CoinDef: A Comprehensive Code Injection Defense for the Electron Framework

Zheng Yang, Simon P. Chung, Jizhou Chen, Runze Zhang, Brendan Saltaformaggio, Wenke Lee
Georgia Institute of Technology

***Abstract*—The increasing popularity of cross-platform frameworks like Electron underscores the appeal of using familiar web technologies for desktop application development. Electron fuses the web and native environments into one single executable. However, this fusion creates unique vulnerabilities and significantly expands the attack surfaces for Electron applications, rendering traditional web defenses ineffective, as they are not designed to operate across both web and native contexts simultaneously. To address these challenges, we propose CoinDef, a centralized defense mechanism that enforces the structural integrity of Abstract Syntax Trees (ASTs) with execution context. CoinDef operates within the JavaScript engine, providing rapid, tamper-proof, and comprehensive mitigation against code injection attacks to Electron applications. CoinDef employs hybrid profiling to collect AST structural profiles, establishing a baseline of expected behavior. Then, CoinDef enforces these profiles for code as it is interpreted at runtime. In an evaluation of CoinDef on 20 representative real-world applications, we demonstrate its effectiveness in blocking exploits, incurring a 3.96% runtime overhead during application startup and negligible overhead during user interaction. Comparing CoinDef to state-of-the-art defenses for Electron applications, we show that CoinDef offers comprehensive protection against sophisticated code injection attacks through DOM manipulations and dynamic code execution.**

1. Introduction

The increasing popularity of the *Electron* framework for developing cross-platform applications highlights the enduring allure of software paradigms that leverage familiar web technologies [1]. However, this novel use of web technologies outside of the sandboxed browser setting also means vulnerabilities from web applications (e.g., *cross-site scripting (XSS)*, *prototype pollution*, etc.) can affect the underlying client machine, potentially resulting in the remote execution of malicious code (RCE).

One such attack with real-world consequences is the Water Labuu campaign discovered in October 2022 [2], which spreads malicious messages through Meiqia, an Electron-based chatting application used by over 400,000 companies for customer service. A simple click on the malicious message injects malicious code into the app and leads to the exploitation of CVE-2021-21220 [3], stealing more than 300,000 US dollars worth of cryptocurrency. Unfortunately, Meiqia is not the only one that is vulnerable. Similar campaigns can be launched against other high-profile apps like *Slack*, *Discord*, *MSTeams*, potentially targeting any subsequently discovered RCE [4]–[10] to execute malicious code of the attackers' choice on millions of machines and devices.

Despite extensive research into defenses against XSS for web applications [11]–[15] and RCE for native environments such as NodeJS [16]–[18], little has been done to address the root cause of such RCE attacks–*code injection*–within Electron applications. These prior solutions are effective within their respective runtime environments (web or native), but they are not equipped to handle the dual environments nature of Electron applications, which fuse web and native environments. This fusion introduces unique vulnerabilities and a broader attack surface [19], rendering existing defenses ineffective for protecting both environments simultaneously. Although recent studies have examined the attack surfaces in Electron applications and proposed mitigation [20]–[22], none have addressed the underlying code injection issues driving these security risks.

In our pursuit of a comprehensive and practical defense, we investigated the underlying cause of code injection attacks in Electron applications, drawing inspiration from Su and Wasserman's work [23], which observed that successful injections must change the intended syntactic structure of a program. Our research builds on this foundational insight and reveals two key observations specific to the Electron application ecosystem: 1) A code injection attack will ultimately change the semantics of the original code, reflected by a structurally different abstract syntax tree (AST), *resulting in a modified or new AST structure*. 2) The JavaScript engine of Electron is the *choke point* to provide comprehensive protection against code injection *for the web and native environments simultaneously*. Consequently, we turned our attention to preserving AST structural integrity in the JavaScript interpreter so that we can prevent code injection by enforcing the AST structural integrity with contextual information.

To this end, we propose CoinDef, a comprehensive Code Injection Defense for the Electron Framework. CoinDef works in the JavaScript interpreter (i.e., V8) of Electron applications to build AST profiles and enforce them at runtime. CoinDef works in two phases. In the learning phase, CoinDef identifies all JavaScript code contained in the application to be protected and generates

DOI 10.1109/SP61157.2025.00195

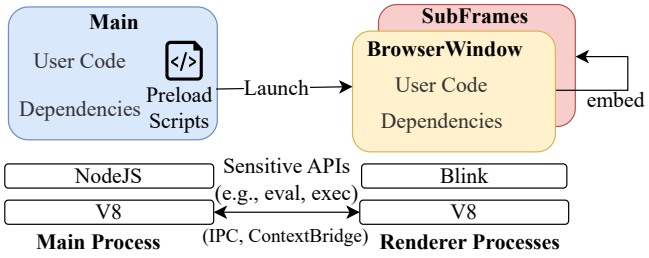


Figure 1: The Process Models of Electron.

the expected AST profiles of all the code with the execution context using both static profiling and dynamic profiling. In the enforcing phase, CoinDef validates every JavaScript code that is being interpreted by comparing the observed AST against those extracted in the learning phase. Furthermore, CoinDef leverages predefined security policies (*security-first* or *usability-first*) at runtime to accommodate unseen AST profiles that are not profiled in the learning phase. As working in the choke point in the execution pipeline for all code, CoinDef enables comprehensive mitigation against maliciously injected code. CoinDef also incurs negligible runtime overhead by taking a free ride (precomputing AST profiles) when the parser is parsing the source code. As such, code only needs to be validated once in its lifetime, incurring no runtime overhead during user interactions.

We evaluated CoinDef on 20 representative real-world Electron applications with code injection and RCE vulnerabilities, listed in Tab. 2. The applications we tested include widely used ones like *Slack* and *MSTeams*, as well as popular open-source projects such as *boostnote* and *joplin*. We first collected the AST profiles as the expected behavior model for each application using the approach discussed in § 3.4. After completing the learning phase, we proceeded to carry out 20 exploits using payloads previously reported as successful. The evaluation results show that CoinDef effectively blocked all the exploits. Furthermore, CoinDef only incurred, on average, a 3.96% *one time* overhead at startup time (§ 4.2) and negligible (Tab. 5) over the remaining lifetime of the Electron application. Additionally, we conducted a comparison with state-of-the-art solutions for Electron applications (§ 4.3) to highlight the comprehensive protection CoinDef provides and discussed where prior solutions failed.

We release the source code of CoinDef and instructions to use it to support further research and development on *https://github.com/ian7yang/CoInDef*.

