2024 IEEE Secure Development Conference (SecDev) | 979-8-3503-9193-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/SecDev61143.2024.00017

# GENIE: Guarding the npm Ecosystem with Semantic Malware Detection

Matías F. Gobbi[*]
*Bundeswehr University Munich, Germany*
matias.gobbi@unibw.de

Johannes Kinder
*LMU Munich, Germany*
johannes.kinder@lmu.de

***Abstract*—Package managers and public repositories such as npm streamline the distribution and maintenance of open source code. At the same time, they have become attractive targets for malicious actors to spread malware to many potential victims. In malware campaigns, families of malicious JavaScript packages exhibit common malicious behavior but differ in their names and syntactic details. We propose to thwart malware campaigns by developing semantic specifications to match similar malware with a single behavioral signature. Specifically, we report on our experience in using CodeQL to describe malicious behavior in JavaScript code, which allows us to employ an existing and mature static analysis framework as a robust building block. We describe a methodology and tool set for developing queries for newly reported and previously undetected malware, so that a single report can be used to take down entire families of similar malware. Applying our approach, we were able to discover 125 previously unreported malicious packages, which we reported and had removed from npm, without producing a single false alarm. As a result, we find that the upfront investment of developing semantic signatures in comparison to automatically learning classifiers pays off with the increased reliability of results by saving on manual effort for validation and relabeling.**

***Index Terms*—malware, npm, static analysis**

## I. Introduction

Package managers provide developers with an effective mechanism for sharing, including, and updating code; yet, they have profound security implications. For every package dependency included in a project, trust is implicitly placed in the registry to ensure authenticity and integrity of that package. When malicious actors target package managers to include malware in packages, this may directly affect developer machines or give rise to software supply chain attacks where malicious code is included in downstream projects.

The package manager npm for Node.js is the largest public database of JavaScript software and hosts more than two million packages with more than fifty billion weekly downloads. Characteristic to npm is the high number of transitive dependencies among packages [1], which is aggravated by the reliance on packages that consist of only a few lines of code [2]. This fragmentation of the codebase increases the attack surface, as each of these packages with its developers and delivery infrastructure has to be trusted [3]. Indeed, these concerns are warranted by numerous attacks targeting npm. For instance, a malicious dependency for attacking cryptocurrency wallets was added to the popular `event-stream` package through social engineering [4]; an update to the `eslint-scope` library contained a data exfiltration script after maintainer credentials were compromised [5]; and the `pure-script` installer was broken when some of its dependencies were leveraged for a denial of service attack [6].

Many attacks do not involve novel techniques. As reported by Ohm et al. [7], malicious code on public repositories often appears in clusters of closely related packages that are published in waves. For instance, the `crossenv` incident refers to a collection of almost 40 packages posted under names similar to existing packages (typosquatting) that performed data exfiltration [8]. Sometimes, these campaigns are not intrinsically malicious, but still violate the terms of service of the respective registry: security researchers hunting after bug bounties commonly publish probe packages under various names in an attempt to exploit *dependency confusion* vulnerabilities, where an external malicious package is given preference over an internal package of the same name [9].

Despite npm scanning packages in search of known malicious content [10], the security of the ecosystem largely relies on its community. Users submit reports of malware via the npm website, which are then manually validated. This means that the responsibility for ensuring the security of the package ecosystem is a joint effort between registry users and registry maintainers. This mechanism is limited in what it can achieve; consequently, external security vendors regularly report malware on npm [11, 12, 13]. As of today, there is no clear solution to malware detection on open source package managers. Several solutions have been proposed in the literature, ranging from overall malware detection [14, 15, 16] and capturing malicious patterns [17, 18, 19, 20, 21] to general approaches for hardening the ecosystem [22, 23, 24, 25].

