RogueOne: Detecting Rogue Updates via Differential Data-flow Analysis Using Trust Domains

Raphael J. Sofaer, Yaniv David
{r.j.sofaer,yaniv.david}@columbia.edu
Columbia University
New York, NY, USA

Mingqing Kang, Jianjia Yu, Yinzhi Cao
{mkang31,jyu122}@jhu.edu
yzcao@cs.jhu.edu
Johns Hopkins University
Baltimore, MD, USA

Junfeng Yang, Jason Nieh
{junfeng,nieh}@cs.columbia.edu
Columbia University
New York, NY, USA

ABSTRACT

Rogue updates, an important type of software supply-chain attack in which attackers conceal malicious code inside updates to benign software, are a growing problem due to their stealth and effectiveness. We design and implement RogueOne, a system for detecting rogue updates to JavaScript packages. RogueOne uses a novel differential data-flow analysis to capture how an update changes a package's interactions with external APIs. Using an efficient form of abstract interpretation that can exclude unchanged code in a package, it constructs an object data-flow relationship graph (ODRG) that tracks data-flows among objects. RogueOne then maps objects to trust domains, a novel abstraction which summarizes trust relationships in a package. Objects are assigned a trust domain based on whether they originate in the target package, a dependency, or in a system API. RogueOne uses the ODRG to build a set of data-flows across trust domains. It compares data-flow sets across package versions to detect untrustworthy new interactions with external APIs. We evaluated RogueOne on hundreds of npm packages, demonstrating its effectiveness at detecting rogue updates and distinguishing them from benign ones. RogueOne achieves high accuracy and can be more than seven times as effective in detecting rogue updates and avoiding false positives compared to other systems built to detect malicious packages.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; • **Security and privacy → Malware and its mitigation**; *Information flow control.*

KEYWORDS

JavaScript, Malicious updates, Malware detection, Node.js, Supply-chain security

ACM Reference Format:
Raphael J. Sofaer, Yaniv David, Mingqing Kang, Jianjia Yu, Yinzhi Cao, and Junfeng Yang, Jason Nieh. 2024. RogueOne: Detecting Rogue Updates via Differential Data-flow Analysis Using Trust Domains. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639199

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04
https://doi.org/10.1145/3597503.3639199

1 INTRODUCTION

Modern application development is accelerated by vast public registries of open-source software packages. These registries have libraries of every size and style, for practically every purpose. Node Package Manager (npm) [26], the package manager for Node.js, has over 3.1 million JavaScript packages. To save development and ongoing maintenance effort, a developer can search for a third-party package on the registry before implementing a new feature themselves. A fully featured application can now be built with a fraction of the code that might have been formerly required.

Attackers are exploiting the open and trusting policies of these registries by mounting *supply-chain attacks*. Adversaries plant malicious code in publicly available packages and trick developers into downloading and incorporating them into their projects. The payload may target the development environment, the server on which the application is deployed, or the client of a web application. Anyone can create a package, and updates are immediately available without review, meaning that updating a dependency package or installing the wrong package is enough to be compromised.

The growing number of, and public interest in, supply chain attacks have inspired the development of tools for detecting malicious packages. Maloss [17] repurposes existing vulnerability-detection techniques and looks for typosquatting attacks in which an attacker names their packages to target common typos in names of widely used packages [77]. Amalfi [60] applies machine learning on features extracted from a package's code, along with metadata such as time between updates. These approaches are effective at detecting malware such as those from automated typosquatting attack campaigns, but not *rogue updates*, which embed malicious code inside existing packages. We show that Amalfi fails to detect almost all rogue updates while Maloss can mistakenly flag most benign updates yet miss detecting the majority of rogue updates. Unlike inherently malicious packages which are easier to avoid, rogue updates are potentially much more insidious as they can affect benign packages that are actually already widely used by developers.

One of the highest impact rogue updates to date was against the `event-stream` package on npm. `event-stream` is a toolkit for working with streaming data, which at the time had 4000 dependent packages. The developer of `event-stream` was no longer maintaining the library and gave custody of it to a new contributor who offered to take over. The new maintainer added a new feature by adding a dependency with an obfuscated malicious file. The update had no unusual metadata and was a small change to a complex but benign code base. Neither AMALFI nor MALOSS can detect it. The attack remained undetected for a month [16] and was downloaded *8 million times*, allowing attackers to steal from end-users through CoPay, a Bitcoin wallet application using `event-stream` as a dependency. Each of the 4000 dependent packages had many more transitive dependents, every one of which caused more users to be instantly exposed to malware.

We present ROGUEONE, a system that detects rogue updates by flagging changes in data-flows from one version of a package to another. ROGUEONE examines the initial and updated versions of a package and uses abstract interpretation to build a fine-grained trace of data-flows among objects. It determines if an update is rogue by detecting new data-flows between system APIs or packages which may steal sensitive data or inject malicious code.

A key challenge is determining if an update is malicious without knowing how the package is used. Unknown code (callers) may call into the package in a myriad of ways, and the package may call into many other packages (dependencies), whose code may not be available or may be infeasible to analyze. We solve this problem by introducing three key ideas: a comprehensive notion of data-flow, differential data-flow analysis, and trust domains.

ROGUEONE uses a comprehensive notion of data-flow that tracks all data and includes potential data-flows in callers and dependencies, not just the package's own code. ROGUEONE tracks data-flows that (1) begin with any data, including *any* static literals such as constant strings or numbers, because any of them may be malicious, (2) end with any object external to the package being analyzed, including all functions of other packages, Node.js built-ins, and the unknown caller of the analyzed package. It is comprehensive in tracking data-flows that begin with any data because a rogue update can insert or modify any code in a package, not just affect inputs from outside the package. It is also comprehensive in tracking data-flows through property references, such as `o1.prop := o2`, because such an assignment allows external unknown code with a reference to object `o1` to receive any data sent to `o2`, resulting in potential data-flows in callers and dependencies even if the analyzed package never accesses the property. This detailed view of a package's data-flows enables effective detection of malware in the presence of unknown callers and unavailable dependencies.

ROGUEONE introduces differential data-flow analysis to examine data-flow changes between the original and updated versions of a package. It exploits the fact that any rogue update must change some data-flow in the target package to introduce malicious functionality. While the idea is intuitively simple, benign updates can also change data-flows, so flagging any change in data-flows would create an unmanageable number of false positives. To apply differential data-flow analysis effectively to complex JavaScript packages, ROGUEONE introduces a novel abstraction: *trust domains*.

Trust domains express the intuitive idea that objects originating in the same dependency package have a common developer who controls the values and code of all those objects. ROGUEONE assigns these objects to the same trust domain since trusting one object created by a package is equivalent to trusting any other object from the same package.

ROGUEONE identifies changes in data-flows across trust domains to flag rogue updates based on the observation that benign updates rarely introduce new data-flows across trust domains. This approach allows effective detection of malware without ROGUEONE incorporating any deny-list of 'suspicious' APIs.

ROGUEONE introduces an Object Data-Flow Relationship Graph (ODRG) to capture how trust domains send and receive data to each other. It uses abstract interpretation to construct ODRGs for the original and updated versions of a package, then detects changes in the data-flows between ODRGs to flag rogue updates. Our implementation of ROGUEONE uses a modern JS abstract interpretation engine based on ODGEN [42] and FAST [32], which allows ROGUEONE to correctly handle JavaScript features such as dynamic typing and prototypical inheritance. ROGUEONE analyzes a package in an update-aware manner, skipping unchanged code when appropriate to avoid unnecessary work. To further increase precision, ROGUEONE recursively analyzes a package's transitive dependencies if their code is available and computes cross-domain flow summaries at the package level, striking a good balance between precision and analysis cost.

We compare the effectiveness of ROGUEONE to MALOSS and AMALFI on hundreds of benign and rogue updates drawn from npm's most popular packages, random packages, and research datasets of malware. ROGUEONE can detect over 75% of rogue updates while keeping false positives under 5%. It is more than twice as effective at detecting rogue updates compared to the closest existing system, and in some cases can be seven times more effective than existing systems. It is also more than seven times as effective in minimizing false positives compared to existing systems.

2 THREAT MODEL

We follow the taxonomy for supply-chain attacks introduced by Ohm et al. [50], which distinguishes between malicious updates to an existing (benign) package versus new malicious packages. ROGUEONE focuses on the former, specifically Ohm et al.'s "Inject into Source" and "Inject into Repository System" categories. Malicious new packages such as typosquatting attacks are out of scope; such attacks are easier to detect and existing tools such as MALOSS already catch them.

We assume that the malicious payload contained in a rogue update has some external effect, either through a system API or a chain of dependencies that reaches a system API. This threat model is common, as demonstrated in attacks such as the `event-stream` attack [65]. Denial-of-service attacks (e.g., Overson, J. [52]) and developer deletion of package functionality are out of scope. These attacks cause damage, but they are less severe, rare, and quickly detected by developers. Similarly, common software vulnerabilities, such as cross-site scripting [8], prototype pollution [33], and arbitrary code execution [67], are out of scope because such packages are attacked during runtime, and many existing tools can already detect such vulnerabilities [68].

