* https://www.youtube.com/watch?v=LUsiFV3dsK8.

[31] *React: A JavaScript library for building user interfaces.* https://reactjs.org/.

[32] *React Native (cross-platform framework).* https://reactnative.dev/.

[33] *React-nodewebkit Starter.* https://github.com/konsumer/react-nodewebkit.

[34] *React starter project for Neutralinojs.* https://github.com/Abdulhafiz-Yusuf/neutralinojs-react.

[35] *Rocket.Chat remote code execution via click event.* https://hackerone.com/reports/899964.

[36] *Rocket.Chat remote code execution via message attachment.* https://hackerone.com/reports/899954.

[37] *Rocket.Chat Remote Code Execution via message attachment.* https://haxx.ml/post/145508617751/hacking-mattermost-2-year-of-nodejs-on-the?is_related_post=1.

[38] *Simplenote remote code execution.* https://hackerone.com/reports/291539.

[39] *Skype: A communication tool for free calls and chat.* https://www.skype.com/en/.

[40] *Slack: A proprietary business communication platform.* https://slack.com/.

[41] *Slack remote code execution.* https://hackerone.com/reports/783877/.

[42] *Snyk Vulnerability Database.* https://security.snyk.io.

[43] *The State of Vulnerabilities in 2019.* https://www.imperva.com/blog/the-state-of-vulnerabilities-in-2019/.

[44] *The State of Web Application Vulnerabilities in 2018.* https://www.imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2018/.

[45] *A study of Electron Security.* https://www.blackhat.com/us-17/briefings/schedule/#electronegativity---a-study-of-electron-security-7320.

[46] *Subverting Electron Apps via Insecure Preload.* https://blog.doyensec.com/2019/04/03/subverting-electron-apps-via-insecure-preload.html.

[47] *The T. J. Watson Libraries for Analysis (WALA) provide static analysis capabilities for Java bytecode and related languages and for JavaScript.* https://github.com/wala/WALA.

[48] *Teams: A business communication platform developed by Microsoft.* https://www.microsoft.com/en-us/microsoft-teams/group-chat-software.

[49] *V8 Context Stack Description.* https://source.chromium.org/chromium/chromium/src/+/main:v8/src/objects/contexts.h;drc=c0fceaa0669b39136c9e780f278e2596d71b4e8a;l=378.

[50] *V8 Zero-cost Async Stack Trace.* https://v8.dev/docs/stack-trace-api#async-stack-traces.

[51] *WhatsApp.* https://www.whatsapp.com/.

[52] *WhatsApp Arbitrary File Read.* https://www.perimeterx.com/tech-blog/2020/whatsapp-fs-read-vuln-disclosure/.

[53] *Wordpress remote code execution.* https://hackerone.com/reports/301458.

[54] Mohammad M Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. Sandtrap: Securing javascript-driven trigger-action platforms. In *USENIX Security Symposium (USENIX Security 2021)*, 2021.

[55] Joey Allen, Zheng Yang, Matthew Landen, Raghav Bhat, Harsh Grover, Andrew Chang, Yang Ji, Roberto Perdisci, and Wenke Lee. Mnemosyne: An Effective and Efficient Postmortem Watering Hole Attack Investigation System. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 787–802. Association for Computing Machinery, 10 2020.

[56] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1687–1700, 2018.

[57] James C Davis, Eric R Williamson, and Dongyoon Lee. A Sense of Time for JavaScript and Node.js: First-class Timeouts as a Cure for Event Handler Poisoning. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[58] Sebastian Egger, Peter Reichl, Tobias Hoßfeld, and Raimund Schatz. "time is bandwidth"? narrowing the gap between subjective time perception and quality of experience. In *2012 IEEE international conference on communications (ICC)*, pages 1325–1330. IEEE, 2012.

[59] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *Proceeding of the 42th IEEE Symposium on Security & Privacy*, IEEE SP 2021. IEEE, 2021.

[60] Aurore Fass, Michael Backes, and Ben Stock. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[61] Aurore Fass, Michael Backes, and Ben Stock. Jstap: a static pre-filter for malicious javascript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 257–269, 2019.

[62] Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 303–325. Springer, 2018.

[63] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *ACM CCS*, 2021.

[64] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

[65] Soroush Karami, Panagiotis Ilia, and Jason Polakis. Awakening the web's sleeper agents: Misusing service workers for privacy leakage. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[66] Soheil Khodayari and Giancarlo Pellegrino. Jaw: Studying client-side csrf with hybrid property graphs and declarative traversals. In *30th USENIX Security Symposium (USENIX Security 2021)*. Usenix, 2021.

[67] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918*, 2018.

[68] Sebastian Lekies, Ben Stock, and Martin 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, October 2013.

[69] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi 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*, pages 268–279, 2021.

[70] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747, 2012.

[71] OpenJS Foundation. About Node.js. https://nodejs.org/en/about/.

[72] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. You've changed: Detecting malicious browser extensions through their update deltas. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 477–491, 2020.

[73] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. Sok: In search of lost time: A review of javascript timers in browsers. In *6th IEEE European Symposium on Security and Privacy (EuroS&P'21)*, 2021.

[74] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 12 angry developers–a qualitative study on developers' struggles with csp. In *ACM CCS*, 2021.

[75] Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, October 2020.

[76] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 361–376, 2018.

[77] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin 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, February 2018.

[78] Marius Steffens, Christian Rossow, Martin Johns, and Ben 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, February 2019.

[79] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise Client-side Protection against DOM-based Cross-site Scripting. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[80] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. Breakapp: Automated, flexible application compartmentalization. In *NDSS*, 2018.

[81] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing dynamic library compromise on node. js via rwx-based privilege reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1838, 2021.

[82] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the node.js ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2951–2968. USENIX Association, August 2021.