In this paper, we propose to tie the existing process of reporting malicious packages on npm into a methodology to greatly enhance its effectiveness. Using a single reported package as a template, we develop a semantic signature as a CodeQL query that will match the same malicious behavior in other packages. Effectively, a single report can thus take down an entire malware campaign. Such semantic signatures have been proposed for malware detection in the past [26, 27, 28, 29, 30, 31] but have not been widely adopted due to brittle tool-chains and the complexity of specification logics and formalisms [32]. However, we believe that static analysis tools designed for robustness and ease of use in

[*]Also with LMU Munich, Germany.

979-8-3503-9193-0/24/$31.00 ©2024 IEEE
DOI 10.1109/SecDev61143.2024.00017

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.

```
1 import javascript
2
3 from ClientRequest request, string url
4 where url = request.getUrl().getStringValue()
5 select request, "Request to " + url
```

**Listing 1.** Example CodeQL query to detect requests.

writing queries, such as CodeQL, are well-equipped to be used in this regard. While machine learning is widely seen as the state of the art in malware detection [15], practical deployments come with challenges of their own and require large and diverse training sets [33, 34]. We argue that it is time for a renaissance of semantic signatures, which promise targeted and explainable detection of malware families from just a single specimen with very low false positive rates. Using our approach, we developed 12 queries from known malicious packages and used these to detect 125 previously undetected malicious packages on npm that were subsequently removed, while not encountering a single false positive. We address code obfuscation, a common impediment to static analysis, by introducing queries aimed at detecting common obfuscation styles in JavaScript. The main contributions of our work are as follows:

- We demonstrate the feasibility of using CodeQL to develop semantic signatures of malware that are effective in practice. Using our queries, we discovered a total of 125 previously unreported malicious packages in npm that were subsequently removed.
- We define queries for detecting obfuscated JavaScript code, study its prevalence and usage on npm, and discuss possible policies for dealing with obfuscation in the open source ecosystem.
- We implemented our approach in GENIE (Guarding thE Npm Ecosystem), which we make available together with all CodeQL queries, the found malware samples, and all data gathered during our analysis.[1][2]

## II. STATIC ANALYSIS WITH CODEQL

CodeQL is a semantic code analysis framework designed to find bugs and vulnerabilities in an application by running queries against a database derived from the source code. The database is created once, before applying any queries. The analysis engine extracts the abstract syntax tree, control flow graph, and data flow graph from source code and encodes it into the database [35, 36].

Queries are written in the declarative, object-oriented logic programming language QL, which is built on Datalog [35, 36]. A set of libraries provides convenience methods and language support. For instance, there are pre-built queries implementing control flow and data flow analyses, and specific support for JavaScript. When applying queries, the produced results are styled as alerts that point directly to code elements. There

```
const trackingData = JSON.stringify({
  hd: os.homedir(),
  hn: os.hostname(),
  un: os.userInfo().username,
});

var postData = querystring.stringify({
                 msg: trackingData,
               });

var options = {
  hostname : "<attacker-controlled-address>",
  method   : "POST",
  ...
};

var req = https.request(options,
            (res) => {res.on("data", (d) => {});}
          );

req.on("error", (e) => {});

req.write(postData);
req.end();
```

**Listing 2.** Simplified source code from malicious package.

are two types of queries: *alert queries* provide a source code location of the match and a description of the issue, while *path queries* display the complete flow of data between the specified source and sink involved in the issue. Path queries can be used for taint-style static information flow analysis, which is particularly powerful for abstractly specifying malicious behavior. CodeQL provides human-readable reports by interpreting the query results based on metadata such as the severity of each query.

Listing 1 shows an example query for detecting all requests, using a SQL-like `from.where.select` clause. The variables `request` and `url` range over all requests in the program and all data flow nodes, respectively. Line 4 forces the second variable to be the URL of the first variable, line 5 then specifies the query results, i.e., tuples of source locations of the requests and custom-built alert messages. Note that this query detects requests made with any of CodeQL-supported JavaScript library. This out-of-the-box library support is a key practical advantage of CodeQL in comparison to alternatives such as Joern [37] or Semgrep, where we would need to define new patterns for each library.