3 OVERVIEW

ROGUEONE distinguishes between rogue and benign updates by comparing sets of data-flows between trust domains, an abstraction for groups of objects that share an origin such as a package. Unlike

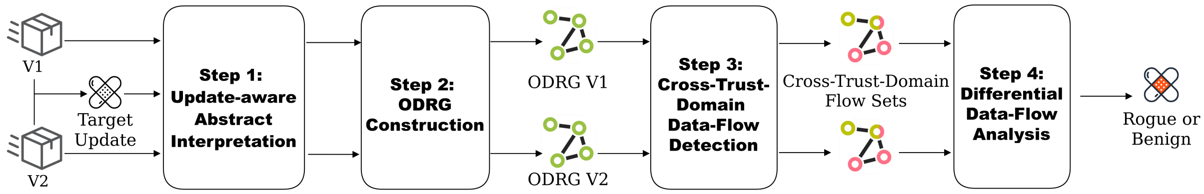


Figure 1: RogueOne architecture.

static analysis approaches that aim to detect vulnerabilities such as taint-analysis, RogueOne cannot restrict itself to data-flows from suspect API sources to sinks in the code being analyzed because a rogue update can be anywhere in the code of a package. Instead, it aims to capture all existing and potential data-flows in and out of the target package and represent them in ODRGs. Fig. 1 shows the high-level steps for this process: (i) update-aware abstract interpretation, (ii) ODRG construction, (iii) cross-trust-domain data-flow detection, and (iv) differential data-flow analysis.

To explain these steps, we use two examples shown in Fig. 2. Fig. 2a shows a rogue update inspired by real-world rogue updates. Before the update, this code fetches a secret from an environment variable (line 1) and uses it to configure (sets the authentication key) a client for a secure store (line 3) provided by the `secure-store` package (imported at line 2). The code also accesses a public store (lines 5-6) provided by the `public-store` package (imported at line 4). An attacker adds a one-line rogue update to leak the secret key to the public store (line 7). Fig. 2b shows a benign update based on an Axios HTTP client example [5]. Before the update, this code imports a package (line 1) that enables performing an HTTP GET (line 6) and writes external data (`response.data` at line 8) to the filesystem (using `fs.writeFileSync` from the `fs` package imported at line 2). The one-line update writes external data (`error.data` at line 11) to the filesystem, but this flow of information between the remote server and the filesystem, or more specifically, between the `axios` and `fs` packages, *already existed* at line 8.

```
1      secret = process.env['SECRET_KEY'];
2      secClient = require('secure-store');
3      secClient.key = secret;
4      pubClient = require('public-store');
5      result = secClient.query('User Query');
6      pubClient.publish(result.publicData);
7  +   pubClient.publish(secret);
```

(a) A rogue update example: the added line 7 leaks a secret key through the call to `pubClient.publish`.

```
1      const axios = require('axios');
2      const fs = require('fs');
3      const instance = axios.create({
4        baseURL: 'https://example.com/api/',
5      });
6      instance.get('/data')
7        .then(function (response) {
8          fs.writeFileSync('response.log', response.data)
9        })
10       .catch(function (error) {
11 +         fs.writeFileSync('error.log', error.data);
12       })
```

(b) A benign update example: the added line 11 adds a second call to `fs.writeFileSync`. However, as the data-flow between `axios` to `fs` existed in the first call to `fs.writeFileSync`, it does not represent a new cross-trust-domain data-flow.

Figure 2: Two update examples, both adding new data-flows, depict the difficulty of flagging only rogue updates.

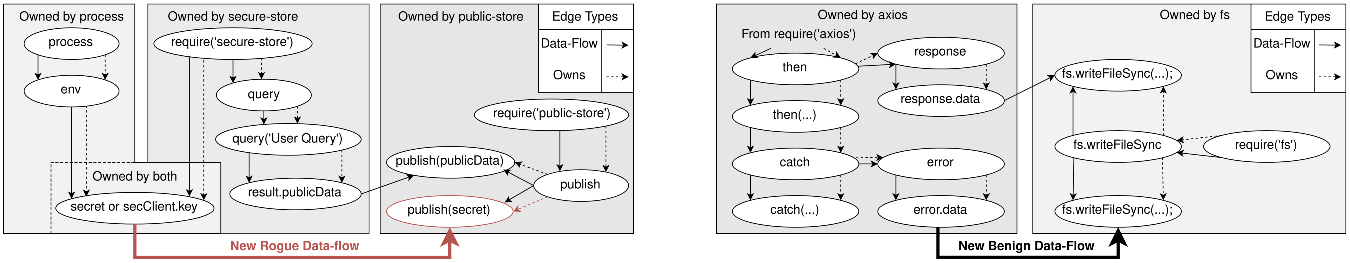
Update-aware abstract interpretation RogueOne examines a target package in isolation, as the identity of the calling package is unknown. To provide a comprehensive notion of data-flow, RogueOne invokes an abstract interpretation engine for the pre- and post-update versions of a package while avoiding analyzing unchanged files between versions of the package to avoid state explosion and prohibitively long analysis times.

RogueOne uses abstract interpretation to reduce complex JS semantics to four high-level operations sufficient for tracking data-flows among objects, including potential data-flows in callers and dependencies, as discussed in §4.1. For example, line 6 in Fig. 2a shows data retrieved using the `secure-store` package, then published back to the web using the `public-store` library. Showing that this data-flow originates with `secure-store` and reaches `public-store` involves the following four operations:

- Package import, e.g., `require('secure-store')`
- Property retrieval, e.g., `secret` is the same object as `process.env['SECRET_KEY']`
- Property assignment, e.g., `secClient` ⟶ `secClient.key`
- External function calls, e.g., `secClient.query` ⟶ `result` and `result.pubData` ⟶ `pubClient.publish(result.pubData)`.

For example, by tracking data-flows through property assignment, RogueOne can identify that any code with a reference to `secClient` may read whatever data is stored in `secret` to capture potential data-flows in callers and dependencies even if the target package in Fig. 2a itself never accesses the property.

ODRG construction Using the set of high-level operations generated by abstract interpretation, RogueOne constructs an ODRG for each version of the package, which it later uses to extract cross-trust-domain data flows. Fig. 3a and Fig. 3b show the ODRGs corresponding to Fig. 2a and Fig. 2b, respectively. The nodes in the graph correspond to objects created during the code's execution, and are connected by *owns* and *data-flow* edges. An owns edge from object A to object B indicates that the code that created A can control B and determine its value. A data-flow edge from object A to object B indicates that the value of B depends on the value of A. Every owns edge has a corresponding data-flow edge, but a data-flow edge may exist without an owns edge. For example, in Fig. 3a, there is a data-flow edge from `secret` to `publish(secret)` because the value of the latter depends on the value of former, but



(a) The rogue update adds the highlighted 'publish(secret)' node. Since the 'secret' node was not previously connected to the `public-store` trust domain, a new cross-trust-domain data-flow is created from `process` to `public-store` and the update is flagged. Static data nodes are omitted for clarity.

(b) The benign update adds the 'error.data' node, as well as the lower call to `fs.writeFileSync`. The new data-flow is across the same trust domain pair that existed in the earlier version, so the update is not flagged. Static data and the region of the `require('axios')` nodes are omitted for clarity.

Figure 3: Object Data-Flow Relationship Graphs for the rogue update and benign update provided in Fig. 2. Shaded areas depict the regions of the object graph 'owned' by a trust domain. If data from one trust domain reaches (solid arrows) the 'owned' area of another trust domain, a cross-trust-domain data-flow is created.

the code for the former does not control the value of the latter. §4.2 provides further details about the ODRG creation process.

Cross-trust-domain data-flow detection After constructing each ODRG, RogueOne uses them to detect data-flows across trust domains. RogueOne defines trust domains by annotating a set of nodes in an ODRG as *trust domain roots* of a specific trust domain. These are objects of known origin whose values are determined by code from that trust domain. The most common example is the object resulting from importing a package, such as in line 1 of Fig. 2a. Nodes that are reachable through owns edges from a trust domain root are considered part of the same trust domain. In Fig. 3, reachability frontiers for owns edges are marked as shaded rectangles annotated with their trust domain.

RogueOne extracts cross-trust-domain data-flows by recording each node which is accessible by data-flow edges from a root of one trust domain and owns edges from a root of another trust domain. For example, in the rogue update depicted in Fig. 3a, there are four cross-trust-domain data-flows. Three exist before the update: (`process` ⟶ `secure-store`) and (`secure-store` ⟶ `process`) are created by the presence of `secret` in both trust domains, and (`secure-store` ⟶ `public-store`) is created by the data-flow from `require('secure-store')` to `publish(publicData)`. One more results from the update: (`process` ⟶ `public-store`) is created by the addition of the new `publish(secret)` node and accompanying data-flow edge from `secret`.