2. Background & Challenges

Process Models. Fig. 1 presents the architectural overview of the Electron framework. Conceptually, the Electron framework consists of two processes. The main process assumes responsibility for native API access with the support of NodeJS. The renderer process focuses on rendering the UI and managing user interactions backed up by Blink, Chromium's renderer engine. Upon initializing the main process, Electron's main process creates a `BrowserWindow` object. This window object launches an HTML page as the UI that runs in the renderer process. Notably, the renderer process relies on Blink and thus inherits the security mechanisms of Chromium, including site isolation, which prevents direct communication between different web frames [24]. To establish communication between the main and the renderer processes, Electron introduces preload scripts, which are bound to the renderer process and provide `BrowserWindow` objects with access to sensitive APIs (e.g., `shell`) defined in those preload scripts through Inter-process Communication (IPC). Additionally, Electron offers developers the `NodeIntegration` option to directly integrate NodeJS's context into the renderer processes, facilitating fast development.

SubFrames. Chatting applications like *Discord* provide "In-App View" features to allow their users to directly view external resources (e.g., watch a YouTube video) in the main application window. However, these applications do not load arbitrary external content. Instead, they have clearly defined the Content Security Policy (CSP) to only load trusted resources and render them in the isolated `iframe`, which we annotated as `SubFrames` in Fig. 1. The JavaScript code running in the `SubFrames` is isolated from the main application to prevent it from tampering with the main application. Since these resources are only presented at runtime, they are unknown to static analysis tools and may not be there when being dynamically profiled.

The shared context between the main and the renderer processes provides UI access to the native OS resources. Meanwhile, it opens security holes for Electron applications: *code injection in the UI can springboard into the main process and achieve RCE* by either directly invoking those APIs through shared context enabled by misconfigured `contextIsolation` [5] or crafting a series of sophisticated payload to abuse the correctly exposed APIs [7]. `contextIsolation` is the critical security feature Electron creates to ensure the JavaScript code running in the UI's process cannot arbitrarily access the powerful NodeJS context running in the main process. With context isolation enabled, Electron creates `contextBridge` to expose functions connected to native OS resources to the UI for desktop experience. Unfortunately, these security suggestions are not well followed by application developers [20], [22]. Even worse, attackers have found ways to bypass these restrictions in some scenarios [25], [26].

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

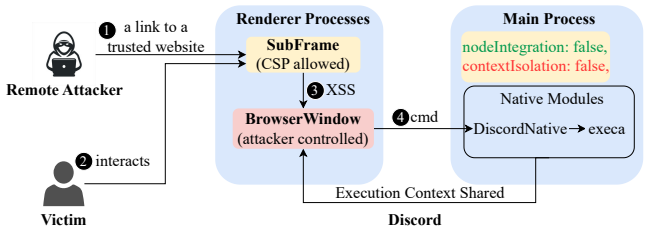


Figure 2: A Real-world Motivating Example.

2.1. A Motivating Example

As shown in Fig. 2, a real attack scenario [5] begins by sending a website link to a potential victim in step ❶. Although *Discord* has defined CSP to prevent code injection, it cannot guarantee all its partners follow the same security practices. As a result, there is still a possibility for trusted partners with insufficient security practices to be exploited and serve as an open gate for the entire system.

In this example, one of the trusted partner websites, *sketchfab.com*, has a cross-site scripting (XSS) vulnerability when rendering annotations for 3D models. The attacker takes advantage of this feature and sends the victim a well-crafted 3D model containing malicious JavaScript code. Then, the XSS bug gets triggered as the victim interacts with the 3D model isolated in a `SubFrame` in step ❷ using the "In-App View" feature. Specifically, the XSS payload successfully navigates the parent frame (the top window or the content of the `BrowserWindow`) to an attacker-controlled webpage in step ❸ by exploiting CVE-2020-15174 [27], which is called a `client-side-redirect` attack. Since this attacker-controlled webpage is now in the context of the top window, it gains access to *Discord*'s `BrowserWindow` context, escaping from the isolated `SubFrame`. Meanwhile, due to the inadequate setting (`contextIsolation: false`) of isolation between the main and the renderer processes, the UI can access the sensitive NodeJS APIs (e.g., `fs`, `child_process`, etc.) even in the renderer process. This misconfiguration gives the attacker further opportunities to compromise the main process. Consequently, the attacker overwrites two JavaScript's built-in methods (i.e., *prototype pollution* attacks) to successfully invoke a privileged NodeJS module `DiscordNative` and achieves remote code execution through `execa` in step ❹.

2.2. Challenges

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

C1. Comprehensive Protection. An effective protective system for Electron applications should address all potential code injection scenarios, covering prevalent web attack vectors like *XSS*, *prototype pollution*, and *client-side-redirect*. This protection must apply across all layers, including first-party code, third-party libraries, and legitimate remote code that Electron applications might load. Ensuring such comprehensive coverage also requires an adaptable approach to usability, preserving core functionality for users. By implementing flexible modes such as *security-first* for heightened protection and *usability-first* for greater functionality, the system can accommodate different user priorities, achieving an optimal balance between security and usability.

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

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

3. System Design

3.1. Threat Model & Assumptions

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

3.2. Design Overview

Fig. 3 illustrates the high-level work flow of CoinDef. CoinDef takes in an Electron application in the learning

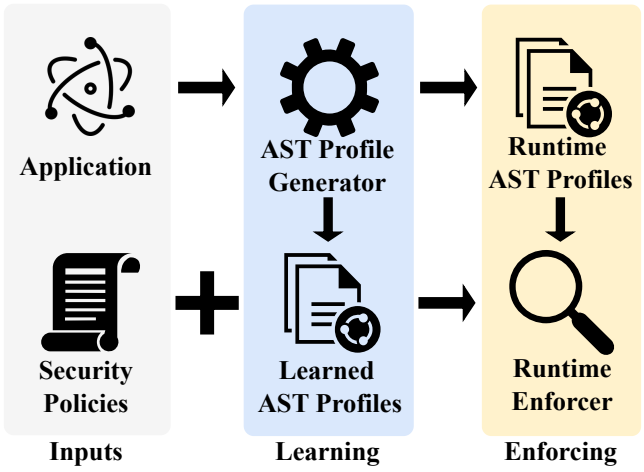


Figure 3: CoinDef Design Overview.

phase to construct its AST profiles for enforcement. In the enforcing phase, CoinDef validates the AST profiles generated at runtime against the learned ones. Unlearned AST profiles encountered at runtime are handled according to predefined security policies (detailed in § 3.5.1) based on the protection mode: *security-first* or *usability-first*. To overcome the challenges outlined in § 2.2, CoinDef operates in the language interpreter, a central place where all JavaScript code (i.e., both web and native) must pass through. This placement not only ensures comprehensive protection (**C1**) but also is tamper-proof (**C3**) from remote attackers since CoinDef has higher privileges than the JavaScript code for residing in the interpreter. Furthermore, CoinDef takes advantage of the existing JavaScript code parsing process to get an almost free ride for constructing AST profiles, incurring negligible runtime overhead (**C2**).

