Mockingbird: Efficient Excessive Data Exposures Detection via Dynamic Code Instrumentation

Chenxiao Xia[13], Jiazheng Sun[2], Jun Zheng[13*], Yu-an Tan[13], Hongyi Su[13]
[1]Beijing Institute of Technology, Beijing, China
[2]Fudan University, Shanghai, China
[3]Key Laboratory of Intelligent Networking Technology for Social Governance,
Ministry of Industry and Information Technology
Emails: chenxiao_xia@bit.edu.cn, jzsun24@m.fudan.edu.cn, zhengjun@bit.edu.cn, tan2008@bit.edu.cn, henrysu@bit.edu.cn

***Abstract*—Excessive Data Exposure (EDE), where an API returns redundant data to the client beyond what is required for its functionality, has become a pervasive and severe security threat. However, automated detection techniques for such vulnerabilities remain underdeveloped, and existing methods, particularly black-box fuzzing, face significant bottlenecks in terms of accuracy and efficiency. To address these challenges, we propose Mockingbird, an automated detection tool based on a statically-assisted dynamic analysis approach. The tool leverages the JavaScript Proxy mechanism for efficient dynamic taint tracking to precisely identify the dangling data that is transmitted from an API response to the client but never consumed by any expected functionality, such as UI rendering or state management. Furthermore, to tackle the lack of a standardized benchmark in this domain, we have constructed and open-sourced EDEBench, the first persistent benchmark for EDE evaluation, comprising 8 popular open-source web projects built on diverse modern technology stacks. Experimental evaluation on EDEBench shows that, compared to the state-of-the-art, Mockingbird achieves an average F1-score improvement of 24.1% (Precision +15.8%, Recall +32.8%), enhances detection speed by nearly 20 times, and demonstrates broad applicability across all tested frameworks. These results provide a clear illustration of our tool's accuracy, applicability, and efficiency. The source code is available at https://github.com/NeoSunJZ/Mockingbird-JS.**

***Index Terms*—Excessive Data Exposure, API Security, Dynamic Code Instrumentation, Gray-box Testing**

I. INTRODUCTION

The security of our data is under persistent threat from Excessive Data Exposure (EDE) vulnerabilities [1]! API endpoints often return redundant data beyond the actual business requirements of the client, thereby delegating the responsibility of data filtering and trimming to the client-side application. This practice, which gives rise to a class of vulnerabilities known as EDE, often stems from a non-optimal engineering trade-off [2]. Faced with evolving requirements and tight delivery schedules, developers opt for expediency to enhance development velocity and circumvent the iterative costs of requirement changes. Indeed, this prioritization of short-term development convenience over long-term security resilience has propelled EDE to rank as the third most critical risk in the OWASP [3] API Security Top 10 for 2023.

*Corresponding author: zhengjun@bit.edu.cn

Despite the persistent threat of EDE, automated detection methods remain scarce. The majority of research focusing on static analysis [4]–[8], dynamic analysis [9]–[15], or hybrid solutions [16], [17] for JavaScript has paid insufficient attention to EDE vulnerabilities. Furthermore, an empirical study by Calzavara et al. [18] revealed that the practical usability of such tools is critically lacking (only one of the 18 evaluated tools was deemed practically viable), indicating a severe shortage of tools capable of conducting broad security assessments on modern web applications. Pan et al. [19] ingeniously circumvented the dependency on JavaScript analysis by employing black-box fuzz testing, proposing the first tool, EDEFuzz, that could effectively detect this vulnerability. Their work established a clear test oracle based on the causal relationship between client-side rendering behavior and API response data: excessively exposed data should not trigger any changes in the Document Object Model (DOM). However, this complete reliance on external behavioral observation leads to significant and intractable false positive and false negative rates. Moreover, its test paradigm of per-field mutation combined with page reloads results in prohibitive time costs, with tests potentially taking hours or even days for complex APIs that often contain thousands of fields. These deficiencies collectively limit its practical applicability.

The primary cause of this predicament lies in the nature of modern web technologies. Websites developed with modern frameworks are typically built using tools like Webpack, a process that compresses, obfuscates, and restructures the source code. Not only are semantic identifiers such as variable and function names lost, but the original file and module boundaries are also destroyed. After hundreds of modules, now devoid of linguistic information, are bundled into one or a few massive files, any attempt to construct precise control-flow or call graphs via static analysis becomes virtually infeasible. However, treating the web client as a completely opaque black box also constrains a tool's potential. In reality, the browser can access and parse HTML and execute JavaScript during page loading. The failure to effectively leverage this available runtime information is a key reason for the accuracy and efficiency bottlenecks in black-box approaches like EDEFuzz.

To address these challenges, we introduce Mockingbird, a

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00247

novel EDE detection tool. Mockingbird employs a **Statically-Assisted Dynamic Analysis** approach. Specifically, we define the response data as the **Source** and a set of function calls representing Expected Client-side Data Consumption as the **Sink**. These sinks encompass not only functions that directly or indirectly cause DOM mutations but also other legitimate data persistence or state management operations, such as updating Web Storage (e.g., LocalStorage). Unlike traditional Taint Analysis, which aims to track whether tainted data reaches a sink, our method seeks to identify **Dangling Data**: data that originates from a source but is ultimately never consumed by any predefined sink function. Such data is considered potentially excessive because, despite being transmitted to the client, it serves no explicit, expected function, thus posing an unnecessary security risk and performance overhead.

Mockingbird identifies dangling data in two main stages. In the first stage, we perform dynamic data propagation tracking using the Proxies. We instrument the browser's native XMLHttpRequest and Fetch APIs to intercept the targeted asynchronous responses and automatically mark all fields within the response data as taint sources. When a tainted data field is used in a computation or assignment that produces a new derived variable, we recursively wrap this new variable with a Proxy as well. This allows the taint mark to be continuously propagated along the data flow until the data is either consumed by a predefined sink or is ultimately left dangling. In the second stage, we analyze the logs collected from the first stage to perform dangling data identification. Data that fail to reach any sink are identified as potential EDE.

By fully leveraging the web application's JavaScript code, Mockingbird can capture more complex and indirect data exposure scenarios compared to black-box methods that rely on observing UI behavior. Furthermore, Mockingbird requires only a minimal number of page refreshes, reducing the time complexity from O(n) for fuzzing-based tools to O(1) and thereby substantially decreasing the testing time.

We have observed a persistent lack of a standardized evaluation benchmark in this domain. Existing studies often rely on evaluating live, online websites, which presents a critical challenge: vulnerabilities, once discovered and reported, are quickly patched. The dynamic and ephemeral nature of online environments prevents subsequent research from reproducing experiments on the same baseline, thus hindering fair comparisons between different methods. To overcome this issue, we have constructed and open-sourced EDEBench, a persistent web benchmark dataset. This dataset features 8 real-world open-source applications built with various mainstream web development frameworks, with multiple typical front-end interaction scenarios preserved in a static state. EDEBench aims to provide a stable and consistent evaluation platform for research in this field. Compared to the SoTA, experiments on EDEBench show Mockingbird achieves an average F1-score improvement of 24.1% (Precision +15.8%, Recall +32.8%), enhancing detection speed by nearly 20 times.

Our contributions are summarized as follows:

- We propose a novel EDE detection method based on the identification of Dangling Data. By tracing the propagation paths of API response data, we identify data that is never consumed for any expected functionality, effectively overcoming the limitations of existing methods that rely on external behavioral observation.
- We design and implement Mockingbird, an efficient and automated detection tool. It employs a statically-assisted dynamic analysis approach to precisely locate Dangling Data by analyzing client-side runtime data flows, achieving significant improvements in both accuracy and efficiency over traditional fuzz testing paradigms.
- We construct and release EDEBench, the first standardized benchmark for EDE evaluation. It comprises multiple persistent, real-world web applications, addressing the critical issues of reproducibility and fair comparison that have challenged previous research relying on dynamic online websites.