## III. THREAT MODEL

We consider an adversary that publishes malicious packages to the npm registry, in an attempt to perform attacks when the package is downloaded and installed on a victim's machine, either as a package dependency or as a standalone application. In this paper, we focus on an attacker interested in reaching a wide range of victims by publishing many packages with semantically similar malicious behavior. The specific malicious goals are diverse and may include stealing sensitive information from the victim and sending it back to the attacker; tampering with, abusing, or destroying systems

[1] https://github.com/lmu-plai/GENIE

[2] https://doi.org/10.5281/zenodo.13143816

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.

| Category | API Function | Source | Sink |
|---|---|---|---|
| Network | `http.get` | × | |
| | `request.write` | | × |
| | `dns.resolve` | × | × |
| CodeGen | `eval` | | × |
| | `vm.runInContext` | | × |
| File System | `fs.readFile` | × | |
| | `fs.writeFile` | | × |
| | `os.homedir` | × | |
| Process | `child_process.exec` | | × |
| | `process.env` | × | |

**Table I.** Examples of commonly seen sources and sinks in malicious packages.

and computational resources; or remote backdoor access to the victim's machine for further attacks.

An attacker may apply one of two major strategies to make their malicious code publicly available in the npm registry. First, they can *infect an existing package* that has an established direct user base or is part of the dependency tree of other packages. This requires either social engineering or other targeted attacks against existing developers and their credentials or the source code repository. Second, they can *submit a new package* containing malicious code and try to trick developers into using or depending on it through *typosquatting* or constructing a *trojan horse*. Typosquatting refers to using names similar to popular packages, hoping for developers to mistype dependencies. Trojan horses are legitimate-looking packages containing hidden malicious behavior. Such new malicious packages are often replicated multiple times to increase the chance of infection and will differ only slightly in functionality [7, 38].

In this work, we focus on raising the bar for attackers using the second strategy by detecting entire families of related malicious code with a single semantic signature. While the queries we develop will also match malicious code inserted into an existing package, we exclude completely novel malicious behavior, which is typically best dealt with using anomaly detection [15, 39, 21]. In Listing 2 we show a simplified version of an actual malicious sample uploaded to the registry, which steals information when installed. The package gathers operating system-related data (lines 1 to 5), makes a request to an adversary-controlled remote address (lines 17 to 19), and sends the data following some simple formatting (line 23).

## IV. Methodology

We propose a new methodology to leverage CodeQL to detect and take down families of malicious code on npm. From malicious samples mined from npm, we define a semantic signature in CodeQL, which is then applied to the entire registry. The process requires a human in the loop with expertise similar to that of a malware analyst in the traditional process of generating static signatures or rules.

To support this methodology, we developed Genie, a framework that allows a developer to maintain a local snapshot of the package registry, detect when a malicious package is taken down, create and query CodeQL databases, while enabling efficient parallel processing and monitoring. A key benefit of Genie is that it enables a third party to do the analysis without having to rely on npm to deploy the approach repository-wide.

| Query | Trigger | Behavior | #D | #R |
|---|---|---|---|---|
| dependency-install | Install | Virus | 1 | 0 |
| dependency-save | Install | Virus | 48 | 0 |
| discord-injection | Runtime | Backdoor | 1 | 0 |
| discord-steal | Runtime | Stealing | 1 | 0 |
| discord-ware | Install | Sabotage | 1 | 0 |
| other-request | Install | Unknown | 1 | 0 |
| other-shell | Runtime | Unknown | 1 | 0 |
| theft-dns | Install | Stealing | 15 | 91 |
| theft-encoded | Install | Stealing | 1 | 0 |
| theft-environment | Install | Stealing | 3 | 11 |
| theft-os | Install | Stealing | 1 | 17 |
| theft-ping | Install | Stealing | 1 | 6 |
| **Total** | | | **75** | **125** |

**Table II.** Taxonomy with the number of malware samples in each class in the dataset of malware we detected as removed (#D), and the number of malware samples in each class detected by applying our queries to the registry (#R).