3.3. AST Profile

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

3.3.1. AST Structural Signature. To generate such AST structures for any code, whether hard-coded (i.e., static) or dynamically generated (i.e., dynamic), CoinDef extracts each AST node's type, value, and position when the interpreter parses the source code. To allow varying legitimate user inputs for dynamic code, CoinDef replaces the concrete values with placeholders for certain *data* nodes. Tab. 1 lists all types of data nodes in an AST including the value of a *literal*, *name*, or *key* node which can only be a string or a number; the value of a *value* node under a property parent can be an object literal, an array literal, a string, or a number. By design, none of these nodes should introduce function definitions or invocations in an AST. If controlled by an attacker, a `variable proxy` (i.e., the variable's identifier) can be pointed to a function or an object. Therefore, CoinDef only allows the value to change when the variable proxy node is not under a `call`, `new`, or `assign` expression.

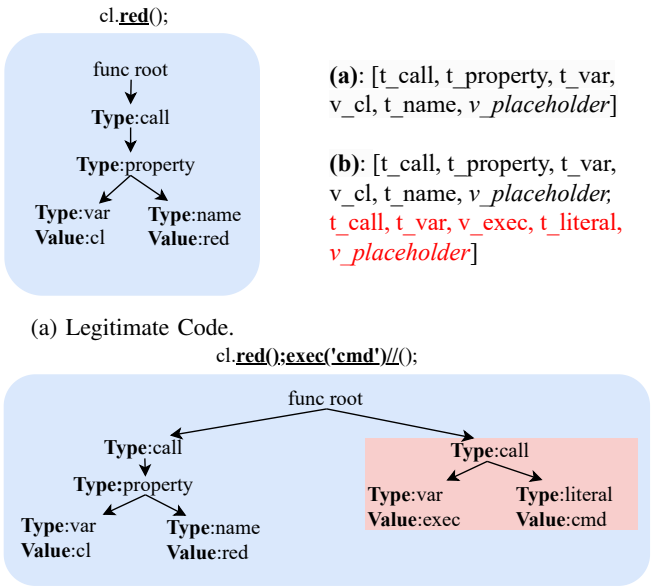
TABLE 1: Data Nodes for AST Structural Signature.

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

Such an AST structural signature prevents the (malicious) inputs from defining a new function, overwriting the prototype functions, calling existing functions, or executing code directly, which simply searching for function-related operations cannot achieve. For instance, given JavaScript code that is vulnerable to code injection attacks: `eval(`cl.${color}()`)`, the dynamic input is `color`. The expected legitimate `color` can be a literal (e.g., `red`). When `eval` executes, an AST is generated as shown in Fig. 4a. However, attackers can change the value of `color` from `red()` to `red();exec('cmd')//`, resulting in arbitrary code execution and a different AST structure as highlighted in Fig. 4b. By checking the code's AST structural signature in Fig. 4b with the baseline in Fig. 4a, CoinDef can detect and reject the malicious code, `exec('cmd')`. Since CoinDef does not consider node's values for `property.name`, any legitimate, expected inputs to `color` (e.g., `yellow`, `green`) can pass the validation.

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

3.3.2. Execution Context Annotation. The execution context is essential for Electron applications to distinguish the privileges of web and native code and generate finer-grained AST profiles for mitigating mimicry attacks [29]. For example, suppose the attackers can leverage Electron's vulnerabilities (e.g., CVE-2022-29247 [30]) to access the native environment



(a) Legitimate Code.

(b) Malicious Code.

Figure 4: AST Structural Signatures of Legitimate and Malicious Code. An example taken from `eval(cl.`${color}()`)` [28] shows that code injection alters the AST structural signature.

through the web layer. In that case, they can directly import NodeJS libraries as what has been defined in the native code to pass the AST structural signature validation. Based on the least-privilege principle, CoinDef isolates the AST profiles based on their running processes (i.e., `MainProcess`, `BrowserWindow`, and `SubFrames`) and annotates each AST profile with the process context and code context, including the caller information and callsite location if it is an `eval`-like call, exemplified in Fig. 5. To facilitate fast enforcement, CoinDef also divides the AST profiles into two categories, static and dynamic, per process.

Static Profile. CoinDef considers an AST profile static when the interpreter interprets a static script file and functions defined in the script file. For instance, when Electron renders an HTML page, the HTML parser invokes V8 to compile code defined in `<script src=''A.js''>`. Currently, the requester is the HTML parser; therefore, static profiles have no execution context other than the running process. Due to the lazy compilation policy[1], not all JavaScript code in a script file is interpreted immediately. That is, some functions, if not invoked immediately, are only parsed and compiled when other code invokes them. Specifically, while parsing the source code of a script, the parser will skip some functions that are not immediately invoked and remember their names and scopes. When those skipped functions are invoked, the parser will parse the function body to generate bytecode for them. At this time, the requester is a

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

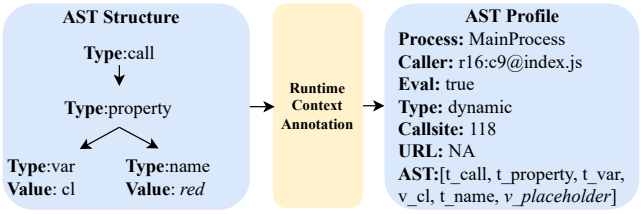


Figure 5: An AST Profile. CoinDef constructs an AST profile by extracting the AST structural signature annotated with runtime context for `eval(cl.`${color}()`)` where `color` is "red".

JavaScript function. However, these functions are defined in the static script file and are not changeable. Therefore, CoinDef still considers AST profiles of such lazily compiled functions static after checking their original script with local files.

Dynamic Profile. CoinDef considers an AST profile dynamic when the code comes from dynamic code generation APIs (e.g., `EventHandler`, `eval`, `document.write`), obtained from the code injection sinks defined in CodeQL [31]. Beyond the code injection sinks, CoinDef also considers the scenario where one script includes or imports another. For example, when script A includes script B by appending another `<script>` tag or runs `eval`-like or `importScript` APIs to execute code dynamically, CoinDef considers the AST profile of Script B or the dynamically evaluated code as a dynamic profile. Note that dynamically generated code also complies with the lazy compilation policy. CoinDef uses the same method to assign types to functions defined in dynamically generated code.

Fig. 5 shows that an `eval`-like API is invoked at the 118th character of a caller function defined at row 16, column 9 in the file "index.js." Since this code is dynamically generated, the profile type is "dynamic" and the URL is marked as "N/A." Because `property.name` is a data node, the value is marked as a placeholder.