II. BACKGROUND

A. Excessive Data Exposure

Excessive Data Exposure is a pervasive API security vulnerability [20], characterized by an API endpoint returning more data than is strictly necessary for the client application to fulfill its intended functionality. This vulnerability typically originates from backend development practices where, for instance, to simplify processing logic or accommodate diverse clients, an API is designed to return a complete database model or business object, thereby delegating the responsibility of data filtering to the client-side.

While this "fat response" pattern may offer short-term convenience in development, it creates a significant risk of inadvertent sensitive information disclosure. The technical barrier for exploiting such vulnerabilities is exceedingly low, as the exposed data is often readily available in a structured format (e.g., JSON), requiring an attacker to merely inspect the network traffic. For example, a vulnerability in the Microsoft CMT submission system, caused by an API (`odata/ConferenceName/Submissions/(ID)`) that excessively exposed the StatusId field, allowed authors to prematurely access the review outcomes of their papers. This incident compromised the confidentiality of the review process for numerous top-tier academic conferences, including IJCAI and ICME. Precisely determining which data fields in an API response are never effectively utilized by the client constitutes the core challenge in automated EDE detection.

B. Browser Automation and Debugging Protocols

The dynamic and complex nature of modern web applications necessitates that any effective security analysis be conducted within a real browser environment. Furthermore, to inject our analysis logic, a mechanism is required to modify the application's code before it is executed by the browser.

The Chrome DevTools Protocol (CDP) [21] provides the critical low-level capabilities for this purpose. It not only allows external tools to control page interactions but, more importantly, offers powerful features for intercepting network

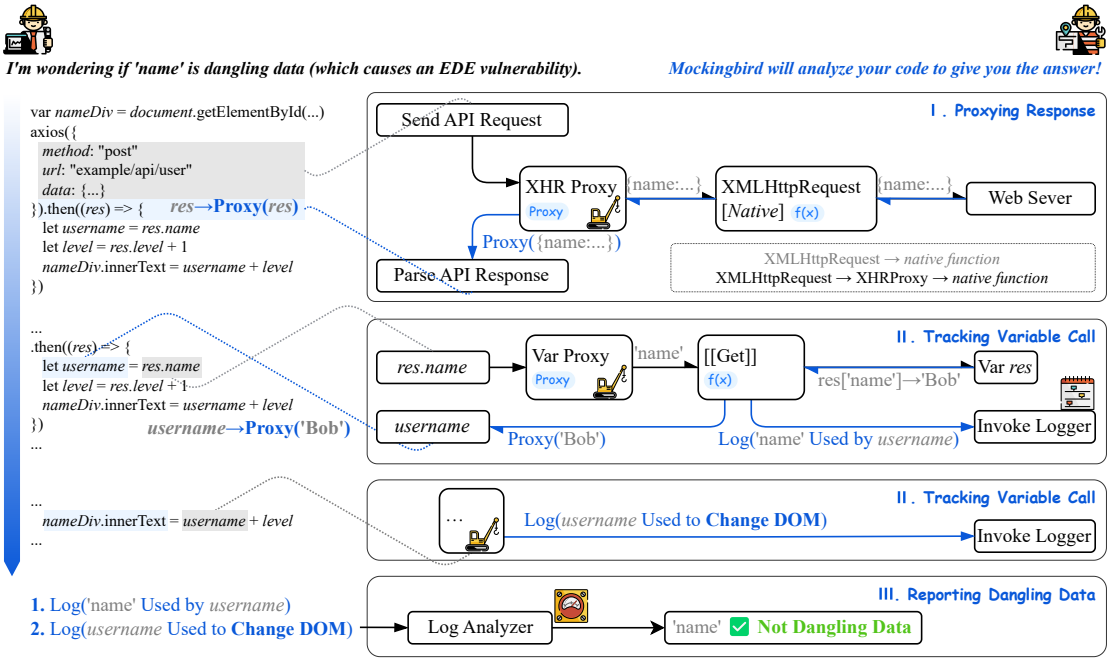

Fig. 1. **An overview of Mockingbird's approach to identifying dangling data by tracing usage chains.** The depicted case illustrates how `res.name` is determined to be non-dangling: I. The API response is proxied to monitor data access. II. Mockingbird tracks a complete usage chain: the tainted data `res.name` is first assigned to the `username` variable, which is subsequently used in a DOM operation (a predefined sink). III. Because `res.name` is effectively consumed via this chain, the Log Analyzer determines it is not dangling data.

requests and modifying their response content. This means we can capture an application's JavaScript files as they are being loaded by the browser, perform code instrumentation, and then deliver the modified code to the browser for execution.

High-level automation frameworks built upon CDP, such as Playwright [22], encapsulate these low-level capabilities into stable and user-friendly APIs. In our approach, Playwright is used not only to automate user interactions for triggering target APIs but, more critically, serves as the foundational mechanism for implementing our code instrumentation. It is through this mechanism that we are able to seamlessly inject our data access tracking logic into the original code.

C. JavaScript Metaprogramming: The Proxy Object

Accurately determining the actual use of a data field hinges on the ability to monitor its access at runtime. The Proxy object, introduced in ECMAScript 6 (ES6) [23], provides a revolutionary metaprogramming mechanism for this task.

It allows for the creation of a proxy for a target object, and through the deployment of Traps, it can intercept and redefine fundamental operations on that object, such as property reads (`get`) and writes (`set`). Compared to traditional instrumentation techniques that involve extensive source code rewriting at build-time, the Proxy offers a more native and flexible way to monitor the usage behavior of data objects at runtime. This capability to intercept operations at the property-access level is the core mechanism that enables our fine-grained taint tracking and the identification of whether a data field has been accessed (detailed in Section III.C).

III. OUR APPROACH

We now provide an overview of Mockingbird and detail the design of its components. We depict the main workflow of Mockingbird in Figure 1.

A. Overview

1) Definitions: **a. Source and Sink**: The response data from an asynchronous request is defined as the Source. A set of function calls representing Expected Client-side Data Consumption is defined as the Sink. The specific scope of these sinks will be detailed in Section III.E. **b. Dangling Data**: A field originating from an API response (the Source) is considered Dangling Data if, throughout its client-side lifecycle, its propagation path never reaches any of the predefined sinks that represent critical client-side behaviors. Such data serves as a direct candidate for constituting an EDE vulnerability.

2) Preparatory Stage - API Endpoint Identification & UI Interaction Capture: A prerequisite for our methodology is the identification of target API endpoints to be tested, the process of which falls beyond the scope of this paper. In practice, we identify these targets by manually inspecting the network traffic during web application interactions. To ensure that the subsequent analysis phase can be fully automated, Mockingbird provides a browser extension that records and generates replayable UI automation scripts. These scripts, which reliably

trigger the target API requests, form the foundation for the subsequent dynamic data propagation tracking stage.

3) Stage I - Dynamic Data Propagation Tracking: To track the source data, Mockingbird overwrites the browser's global `XMLHttpRequest` and `Fetch` objects to intercept all asynchronous network requests. Upon intercepting a response from a target API, the returned data object is immediately wrapped by a **Var Proxy** object before being passed to the subsequent business logic.