Differential data-flow analysis RogueOne compares the cross-trust-domain data-flows before and after the update. The rogue update in Fig. 3a is flagged, since the cross-trust-domain data-flow (`process` ⟶ `public-store`) is new. However, the benign update in Fig. 3b is not flagged, since the 'new' cross-trust-domain data-flow (`axios` ⟶ `fs`) was already present in the previous version.

4 DESIGN

4.1 Update-Aware Abstract Interpretation

RogueOne employs abstract interpretation to obtain all object data-flow relations from an npm package. An abstract interpretation engine [11] simulates the execution of code, maximizing code coverage by executing the code with different abstract values to explore all code paths. This captures a superset of the program's possible states. The superset will be closer to the actual set of possible states if the abstract interpretation engine is more precise in its simulation. Precision can include being flow-sensitive so that the simulation depends on the control flow of the code, path-sensitive so that the simulation does not evaluate mutually exclusive paths of code together, and context-sensitive so that the simulation of a function accounts for its calling context.

To more precisely track object data-flow relations, RogueOne employs a flow-sensitive, path-sensitive, and context-sensitive abstract interpretation engine for npm packages, ODGen. We use ODGen to refer to the latest version of the engine, including the FAST [32] improvements, implemented as a patch to ODGen. Normally, ODGen starts the simulation in the package's main entry point and calls every function in the package's API [25] to maximize branch and state coverage. When encountering abstract values used as callees in call sites, ODGen creates abstract results, and simulates the execution of passed callbacks. During its simulation, ODGen meticulously records every operation performed on each object at every statement, following each execution path and tracking states, call stacks, and control flow. For example, ODGen allows traversing from a variable to every object which has been referred to by that variable. ODGen is a cloning-based analysis, meaning each different possible value for a variable is represented independently in its result.

Abstract interpretation engines like ODGen strive for precision, but this design choice has a price: real-world code bases can cause state-explosion and require significant time to conclude their analyses. To tackle this problem, RogueOne invokes ODGen's abstract interpretation engine in an update-aware manner to side-step its inherent limitations, allowing RogueOne to examine updates to large complex packages. RogueOne focuses ODGen on parts of the code that change as a result of an update so fewer states are explored and less code requires simulation. To do this, RogueOne begins analyzing a package by examining the two versions and building a

```
// process is a built-in global object treated as imported
process := require('node:process');               // L1
env := process.env;                               // L1
secret := env.SECRET_KEY;                         // L1
secClient := require('secure-store');             // L2
secClient.key := secret;                          // L3
pubClient := require('public-store');             // L4
secQuery := secClient.query;                      // L5
// The implied `this` is treated as a parameter.
result = secQuery(secClient, 'User Query');       // L5
pubData := result.publicData;                     // L6
publish := pubClient.publish;                     // L6
pubResult1 := publish(pubClient, pubData);        // L6
pubResult2 := publish(pubClient, secret);         // L7
```

Figure 4: High-level object and data-flow operations resulting from abstract interpretation of rogue update in Fig. 2a.

list of changed files. Then, RogueOne interposes itself in ODGen's simulation of code importing statements (`require` and `import`). Import attempts against large unchanged files may be aborted to save analysis effort. This results in an opaque imported abstract value, whose properties are themselves abstract values. These are manipulated as required by ODGen, which still tracks data-flow into and out of the imported object.

By invoking ODGen's abstract interpretation engine, RogueOne obtains a record of JavaScript execution for a package, from which it extracts just the object and data-flow related operations necessary and sufficient for trust domain analysis. This reduces the complex semantics of JavaScript to four high-level operations: (1) Package import: `o := require(package)`, (2) Property retrieval: `o2 := o1.prop`, (3) Property assignment: `o1.prop := o2`, and (4) External function call: `result := func(p1,p2,...)`. Package import must be included to determine when external code enters the target program. Object property relationships defined by property retrieval and assignment operations must be included since any code with a reference to a parent object naturally has a reference to all properties of that object, and can read and write the data inside. External function calls must be included to model the APIs of imported packages, system APIs, and core operations such as addition and subtraction. All other elements of JavaScript semantics either do not affect the value of an object or can be modeled with these operations.

For example, RogueOne's use of abstract interpretation on the rogue update example in Fig. 2a results in the set of operations shown in Fig. 4, with a comment on each line indicating from which original line of code in Fig. 2a it was obtained. The built-in `process` global is modeled as a package import, property access is decomposed into separate operations for simplicity, and each function call has the implicit `this` moved into the parameter list. In a more complex program, branches, internal function calls, and loops would all be unrolled into one series of object operations.

4.2 ODRG Construction

Using the set of high-level operations that result from abstract interpretation, RogueOne constructs an ODRG, which captures captures relations between the objects created by the code in the package, as well as objects passed as input to and from the package. The ODRG's nodes represent objects connected by two sets of edges: data-flow and owns. We write the former as $\rightarrow$ and the latter as $\xrightarrow{owns}$. RogueOne processes the above-mentioned four operations to construct the ODRG as follows:

(1) Package import `o := require(package)`: Creates a new node representing `o`, containing the imported external package object.
(2) Property retrieval `o2 := o1.prop`: Finds the set of objects referenced by `o1.prop` in the ODRG and makes them available as `o2`. If no child object node is present in the ODRG (e.g., `o1` is passed in as a parameter), creates one and adds an owns edge `o1` $\xrightarrow{owns}$ `o2` and a data-flow edge `o1` $\rightarrow$ `o2`. If one or more objects are already present in the ODRG as `o1.prop`, this statement does not change the ODRG.
(3) Property assignment `o1.prop := o2`: Adds `o2` to the set of objects reachable through `o1.prop` showing that `o2` may carry data from `o1`, and an owns edge `o1` $\xrightarrow{owns}$ `o2` showing that later code which holds a reference to `o1` may read `o2`.
(4) External function call `result := func(p1,p2,...)`: Adds data-flow edges `func` $\rightarrow$ `result`, `p1` $\rightarrow$ `result`, `p2` $\rightarrow$ `result`,... because the function and the parameters all influence `result`. Adds an owns edge `func` $\xrightarrow{owns}$ `result` because `func` controls `result` and can receive future data flows into `result`. Fig. 3b shows an example as the return value of `instance.get('data')` must be part of the `axios` trust domain, since it carries data and functions from `axios`.

For example, RogueOne follows these steps to construct the post-update ODRG in Fig. 3a from the set of operations in Fig. 4 for the rogue update example. The pre-update ODRG is constructed in a similar manner. Since none of these processing steps overwrite the values of objects or remove information from the resulting graph, order is not important. As a result, the input to ODRG construction can be regarded as an unordered set of operations.

4.3 Cross-Trust-Domain Data-Flow Detection

Data-flow extraction After building each ODRG, RogueOne computes the set of cross-trust-domain data-flows. First, RogueOne designates all objects created via the `require(package)` operation as trust domain roots, and tags them with `package`. Each trust domain root 'owns' the nodes reachable through owns edges; these nodes are part of that root's trust domain. Similarly, each trust domain root may send data to any node reachable through data-flow edges. The flow set is calculated accordingly:

$$
\begin{aligned}
&OwnsReaches(trustDomain) := \{\text{obj} \mid \\
&\qquad \text{obj is reachable along owns edges from a root of } trustDomain\ \} \\
&DataFlowReaches(trustDomain) := \{\text{obj} \mid \\
&\qquad \text{obj is reachable along data-flow edges from a root of } trustDomain\ \} \\
&Flows(ODRG) := \{\ (A \rightarrow B) \mid \exists\ \text{obj such that} \\
&\qquad \text{obj} \in DataFlowReaches(A) \wedge \text{obj} \in OwnsReaches(B)\ \}
\end{aligned}
$$

The result is a set of cross-domain data-flows for the target package: `{(t1 → t2), (t3 → t4),...}`.

For example, for the rogue update in Fig. 2a, three trust domains are present: `process`, `secure-store`, and `public-store`. For each trust domain, we calculate the two sets of reachable nodes described above, described here using the names in Fig. 4, with the node added in the update in asterisks:

$$\begin{aligned}
OwnsReaches(process) &= \{process, env, secret\} \\
OwnsReaches(secure\text{-}store) &= \{secClient, secret, secQuery, result, \\
&\quad pubData\} \\
OwnsReaches(public\text{-}store) &= \{pubClient, publish, pubResult1, \\
&\quad *pubResult2*\} \\
DataReaches(process) &= \{process, env, secret, *pubResult2*\} \\
DataReaches(secure\text{-}store) &= \{secClient, secret, secQuery, result, \\
&\quad pubData, pubResult1\} \\
DataReaches(public\text{-}store) &= \{pubClient, publish, pubResult1, \\
&\quad *pubResult2*\}
\end{aligned}$$

Then, for each ordered pair of distinct trust domains we observe if and where that cross-trust-domain data-flow occurs:

$$\begin{aligned}
(process \to secure\text{-}store) &: \text{Occurs at } secret \\
(secure\text{-}store \to process) &: \text{Occurs at } secret \\
(secure\text{-}store \to public\text{-}store) &: \text{Occurs at } pubResult1 \\
(process \to public\text{-}store) &: \text{Occurs post-update at } *pubResult2*
\end{aligned}$$