3.4. AST Profile Collection

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

CoinDef collects AST profiles at the function level to preserve AST structural integrity within the execution

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

Static Profiling. CoinDef customizes D8[2], a shell interface to V8, to exclusively invoke V8's parser and generate AST structural signatures. This customization disables the lazy compilation described in § 3.3.2, forcing the parser to build the AST structural signature for every JavaScript function defined in the local files. This enables static profiling to provide CoinDef with a comprehensive model of the application. Specifically, CoinDef obtains P_S as:

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

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

Dynamic Profiling. CoinDef comprehensively exercises each application to supplement the static profiles (P_S) using a semi-automated approach that simulates user interactions, similar to other state-of-the-art techniques [20], [32], [33]. This approach begins with a crawler that systematically interacts with the application's UI, clicking on buttons and menus and typing text. To improve the performance of this approach by covering more complex features, we supplement it with manual exercises based on the application user manuals. If available, we also employ end-to-end test cases to simulate user interactions and trigger features, ensuring we cover all relevant functionalities. Given a dynamic code trigger action T in such a dynamic profiling procedure, CoinDef obtains:

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

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

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

Algorithm 1: AST Profile Enforcement.

Data: Current Runtime AST Profile r, Learned AST Profile P, Security Policies S
Result: Whether to generate the function object

```
1  begin
       // r is exemplified in Fig. 5.
2      if r.type == static then
3          P_S ← a nested hash map keyed by url,
             callsite for r.process;
4          p ← P_S.find(r.url, r.callsite);
5          return p.validate(r.ast);
6      end
7      P_D ← a nested hash map keyed by url, caller,
         callsite for r.process ;
8      ps ← P_D.find(r.url, r.caller, r.callsite);
9      if ps ≠ ∅ and ps.validate(r.ast) then
10         return true ;
11     end
       // security policy for other cases.
       // local or remote dataflow scope of r.
12     scope ← localDataFlow(r) ;
       // empty-, cross- or same-origin
13     origin ← originCheck(r.url) ;
       // obtained from hybrid profiling.
14     allowedOrigins ← S.allowedOrigins ;
15     if scope == local then
16         return true;
17     end
18     if r.process == MainProcess then
19         return false;
20     end
21     if r.process == BrowserWindow then
22         if origin != same then
23             return false;
24         end
           // S.SameOriginInBrowser is false by
              default.
25         return S.SameOriginInBrowser;
26     end
27     if r.process == SubFrames then
           // S.SubFrame is false by default.
28         if not S.SubFrame then
29             return false;
30         end
31         fOrigin ← frameOrigin(r.caller);
32         return fOrigin ∈ allowedOrigins;
33     end
34 end
```

3.5. Runtime Enforcement

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

according to predefined security policies. Specifically,

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

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

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

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

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

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

3.6. Security Analysis

Now, we formalize the security analysis as follows.

Definitions.

$\mathcal{T}_{model}$: an AST representation of the program execution learned in the profiling phase.
$\mathcal{T}_{runtime}$: an AST resulting from runtime input (e.g., user input, dynamic code) in the enforcing phase.
$\mathcal{V}$: the set of $\mathcal{T}_{model}$ learned from the whole program, and $\mathcal{T}_{model} \in \mathcal{V}$.
$\mathcal{C}$: the execution context of a $\mathcal{T}_{model}$ at a given program point, and $\mathcal{T}_{model} \in \mathcal{V}_\mathcal{C}, \mathcal{V}_\mathcal{C} \subseteq \mathcal{V}$.
$\mathsf{DataNode} \subset \mathcal{T}$: the set of leaf nodes representing data values (e.g., string literal) expected to vary at runtime.
$\mathsf{ExecNode} \subset \mathcal{T}$: the set of internal nodes representing executable logic (e.g., call expression).
$\mathsf{IntermediateNode} \subset \mathcal{T}$: the set of internal nodes representing expressions and statements (e.g., block statement).
$\mathsf{Children}(n)$: each node $n \in \mathsf{IntermediateNode}$ has a sequence of children forming a subtree, denoted by $\mathsf{Children}(n) = [n_1, n_2, \ldots, n_k]$.

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

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

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

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

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

1) Type mismatch is disallowed:

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

2) Data nodes must match in type:

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

3) Exec nodes must match exactly:

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

4) Intermediate nodes must match recursively on their children:

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

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

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

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

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

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

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

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

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

For cases where input validation cannot be definitively resolved–due to incomplete learning, ambiguous context, or mixed data types–CoinDef defers to a predefined set of security policies (Alg. 1). These policies combine static and dynamic analyses, such as local data flow inspection and runtime execution context profiling, to maintain security guarantees while supporting usability.

3.7. Implementation

As illustrated in Fig. 6, CoinDef integrates instrumentation hooks into V8's interpreter to gather AST node information, modeling both static and dynamic code behaviors.

When code is processed, the language parser tokenizes it and constructs two objects: `script_info` and `parser_info`. These objects contain metadata about the script, including the AST, source location, text range, and the origin of the code (static or dynamic). Before bytecode generation begins, CoinDef collects contextual

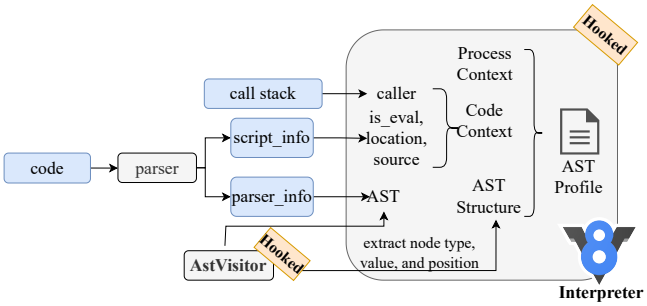


Figure 6: AST Profile Hooks Implementation.

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

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

4. Evaluation

To evaluate CoinDef 's effectiveness and practicability in protecting users from code injection attacks and whether it effectively addresses the challenges presented in §2.2, we address the following research questions:

RQ1: How effectively can CoinDef protect users from code injection attacks and the subsequent RCE?
RQ2: How do FP and FN impact user experience?
RQ3: What is the runtime overhead?
RQ4: How comprehensive is CoinDef compared with the SOTA tools?

To answer these research questions, we evaluate CoinDef on 20 diverse applications that are vulnerable to code injection and RCE, using their real-world exploits. These applications are selected for their broad representation across software categories, scales, types of vulnerabilities, and use cases. We source them from the GitHub Advisory Database [41], articles, and technical blogs [4], [5], [7], [8], [34].

TABLE 2: A Diverse Set of Applications Vulnerable to Code Injection and RCE Attacks.