This Proxy object is the core of our dynamic tracking mechanism. When the client-side code reads any property of the proxied object via a get operation, a predefined trap is triggered. This trap meticulously logs the access information, the execution context, and the function call stack for the current property. However, the Proxy mechanism cannot inherently track independent variables that result from new computations or assignment expressions involving the accessed property. To ensure the taint continues to propagate along the data flow, Mockingbird must dynamically apply **Supplementary Instrumentation** to these newly derived variables, transforming them into tracked Proxy objects as well. This process is iterative. Since each round of supplementary instrumentation involves modifying the source code, the UI automation script must be replayed. This cycle continues until all relevant derived variables are correctly wrapped by Proxy objects. The tracking stage concludes once the client-side execution path triggered by the asynchronous response stabilizes.

4) Stage II - Dangling Data Identification and Analysis: Following the completion of the tracking in *Stage I*, Mockingbird performs an offline analysis of the collected logs. The analysis process iterates through each field in the API response source, reconstructing its complete propagation path within the client-side based on the logs, up to the final expressions or statements it participates in. By examining whether these terminal consumption points belong to any of the predefined, legitimate sinks, Mockingbird can make a determination. If a field's propagation path never reaches any sink, or if the logs indicate it was never accessed at all, it is identified as dangling data and reported as a potential EDE vulnerability.

B. API Response Interception

To initialize API response data as taint sources, it is essential to intercept the entry points of all asynchronous network requests within the browser. To this end, we have designed and implemented a unified interception and data-wrapping workflow targeting the two core mechanisms in modern browsers: `XMLHttpRequest` (XHR) and the `Fetch` API.

1) XHR Interception: Mockingbird intercepts XHR by overriding its global constructor. Whenever the application code instantiates an XHR object, an **XHR Proxy** instance is returned instead. This proxy internally manages a native XHR object and listens for its state changes. When a request to a target API completes successfully, the proxy first creates a **Var Proxy** for the response data (e.g., `responseText`), as described in Section III.C, before passing it to the application's callback functions (e.g., `onload`).

2) Fetch API Interception: For the Fetch API, Mockingbird overrides the global fetch function. Similar to the XHR interception, the **Fetch Proxy** function calls the native fetch and registers a pre-processing handler for the returned Promise via `.then()`. This handler executes before the application's own callback chain. Once the Promise resolves to a Response object, our handler creates a **Var Proxy** for it.

Through this interception mechanism, which covers both predominant APIs, we ensure that all target response data is brought under monitoring from the moment it enters the client-side execution environment, laying a solid foundation for the subsequent Proxy-based dynamic taint tracking.

C. Data Access Tracking via Var Proxy

The core of EDE detection lies in precisely determining whether API response data is effectively utilized by the client. In JavaScript, this usage behavior is primarily manifested as the reading of object properties, an operation defined by the `[[Get]]` internal method in the language specification. Since the `[[Get]]` method cannot be directly accessed or overridden, we employ the Proxy object, introduced in ES6, as our core monitoring mechanism.

By creating a **Var Proxy** for a data object, we leverage its **Get Trap** to effectively intercept all `[[Get]]` operations on its properties. When any tracked property is accessed, our Get Trap is triggered. At this point, we record the details of the access event, capture the current call stack, and store this information in a log for subsequent analysis.

Algorithm 1 Recursive Variable Proxy

Input: $data$: variables to be proxied
Output: $data_{\text{proxy}}$: proxied variables
function VARIABLEPROXY
  if isProxied($data$) **then**
    $data_{\text{proxy}} \leftarrow$data
  end if
  $handler \leftarrow$ createGetTrapHandler()
  if isObject($data$) **then**
    $data_{\text{proxy}} \leftarrow$ new Proxy($data, handler$)
    for each $child$ **of** $data_{\text{proxy}}$ **do**
      $child \leftarrow$ VariableProxy($child, handler$)
    end for
  else if isPrimitiveType($data$) **then**
    $wrapperObj \leftarrow$ convertToObject($data$)
    $data_{\text{proxy}} \leftarrow$ new Proxy($wrapperObj, handler$)
  end if
end function

An inherent challenge is that the Proxy mechanism is only applicable to objects and cannot wrap JavaScript's Primitive Values, such as strings or numbers, because direct access to them does not trigger a `[[Get]]` operation. However, we observed that the JavaScript runtime performs autoboxing in certain scenarios (e.g., when accessing a property or calling a method on a primitive, such as `'hello'.length`), temporarily converting the primitive into its corresponding wrapper object to perform the operation. Inspired by this behavior, we devised a Proactive Boxing strategy. Before applying the proxy to the API response data, we first traverse the data and explicitly convert all primitive values into instances of

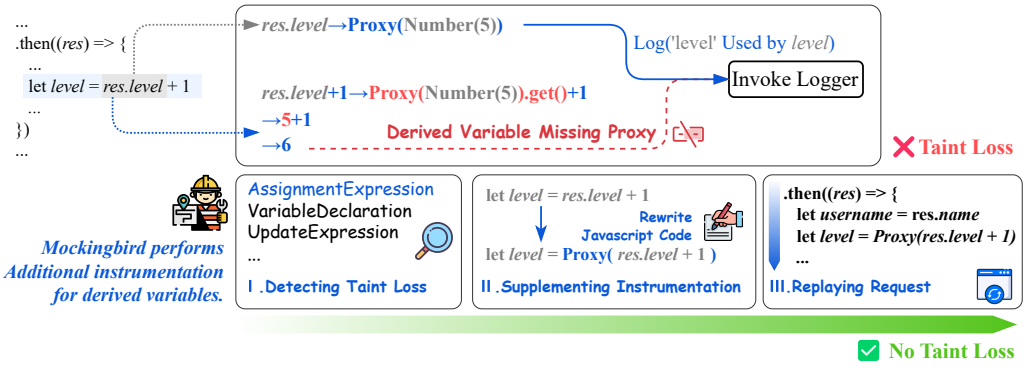

Fig. 2. **Mockingbird's mechanism for preventing taint loss.** To address taint loss from derived variables (e.g., `level` in `let level = res.level + 1`), Mockingbird first detects these un-proxied variables, then rewrites the code to wrap them in a new Proxy, and finally replays the execution. This cycle repeats until all data-flow paths are instrumented and no taint loss is detected.

their corresponding wrapper objects (e.g., `String`, `Number`). Through this preprocessing step, all data, regardless of its original type, is unified into an object form. This allows the subsequent Proxy mechanism to be applied uniformly across all data fields, ensuring that any property access can be precisely captured. This process is detailed in Algorithm 1.

D. Supplementary Instrumentation for Derived Variables

While Proxy can effectively monitor direct access to response data, the tracking chain is severed when a tracked value is used in a computation and assigned to a derived variable. We term this phenomenon **Taint Loss**.

For a concrete illustration, consider the statement `newVar = proxiedVar + 1`. Assuming `proxiedVar` is a proxied `Number` object, the JavaScript runtime implicitly converts it to a primitive value before performing the addition. Consequently, the derived variable `newVar` is assigned a pure, unproxied `number`, causing a break in the data flow tracking.

To address this problem, we propose an iterative strategy for **Supplementary Instrumentation**. The core idea is to analyze and identify the code locations causing taint loss after each execution, and then modify the code to apply instrumentation to the newly created derived variables. This **Execute-Analyze-Rewrite-Replay** cycle continues until a complete execution pass identifies no new taint loss points. The core principle of this process is illustrated in Figure 2.

1) Taint Loss Scenario Identification: We identified that the root cause of taint loss lies in assignment operations: when a monitored variable is part of a computation whose result is assigned to a new variable or used to update itself, the tracking link is broken. Therefore, our rewriting strategy focuses on the following three Abstract Syntax Tree (AST) node types that are directly related to variable creation and updates:

- AssignmentExpression (e.g., `x += y`)
- VariableDeclaration (e.g., `let x = y`)
- UpdateExpression (e.g., `x++`)

Notably, expressions that do not assign their result to a named variable (such as a standalone binary operation `a + b`) are outside the scope of our rewriting, as they do not create a state that requires persistent tracking.