Cross-package data-flow elimination The cross-domain data-flow algorithm is specific to one package and yields a package-granularity depiction of data-flows. These data-flows may be to and from packages that can themselves be analyzed by RogueOne. Many packages on npm have no external effects; they expose an API that performs some calculation and returns a value. By analyzing a dependency and observing that there would be no transitive cross-domain data-flows through it, RogueOne can eliminate cross-domain data-flows to that dependency from the flow set. This reduces false positives, especially those resulting from new dependencies, as discussed in §7.3. Where possible, RogueOne performs cross-package analysis as follows:

(1) When an update is analyzed, all available dependencies and transitive dependencies of the pre- and post-update versions are also analyzed, resulting in a list *FlowSets* of data-flow sets, along with $Flows(T)$ for the original target package.
(2) If, for any dependency D:

$$Flows(D) \subseteq (\{(T \to D), (D \to T)\} \cup Flows(T))$$

then D is a dead-end for data-flows. Consider an update to the target package which adds the data-flow $(X \to D)$, where X is some trust domain. To have a malicious effect, data must eventually reach some external API. However, since the flows in D are a subset of the flows in the target and dependencies (other than flows to and from D), we know that data from X cannot go anywhere it did not go before. No data-flow to D can cause information to reach new un-analyzed code, including any system API. RogueOne removes D from *FlowSets* and deletes any data-flow containing D from all elements of *FlowSets* and $Flows(T)$.
(3) The previous step is repeated until no such 'dead end' dependencies remain.

The resulting $Flows(T)$ is used for differential analysis.

4.4 Differential Data-Flow Analysis

The set of cross-domain data-flows for a typical JavaScript package is large, and any particular data-flow which might be used by a malicious package will also be used by many benign packages. To overcome this challenge, RogueOne takes advantage of the stability of the cross-domain flow set of a typical package. RogueOne assumes that the earlier version of a package is benign, and is looking for a malicious update. It calculates the set difference of cross-domain data-flows to find new cross-domain data-flows resulting from an update:
$newFlows := flowsAfterUpdate/flowsBeforeUpdate$.
If the set of new flows is empty, the package update is considered benign. Otherwise, the update is flagged as a rogue update. Referring back to the flows detected in §4.3, we see that $(process \to public\text{-}store)$ only occurs in the post-update version, and the update is flagged.

5 IMPLEMENTATION

The RogueOne implementation consists of 13K Lines of Code (LoC), mainly Python and JS, to support update-aware abstract interpretation, construct the ODRG, perform trust domain analysis, and perform differential analysis. Our engine is a fork of the open-source ODGen repository [40], to which we add update-aware analysis, ODRG construction, and trust domains. The upgrades to ODGen described in FAST [32] have also been merged [31].

The current implementation supports JavaScript features up to ECMAScript 5 with limited support for features up to ECMAScript 2018. JavaScript behavior is modeled within ODGen using a statement-by-statement simulation of the behavior of Node.JS when running the target code. All objects which may be created by the program in all possible branches are modeled symbolically. Newer features such as dynamic imports are not supported but can be converted via transpilers such as Babel. When unsupported features of JavaScript are encountered, they may have no effect or may cause errors.

Extracting the ODRG We augment ODGen by tracking several new object relationships. These include: (1) Linking an external (code not within the scope of analysis) function object to each possible return value. (2) Linking an external function object to the parameters of callbacks passed to that function object (see Fig. 2b). (3) Labelling objects accessible through `module.exports`. (4) Labelling function objects not linked to definitions. However, these additional edges only require the addition of record-keeping to ODGen, no additional simulation is necessary. With these additions, the set of object operations discussed in §4 is complete, and all data needed for RogueOne's analysis is present.

Handling Prototypical Inheritance In all JavaScript runtimes, upon creation, every object contains a `obj.__proto__` property connecting it to its class' prototype chain[47]. Traversing the `obj.__proto__` chain to its end will always reach `Object.prototype`. This is a stumbling block for our information flow analysis, as to handle a case like the one seen on line 3 of Fig. 2a, we must consider data-flows that happen exclusively across object property connections, $\xrightarrow{owns}$ connections in the ODRG. To prevent every trust domain from simultaneously owning `Object.prototype`, RogueOne maintains a list of built-in objects which are present before any code is executed and do not normally carry information. These include the Object, Function, String, Number, and Array prototypes. In addition, the data-flow properties of the built-in functions in the Array and String prototypes are explicitly modeled. Data stored in these prototypes is still tracked, but cross-domain data-flows do not occur solely because of the convergence of object prototype chains.

Additional Trust Domain Roots In §4.2 we describe trust domain roots which are created through package import. In our implementation, three more cases must be considered. First, built-in objects. Before the first line of a JavaScript program is executed, the global and module scopes are populated with numerous objects [24]. These objects provide OS-related data like `process` or code generation APIs like `eval` that RogueOne must track dataflow into. Therefore, our implementation tags many built-in objects modeled by ODGen with trust domains as if they were the result of external imports.

Next, RogueOne considers data which comes from the caller and the surrounding program which has imported the target program. When the caller imports the package under analysis, they receive a reference to the `module.exports` object. This is the object which contains all the APIs the package exports, and which ODGen uses to simulate all entrypoints of a package. The caller can access all properties of it, call functions stored in it, etc. That means the caller owns any object which is accessible through `module.exports`, and supplies the parameters to any function call to the `module.exports` object any of its properties. These objects are tagged with a special `:caller` [1] trust domain. This allows RogueOne to track data-flow not only to packages imported by the target program, but also to the package which imported the target program.

The final source of non-import trust domains is local data. The model language in §4 contains no values like strings or numbers, but malware often introduces new values such as attacker IP addresses or obfuscated code to by unpacked. To detect if these values are passed to external APIs, each static value in a program is treated as having a `:local` trust domain, as if the string `'example.com'` was created by a `require(':local:example.com')` statement. The exact trust domains which are created can be configured as a tradeoff between precision and sensitivity. By default, RogueOne places all local values into one trust domain: `:local`. In our evaluation we also demonstrate an alternate 'Paranoid' configuration in which each value has a unique trust domain, giving greater sensitivity at the cost of precision.

6 LIMITATIONS

When analyzing arbitrary code, it is necessary to accept some drawbacks in order to analyze even a reasonable portion of target code [19]. RogueOne is unsound and can have both false positives and false negatives. Like all malware detection techniques, RogueOne can be evaded.

6.1 Limitations of Abstract Interpretation

Timeouts Any abstract interpretation or symbolic execution system must contend with the fact that a complete analysis of a program cannot be guaranteed to terminate. Although ODGen models many complex JavaScript features, the main causes of incompleteness and unsoundness in ODGen are the same language features which inhibit analysis in any language: unbounded loops and recursion, race conditions. In particular, loops with heavily branching bodies that require the abstract interpretation engine to explore an exponentially expanding tree of branches can stop abstract interpretation from completing.[2]

FAST Analysis Scaling In FAST [32], the authors improve the ability of ODGen to scale across a large dataset by targeting particular APIs such as `child_process.spawn` and pruning execution paths which cannot lead to the APIs of interest. This approach does not apply to RogueOne as nearly all external APIs must be tracked, so almost no execution paths are pruned. Although these mitigations are included in RogueOne, they are not enabled by default as they increase analysis time overall. Future work might adapt them to improve performance for RogueOne. Instead, RogueOne uses update-aware analysis that excludes unchanged code to mitigate timeouts. The current implementation excludes unchanged code at file granularity; excluding unchanged code at function or scope granularity is left to future work.

Loop Abort JavaScript packages such as web servers or parsers may have unbounded inputs and outputs. Abstract interpretation of such a package must restrict its fidelity to complete the analysis without timing out. ODGen measures statement coverage during consecutive loop or recursive evaluations to restrict branch exploration and abort loop evaluation once no new code is being evaluated. This can result in code not being evaluated (being incorrectly ignored as dead), if the abort heuristic is triggered before the problematic code is reached. An attacker could use this to construct an update which evades RogueOne by delaying the malicious functionality until after a state change which ODGen does not model, such as a change in the result of a system API. Such an attack would create a new trigger of the timeout mitigation, which could be used as an additional flagging criterion in future work.

Race Conditions JavaScript's built-in asynchronous features make some code execution non-deterministic. By creating a set of callbacks which are unpredictably interleaved, the programmer can create a large number of implicit branches through race conditions.[3] Exploring all possible interleavings of asynchronous code is obviously infeasible. For simplicity, ODGen assumes that all callbacks are executed after the current entry point (exported function or module level scope) is complete. Future work may draw on race condition detection work such as NodeRacer[18] to identify what interleavings should be tested to find new behavior.

6.2 Other Limitations

Update-Aware Imprecision If RogueOne excludes unchanged code from abstract analysis, analysis of changed code may become less precise if it called into the excluded code. This is beneficial if a timeout can be avoided by the exclusion, but a disadvantage if no timeout would have occurred. The current criteria RogueOne uses to exclude files are targeted at large unchanging 'vendored'

[1]The `':'` character is used as it is excluded from Node.js package names.