#	Application	Vulnerable Version	Electron Version	Line of Code	GitHub Stars Reference	Injection Description	Attack Vectors
1	MSTeams	v1.4.00.4855	v8.5.5	187,295	N/A [7]	a message to achieve template injection attacks	TI, S-XSS, RCE
2	Slack	v4.3.2	v7.1.9	153 (minified)	N/A [4]	an embedded frame opening a malicious webpage	CSR, RCE
3	Discord	v0.0.14	v11.4.2	10,932	N/A [5]	a embedded frame opening a malicious webpage	CSR, PP, RCE
4	VSCode	v1.63.1	v11.2.1	3,114 (minified)	147k [34]	a local file exploiting markdown preview	MP, RCE
5	GraSSHopper	v1.1.7	v12.0.6	71 (minified)	N/A [21]	a text rendered as HTML in a popup window	D-XSS, RCE
6	ARDM	v1.4.9	v11.4.9	11,271	25.8k [21]	a text rendered as HTML	S-XSS, RCE
7	Joplin	v2.9.12	v19.0.10	172,199	36.2k [35]	the language indicator for the markdown code format	MP, RCE
8	Boostnote	v0.22.0	v12.0.14	103,203	20.6k [21]	code rendered as HTML for the markdown code format	MP, RCE
9	Altair-graphql	v4.0.11	v14.0.1	1,804	4.7k [21]	query description rendered as HTML	S-XSS, RCE
10	Appium-desktop	v1.22.0	v7.1.2	133 (minified)	4.5k [36]	incoming http request reflected as HTML	R-XSS, RCE
11	Simplenote	v2.9.0	v9.1.0	20,615	4.4k [10]	markdown file not being properly sanitized	MP, RCE
12	BlockBench	v3.9.3	v13.1.2	49,280	2.1k [21]	a filename rendered as HTML	D-XSS, RCE
13	electron-crud	v2.8.0	v10.0.0	1,168	1.5k [21]	database records rendered as HTML	S-XSS, RCE
14	arc-electron	v16.0.1	v13.1.1	9,971	1.3k [9]	HTTP header rendered as HTML	S-XSS, RCE
15	vmd	v1.34.0	v3.0.9	1,976	1.2k [37]	markdown file not being properly sanitized	MP, RCE
16	antares-sql	v0.5.6	v14.0.1	256,525	1.1k [21]	database table names rendered as HTML	S-XSS, RCE
17	Markdownify	v1.4.1	v7.2.4	10,337	868 [38]	markdown file not being properly sanitized	MP, RCE
18	Poddycast	v0.8.0	v11.2.1	2,395	160 [39]	bookmark rendered as HTML	S-XSS, RCE
19	OhHai Browser	v3.4	v8.2.5	2,736	52 [40]	bookmark rendered as HTML	S-XSS, RCE
20	Jukeboks	v2.2.2	v11.2.3	1,360	23 [21]	filename rendered as HTML	D-XSS, RCE

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

4.1. Effectiveness

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

4.1.1. Diverse Code Injection Vulnerabilities. As described in Tab. 2, the code injection points of the vulnerabilities generally fall into three categories: messages, remote resources, and markdown files, covering the prevalent code injection attack vectors for Electron applications, including *template injection*, *markdown preview*, *client-side-redirect*, *prototype pollution*, and *all types of XSS*. Among these attack vectors, *client-side-redirect* is particularly threatening in Electron applications and is not handled in prior work [20], [21], because it opens a new website and completely takes over the `BrowserWindow`, resulting in a legitimate call chain through IPC to the `MainProcess`. For example, collaboration applications like *MSTeams*, *Slack*, and *Discord* have injection points within the messaging feature. In the case of *MSTeams*, an attacker can exploit a vulnerability in rendering the display name of a mentioned user in a group chat to launch a code injection attack on AngularJS's template engine. For *Slack*, the injection point is a malicious file uploaded to *file.slack.com* and then sent to a victim via messaging. Similarly, for *Discord*, attackers can inject malicious code through messages. These two attacks leverage malicious iframes and a vulnerability of Electron to launch an *client-side-redirect* attack to load attacker-controlled content in the `BrowserWindow`, which pollutes the prototype of certain built-in functions to achieve RCE. Some applications load remote resources and injection points can be found within these resources. For example, in *electron-crud*, attackers can inject JavaScript code into the records of the connected database to launch a Stored XSS. The malicious records allow attackers to execute arbitrary code. Productivity applications like *VSCode* have injection points within markdown files. These files may contain well-crafted HTML payloads that are not properly sanitized. When users interact with these markdown files, the injected code executes, leading to RCE.

4.1.2. Experiment Setup. The primary objective of evaluating CoinDef is to assess its effectiveness in blocking code injection attempts while allowing legitimate user inputs. To ensure realistic testing, we use the vulnerable versions of each application listed in Tab. 2, ensuring that all code injection attacks constitute novel (i.e., 0-day) attacks for each application and CoinDef is agnostic to these attack vectors.

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

4.1.3. Learning Phase. Using the methods outlined in § 3.4, we collect AST profiles for all 20 applications. Tab. 3 presents the size of these AST profiles alongside the human effort required for their collection, measured in hours. The AST profiles are organized by process (i.e., `MainProcess`, `BrowserWindow`, and `SubFrames`)

TABLE 3: AST Profiles Collected in the Learning Phase.

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

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

TABLE 4: Effectiveness of CoinDef under Enforcement.

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

*: code coverage during enforcement testing.

and type (i.e., `static` or `dynamic`). Notably, only three collaborative applications (i.e., *MSTeams*, *Slack*, and *Discord*) and *VSCode* exhibit dynamic code in `SubFrames`. This is because collaborative applications include "In-App View" features, while *VSCode* executes extensions in isolated environments. Our results show that 18 out of the 20 applications generate and run code dynamically, with more complex applications producing and executing greater amounts of dynamic code. The applications also run more dynamic code within the `BrowserWindow` process compared to `MainProcess`, which is expected since users interact primarily through the UI components, triggering dynamic code generation in `BrowserWindow`. Notably, achieving a converged state of learned AST profiles required approximately 4 hours for complex applications and about 30 minutes for simpler ones, as shown in the last column of Tab. 3. In the converged state, the number of AST profiles no longer increases, indicating that our exercising strategies have fully covered the necessary code paths.

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

Code Coverage During Enforcement Testing. To fairly evaluate the defensive mechanism for CoinDef, we adopt function-level code coverage on first-party code, as defined by V8 [42]. This choice is motivated by the well-documented bloat of JavaScript dependencies, where developers may import a single function from a large module, resulting in low overall code coverage if third-party code is included. Moreover, we focus on function-level rather than block-level code coverage, as CoinDef enforces the integrity of the AST profile for JavaScript functions, not their individual blocks. Once a function passes validation, it is allowed to execute in full. We use Acorn [43], a popular JavaScript parser, to statically count the number of functions in the first-party code as our ground truth. During the testing phase, we log the number of functions invoked to calculate the test coverage. As shown in Tab. 4, the applications were exercised thoroughly. For simple and small applications (three out of 20), achieving 100% code coverage was straightforward. For medium size applications (five out of 20), we achieved over 90% code coverage. For more complex applications (12 out of 20), such as *Slack* and *VSCode*, we achieved over 75% code coverage, indicating robust test coverage across varying application complexities.