2) Source Code Rewriting: Since scripts that have already been loaded and executed cannot be modified in-flight, our strategy is to record all code locations requiring supplementary instrumentation at the end of each iteration, and then reload the page. During the next page load, Mockingbird intercepts the corresponding JavaScript resource request, performs static analysis on the fetched source code to parse its AST, applies supplementary instrumentation at the targeted locations, and returns the modified script. Finally, the UI automation script recorded in the preparatory stage is used to re-trigger the asynchronous request and execution flow. Our code instrumentation algorithm is presented in Algorithm 2.

This process is repeated cyclically: in each iteration, new derived variables are brought under monitoring, which in turn may generate deeper levels of derivations. The termination condition for this loop is a full execution cycle in which no new taint loss points are discovered.

3) Avoiding Taint Spread: When applying supplementary instrumentation to applications built with modern web frameworks, a key challenge is to avoid unnecessary **Taint Spread**. If an expression within a generic library function or internal framework utility is directly proxied, it can cause all data flowing through that expression to be indiscriminately wrapped in a proxy. For instance, a common rendering function, after being instrumented because it processed a tainted variable A, might subsequently and incorrectly proxy an unrelated variable B.

To counter this challenge, we designed a context-aware instrumentation strategy. Its core principle is to proxy the output of an expression if and only if at least one of its inputs is already a tracked object. This process is illustrated in Figure 3.

We perform this validation at runtime: when instrumenting a candidate expression `E`, the proxying function receives all of its statically-analyzed input variables as arguments. It then inspects the types of these arguments in the current calling

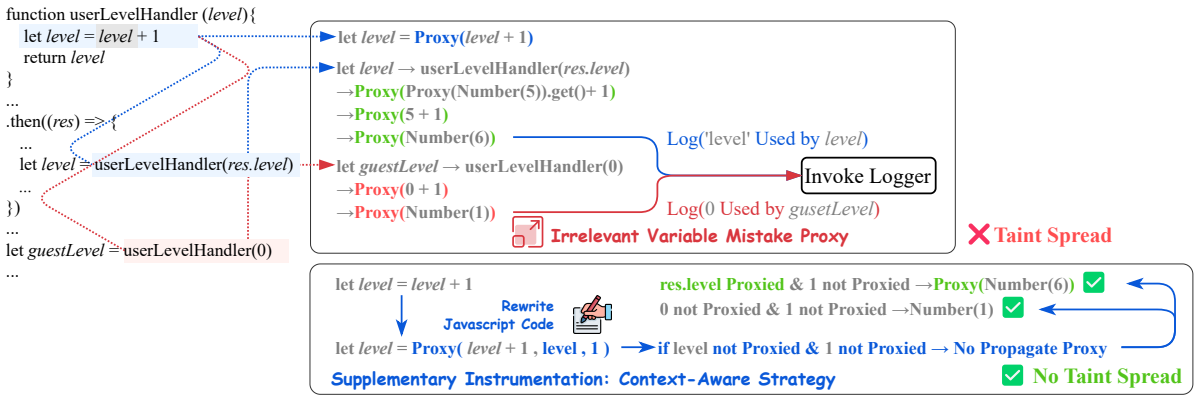

Fig. 3. **Mockingbird's context-aware instrumentation for preventing taint spread.** To avoid taint spread from generic functions, this strategy proxies an expression's output only when an input is already a proxy. As illustrated, the output of `userLevelHandler(res.level)` is proxied, whereas the output of `userLevelHandler(0)` is ignored, effectively preventing unnecessary proxying.

Algorithm 2 AST-based Source Code Re-writer

Input: *node*: An AST node.
Output: *node'*: The modified AST node.
function TRANSFORMNODE(*node*)
  if *node* **is an** AssignmentExpression (*target* = *expression*) **then**
    newExpression ← `varProxy`(*expression*)
    return ASTNode(*target* = *newExpression*)
  else if *node* **is a** VariableDeclaration (**let** *target* = *expression*) **then**
    newExpression ← `varProxy`(*expression*)
    return ASTNode(**let** *target* = *newExpression*)
  else if *node* **is an** UpdateExpression **then**
    parentStmt ← GetParentStatement(*node*)
    if *parentStmt* **is an** ExpressionStatement **then**
      node' ← ConvertToEquivalentAssignment(*node*)
      return *node'*
    else
      if operator in *node* **is prefix then**
        node' ← CreateSelfUpdatingAssignmentForm(*node*)
        return *node'*
      else if operator in *node* **is postfix then**
        variable ← GetOperand(*node*)
        op ← GetBaseOperatorFromUpdate(*node*)
        inverseOp ← GetInverseOperator(*op*)
        assignmentPart ← ASTNode(*variable* = *variable* op 1)
        valuePart ← ASTNode(*variable* inverseOp 1)
        node' ← SequenceExpression(*assignmentPart*, *valuePart*)
      end if
    end if
  end if
  return *node'*
end function

context. The proxy operation is executed only after confirming that at least one input value is an instance of our proxy, thereby preventing taint spread.

E. Dangling Data Identification and Analysis

Following the tracking stage, Mockingbird will have collected extensive logs regarding the usage of API response data (provided by the Proxy mechanism detailed in Section III.C). We identify dangling data for each field from the API response based on two key pieces of information: **a.** The field's complete propagation path; **b.** The expression and statement context at the moment the field was accessed.

The core of this identification process lies in concretizing the definition of a **Sink**. In Mockingbird, a Sink is concretely defined as a code point that performs any of the following key client-side behaviors:

a) **DOM Operations or Web Storage**: The field (or a value derived from it) is used as an argument to a DOM manipulation API (e.g., `document.write`) or a Web Storage API (e.g., `document.cookie`).
As different modern front-end frameworks [24]–[28] (e.g., React, Vue) often maintain their unique virtual DOM or rendering engines, for applications built with them, analysts should conduct empirical framework-specific analysis and, when necessary, supplement the Sinks with framework-specific re-render functions.
b) **Critical Function Calls**: The field is passed as an argument to a function that is critical to the application's functionality. This includes, but is not limited to:
- Functions that execute page navigation or initiate new network requests.
- Other core business logic functions pre-marked by an analyst.

c) **Control-Flow Decisions**: The field is used as the condition in a conditional statement (e.g., `if`, `switch`) or a loop (e.g., `for`, `while`), directly influencing the application's execution flow and final presentation.

We classify each field into one of three categories, with the specific classification rules and decision logic as follows:

a) **Unused Data:** Fields that were never accessed during the entire tracking period. These fields are **clearly dangling**. At the conclusion of the tracking period, any source field that never triggered a get trap within our Var Proxy monitoring is directly classified as Unused.
b) **Necessary Data:** Fields that were accessed, and their propagation path explicitly reached one or more predefined Sinks. These fields are **not considered dangling**. For each accessed field, Mockingbird examines all of its

execution traces. If any single trace satisfies any of the Sink rules above, the field is classified as Necessary.

c) **Potentially Dangling Data:** Fields that were accessed, but their propagation paths did not reach any known Sink. These fields represent **a more subtle form of dangling data**; their propagation paths exist within the client-side code, but they do not contribute to any observable, critical behaviors. Any field that was accessed during the tracking period but fails to satisfy the rules for Necessary Data is assigned to this category. Typical scenarios include:
   - Usage in temporary internal computations whose final results do not impact the UI, storage, or control flow.
   - Usage solely for logging or debugging purposes (e.g., `console.log`).
   - Involvement in a conditional check where the corresponding branch is never activated in any of the test cases.

