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, Yinzhı 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE ’24, April 14–20, 2024, Lisbon, Portugal © 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9217-4/24/04
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 dependencies, 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
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.

**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

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

(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 1: RogueOne architecture.**

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

- **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 own 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 pubClient.publish(secret) because the value of the latter depends on the value of former, but...
4 DESIGN

4.1 Update-Aware Abstract Interpretation

In the code for the former, 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.

(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.

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

4.2 ODRG Construction

Using the set of high-level operations that result from abstract interpretation, RogueOne constructs an ODRG, which 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 → and the latter as −→.

(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 −→ o2 and a data-flow edge o1 → 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 −→ 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 → result, p1 → result, p2 → result... because the function and the parameters all influence result. Adds an owns edge func −→ 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.

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:

\[
\text{OwnsReaches}(\text{trustDomain}) = \{ \text{obj} | \text{obj} \text{ is reachable along owns edges from a root of trustDomain} \}
\]

\[
\text{DataFlowReaches}(\text{trustDomain}) = \{ \text{obj} | \text{obj} \text{ is reachable along data-flow edges from a root of trustDomain} \}
\]

\[
\text{Flows}(\text{ODRG}) = \{ (A \rightarrow B) | \exists \text{ obj such that obj \in DataFlowReaches(A) \land obj \in OwnsReaches(B)} \}
\]

The result is a set of cross-domain data-flows for the target package: \{(t1 \rightarrow t2), (t3 \rightarrow 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:

- \texttt{OwnsReaches(process)} = \{ \texttt{process}, \texttt{env}, \texttt{secret} \}
- \texttt{OwnsReaches(secure-store)} = \{ \texttt{secClient}, \texttt{secret}, \texttt{secQuery}, \texttt{result}, \texttt{pubData} \}
- \texttt{OwnsReaches(public-store)} = \{ \texttt{pubClient}, \texttt{publish}, \texttt{pubResult1}, \texttt{pubResult2} \}

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

- \texttt{(process \rightarrow secure-store)}: Occurs at \texttt{secClient}
- \texttt{(secure-store \rightarrow process)}: Occurs at \texttt{secret}
- \texttt{(secure-store \rightarrow public-store)}: Occurs at \texttt{pubResult1}
- \texttt{(process \rightarrow public-store)}: Occurs post-update at \texttt{* pubResult2*}

**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 \textsc{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 to that dependency from the flow set. This reduces false positives, especially those resulting from new dependencies, as discussed in §7.3. Where possible, \textsc{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 \texttt{FlowSets} of data-flow sets, along with \texttt{Flows(T)} for the original target package.
2. If, for any dependency \(D\):

   \[
   \text{Flows}(D) \subseteq (\{(T \rightarrow D), (D \rightarrow T)\} \cup \text{Flows}(T))
   \]

   then \(D\) is a dead-end for data-flows. Consider an update to the target package which adds the data-flow \((X \rightarrow D)\), 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. \textsc{RogueOne} removes \(D\) from \texttt{FlowSets} and deletes any data-flow containing \(D\) from all elements of \texttt{FlowSets} and \texttt{Flows(T)}.

3. The previous step is repeated until no such 'dead end' dependencies remain.

The resulting \texttt{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, \textsc{RogueOne} takes advantage of the stability of the cross-domain flow set of a typical package. \textsc{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: 

\[
\text{newFlows} = \text{flowsAfterUpdate} \setminus \text{flowsBeforeUpdate}
\]

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 \((\text{process} \rightarrow \text{public-store})\) only occurs in the post-update version, and the update is flagged.

### 5 IMPLEMENTATION

The \textsc{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 ECMA Script 5 with limited support for features up to ECMA Script 2018. JavaScript behavior is modeled within ODGen using a statement-by-statement simulation of the behavior of NodeJS 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 \texttt{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 \textsc{RogueOne}'s analysis is present.

**Handling Prototypical Inheritance**  
In all JavaScript runtimes, upon creation, every object contains a \texttt{obj.__proto__} property connecting it to its class' prototype chain [47]. Traversing the \texttt{obj.__proto__} chain to its end will always reach \texttt{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, \texttt{obj.__proto__.\_owms}, which are used for differential analysis. To prevent every trust domain from simultaneously owning \texttt{Object.prototype}, \textsc{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 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.