Protection Result. As illustrated in Tab. 4, we executed a total of 79 attacks across 20 applications, with each application facing between two and eight targeted attacks. Each individual attack focused on a specific code injection point. However, in some cases, reaching RCE required a series of attacks in a kill chain, as illustrated in § 2.1. For instance, six attacks launched against *Discord* formed two separate attack sessions, each involving three interdependent attacks (i.e., *client-side-redirect*, *prototype pollution*, and *RCE*) that combined to achieve RCE. In the *security-first* mode, CoinDef successfully blocked all code injection attempts for all 20 applications with few false positives. In *usability-first* mode, CoinDef allowed two code injections to bypass protection in two applications but still prevented the final RCE with zero

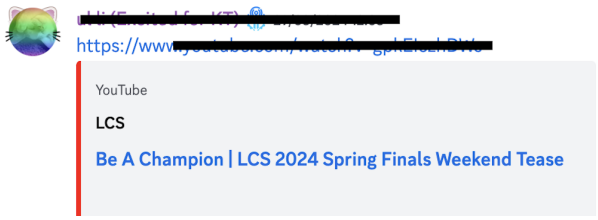


Figure 7: The False Positive Example Observed on Discord. Although the "In-App View" video player is disabled, the user can still click the link to open the video using a browser.

TABLE 5: Runtime Overhead During User Interaction.

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

false positives. This result suggests that for applications primarily used locally without loading remote content (e.g., *GraSSHopper*, *SimpleNote*, *Markdownify*), CoinDef reliably prevents code injection attacks under both modes. For applications (e.g., *Slack*, *Discord*) with more interactive features, such as "In-App View" functionality, CoinDef may miss one attack targeting `SubFrames` but it nonetheless delivers robust protection by securing critical processes in the `BrowserWindow` and `MainProcess` contexts, blocking any code injections into these processes and subsequent RCE attacks (**RQ1**).

False Positives. In our evaluation, we observed 17 false positives in the security-first mode across 3 of the 20 applications tested, while the usability-first mode produced zero false positives. These 17 cases fall into two categories: 1) unlearned features provided by remote content and 2) unlearned updates. Specifically, 12 false positives resulted from "In-App View" features in *Slack* and *Discord*, such as video playback or file previews within isolated `SubFrames`. In security-first mode, scripts running within `SubFrames` are disabled, blocking this content, as illustrated in Fig. 7. Although this leads to an empty video window, users can still click the link to open it in an external browser. The remaining five false positives occurred in *VSCode* due to extension updates that introduced new script files. We suggest that users can temporarily enable learning mode for these extensions to gather the necessary AST profiles for these updates.

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

The FP and FN analysis suggest that the impact on the user experience is minimal (**RQ2**). If the users want to enjoy the "In-App View" features, they can opt-in for the *usability-first* mode, in which the security in the main application remains guaranteed.

4.2. Runtime Overhead

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

Page Load. In the first experiment, we measured the overhead caused by CoinDef during the page loading process, which is in line with prior work [20], [44]–[47]. We initiated a timer when the `navigationStart` event was triggered and concluded the measurement when the `loadEventEnd` event was fired on the initial web page. This time difference represented the page load time. As some applications load remote websites as their first page (e.g., *Slack* and *Discord*), we ran the application once before testing to warm up the network cache, thus reducing the impact of network latency. For more complex applications like *VSCode*, which encompass multiple pages/frames, we only measured the overhead for the main frame (e.g., the editor window for VSCode).

To calculate the overhead for page load, we executed the application ten times with and without CoinDef enabled, selecting the median time cost. As indicated in the last column of Tab. 4, CoinDef introduced an average overhead of only 3.96%, with the highest overhead of 9.31% observed for *Slack* and the lowest of 1.04% for *Poddycast*. Although the overhead for *Slack* appears relatively high, the additional time contributed by CoinDef is merely 19 milliseconds. Notably, CoinDef exhibits higher overhead for applications that require network connections, such as *MSTeams* and *Discord*. For instance, *Discord* performs update checks before loading the first page, and our testing environment's first page comprises rich content with 57 script files sourced from *discord.com*. These 57 script files triggered validation for over 20,000 AST profiles.

Interaction. Consistent with previous work [21], we utilized the Speedometer 2.0 benchmark suite [48] to measure the overhead for user interactions. The Speedometer 2.0 benchmark suite comprises 17 uniquely implemented *TodoMVC* applications. The benchmark simulates user actions of adding, completing, and removing items from a to-do list. In this process, the benchmark sequentially executes the 17 *TodoMVC* applications. For each application, the test begins by adding 100 items one at a time, with each item containing

TABLE 6: CoinDef Compared With the State-of-The-Art.

Attack Vectors	# of Attacks	Synode [16]		DomTyping [21]		XGuard [20]		CoinDef*	
		FP	FN	FP	FN	FP	FN	FP	FN
Discord									
CSR	1	0	1	0	1	0	1	1	0
PP	1	0	1	0	1	0	1	0	0
RCE	1	0	1	0	1	0	0	0	0
jukebox									
D-XSS	1	0	1	0	0	0	1	0	0
RCE	1	0	1	0	0	0	0	0	0
AllinOne†									
D-XSS$_D$	2	0	2	1	0	0	2	0	0
D-XSS$_E$	2	1	0	0	2	0	2	0	0
S-XSS$_D$	2	0	2	1	0	0	2	0	0
S-XSS$_E$	2	1	0	0	2	0	2	0	0
CSR	2	0	2	0	2	0	2	1	0
PP	1	0	1	0	1	0	1	0	0
RCE	10	0	6	0	6	2	0	0	0
Total CI	14	2	10	2	9	0	14	2	0
Total RCE	12	0	8	0	7	2	0	0	0

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

some content. Subsequently, the test iteratively marks each item as complete. Finally, the benchmark concludes by removing all the items individually. We repeated this process ten times, with and without CoinDef. As shown in Tab. 5, the time cost for running all 17 *TodoMVC* applications ranges from 111.9 milliseconds to 119.9 milliseconds for the baseline version of Electron, while it is between 113.9 milliseconds and 120.1 milliseconds. The interactive overhead is as small as 0 and as large as 1.78%. By design, CoinDef is not supposed to introduce runtime overhead for user interaction because CoinDef does not hook in function calls. Therefore, we applied all the data samples to measure their confidence intervals and concluded that the two sample sets originate from the same distribution, which indicates that CoinDef does not introduce any runtime overhead during user interactions. Note that in scenarios where the application loads remote scripts, there will be minimal overhead as we measured for app startup (shown in Tab. 4). However, compared with the network latency during user interaction, such milliseconds overhead is negligible.

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

4.3. Comparison To State-of-The-Art

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

Experiment Setup. For a representative comparison, we selected three applications: *Discord*, a complex application discussed in § 2.1; *Jukeboks*, a simpler application; and a custom-built application (i.e., *AllinOne*) that incorporates a range of attack vectors, including XSS, OR, PP, and RCE. Specifically, we derived code injection attacks based on the CVEs identified in our evaluation dataset and the XSS cheat sheet from OWASP [49] into *AllinOne*. These attacks include DOM-based XSS via DOM manipulation (D-XSS$_D$) and dynamic code execution using `eval` (D-XSS$_E$), Stored XSS through DOM manipulation (S-XSS$_D$) and dynamic code execution using `eval` (S-XSS$_E$), client-side-redirect (CSR) from a `SubFrame`, prototype pollution (PP), and the final RCE with each code injection attack leads to except for PP. In total, the *AllinOne* contains 11 code injection points leading to 10 RCE attacks. Notably, we set CoinDef to the default *security-first* mode for this comparison.