[2]An example of this can be seen in the package `jade`, included in our artifact.

[3]For example, see `tests/dataflow_fixtures/dual_version/design_ex` in our artifact.

libraries such as jQuery, which tend to be intractable. There is an inherent tradeoff in the choice of criteria: The more files are excluded, the less timeouts will be encountered, but the more false positives will occur. In addition, the presence of a vulnerability such as unsanitized dynamic access to `eval` or `require` in the excluded code could make RogueOne unable to detect a malicious data-flow. Improving the granularity of code exclusion and the resilience of the abstract interpretation engine to timeouts will help eliminate these cases.

Unseen Mutations RogueOne's threat model is targeted at a particular update, which may or may not be malicious. As a result, we assume that when an object is passed to an external trust domain, the foreign code does not mutate the object it has received. This can result in a break in a multi-package data flow which occurs through mutation of a local object. If a dependency had such behavior,[4] it would create an invisible mutation to a local object, which could create a new data-flow if it were subsequently passed to another trust domain. Detecting these data-flows would require RogueOne to record new data flow edges from a function to mutable arguments to that function. If a dependency mutates its input in a way that enables an attack, RogueOne cannot detect that mutation. New dependencies, such as `flatmap-stream` in the `event-stream` attack, will be detected.

False Positives RogueOne gives a false positive whenever a package makes a benign change that creates a cross-trust-domain data-flow. For example, a package may add a new option to save its output to a file, creating an innocuous but flagged data-flow to `fs`. Future work may restrict analysis to entry points actually called by the RogueOne user, eliminating new branches that are not reached in the actual application. The remaining false positives represent changes in functionality, in which case RogueOne may assist developers in quickly locating and analyzing the changes.

Obfuscation Benign and malicious JavaScript packages on npm are frequently obfuscated using a variety of tools and techniques[46]. We did not observe any effect from common transformations such as minification and control flow flattening on our analysis. In addition, where two malicious samples contained obfuscated and unobfuscated versions of the same payload, the analysis result was unaffected. However, some obfuscation techniques such as global arrays will cause over-approximation and spurious cross-trust-domain data-flows, concealing new data-flows. The addition of an obfuscated payload to an unobfuscated package would still be detected.

7 EVALUATION

We evaluate the effectiveness of RogueOne at detecting rogue updates and compare it against state-of-the-art approaches for detecting malicious packages. We also measure the runtime performance of RogueOne.

7.1 Experiment Setup

Datasets. We collected two datasets with a mix of benign and rogue updates and one consisting of all presumed benign updates. Each data point consists of two consecutive versions of a package, e.g., `(0.1.1,0.1.2)`.

- Multi-version dataset with 333 updates from 12 distinct packages. This dataset was used by the authors of Amalfi for evaluating their system and includes multiple updates per package of which at least one is rogue.[5]
- Single-update dataset with 341 updates, each from a distinct package. We constructed this dataset by collecting the latest update from 150 "most depended upon" packages [34], 150 randomly selected packages from the npm repository, 29 rogue updates from the Backstabber's Knife [50] malware collection, and 12 other publicly reported rogue updates.

Evaluation methodology. Experiments were performed on a machine with a 32-core i9-13900KF and 128 GB of RAM, running Ubuntu 22.04. We enforced a timeout of one hour for processing one update for all competing systems; errors or timeouts were considered benign updates by default to reflect the most likely mode of use of a malware-finding tool in the software development industry.

Baselines. We evaluated RogueOne against two other systems:

- Amalfi [60] (a state-of-the-art ML-based system). The Amalfi artifact provides classification results (but *not* code or models) on the multi-update dataset with three different classifiers, Bayes, D-Tree, and SVM. The Amalfi authors indicated[57] that they could not provide their code or trained model for legal reasons; therefore, we could only report the results for Amalfi for their own dataset and could not evaluate Amalfi on any other dataset.
- Maloss [17] (a state-of-the-art malicious package detection system). The Maloss system is composed of three analysis components: (1) static, (2) dynamic, and (3) metadata. The static analysis component examines suspect packages by detecting calls to suspicious APIs and taint-flows between them and is reported as Maloss-STATIC. We also show results from Maloss's static differential mode under the name Maloss-SDIFF[6], which accepts two versions of a package and only flags *new* suspicious APIs. Maloss's dynamic analysis component installs the target package in a sandbox and records all network communication, file access, and process creation. Dynamic analysis detects install time attacks[50], not require-time attacks like `event-stream`. The combination of static and dynamic analysis is reported as Maloss. In our evaluation, we did not run Maloss's metadata analysis component as it is ineffective against rogue updates. We used Maloss's published docker image on DockerHub (Digest 69de276a4552) and ran it with the default configuration settings providing the updated package to Maloss and the original and updated version to Maloss-SDIFF as inputs.

Configurations. We evaluated two RogueOne configurations:

- RogueOne. The default version of RogueOne assigns all static objects and literals in the update itself to a single `:local` trust domain (See trust domain assignment: §5).
- RogueOne-Paranoid. RogueOne-Paranoid employs the opposite strategy, by assigning every static object and literal a unique trust domain for maximum sensitivity. RogueOne-Paranoid creates detailed descriptions of program data-flows usable for

[4] `tests/dataflow_fixtures/single_version/third_party_injection` in the artifact.

[5] We omit any package without updates.

[6] In the Maloss system this component is called `compare-ast`.

System	Rogue Updates				Benign Updates			
	Flagged		Missed		Correct		FalsePos	
RogueOne	7	100%	0	0%	309	94.8%	17	5.2%
RogueOne-Paranoid	7	100%	0	0%	243	74.5%	83	25.5%
Amalfi-Bayes	2	28.6%	5	71.4%	314	96.3%	12	3.7%
Amalfi-DTree	1	14.3%	6	85.7%	285	87.4%	41	12.6%
Amalfi-SVM	0	0%	7	100%	320	98.2%	6	1.8%
Maloss	7	100.0%	0	0.0%	54	16.6 %	272	83.4%
Maloss-STATIC	2	28.6%	5	71.4%	241	73.9%	85	26.1%
Maloss-SDIFF	1	14.3%	6	85.7%	324	99.4%	2	0.6%

Table 1: Classification Accuracy on Multi-version Dataset

System	Rogue Updates				Benign Updates			
	Flagged		Missed		Correct		FalsePos	
RogueOne	31	75.6%	10	24.4%	286	95.3%	14	4.7%
RogueOne-Paranoid	38	92.7%	3	7.3%	180	60.0%	120	40.0%
Maloss	14	34.1%	27	65.9%	190	63.3%	110	36.7%
Maloss-STATIC	13	31.7%	28	68.3%	197	65.7%	103	34.3%
Maloss-SDIFF	5	12.2%	36	87.8%	292	97.3%	8	2.7%

Table 2: Classification Accuracy on Single-update Dataset

post-processing heuristics commonly used in the security industry for directing manual examination efforts.

7.2 Detecting Rogue Updates

Table 1 presents the number of benign and rogue updates that were classified correctly in the multi-version dataset. All configurations of RogueOne correctly identified all rogue updates in the dataset while controlling false positives. Maloss detects all rogue updates but flags most (83%) benign updates as rogue, while the other baselines detected two at best. RogueOne exhibits a false positive rate that is equivalent on average to Amalfi's variants, while Maloss-STATIC suffers from a false positive rate that is similar to RogueOne-Paranoid without the sensitivity. §7.3 provides further details with a breakdown (Table 4) and discussion of false positives.

Table 2 presents the number of benign and rogue updates that were classified correctly in the single-update dataset. Amalfi is excluded since the Amalfi classifier is not publicly available. This experiment showcases the trade-off of sensitivity with false positives provided by RogueOne and Maloss. RogueOne correctly detects more than 70% of rogue updates while suffering only 5% false positives. The even more sensitive RogueOne-Paranoid detects almost 93% of rogue updates yet suffers from 40% false positives. In contrast, the most sensitive variant of Maloss detects 32% of rogue updates with a similar false positive rate (37%). Maloss-SDIFF is less prone to false positives (2.7% of benign updates), but detects only 12% of rogue updates. Errors and timeouts were considered to be benign for classification purposes. 11 benign updates and one rogue update caused errors for Maloss, while nine benign updates caused errors for RogueOne.

The results across both multi-version and single-update datasets show that RogueOne identifies more rogue updates than other systems while maintaining similar false positive rates. It is important to note that the Maloss system, including the metadata analysis component, was designed to detect packages which were malicious at the moment of publishing, leading to degraded performance on this subset of malware.

Attack	Multi	Single	Total	RogueOne-Paranoid	RogueOne
C-Harvest	0	22	22	100%	100%
CC-Harvest	0	10	10	80%	0%
S-Harvest	2	1	3	100%	100%
Dependency	1	1	2	100%	100%
M-FS-Access	1	3	4	75%	75%
Remote Ctrl	3	4	7	100%	100%
Total	7	41	48	94%	79%

Table 3: True Positives and Effectiveness By Attack Type

Table 3 provides a breakdown of the types of rogue updates in both datasets and RogueOne's overall detection effectiveness for each type of attack. We describe all of them, as well as the cause of any false negatives, below.