IV. EXPERIMENTAL EVALUATION

Our experimental evaluation is designed to answer the following research questions:

- **(RQ-1) Accuracy:** What are the Precision and Recall of Mockingbird in identifying excessively exposed data?
- **(RQ-2) Applicability:** Can Mockingbird be applied to a range of modern web applications that use different mainstream front-end frameworks and architectures?
- **(RQ-3) Efficiency:** How much manual effort and computation time are required to use Mockingbird?

A. Benchmark: EDEBench

To accurately answer RQs 1-3, we need to evaluate Mockingbird and other comparable works. However, a key obstacle impeding progress in the field of EDE detection is **the absence of a standard evaluation benchmark**. This situation hinders the quantitative evaluation of different testing tools and makes fair comparisons between methods difficult to achieve. As mentioned in our related work, existing research (e.g., EDE-Fuzz) typically conducts tests directly on dynamically changing, live web applications. While this in-the-wild evaluation approach can demonstrate a method's practical potential, its inherent uncertainty presents two critical challenges:

a) **Reproducibility Challenge**: The API endpoints and front-end logic of online applications are subject to frequent changes beyond the control of researchers, and discovered EDE vulnerabilities may be patched at any time. This leads to a significant transience problem for any experiment based on live targets, making their results difficult to reproduce and failing to provide a stable cornerstone for subsequent research.

b) **Ground Truth Validity Challenge**: For complex APIs that return a massive number of fields, establishing a ground truth by relying on black-box methods like external behavior observation is not only prohibitively costly and error-prone but also fails to guarantee the accuracy and consistency of the annotations. This ultimately compromises the validity of the evaluation conclusions.

To address these issues, we have designed and constructed **EDEBench**, an open-sourced benchmark specifically for EDE detection research. The construction of EDEBench followed a rigorous, multi-stage process to ensure its **representativeness, diversity, and reproducibility**:

Stage I: Candidate Project Pool Screening. We searched GitHub using keywords for mainstream front-end frameworks (e.g., `Vue`, `React`) and application scenarios (e.g., `admin management`, `e-commerce`). The screening criteria were: (1) Over 5,000 GitHub stars to ensure project popularity and impact; however, for frameworks with fewer popular projects, we included those with over 1,000 stars. (2) Code commits within the last year to ensure they represent modern development practices. (3) A clear front-end/back-end separated architecture. This stage yielded a pool of over 50 candidate projects.

Stage II: Project Feasibility Assessment and Final Selection. We then evaluated each candidate project individually, primarily assessing: (1) whether it had clear deployment documentation and configuration to ensure a reproducible experimental environment, and (2) whether its core functionalities could be run stably. This round of screening reduced the pool to approximately 20 projects. Finally, we meticulously selected **8** projects to form EDEBench, with the ultimate principle of maximizing the diversity of technology stacks and application scenarios. These projects have an average of over 10,000 GitHub stars, cover mainstream front-end technologies including `Native JS`, `Vue`, `React`, `Svelte`, `jQuery`, and `Angular`, and encompass diverse application scenarios such as admin management (`RuoYi`), e-commerce (`ArialShop`), news portals (`AngularHN`), media applications (`Koel`), and Q&A communities (`Answer`).

Stage III: Test Case Extraction and Encapsulation. To ensure that our tests were independent and focused, we did not use the entire projects as test subjects. Instead, for each project, we identified 3-5 core business scenarios (e.g., user permission management, online audio playback, article list rendering). For each scenario, we identified its core API endpoint and corresponding client-side consumption logic, encapsulating it into an independent test case. This approach ensures our evaluation is centered on the application's core functionalities rather than peripheral or randomly chosen features.

Stage IV: White-Box Ground Truth Labeling. The core value of a benchmark lies in the reliability of its ground truth annotations. To this end, we assembled an expert panel of five experienced web development and security engineers to perform manual labeling using a source-code-based, white-box approach. All members received unified training before labeling to reach a strict consensus on the definition of Necessary Data. Before commencing full-scale labeling, we conducted a pilot labeling of two test cases and calculated the initial inter-rater reliability to ensure consistency. Full-scale work began only after the Kappa coefficient value exceeded 0.85. During the labeling process, annotators located the API's client-side call sites and meticulously traced every access and propagation of the response data throughout its complete execution path, finally using cross-validation to ensure the accuracy of every

single annotation.

B. Experimental Procedure

We evaluate each test case in EDEBench using Mockingbird according to the following three steps. This process is consistent with the tool's workflow outlined in Section III.A, and the detailed steps of the automated run are described below.

a) **Step I: Interaction Script Generation.** For each test case in EDEBench, the target API endpoint to be tested is predefined. We use Mockingbird's accompanying browser extension to manually perform the client-side interaction sequence (e.g., page loading, button clicks, form inputs) required to trigger the API. The extension records this process and automatically generates a replayable UI automation script. This script is subsequently used to replay requests during the supplementary instrumentation phase of data propagation tracking (see Section III.A.2).

b) **Step II: Running Mockingbird.** After the interaction script is generated, we launch Mockingbird to perform a fully automated dynamic analysis. The core of this stage is an iterative Execute-Analyze-Rewrite-Replay loop. In each iteration, Mockingbird will: (a) execute the script and track the propagation of known tainted data using Proxy objects; (b) identify the statement locations where taint loss occurred during that execution run (see Section III.D); (c) based on an AST analysis of these statements, perform supplementary instrumentation on the JavaScript source code to cover newly discovered derived variables; and (d) reload the page and replay the request. This iterative process continues until no new taint loss points are discovered within a complete execution cycle, ensuring the complete capture of data propagation paths. After the iteration converges, Mockingbird performs a final, stable run with the fully instrumented code. During this run, the system records a comprehensive and stable execution trace, including a detailed access log for every field in the API response and its derived variables, complete with the call stack and code context for each access.

c) **Step III: Result Analysis and Collection.** Mockingbird's analysis module parses the stable execution trace collected in the previous step. According to the rules defined in Section III.E, the analyzer reconstructs the client-side propagation path for each field from the API response and checks whether its final consumption points hit any predefined Sinks, such as DOM operations, critical function calls, or control-flow decisions. Finally, Mockingbird automatically classifies each field as Unused Data, Necessary Data, or Potentially Dangling Data.

For other related works, we followed their recommended experimental procedures and settings.

C. Evaluation Metrics and Baselines

1) Metrics: To answer the research questions, we define the following metrics:

Accuracy (RQ-1): We will adopt the standard metrics of **Precision**, **Recall**, and **F1-Score** to evaluate the accuracy of Mockingbird. These will directly measure the performance of our algorithm.

Efficiency (RQ-3): We will measure efficiency from the following three dimensions:

a) **Manual Setup Time (MST):** The total time required for a researcher to generate a replayable UI script for each test case.

b) **Automated Analysis Time (AAT):** The total time required for Mockingbird to complete all its supplementary instrumentation.

c) **Total Execution Time (TET):** The total time required to perform one complete vulnerability analysis. For Mockingbird, this includes the automated analysis overhead and the time for the final stable run to generate the log.

2) Baselines: To comprehensively evaluate the performance of Mockingbird, we selected the following baseline works for comparison:

a) **EDEFuzz:** As the only published automated detection tool specifically targeting web EDE vulnerabilities to date, it is an essential point of comparison. We acknowledge the fundamental methodological differences between EDEFuzz (a black-box fuzzer) and Mockingbird (a hybrid analysis tool). Therefore, the purpose of this comparison is not merely to determine superiority, but rather to reveal the trade-offs and differences between the two technical approaches in terms of accuracy, efficiency, and detection capabilities.

b) **Mockingbird(*S*):** To validate the necessity of our proposed iterative supplementary instrumentation strategy (Section III.D), we implemented a simplified version of Mockingbird. This version performs a one-time wrapping of the API response object with a Proxy but does not track new variables derived from it. By comparing it with the full version of Mockingbird, we can quantify the critical role of the iterative analysis in discovering complete data propagation paths and avoiding false negatives.