Result. We present the comparison result in Tab. 6. Synode failed to block 10 code injection attacks, eight of which ultimately resulted in RCE. This is because Synode cannot cover code injection through DOM modifications including *client-side-redirect*. Moreover, Synode produces two false positives for dynamic code execution through `eval` due to the inaccuracy of static analysis on implicit data flow for string concatenation. DomTyping failed to prevent nine code injection attacks, leading to seven cases of RCE. this is because DomTyping does not handle *client-side-redirect* and dynamic code execution. Furthermore, DomTyping reported two false positives for legitimate DOM modifications. Although XGuard successfully prevented all RCE attacks, it failed to block all 14 code injection attempts, meaning that attacks could progress to critical stages before being mitigated. Meanwhile, XGuard reported two false positives due to the incomplete call chain modeled based on minified JavaScript code. In contrast, CoinDef provided complete protection, successfully preventing all code injection and RCE attacks across all scenarios. This underscores CoinDef 's comprehensive coverage and robust defensive capabilities compared to existing solutions.

Case Study. To better understand the comprehensive protection CoinDef offers, we elaborate the experiment process with a case study for *Discord* in Fig. 8. The attacker first injected JavaScript code onto the DOM of the vulnerable website. The injected code launched a *DOM-based XSS* attack to initiate the *client-side-redirect* attack. When compiling the code for navigating the browser window of *Discord*, CoinDef disabled any code running in `SubFrames`, and thus rejected the compilation under the *security-first* mode. Subsequently, the interpreter returned a `noop` function, and the *client-side-redirect* attack was prevented in step ❶. What if the user was under *usability-first* mode and the attacker reached ❷? To show that CoinDef can still provide protection and prevent the attack from evolving into a remote code execution attack, we turned on the *usability-first* mode. When compiling the malicious script inside the malicious

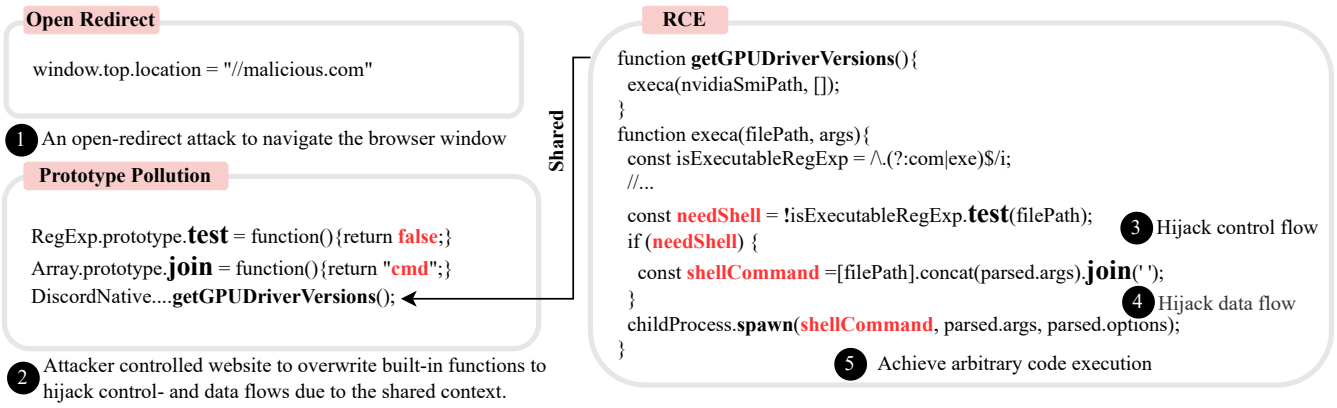


Figure 8: The Detailed Attack Chain on Discord.

website, CoinDef detected that the malicious JavaScript code was going to run in the context of `BrowserWindow`. Then, CoinDef pulled out the AST profiles built for `BrowserWindow` and found no entries for the malicious script. Subsequently, CoinDef rejected the compilation and the interpreter returned a `noop` function. If the attacker was smarter and could craft the malicious code to pass the validation and get executed in step ❷. To simulate this effect, we modified the code and let the attack proceed to this step. However, in order to invoke the privileged module (`DiscordNative` shared by the `MainProcess`) to launch RCE, the attacker must overwrite two methods for two built-in JavaScript objects: `test` for `RegExp` to hijack the control flow in step ❸ and `join` for `Array` to hijack the data flow in step ❺. No matter how sophisticated the attacking code became when being executed, it had to create two functions, which created two new AST profiles. These two new dynamic profiles were never found in the AST profiles for `BrowserWindow` given its running context. Thus, CoinDef blocked the *prototype pollution* and subsequently prevented the RCE.

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

5. Discussions

Robustness. CoinDef demonstrates its robustness by successfully blocking all attacks, among which are 20 cited attacks and 59 variants trying to evade CoinDef. These attacks include DOM manipulation, dynamic execution, and other evasion techniques built on top of our domain expertise. To achieve this robustness, CoinDef limits attackers' ability to inject foreign code through: execution context integrity enforcement (§ 3.3.2) and AST structural integrity enforcement (§ 3.3.1) as formalized in § 3.6. First, CoinDef tracks execution context integrity, ensuring that even structurally valid but behaviorally malicious scripts are detected. Attackers cannot arbitrarily construct a legitimate AST for a given injection point, as the execution context restricts possible AST modifications. Then, CoinDef only allows data nodes to change to accommodate legitimate user inputs. Any injected code inevitably modifies a data node into a subtree, which CoinDef blocks. This enforcement applies across DOM-based injections, template literals, and `eval`-like APIs, preventing attackers from rewriting execution logic.

Deployment. CoinDef identifies the required Electron version and downloads a corresponding instrumented Electron version, leaving the application code intact. This convenient integration offers improved code injection security without modifications or source code access. To generate AST profiles, application developers can leverage the existing UI testing cases to provide comprehensive AST profiles. As for IT administrators and home users, they can leverage automated crawlers and manual exercises to build AST profiles that are customized for their use patterns, in line with prior work [14], [21], [50].

Portability. The modifications CoinDef has made for AST profile generation and enforcement are minimal, numbering fewer than 150 lines in the V8's code base and two self-contained files for logging and validation, accounting for 500+ lines of C++ code. These files containing CoinDef's hooks remain almost consistent across ten major Electron versions evaluated for CoinDef (Tab. 2), simplifying porting and reducing maintenance and compatibility efforts.