Client-side Data Harvest (C-Harvest) A client-side data harvest attack steals sensitive user data—e.g., passwords, credit card numbers, and session tokens—from browser forms, AJAX requests, and cookies, and sends it to the attacker's server. An adversary injects their malicious payload as a rogue update into JavaScript code meant for the browser. RogueOne in all its configurations flags all 22 of these packages, with the typical triggering data-flow being `:local` to `:JS:frontend:document`.

Camouflaged Client-side Data Harvest (CC-Harvest) A variant of the client-side data harvest attack places the payload into benign code which already has cross-domain data-flows corresponding to the attack. In other words, the APIs used for malicious functionality are already in use. Default RogueOne cannot detect these attacks because with all string literals contracted into the `:local` trust domain, the new data-flows, e.g. `:local` to `:JS:frontend:document`, already exist in the previous program or a dependency. As a result, default RogueOne has ten false negatives from this category. This highlights the utility of RogueOne-Paranoid. By increasing the granularity of the trust domains of local data, RogueOne-Paranoid recognizes that the data flows to those APIs are from new static strings, and flags all examples of this attack for which our analysis completes. Abstract interpretation fails to correctly handle branching loops in code unrelated to the payload for two of these samples (§6.1), causing false negatives for RogueOne-Paranoid as well.

Server-side Data Harvest (S-Harvest) A server-side attacker can also execute a data harvest attack. They must use two sets of APIs: data harvesting (e.g., via `process.env`) and data exfiltration (e.g., via `https`). RogueOne detects all of these samples, most often by the data-flow from `:local` (the attacker's hard-coded server) to the exfiltration API. In one case RogueOne detects the data-flow `public-ip` to `:local`, as the updated malware begins collecting the public IPs of victims.

Malicious Dependency (Dependency) This attack hides the malicious payload in another package, then updates the target package by adding a `require` call that triggers the run-time import and execution of the malicious dependency. This is how the `event-stream` project was attacked [16]. Since there is no actual use of the malicious dependency, there is no call to any dependency methods. In this case, the dependency is added and assigned to a property of `module.exports`, making it available to the caller. RogueOne recognizes that in addition to the new package, there is a data-flow from

FP Cause	Multi	Single	Total	RogueOne-Paranoid	RogueOne
Dependencies	14	27	41	6.5%	2.1%
System APIs	3	5	8	1.3%	0.8%
New Data	43	84	127	20.3%	0.0%
Install Scripts	14	0	14	2.2%	0.8%
Built-ins	3	1	4	0.6%	0.6%
Flow Analysis	6	3	9	1.4%	0.5%
Total	83	120	203	32.4%	4.8%

Table 4: False Positives Across RogueOne Configurations

`flatmap-stream` to `:caller`, due to the caller receiving the exported library object.

Malicious Filesystem Access (M-FS-Access) This attack maliciously modifies the file system, e.g., sabotaging another npm package if it exists, or appending the attacker's ssh key to local `/home/*/.ssh/authorized_keys` files. The default RogueOne detects three out of four such attacks via data-flows that begin or end at `fs`. The last one uses JavaScript syntax unrelated to the payload that our abstract interpretation engine does not support. This causes abstract interpretation to fail, giving a false negative result.

Remote Control (Remote Ctrl) This attack establishes a long-term hold on the victim computer (e.g., via a reverse shell or cryptocurrency miner). To do this, the attacker must use one or more of the network, filesystem, or OS process (such as `child_process`) APIs. RogueOne detects the APIs used by the malware and fully captures the attacks in the ODRG, representing them as data-flows from `:local` to the package used to control the network, filesystem or OS interaction. For example, a static IP address string would be in the `:local` trust domain and may flow to `http`, causing a flag.

7.3 False Positives and Causes

Table 4 lists false positives across both configurations of RogueOne, divided into six different causes which are described below. Since RogueOne-Paranoid has the highest number of false positives, the total counts necessarily correspond to its false positives, with RogueOne having less. The first three causes are related to trust domain analysis, and the last three are related to abstract interpretation.

- New Dependencies. RogueOne recursively analyzes dependencies and composes trust domain relationships. If a dependency cannot be analyzed due to a timeout or unsupported JavaScript feature, it is left as a 'black box' and treated identically to a system API. Then, any new cross-domain data-flow to or from the dependency will be flagged, causing a false positive if the dependency is actually benign.
- New System API Use. RogueOne considers any update that introduces new system APIs such as network and filesystem access to be rogue. Updates to mature packages which use new system APIs are rare, but when they happen they can only be distinguished from malicious additions by manual analysis. At the same time, the output generated by RogueOne can greatly assist the manual examination of these cases.
- New Data. This cause is specific to RogueOne-Paranoid, which marks every piece of data in a program as a potential threat. However, these potential threats could be as simple as changed logging messages, references to renamed files, or adjusted regular expressions. As a result, over the two datasets, 173 additional packages are flagged over RogueOne.
- Incomplete modeling of install scripts. npm provides package developers with "Install hooks"[27] which execute arbitrary code during package install. RogueOne models these hooks as `require` calls when they run JavaScript files, but non-JavaScript scripts result in RogueOne flagging the update as rogue.
- Unrecognized Built-ins. RogueOne assumes that any unrecognized built-in function is a sensitive external API, which leads to false positives.
- Incomplete Flow Analysis. The underlying abstract interpretation engine that RogueOne uses does not support all of JavaScript. This results in disrupted data flows, which create false positives.

Reviewing a false positive typically takes no more than two minutes, and was done through a simple custom web interface displaying new trust domain relations and a diff.

7.4 RogueOne Performance and Efficiency

RogueOne's abstract interpretation engine aspires to high fidelity, making timeouts inevitable. However, we do not observe any rogue update in the wild in which the malicious payload causes a timeout. In the two datasets together, five updates timed out in 1 hour. Among the rest, the average processing time was 226.7 seconds, of which 84% was abstract interpretation and 16% was post-processing. 95% of packages finish analysis in less than 30 minutes, and 90% take less than four seconds. In the cross-package portion of our analysis, we use a shorter 20 minute timeout and cache analysis results for use by any dependents. Using a multi-core harness, processing all updates across both datasets takes approximately 11 hours. Without our update-aware optimizations, ODGen times out on 140 samples in the single-update and multi-version datasets. With update-aware abstract interpretation, this is reduced to five.

8 RELATED WORK

Software supply-chain attacks. Software supply-chain attacks have become a popular topic in recent years [38, 51] Such supply-chain attacks could range from developing malicious package from scratch [50] to name confusing (e.g., Typosquatting [71] and Brandjacking [62]). Various tools attempt to detect and negate these attacks [17, 21, 28, 60, 76]. Maloss repurposed vulnerability detection and monitoring tools to find and characterize malicious packages. Amalfi [60] and JStap [21] apply machine learning techniques to improve the performance of program-analysis-based malicious code detectors. Our results show that Amalfi and Maloss are effective on malware in general, but ineffective for detecting rogue updates. JStap also trains its classifier on control and data-flow information, but HideNoSeek [20] shows that machine learning-based methods are vulnerable to rogue updates that camouflage themselves within the AST of the target. HideNoSeek's method of camouflage does not affect RogueOne because it does not conceal data-flows.

Software isolation is a popular defense to isolate potentially malicious scripts that has been applied to supply-chain attacks. JSIsolate [76] extracts the dependency relationship of different JavaScript components for functionality-wise isolation to defend against malicious code executing in the browser, but does not detect rogue updates. Enclosure [28] replaces static policy inference with explicit inline policies for Python and Go but requires developers to write policies, which is not common practice. Other software isolation approaches focus on vulnerabilities and do not address supply-chain attacks. MIR [72] performs language-runtime-level software isolation via file-based policies limiting the usage of external code, preventing the exploitation of vulnerabilities to call other external APIs. `sysfilter` [14] restricts the syscalls available to an application to prevent the exploitation of vulnerabilities in benign native applications. These techniques are vulnerable to the camouflaged client-side data harvest attacks in our evaluation, which fit within the existing "permissions" of the victim package.

Information flow analysis. Static and dynamic information flow analysis are used for vulnerability discovery, test and input generation, and malware analysis. Taint analysis, a form of information flow analysis, is common in security applications. Examples include [70] presents TAJ, a static taint-analysis tool based on slice-construction balancing context-sensitive and insensitive analysis, and [58] a dynamic taint-analysis engine that runs parts of the program using forward symbolic execution. Later, [29] performed unified taint analysis with points-to-analysis, an important subproblem. StubDroid [4] produces data-flow summaries for Android libraries. Unlike these approaches and others that attempt to extract input-output relations or connect inputs with covered paths in the program, RogueOne captures all flows, including to and from external packages, with less regard for how specific values affect the execution. Our evaluation compares RogueOne to Maloss, which employs static taint analysis as well as dynamic analysis, and shows the advantages of our approach.