We also conducted an in-depth investigation into the feasibility of adapting existing JS information-flow analysis frameworks [29]–[33], [35]–[38] for EDE detection. However, the design goals of these tools are in fundamental conflict with EDE detection: **1. Different Analysis Goals:** They aim to track data from untrusted sources (e.g., URL parameters) to dangerous sinks (e.g., innerHTML) to detect vulnerabilities like XSS. In contrast, EDE detection aims to track data from trusted sources (API responses) to determine if it is not consumed by any legitimate sink (e.g., UI rendering). The analysis logic is effectively reversed. **2. Different Data Models:** XSS analysis primarily focuses on string-type data, whereas EDE vulnerabilities involve arbitrary data types (numbers, booleans, nested objects), which existing tools are ill-equipped to handle directly. We provide a detailed discussion of this in Chapter V. **Given these fundamental differences, forcibly adapting these tools would not only require a massive engineering effort but also fail to provide a fair comparison**. Therefore,

we believe our current selection of baselines is the most reasonable and convincing.

D. Results

1) RQ1. Accuracy: To answer RQ-1, we quantitatively evaluated the accuracy of Mockingbird, the baseline method EDEFuzz, and an ablation version, Mockingbird(*S*), on the eight projects of the EDEBench. We report the Precision, Recall, and F1-Score for each tool on every project in Table I, respectively, and summarize the overall average performance.

Overall Performance: The experimental results show that Mockingbird demonstrates a significant advantage in overall performance. Compared to EDEFuzz, Mockingbird's precision, recall, and F1-score increased by 15.8, 32.8, and 24.1 percentage points, respectively. This comprehensive and robust performance improvement strongly validates the effectiveness and advancement of our method in the EDE detection task.

Precision Analysis: Mockingbird's high precision is primarily attributed to its fine-grained semantic analysis of client-side data flow. This approach accurately distinguishes data fields that, while not directly triggering DOM changes, are legitimately consumed in internal state management. In contrast, EDEFuzz, which relies entirely on external DOM mutations as its test oracle, has inherent limitations in its judgment logic. Although it achieves reasonable precision in applications where data usage is strongly correlated with UI presentation (e.g., `Navidrome` and `AngularHN`), EDEFuzz generates a large number of false positives as soon as data consumption becomes indirect (e.g., a field is used to update an internal state that is not immediately rendered). This is because it cannot observe a direct DOM impact, leading it to misclassify necessary data as over-exposed.

Recall Analysis: Mockingbird's performance in terms of recall is particularly outstanding, thanks to its white-box analysis method that effectively resists interference from irrelevant DOM changes. Specifically, in the `Koel` and `ArialShop` projects, EDEFuzz's recall was significantly low. We found this was mainly due to periodic or non-API-data-driven DOM updates in these applications (e.g., the scrolling of the music progress bar in `Koel`, or the timed appearance of a promotional pop-up in `ArialShop`). This background noise unrelated to the API response severely interfered with EDEFuzz's differential testing logic based on field removal. It caused a large number of actually unused fields to be misjudged as necessary, resulting in serious false negatives. By directly tracing data usage within the code to construct its test oracle, Mockingbird can fundamentally circumvent such interference, maintaining extremely high recall even on pages with complex dynamic backgrounds.

Ablation Study Analysis: Finally, by comparing with the ablation version, we verified the necessity of the iterative supplementary instrumentation strategy. As shown in Table I, Mockingbird(*S*) failed to trace the complete derived variable call chains, leading to a large number of false positives in several projects (e.g., its precision was only 35.64% in `angularHN`, causing its F1-score to be significantly lower than the full version of Mockingbird). This demonstrates that our iterative analysis is crucial for capturing complete data propagation paths and avoiding the erroneous reporting of legitimately used data as over-exposed.

2) RQ2. Applicability: To answer RQ-2, we evaluated Mockingbird's applicability across a diverse set of web applications. The experiment shows that Mockingbird can be successfully applied to all 8 test projects in EDEBench, but its analysis process also reveals common challenges that modern front-end frameworks pose to program analysis.

A core challenge for a program analysis-based method like Mockingbird is to accurately identify data consumption behaviors that are related to UI rendering but deeply encapsulated by modern frameworks. We found that, despite different implementations, the endpoint of such consumption behaviors is universally characterized by a handoff of data from the application logic to the framework's rendering engine. For example, in Angular applications, the endpoint of the data flow is a low-level rendering function of its Ivy engine (e.g., `textInterpolate`). In React applications, this boundary is manifested when data is embedded as properties (props) into a JSX structure and returned as a React element in a return statement. At this moment, the data is encapsulated and handed over to React's `Reconciliation` algorithm for processing.

The Svelte framework, due to its unique compile-time mechanism, presents an even more distinct pattern. We observed that API response data is often restructured within a function and returned as a new state object, at which point our tracking is interrupted. We hypothesize this is because the returned object is passed to Svelte-generated functions that directly manipulate the DOM, and subsequent access no longer involves standard property reads/writes, thus escaping the monitoring scope of the Var Proxy. To handle this phenomenon rigorously, we define such functions that return new state objects as a framework-specific Sink, representing the data's entry into Svelte's rendering pipeline.

It is important to emphasize that EDEFuzz, as a black-box fuzzing method, is theoretically more applicable than white-box or gray-box methods because it is agnostic to the application's internal implementation. However, our experiments prove that by identifying the aforementioned framework-specific handoff patterns as Sinks, Mockingbird successfully overcomes the fragility of traditional program analysis methods when facing framework abstractions, achieving a broad applicability comparable to that of black-box methods. This in itself is a significant achievement. Nevertheless, we acknowledge that adapting Mockingbird to new frameworks still requires defining new Sink patterns, and further improving its applicability will be an important direction for future work.

3) RQ3. Efficiency: To quantitatively evaluate the operational efficiency of Mockingbird, we measured its performance across three dimensions and compared it with the baseline method EDEFuzz.

First, regarding the Manual Setup Time (MST), our researchers spent an average of approximately 2 minutes generating a replayable UI script for each test case in EDEBench.

TABLE I
COMPARISON OF PRECISION, RECALL, F1-SCORE, AAT AND TET ON EDEBENCH

Metric	Tool	Evaluated Web Projects								Avg.
		RuoYi	IceCMS	Answer	Navidrome	Koel	GoFilm	AngularHN	ArialShop	
Precision(%)	EDEFuzz	0.0	57.8	21.1	94.2	73.2	36.7	88.7	72.1	68.1
	Mockingbird	**46.3**	**81.5**	**100.0**	**95.6**	**77.3**	**65.5**	**100.0**	**99.1**	83.9
	Mockingbird(S)	18.8	38.3	**100.0**	84.8	30.3	35.0	37.0	92.9	44.6
Recall(%)	EDEFuzz	0.0	84.1	61.5	73.0	51.9	**100.0**	85.8	37.3	63.2
	Mockingbird	**100.0**	**100.0**	**69.2**	97.2	**99.4**	**100.0**	**100.0**	**88.1**	96.0
	Mockingbird(S)	**100.0**	**100.0**	**69.2**	**100.0**	**99.4**	**100.0**	**100.0**	**88.1**	96.7
F1-Score(%)	EDEFuzz	0.0	68.5	31.4	82.3	60.7	53.7	87.2	49.2	65.5
	Mockingbird	**63.3**	**89.8**	**81.8**	**96.4**	**87.0**	**79.1**	**100.0**	**93.3**	89.6
	Mockingbird(S)	31.7	55.4	**81.8**	91.8	46.5	51.8	54.0	90.4	61.1
AAT(s)	Mockingbird	52	95	26	289	248	205	312	25	157
TET(s)	EDEFuzz	3332	1507	848	2557	5938	1562	4889	2581	2902
	Mockingbird	59	104	32	299	263	217	325	32	166

This represents a one-time upfront investment to ensure the automation and reproducibility of the subsequent analysis.