6. Limitations

Code Coverage. As observed in § 4.1, CoinDef will incur false positives in the security-first mode due to unlearned AST profiles from remote resources. It is well understood that no single security system can promise full code coverage during the dynamic profiling of extensive programs, meaning some code may not be profiled. Using the hybrid profiling approach discussed in § 3.4, the AST profiles we collected support 20 applications with a testing code coverage of 87.03% on average, with only incurring a few false positives. We consider further increasing the code coverage for CoinDef as an orthogonal problem. CoinDef can benefit from the recent research in debloating and fuzzing web applications [32], [33], [51], [52] for expanded coverage.

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

7. Related Work

Code Injection Mitigation. Security researchers have extensively studied code injection vulnerabilities in the web ecosystem over the past decade. Prior solutions, such as dynamic taint analysis [11]–[13], [15], achieve effective mitigation by instrumenting the browser engine to track data flows. However, these approaches often incur significant runtime overhead and are challenging to maintain due to the frequency of browser updates. Other efforts [16], [17], attempt to replace the dangerous `eval` with "safe" `eval` by modifying the way to invoke `eval`. While useful, this approach does not extend well to the Electron framework due to its broad attack surface, as discussed in § 4.3, and lacks tamper-resistance as they are in the same privilege layer as attackers for Electron applications. Additionally, whitelist-based solutions [14], [50], [53] enforce AST integrity to prevent web-based code injection, but they enforce policies before scripts are parsed, incurring high runtime overhead. These methods also lack the execution context needed to counter mimicry attacks in Electron applications, limiting their effectiveness in complex cross-environment applications. CoinDef stands apart from existing solutions for two main reasons. First, it maintains the AST's structural integrity along with its running context, making it effective against mimicry attacks that other defenses miss. Second, CoinDef works directly within the interpreter engine, allowing it to protect both native and web environments together, covering all code injection scenarios in Electron applications with negligible runtime overhead. This combined approach makes CoinDef particularly effective for securing applications with mixed web and native elements.

JavaScript Security. Various program analysis techniques have been developed to analyze JavaScript programs and uncover vulnerabilities. Symbolic analysis-based solutions [54]–[59] are commonly employed to detect *prototype pollution* vulnerabilities by examining possible object inheritance issues and property changes. Dynamic analysis-based methods [44]–[46], [60]–[62] focus on instrumenting the browser engine to gather runtime traces of JavaScript code, providing insights into potential vulnerabilities through behavior tracking. Although these techniques are effective for analyzing JavaScript behavior in traditional web environments, they cannot be directly applied to Electron applications for defending against code injection, due to the unique attack surfaces and shared contexts present in these hybrid applications.

Electron Security. Recent studies highlight the security risks in applications built on the Electron framework. Xiao et al. [20] showed how shared contexts in Electron could escalate XSS attacks to severe RCE incidents. They developed XGuard to prevent RCE, but it addresses only the symptoms, not the root cause of code injection attacks. Jin et al. [21] studied vulnerabilities in the UI components of Electron applications, proposing DomTyping to enforce DOM integrity, which effectively prevents code injection through DOM modifications but doesn't address dynamic code execution or *client-side-redirect* issues. Ali et al. introduced Inspectron [22], designed to identify misconfigurations in Electron applications. However, even with proper configurations, attackers can exploit Electron vulnerabilities [19] to gain privileged access. In contrast, CoinDef directly targets the root cause of RCE: code injection in Electron applications. It provides comprehensive protection and covers both dynamic code execution and DOM manipulation scenarios, making it resilient against a broader range of code injection threats.

8. Conclusion

In this paper, we introduce CoinDef, an advanced security tool designed to prevent code injection attacks in Electron applications. CoinDef enforces the AST structural integrity with contextual information at all code execution points within the language interpreter. Our evaluation demonstrates that CoinDef effectively mitigates various code injection forms, including DOM manipulations and JavaScript's dynamic code execution APIs, preventing potential remote code execution exploits. Additionally, CoinDef operates with negligible runtime overhead and minimal false positives, offering both security-first and usability-first modes to accommodate varying security and usability needs. We release the source code of CoinDef and instructions to use it to support further research and development on *https://github.com/ian7yang/CoInDef*.

9. Acknowledgments

We thank the anonymous reviewers and our shepherd for their helpful and informative feedback. This material was supported in part by National Science Foundation (NSF) under grants No. CNS-2229876; the Office of Naval Research (ONR) under grants N00014-17-1-2179; the Defense Advanced Research Projects Agency (DARPA) under contract N66001-21-C-4024 Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, ONR, or DARPA.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] F. Xiao, Z. Yang, J. Allen, G. Yang, G. Williams, and W. Lee, "Understanding and mitigating remote code execution vulnerabilities in cross-platform ecosystem," in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, US, Nov. 2021, pp. 2975–2988.

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

[22] M. M. Ali, M. Ghasemisharif, C. Kanich, and J. Polakis, "Rise of inspectron: Automated black-box auditing of cross-platform electron apps,"

in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[23] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, Charleston, South Carolina, Jan. 2006.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[44] J. Allen, Z. Yang, M. Landen, R. Bhat, H. Grover, A. Chang, Y. Ji, R. Perdisci, and W. Lee, "Mnemosyne: An Effective and Efficient Postmortem Watering Hole Attack Investigation System," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Nov. 2020, pp. 787–802.

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

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

[47] J. Allen, Z. Yang, F. Xiao, M. Landen, R. Perdisci, and W. Lee, "Webrr: A forensic system for replaying and investigating web-based attacks in the modern web," in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

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

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

[50] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, "Cspautogen: Black-box enforcement of content security policy upon real-world websites," in *Proceedings of the 23rd ACM Conference on*

Computer and Communications Security (CCS), Vienna, Austria, Oct. 2016, pp. 653–665.

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

[52] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities," in *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2023.

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

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

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

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

[57] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. N. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability," in *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2023.

[58] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, "{Arcus}: Symbolic root cause analysis of exploits in production systems," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Conference, Aug. 2021.

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

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

[61] R. P. Kasturi, Y. Sun, R. Duan, O. Alrawi, E. Asdar, V. Zhu, Y. Kwon, and B. Saltaformaggio, "Tardis: Rolling back the clock on cms-targeting cyber attacks," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, Virtual Conference, May 2020.

[62] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper presents a defense mechanism, CoinDef, to mitigate code injection attacks in Electron applications. The mechanism enforces structural integrity of Abstract Syntax Trees using a combination of hybrid profiling and runtime enforcement.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) The paper creates a new tool to enable future science. A significant amount of engineering went into the development of CoinDef, and the authors make the implementation publicly available, thus enabling new research to further evaluate and improve the system.
2) The paper provides a valuable step forward in an established field. The protection of the integrity of Abstract Syntax Trees has been understood as a useful defense mechanism for a while, and this paper builds a valuable instantiation of this technique for the important class of Electron applications.

A.4. Noteworthy Concerns

1) The evaluation of the usability of CoinDef could be further improved: usability has only been evaluated through use by the authors.