Abstract interpretation. Abstract interpretation has been applied to JavaScript static analysis [30, 35, 48]. Recent works build various forms of graphs for various applications, especially vulnerability detection [32, 41, 42]. ODGen [42] uses abstract interpretation to construct an Object Dependence Graph which enables queries for the offline detection of a wide range of Node.js vulnerabilities. RogueOne modifies ODGen to track more data-flows and adopts update-aware analysis to prioritize code that is related to changes in the target. This reduces the likelihood of timeouts. FAST[32] mitigates timeouts in ODGen by pruning code which does not lead to APIs of interest. Empirical evaluation of the FAST mitigations in RogueOne shows that when considering all possible external APIs, the overhead of FAST outweighs the benefit. RogueOne is the first of its kind to use abstract interpretation for rogue update detection.

Static or dynamic analysis of JavaScript. Static and dynamic program analysis have been used to detect a wide range of vulnerabilities, such as browser extension vulnerabilities [22], Regular Expression Denial of Service (ReDoS) [7, 13, 66], debloating [37], hidden property abuse [74], and prototype pollution [3, 33, 63]. These tools do not work for rogue update detection, in which malicious code is embedded into the program and possibly obfuscated, rather than being injected in through a vulnerability.

SAFE [39] and SAFEWAPI [6] convert JavaScript to an Intermediate Representation (IR) form for further static analysis to detect bugs in JavaScript code, but are restricted to Web-IDL specified APIs and do not detect rogue updates. SAFE$_{DS}$ [53] adopts Jalangi [61] , a dynamic analysis tool which selectively records and replays front- and back-end JavaScript programs, to build dynamic shortcuts on top of SAFE, which speeds up static analysis of large packages such as Lodash. SAFE$_{DS}$ presents an alternate approach to optimizing abstract interpretation using external knowledge, but is not applicable to rogue updates as they often use conditional triggers to hide from dynamic analysis tools.

JavaScript call graph construction [1, 2, 9, 15, 23, 49, 59, 69] using static [1], dynamic [69], or hybrid [2] analysis, is usually the first step of static analysis, including for RogueOne's abstract interpretation engine. JavaScript symbolic execution [45, 55, 56] has also been used for static analysis.

Patch analysis. Patches, e.g., those related to security updates, are often studied to fix security vulnerabilities. Approaches include hot-patching [10, 54, 75] and backporting security patches [64]. Similarly, approaches have been developed to infer correct patches from normal test cases [36, 73] or existing human-written patches [43, 44]. Recently, UPGRADVISOR [12] employed differential analysis to determine whether a dependency update will break a Python application, using a combination of static analysis and a hardware-based tracer for interpreted languages. All of these works assume patches are benign and do not detect rogue updates. In contrast, in RogueOne's threat model patches may be malicious.

9 CONCLUSIONS AND FUTURE WORK

RogueOne is the first system designed to automatically detect rogue updates to npm packages. It uses update-aware abstract interpretation to capture all existing and potential data-flows in and out of packages with a high-degree of precision while avoiding timeouts even when analyzing large, complex packages. It constructs object data-flow relationship graphs for pre and post-update versions of a package, and groups objects in trust domains to express trust among objects of a common origin. It then uses the graphs and trust domains to detect data-flows across trust domains and applies differential data-flow analysis to identify changes in cross-trust-domain data-flows, reflecting changes in data-flows to and from external APIs. RogueOne uses the data-flow changes to flag potential rogue updates. Our evaluation across hundreds of npm packages shows that RogueOne can detect over 75% of rogue updates while keeping false positives under 5%. Compared to other malware detection tools, RogueOne can be seven times more effective at detecting rogue updates and minimizing false positives.

While RogueOne focuses on npm packages, future work will consider other languages and repositories, such as Rust/Cargo and Ruby/Rubygems. JavaScript's dynamic prototype-based type system makes data-flow analysis challenging, and we believe RogueOne would be even more effective in a less dynamic language.

ACKNOWLEDGMENTS

Matthew Luo assisted with system implementation and experiment scaling. This work was supported in part by NSF grants CNS-2046361, CNS-2052947, CNS-2154404, CNS-2247370, and CCF-2124080, DARPA contract N66001-21-C-4018, a DARPA Young Faculty Award, as well as research awards and gifts from Google, Amazon, Accenture, and Visa.

REFERENCES

[1] Gábor Antal, Péter Hegedus, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Static JavaScript Call Graphs: A Comparative Study. In *Proceedings of the 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM '18)*. Madrid, Spain, 177–186. https://doi.org/10.1109/SCAM.2018.00028

[2] Gábor Antal, Zoltán Tóth, Péter Hegedűs, and Rudolf Ferenc. 2020. Enhanced bug prediction in JavaScript Programs with Hybrid Call-Graph Based Invocation Metrics. *Technologies* 9, 1 (2020), 3. https://doi.org/10.3390/technologies9010003

[3] Olivier Arteau. 2018. Prototype Pollution Attack in NodeJS Application. NorthSec. Retrieved 2 Feb 2023 from https://github.com/HoLyVieR/prototype-pollution-nsec18/blob/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf

[4] Steven Arzt and Eric Bodden. 2016. StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Austin, Texas, 725–735. https://doi.org/10.1145/2884781.2884816

[5] Axios 2024. *Minimal Example | Axios Docs*. Axios. Retrieved 12 Jan 2024 from https://axios-http.com/docs/example

[6] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: Web API Misuse Detector for Web Applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. Hong Kong, China, 507–517. https://doi.org/10.1145/2635868.2635916

[7] Zhihao Bai, Ke Wang, Hang Zhu, Yinzhi Cao, and Xin Jin. 2021. Runtime Recovery of Web Applications under Zero-Day ReDoS Attacks. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP '21)*. San Francisco, CA, 1575–1588. https://doi.org/10.1109/SP40001.2021.00077

[8] Yinzhi Cao, Chao Yang, Vaibhav Rastogi, Yan Chen, and Guofei Gu. 2015. Abusing Browser Address Bar for Fun and Profit - An Empirical Investigation of Add-On Cross Site Scripting Attacks. In *Proceedings of the International Conference on Security and Privacy in Communication Networks*. Beijing, China, 582–601. https://doi.org/10.1007/978-3-319-23829-6_45

[9] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. 2022. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs. In *Proceedings of the 36th European Conference on Object-Oriented Programming (ECOOP '22)*, Vol. 222. Berlin, Germany, 3:1–3:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.3

[10] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. 2018. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS '18)*. The Internet Society, San Diego, CA. https://par.nsf.gov/servlets/purl/10053521

[11] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. Los Angeles, California, 238–252. https://doi.org/10.1145/512950.512973

[12] Yaniv David, Xudong Sun, Raphael J Sofaer, Aditya Senthilnathan, Junfeng Yang, Zhiqiang Zuo, Guoqing Harry Xu, Jason Nieh, and Ronghui Gu. 2022. UPGRADVISOR: Early Adopting Dependency Updates Using Hybrid Program Analysis and Hardware Tracing. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. Carlsbad, CA, 751–767. https://par.nsf.gov/biblio/10356086

[13] James C Davis, Eric R Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *Proceedings of the 27th USENIX Security Symposium (SEC '18)*. Baltimore, MD, 343–359. https://www.usenix.org/conference/usenixsecurity18/presentation/davis

[14] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '20)*. Virtual, 459–474. https://www.usenix.org/conference/raid2020/presentation/demarinis

[15] J Dijkstra. 2014. *Evaluation of static JavaScript call graph algorithms*. Ph. D. Dissertation. Software Analysis and Transformation.

[16] dominictarr. 2023. event-stream. Retrieved 2 February 2023 from https://github.com/dominictarr/event-stream/issues/116

[17] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium (NDSS '21)*. The Internet Society, Virtual. https://doi.org/10.14722/ndss.2021.23055

[18] André Takeshi Endo and Anders Møller. 2020. NodeRacer: Event Race Detection for Node.js Applications. In *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST '20)*. Porto, Portugal, 120–130. https://doi.org/10.1109/ICST46399.2020.00022

[19] Dawson Engler, Ben Chelf, Andy Chou, and Seth Hallem. 2008. A Couple Billion Lines of Code Later: Static Checking in the Real World. In *17th USENIX Security Symposium (SEC '08)*. San Jose, CA. https://www.usenix.org/conference/17th-usenix-security-symposium/couple-billion-lines-code-later-static-checking-real-world

[20] Aurore Fass, Michael Backes, and Ben Stock. 2019. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. London, United Kingdom, 1899–1913. https://doi.org/10.1145/3319535.3345656

[21] Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC '19)*. San Juan, Puerto Rico, 257–269. https://doi.org/10.1145/3359789.3359813

[22] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Virtual, 1789–1804. https://doi.org/10.1145/3460120.3484745

[23] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE '13)*. San Francisco, CA, 752–761. https://doi.org/10.1109/ICSE.2013.6606621

[24] OpenJS Foundation. 2023. Global objects | Node.js v21.1.0 Documentation. Retrieved 13 November 2023 from https://nodejs.org/api/globals.html

[25] OpenJS Foundation. 2023. Modules: CommonJS modules | Node.js v21.2.0 Documentation. Retrieved 15 November 2023 from https://nodejs.org/api/modules.html#moduleexports

[26] OpenJS Foundation. 2023. npm. Retrieved 4 February 2023 from https://www.npmjs.com

[27] OpenJS Foundation. 2023. scripts | npm Docs. Retrieved 12 April 2023 from https://docs.npmjs.com/cli/v9/using-npm/scripts

[28] Adrien Ghosn, Marios Kogias, Mathias Payer, James R Larus, and Edouard Bugnion. 2021. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Virtual, 255–267. https://doi.org/10.1145/3445814.3446728

[29] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-to and Taint Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 102 (Oct 2017), 28 pages. https://doi.org/10.1145/3133926

[30] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proceedings of the 2009 International Static Analysis Symposium (SAS '09)*. Los Angeles, CA, 238–255.