The core of the evaluation focuses on the automated execution phase. Table I details the Automated Analysis Time (AAT) and Total Execution Time (TET) for both tools. The data indicates that Mockingbird demonstrates a significant performance advantage in all test cases. When processing an API response containing 1,000 fields, the average analysis time required by EDEFuzz is approximately 6,000 seconds, whereas Mockingbird completes the same task in only about 300 seconds. This result means that Mockingbird's processing speed is improved by about 20 times compared to EDEFuzz, which is critical for scenarios requiring rapid iteration and large-scale deployment.

This significant performance difference stems from the fundamental disparity in their detection paradigms. The core performance bottleneck of EDEFuzz lies in its per-field mutation and full-page reload strategy. For a response containing N fields, the time complexity of this method is linear with respect to the number of fields, O(N), because each mutation forces a full page load and analysis cycle. As N increases, the cumulative time cost rises sharply. In contrast, Mockingbird's analysis process is based on the convergence of call chains, and its iterative process typically completes within a few rounds. This approach avoids the repetitive page-loading overhead proportional to the number of fields, thereby fundamentally improving analysis efficiency.

E. Threats to Validity

The conclusions of our work may be subject to the following potential threats to validity.

1) External Validity: This concerns the generalizability of our research findings.

a) **Representativeness of the Benchmark:** This is the most significant threat. Although EDEBench includes 8 popular open-source projects covering various technologies and scenarios, it still cannot represent the infinite diversity of all modern web applications. EDE patterns found in other types of applications (e.g., graphics-intensive applications that heavily rely on Canvas, or WebAssembly [34] applications) may differ from those studied here. Our conclusions should be interpreted as a proof of effectiveness within mainstream business-oriented web applications, and their universality requires future validation on larger and more diverse datasets.

2) Internal Validity: This concerns the reliability of the causal conclusions drawn from our experimental process.

a) **Bugs in Mockingbird's Implementation:** There may be bugs in the implementation of the Mockingbird tool. For instance, the supplementary instrumentation logic may be incomplete, which could affect the accuracy of the final results. To mitigate this threat, we have conducted unit testing and code reviews for the core modules.
b) **Insufficient Test Coverage:** Since user interactions cannot fully guarantee the coverage of all possible execution paths, if a data field is consumed in a code branch not covered by the tests, Mockingbird will miss this usage, leading to false positives.

3) Construct Validity: This concerns whether our measurements truly reflect the concepts we claim to be measuring.

a) **Completeness of the Sink Definition:** Our definition of Necessary Data relies on a predefined set of Sinks. While this definition covers major scenarios like DOM operations, state management, and control flow, there may still exist legitimate but more subtle forms of data usage that we have not covered. This could lead to false positives.
b) **Subjectivity of the Ground Truth:** The Ground Truth for EDEBench relies on manual annotation by experts. Although we employed a rigorous process of training, consensus, and cross-validation, human judgment can never be completely eliminated. Especially in some borderline cases, the judgment of whether a field is necessary may be debatable.

V. RELATED WORK

A. Code Security Analysis for Web Applications

Automated security analysis for web applications has long been a focal point of research in both academia and industry. Given the central role of JavaScript in modern web development, a vast body of research has emerged, primarily following three technical lines: static analysis [4]–[8], dynamic analysis [9]–[15], and hybrid analysis [16], [17]. However, the focus of this research has predominantly been on detecting traditional vulnerabilities such as Cross-Site Scripting (XSS) and code injection, with insufficient attention paid to emerging API security risks like EDE.

Directly adapting existing tools for vulnerabilities like XSS to detect EDE is exceptionally difficult. First, XSS is a classic taint analysis problem, where the objective is to track whether untrusted external input flows into a dangerous execution sink without proper sanitization. The goal of EDE detection, in contrast, is the inverse: it must track data from a trusted source (i.e., the application's own API) to determine which fields are never consumed by any legitimate business function (a sink, such as UI rendering or state updates). Second, XSS payloads almost always exist as strings, focusing the analysis on string operations. Excessively exposed data, however, can be of any data type—including numbers, booleans, arrays, or complex nested objects. This requires the detection tool to be capable of handling nearly all data structures. These fundamental differences in analysis objectives, data types, and detection logic mean that the analysis paradigms for the two vulnerabilities are entirely distinct and cannot be easily repurposed.

More critically, the usability and effectiveness of existing analysis tools continue to face significant challenges. The empirical study by Calzavara et al. [18] systematically revealed this predicament, noting that "most analysis tools are unavailable, hard to run or do not support modern browser automation frameworks and, even when they run, their results are largely unsatisfactory." The root cause of this situation lies in the complexity of modern web technologies. Developers commonly use tools like Webpack and Vite to bundle, minify, and obfuscate JavaScript source code to ensure efficient delivery over various network conditions. As a result, the code in production environments loses most of its semantic information (e.g., variable and function names), original file and module boundaries are destroyed, and techniques like control-flow flattening can fundamentally alter the program's execution logic. These factors, combined with JavaScript's inherent language features such as dynamic typing and prototypal inheritance, collectively constitute formidable barriers to analysis.

B. EDE Vulnerability Detection

Compared to other vulnerabilities, research focused on the automated detection of EDE in web applications is even more scarce. To the best of our knowledge, EDEFuzz by Pan et al. [19] was the only prior automated detection solution in this domain. As a black-box fuzzing tool, it cleverly circumvents direct analysis of JavaScript code by proposing a core test oracle: Excessively Exposed Data does not trigger DOM changes. However, this complete reliance on external behavioral observation has clear limitations. For instance, some data fields may not directly affect the DOM but could be used for subsequent API requests. Conversely, some DOM changes may be triggered by timers or asynchronous user actions, unrelated to any specific data field. These scenarios can cause EDEFuzz to generate significant false positives and false negatives.

In the non-web domain, Koch et al. [39] proposed a white-box mechanism for identifying EDE vulnerabilities in Android applications written in Java. This approach relies on decompiling Java bytecode, followed by code instrumentation and static data-flow analysis. While decompiled Java code typically retains much of its program structure and semantic information, the aforementioned characteristics of JavaScript—such as compression and obfuscation during the build process, dynamic typing, and prototypal inheritance—make such white-box approaches that depend on precise static analysis difficult to apply directly to the modern web environment.

Therefore, to effectively overcome the shortcomings of prior work, Mockingbird adopts a statically-assisted dynamic analysis approach, featuring a novel data propagation tracking scheme designed for the semantic characteristics of JavaScript. To our knowledge, this is the first automated EDE vulnerability detection tool for web applications based on a hybrid code analysis approach.

VI. DISCUSSION AND FUTURE WORK

This paper proposes Mockingbird, a white-box analysis framework designed to address the challenge of EDE detection in modern web applications. Experimental results show that Mockingbird achieves significant advantages in accuracy, applicability, and efficiency compared to existing baseline methods. Its innovative data flow analysis paradigm demonstrates two core advantages over black-box methods that rely on observing UI behavior: first, it can capture complex, indirect data usage, thereby improving detection accuracy; and second, it avoids the repetitive page-loading overhead proportional to the number of fields, fundamentally improving analysis efficiency. The primary limitation of this work is that adapting Mockingbird to new front-end frameworks may still require the manual pre-definition of relevant rules. Future research can be expanded in several key areas. A primary direction is to shift detection capabilities left, integrating them into IDEs or CI/CD pipelines to provide developers with real-time feedback during the development phase. To enhance the effectiveness of this early-stage detection, future work will also focus on deepening the semantic analysis to better judge complex and indirect data usage. Furthermore, to ensure the comprehensiveness of the dynamic analysis component, we plan to improve test coverage through automated UI exploration techniques for a more thorough evaluation.