In the following, we detail the four steps involved in developing semantic signatures with Genie (§IV-A–§IV-D). Using the malware example from Listing 2 as inspiration, we then develop a specific query capturing its behavior in Listing 3 (§IV-E). Finally, we demonstrate how to address the issue of code obfuscations (§IV-F).

### A. Detecting Removed Packages

First, we detect packages that have been removed from the registry, as this can be a sign for a malware report. When a malicious package is reported to npm, the npm security team publishes a security placeholder in its place, alerting the community and stopping any further damage. We use this fact to detect recent malware reports and employ them to find further instances of ongoing malware campaigns. Although not every removed package is malicious, and not every malicious package belongs to a wider malware campaign with additional, similar packages, we can show that this method is very effective in practice. Central to this technique is monitoring the package repository for removed packages.

We periodically mirror the repository in its entirety such that we still have a copy of a malicious package after removal. If the registry maintainer cooperates, this step could be simplified and deployed at the site of the registry itself. Through Genie, we can perform this action while being external to npm.

### B. Manual Inspection

Second, a manual source code analysis searches for common patterns seen in malware [7, 14, 15], such as the requests made, the spawning of processes, the interactions with the file system, or the runtime generation and loading of code. This is

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.

```
class TTConfiguration extends TaintTracking::Configuration {

  predicate isSource(Node source) {
    exists( SourceNode os
          | os = moduleMember("os", ["hostname", "homedir", "userInfo"])
          | os = source.(InvokeNode).getCalleeNode()
          )
  }
  predicate isSink(Node sink) {
    exists( ClientRequest client
          | sink = client.getAMemberCall("write").getAnArgument()
          )
  }
  predicate isAdditionalTaintStep(Node pred, Node succ) {
    exists( PropWrite propWrite
          | propWrite.writes(succ, _, pred)
          )
  }
}

from TTConfiguration cfg, PathNode source, PathNode sink
where cfg.hasFlowPath(source, sink)
select source, sink, "Detected suspicious flow of information"
```

**Listing 3.** Simplified query that matches the malicious package.

inherently a custom process, but it generally follows a common strategy. Particularly, we aim to identify information flows between meaningful sources and sinks in the source code. Table I lists examples of sources and sinks frequently seen in malicious packages. We refer the reader to Duan et al. [14] for an extensive list of manually labeled APIs. Besides data flows, installation routines and libraries required by a package provide hints of where to look for malicious code. Following the example of Listing 2, we can identify potential sources in the calls to `os.hostname()`, `os.homedir()`, and `os.userInfo()`, where specific operating system information is gathered, and a potential sink that uploads this information to a web server with a POST request in `req.write(...)`.

This step also distinguishes malware that is out of scope for source code analysis, e.g., due to the malicious behavior residing in executable binaries, which should be addressed through orthogonal detection methods.

### C. *Query Development*

Third, having identified the malicious behavior in source code, we define a corresponding CodeQL query. We start by matching either the source or the sink of the flow with CodeQL, and then connect the other end via the definition of increasingly general predicates. Following the example in Listing 2, besides having predicates for the identified sources and sink, we need to make sure that the data flow connects both nodes. We declare an additional predicate to propagate the taint from the initial variable to the entire object in the argument of the `querystring.stringify(...)` method call. As this has the potential for generating false positives when applying the query to the entire registry, in practice we have to refine the query further. For instance, in this case we only query for packages with detected flows from all three of the identified sources to the sink (omitted from the example for brevity).