[31] Mingqing Kang. 2023. fast. Retrieved 14 November 2023 from https://github.com/fast-sp-2023/fast

[32] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V.N. Venkatakrishnan, and Yinzhi Cao. 2023. Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '23)*. San Francisco, CA, 1059–1076. https://doi.org/10.1109/SP46215.2023.10179352

[33] Zifeng Kang, Song Li, and Yinzhi Cao. 2022. Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites. In *29th Annual Network and Distributed System Security Symposium, (NDSS '22)*. The Internet Society, San Diego, CA. https://www.ndss-symposium.org/ndss-paper/auto-draft-207/

[34] Andrei Kashcha. 2023. npm rank. Retrieved 2 February 2023 from https://gist.github.com/anvaka/8e8fa57c7ee1350e3491

[35] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. Hong Kong, China, 121–132. https://doi.org/10.1145/2635868.2635904

[36] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE '13)*. San Francisco, CA, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[37] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '20)*. Virtual, 121–134. https://www.usenix.org/conference/raid2020/presentation/koishybayev

[38] P. Ladisa, H. Plate, M. Martinez, and O. Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP '23)*. San Francisco, CA, 1509–1526. https://doi.org/10.1109/SP46215.2023.10179304

[39] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *19th International Workshop on Foundations of Object-Oriented Languages (FOOL '12)*, Vol. 10. Tuscon, AZ. https://github.com/sukyoung/safe

[40] Song Li. 2021. ODGen Source Code. Retrieved 11 January 2024 from https://github.com/Song-Li/ODGen/.

[41] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. 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 (ESEC/FSE 2021). Athens, Greece, 268–279. https://doi.org/10.1145/3468264.3468542

[42] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *31st USENIX Security Symposium (SEC '22)*. Boston, MA, 143–160. https://www.usenix.org/conference/usenixsecurity22/presentation/li-song

[43] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Paderborn, Germany, 727–739. https://doi.org/10.1145/3106237.3106253

[44] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. St. Petersburg, FL, 298–312. https://doi.org/10.1145/2837614.2837617

[45] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN '17)*. Santa Barbara, CA, 196–199. https://doi.org/10.1145/3092282.3092295

[46] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '21)*. Taipei, Taiwan, 569–580. https://doi.org/10.1109/DSN48987.2021.00065

[47] Mozilla. 2023. Inheritance and the prototype chain - JavaScript | MDN. Retrieved 13 November 2023 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

[48] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: Feedback-Driven Static Analysis of Node.Js Applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. Tallinn, Estonia, 455–465. https://doi.org/10.1145/3338906.3338933

[49] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular Call Graph Construction for Security Scanning of Node.Js Applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21)*. Virtual, 29–41. https://doi.org/10.1145/3460319.3464836

[50] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '20)*. Lisbon, Portugal, 23–43. https://doi.org/10.1007/978-3-030-52683-2_2

[51] Chinenye Okafor, Taylor R. Schorlemmer, Santiago Torres-Arias, and James C. Davis. 2022. SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED'22)*. Los Angeles, CA, 15–24. https://doi.org/10.1145/3560835.3564556

[52] Jarrod Overson. 2021. How Two Malicious NPM Packages Targeted & Sabotaged Others. Retrieved 2 February 2023 from https://jsoverson.medium.com/how-two-malicious-npm-packages-targeted-sabotaged-one-other-fed7199099c8

[53] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. 2021. Accelerating JavaScript Static Analysis via Dynamic Shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. Athens, Greece, 1129–1140. https://doi.org/10.1145/3468264.3468556

[54] Mathias Payer and Thomas R Gross. 2013. Hot-patching A Web Server: A Case Study of ASAP Code Repair. In *Proceedings of the 11th Eleventh Annual Conference on Privacy, Security and Trust (PST '13)*. Tarragona, Spain, 143–150. https://doi.org/10.1109/PST.2013.6596048

[55] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. 2018. Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*. Frankfurt am Main, Germany, Article 11, 14 pages. https://doi.org/10.1145/3236950.3236956

[56] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. Oakland, CA, 513–528. https://doi.org/10.1109/SP.2010.38

[57] Max Schaefer. 2023. Amalfi Classifier. Private email communication.

[58] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and privacy (SP '10)*. Oakland, CA, 317–331. https://doi.org/10.1109/SP.2010.26

[59] Dominik Seifert, Michael Wan, Jane Hsu, and Benson Yeh. 2022. An Asynchronous Call Graph for JavaScript. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22)*. Pittsburgh, PA, 29–30. https://doi.org/10.1145/3510457.3513059

[60] Adriana Sejfia and Max Schäfer. 2022. Practical Automated Detection of Malicious Npm Packages. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Pittsburgh, PA, 1681–1692. https://doi.org/10.1145/3510003.3510104

[61] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. Saint Petersburg, Russia, 488–498. https://doi.org/10.1145/2491411.2491447

[62] Ax Sharma. 2020. Trick or treat: that 'twilio-npm' package is brandjacking malware in disguise! Retrieved 10 March 2023 from https://blog.sonatype.com/twilio-npm-is-brandjacking-malware-in-disguise

[63] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. 2023. Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js. In *Proceedings of the 32th USENIX Security Symposium (SEC '23)*. Anaheim, CA. https://www.usenix.org/conference/usenixsecurity23/presentation/shcherbakov

[64] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. 2022. Backporting Security Patches of Web Applications: A Prototype Design and Implementation on Injection Vulnerability Patches. In *31st USENIX Security Symposium (SEC '22)*. Boston, MA, 1993–2010. https://www.usenix.org/conference/usenixsecurity22/presentation/shi

[65] Snyk. 2022. A post-mortem of the malicious event-stream backdoor | Snyk. Retrieved 2 February 2023 from https://snyk.io/blog/a-post-mortem-of-the-malicious-event-stream-backdoor

[66] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *Proceedings of the 27th USENIX Security Symposium (SEC '18)*. Baltimore, MD, 361–376. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

[67] Liran Tal. 2019. Malicious remote code execution backdoor discovered in the popular bootstrap-sass Ruby gem. Retrieved 1 August 2023 from https://snyk.io/blog/malicious-remote-code-execution-backdoor-discovered-in-the-popular-bootstrap-sass-ruby-gem

[68] Matthew Taylor, Ruturaj K. Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending Against Package Typosquatting. In *Proceedings of the 14th International Conference on Network and System Security (NSS '20)*, Vol. 12570. Springer, Melbourne, Australia, 112–131. https://doi.org/10.1007/978-3-030-65745-1_7

[69] Tajkia Rahman Toma and Md Shariful Islam. 2014. An efficient mechanism of generating call graph for JavaScript using dynamic analysis in web application. In *Proceedings of the International Conference on Informatics, Electronics & Vision (ICIEV '14)*. Dhaka, Bangladesh, 1–6. https://doi.org/10.1109/ICIEV.2014.6850807

[70] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87–97. https://doi.org/10.1145/1542476.1542486

[71] Ruturaj K. Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. 2021. Security Issues in Language-based Software Ecosystems. arXiv:1903.02613 http://arxiv.org/abs/1903.02613

[72] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing Dynamic Library Compromise on Node.Js via RWX-Based Privilege Reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Virtual, 1821–1838. https://doi.org/10.1145/3460120.3484535

[73] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Gothenburg, Sweden, 1–11. https://doi.org/10.1145/3180155.3180233

[74] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. 2021. Abusing Hidden Properties to Attack the Node.js Ecosystem. In *Proceedings of the 30th USENIX Security Symposium (SEC '21)*. Virtual, 2951–2968. https://www.usenix.org/conference/usenixsecurity21/presentation/xiao

[75] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. 2020. Automatic Hot Patch Generation for Android Kernels. In *29th USENIX Security Symposium (SEC '20)*. Virtual, 2397–2414. https://www.usenix.org/conference/usenixsecurity20/presentation/xu

[76] Mingxue Zhang and Wei Meng. 2021. JSISOLATE: Lightweight in-Browser JavaScript Isolation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. Athens, Greece, 193–204. https://doi.org/10.1145/3468264.3468577

[77] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Smallworld with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC '19)*. Santa Clara, CA, 995–1010. https://www.usenix.net/system/files/sec19-zimmermann.pdf