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 which 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. 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’

\[\text{footnote}[1]{An example of this can be seen in the package jade, included in our artifact.}
\[\text{footnote}[2]{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, 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 69de276a4d52) 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

\(^5\)We omit any package without updates.

\(^6\)In the Maloss system this component is called compare-ast.
Table 1: Classification Accuracy on Multi-version Dataset

<table>
<thead>
<tr>
<th>System</th>
<th>Rogue Updates</th>
<th>Benign Updates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Flagged</td>
<td>Missed</td>
</tr>
<tr>
<td>RogueOne</td>
<td>320 98.2%</td>
<td>5 12.2%</td>
</tr>
<tr>
<td>RogueOne-Paranoid</td>
<td>7 100%</td>
<td>0 0%</td>
</tr>
<tr>
<td>AmaFi-Bates</td>
<td>28.6% 5 71.4%</td>
<td></td>
</tr>
<tr>
<td>AmaFi-DTree</td>
<td>1 14.3% 6 85.7%</td>
<td></td>
</tr>
<tr>
<td>AmaFi-SVM</td>
<td>0 0% 7 100%</td>
<td></td>
</tr>
<tr>
<td>Maloss</td>
<td>7 100% 0 0%</td>
<td></td>
</tr>
<tr>
<td>Maloss-STATIC</td>
<td>28.6% 5 71.4%</td>
<td></td>
</tr>
<tr>
<td>Maloss-SDIFF</td>
<td>1 14.3% 6 85.7%</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Classification Accuracy on Single-update Dataset

<table>
<thead>
<tr>
<th>System</th>
<th>Rogue Updates</th>
<th>Benign Updates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Flagged</td>
<td>Missed</td>
</tr>
<tr>
<td>RogueOne</td>
<td>31 75.6% 10 24.4%</td>
<td></td>
</tr>
<tr>
<td>RogueOne-Paranoid</td>
<td>38 92.7% 3 7.3%</td>
<td></td>
</tr>
<tr>
<td>Maloss</td>
<td>14 34.1% 27 65.9%</td>
<td></td>
</tr>
<tr>
<td>Maloss-STATIC</td>
<td>13 31.7% 28 68.3%</td>
<td></td>
</tr>
<tr>
<td>Maloss-SDIFF</td>
<td>5 12.2% 36 87.8%</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: True Positives and Effectiveness By Attack Type

<table>
<thead>
<tr>
<th>Attack</th>
<th>Multi</th>
<th>Single</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Harvest</td>
<td>0</td>
<td>22</td>
<td>100% 100%</td>
</tr>
<tr>
<td>CC-Harvest</td>
<td>0</td>
<td>10</td>
<td>80%   0%</td>
</tr>
<tr>
<td>S-Harvest</td>
<td>2</td>
<td>1</td>
<td>100% 100%</td>
</tr>
<tr>
<td>Dependency</td>
<td>1</td>
<td>1</td>
<td>100% 100%</td>
</tr>
<tr>
<td>M-FS-Access</td>
<td>1</td>
<td>3</td>
<td>75%   75%</td>
</tr>
<tr>
<td>Remote Ctrl</td>
<td>3</td>
<td>4</td>
<td>100% 100%</td>
</tr>
<tr>
<td>Total</td>
<td>7</td>
<td>41</td>
<td>94%   79%</td>
</tr>
</tbody>
</table>

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 AmaFi’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. AmaFi is excluded since the AmaFi 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 variant of Maloss 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.

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.

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
Table 4 lists false positives across both configurations of ROGUEOne.

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, 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.

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 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, 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.
updates. RogueOne [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 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 sub-problem. 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. SAFEIR [53] adopts Jalangi [61], a dynamic analysis tool which selectively records and replays front-end and back-end JavaScript programs, to build dynamic shortcuts on top of SAFE, which speeds up static analysis of large packages such as Lodash. SAFEIR 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.