ACKNOWLEDGMENT

This work was supported by the National Key Research and Development Program of China under Grant 2023YFC3305404.

REFERENCES

[1] S. Ikeda, "Massive optus data leak prompts new privacy rules in Australia," CPO Magazine, 2022.

[2] N. S. R. Alves, T. S. Mendes, M. G. De Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," Inf. Softw. Technol., vol. 70, pp. 100-121, Feb. 2016.

[3] OWASP Foundation, "OWASP Top 10 API Security Risks," 2023. [Online]. Available: https://owasp.org/API-Security/editions/2023/en/0x11-t10/. [Accessed: Sep. 12, 2025].

[4] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Proc. Int. Static Anal. Symp. (SAS)*, ser. Lecture Notes in Computer Science, vol. 5673, 2009, pp. 238–255.

[5] V. Kashyap *et al.*, "JSAI: a static analysis platform for JavaScript," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2014, pp. 121–132.

[6] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript," in *Proc. 19th Int. Workshop Found. Object-Oriented Lang. (FOOL)*, 2012, p. 96.

[7] IBM Research, "T.J. Watson Libraries for Analysis (WALA)," 2006. [Online]. Available: http://wala.sf.net. [Accessed: Sep. 12, 2025].

[8] M. Ferreira, M. Monteiro, T. Brito, M. E. Coimbra, N. Santos, L. Jia, and J. F. Santos, "Efficient static vulnerability analysis for JavaScript with multiversion dependency graphs," Proc. ACM on Programming Languages, vol. 8, no. PLDI, pp. 417-441, Jun. 2024, doi: 10.1145/3656394.

[9] E. Andreasen *et al.*, "A survey of dynamic analysis and test generation for JavaScript," *ACM Comput. Surv.*, vol. 50, no. 5, Art. no. 66, 2017.

[10] R. Wang, G. Xu, X. Zeng, X. Li, and Z. Feng, "Tt-xss: A novel taint tracking based dynamic detection framework for dom cross-site scripting," *J. Parallel Distrib. Comput.*, vol. 118, pp. 100–106, 2018.

[11] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: Evolutionary fuzzing for black-box xss detection," in *Proc. 4th ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, 2014, pp. 37–48.

[12] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *Proc. 21st USENIX Secur. Symp. (USENIX Security '12)*, 2012, pp. 523–538.

[13] D. Klein, T. Barber, S. Bensalim, B. Stock, and M. Johns, "Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions," in Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P), 2022, pp. 236–250.

[14] A. Chudnov and D. A. Naumann, "Inlined Information Flow Monitoring for JavaScript," in Proc. ACM Conf. Comput. Commun. Secur. (CCS), Denver, CO, USA, 2015, pp. 629–643.

[15] E. Trickel et al., "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect SQL and command injection vulnerabilities," in Proc. 2023 IEEE Symp. Secur. Privacy (SP), 2023, pp. 1569–1586.

[16] B. Sayed, I. Traoré, and A. Abdelhalim, "If-transpiler: Inlining of hybrid flow-sensitive security monitor for JavaScript," *Comput. Secur.*, vol. 75, pp. 92–117, 2018.

[17] O. Tripp, P. Ferrara, and M. Pistoia, "Hybrid security analysis of web JavaScript code via dynamic partial evaluation," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, 2014, pp. 49–59.

[18] S. Calzavara, S. Casarin, and R. Focardi, "Dynamic security analysis of JavaScript: Are we there yet?," in *Proc. ACM Web Conf. 2025*, 2025, pp. 1105-1115.

[19] L. Pan, S. Cohney, T. Murray, *et al.*, "Edefuzz: A web api fuzzer for excessive data exposures," in *Proc. 46th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, 2024, pp. 1-12.

[20] OWASP Foundation, "API3:2019 - Excessive Data Exposure," in *OWASP API Security Top 10*, 2019. [Online]. Available: https://owasp.org/API-Security/editions/2019/en/0xa3-excessive-data-exposure/. [Accessed: Sep. 12, 2025].

[21] The Chromium Authors, "Chrome DevTools Protocol." [Online]. Available: https://chromedevtools.github.io/devtools-protocol/. [Accessed: Sep. 12, 2025].

[22] Microsoft, "Playwright Documentation." [Online]. Available: https://playwright.dev/. [Accessed: Sep. 12, 2025].

[23] Ecma International, "ECMAScript Language Specification," ECMA-262, 15th ed., Jun. 2024. [Online]. Available: https://ecma-international.org/publications-and-standards/standards/ecma-262/. [Accessed: Sep. 12, 2025].

[24] E. You, "Vue.js." [Online]. Available: https://vuejs.org/. [Accessed: Sep. 12, 2025].

[25] Meta and community, "React." [Online]. Available: https://react.dev/. [Accessed: Sep. 12, 2025].

[26] Google, "Angular." [Online]. Available: https://angular.dev/. [Accessed: Sep. 12, 2025].

[27] The jQuery Team, "jQuery." [Online]. Available: https://jquery.com/. [Accessed: Sep. 12, 2025].

[28] R. Harris, "Svelte." [Online]. Available: https://svelte.dev/. [Accessed: Sep. 12, 2025].

[29] D. Klein, T. Barber, S. Bensalim, B. Stock, and M. Johns, "Hand sanitizers in the wild: A large-scale study of custom JavaScript sanitizer functions," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, 2022, pp. 236–250.

[30] R. Kanyal and S. R. Sarangi, "PanoptiChrome: A modern in-browser taint analysis framework," in *Proc. ACM Web Conf. (WWW)*, 2024, pp. 1914–1922, doi: 10.1145/3589334.3645699.

[31] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: tracking information flow in JavaScript and its APIs," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2014, pp. 1663–1671.

[32] A. Chudnov and D. A. Naumann, "Inlined information flow monitoring for JavaScript," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2015, pp. 629–643.

[33] B. Sayed, I. Traoré, and A. Abdelhalim, "If-transpiler: Inlining of hybrid flow-sensitive security monitor for JavaScript," *Comput. Secur.*, vol. 75, pp. 92–117, 2018.

[34] The WebAssembly Community Group, "WebAssembly." [Online]. Available: https://webassembly.org/. [Accessed: Sep. 12, 2025].

[35] A. L. Scull Pupo, L. Christophe, J. Nicolay, C. De Roover, and E. Gonzalez Boix, "Practical information flow control for web applications," in *Proc. Int. Conf. Runtime Verif. (RV)*, ser. Lecture Notes in Computer Science, vol. 11237, 2018, pp. 372–388.

[36] L. Christophe, "LinvailTaint," 2023. [Online]. Available: https://github.com/lachrist/aran/blob/664f0a304b555bcb106f24e72734ad8c88dac429/graveyard/test/live/linvailtaint.js. [Accessed: Sep. 12, 2025].

[37] Z. Ahmad, S. Casarin, and S. Calzavara, "An empirical analysis of web storage and its applications to web tracking," *ACM Trans. Web*, vol. 18, no. 1, Art. no. 7, 2024.

[38] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs, "Jalangi: a selective record-replay and dynamic analysis framework for JavaScript," in Proc. ACM Jt. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), Saint Petersburg, Russia, 2013, pp. 488–498.

[39] W. Koch, M. Müller, A. B. Jones, and E. P. Lee, "Semi-automated discovery of server-based information oversharing vulnerabilities in Android applications," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, 2017, pp. 98–109.