Having captured the malicious flow in the package, we restrict the query to make it specific to the sample we are working on. It has to be general enough to be able to find other malware samples in the wild, but at the same time, needs to be accurate enough to match only actually malicious packages avoiding false alarms. For that, the query developer has to capture, with the written query, the semantics of each relevant step from the flow of data in the sample. A common pattern seen in information stealing packages, is to encode the stolen data before exfiltration; for instance, applying `Buffer.from(...).toString("<encoding>")`. A query developer should aim to express this serialization step in the captured flow. In practice, one can define a taint tracking configuration for the flow from the source to this intermediate step and another one from the intermediate step to the sink. While code like Listing 2 is prohibited on npm, there is no clear policy regulating user tracking, so we have to distinguish the type of information being exfiltrated.

### D. *Application to Registry*

Finally, after developing a query for a removed malicious package, we apply it to the entire registry. For each matched sample, the engine generates an alert message with the requested information specified in the query together with the location in the source code responsible for the match. After manually validating the produced alerts in the package source code correspond to actual malicious code, we report them to the npm security team.

### E. *Example Query*

Following our proposed methodology, we develop a query specifically matching the behavior from the source code described in Listing 2, to detect similar packages in the registry. In Listing 3, we define a custom taint tracking configuration

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.

(lines 1 to 19) for this attack pattern. Extending this class enables us to capture even non-value-preserving data flows, such as those passing through the `querystring.stringify(...)` method call. We declare the source node (lines 3 to 8) of the configuration, which retrieves specific operative system information via the calls to `os.hostname()`, `os.homedir()`, and `os.userInfo()`. We declare the sink node (9 to 13) of the configuration, which uploads this information to a web server via passing it as an argument of a POST request, `req.write(...)`. We define an additional taint step (14 to 18) specifying that storing information in an object property should propagate the taint between nodes. Finally, with the `from.where.select` clause (lines 21 to 23), we select all source-sink pairings where there is a flow satisfying the conditions from the defined configuration.

### *F. Detecting Code Obfuscation with CodeQL*

Static analysis approaches to malware detection are known for their limitations when faced with certain forms of code obfuscation [32, 40]. Obfuscation transforms code into an equivalent form that is harder to interpret for both human analysts and program analysis tools. Ohm et al. [7] report that almost half of malware they gathered in a study leveraged some kind of obfuscation technique.

Due to the nature of its analysis, CodeQL is immune to basic obfuscation techniques such as *minification* (removing all unnecessary characters from a program's source code) or *variable renaming* (replacing identifiers with randomly generated names). Hence, our approach by design directly supports the analysis of minified source files, which are relatively common in deployed packages.

Examples of transformations that do thwart CodeQL are *control-flow flattening* (having basic blocks inside a loop where a dispatcher controls the program flow), *encoding* (converting data to a different representation), *string concealing* (computing strings literal values during runtime), *variable masking* (hiding variables inside arrays), and *dot-to-bracket notation* (performing property accesses using dynamic keys instead of doing it with static keys).

A popular open-source obfuscator used by numerous npm packages is the *JavaScript Obfuscator*.[3] The tool provides a large set of potent code transformations that can be applied to a target program. Based on the chosen degree of protection and the acceptable performance hit to the application, there are multiple preset configurations. Even in its default settings, the obfuscator performs transformations that prevent CodeQL from completely modeling the dataflow. For example, the statement `console.log("Hello World!")` can be obfuscated to the equivalent code in Listing 4.

```
var _0x3b8fdd=_0x599d;function _0x10ad(){var
    ↪ _0x211b6a=['36gLyCea','539775bVxIep','
    ↪ Hello\x20World!','14SYFEtK','5154824AUhDey
    ↪ ','2631912ISGkUi','253730ViqKpg','log','
    ↪ 3500740lAyOos','1503258dhExgS','98412
    ↪ VnHkaO','6IUxeuq'];_0x10ad=function(){
    ↪ return _0x211b6a;};return _0x10ad();}
    ↪ function _0x599d(_0x102814,_0x3deeb4){var
    ↪ _0x10ad9e=_0x10ad();return _0x599d=
    ↪ function(_0x599d03,_0x431549){_0x599d03=
    ↪ _0x599d03-0x1d7;var _0x34436c=_0x10ad9e[
    ↪ _0x599d03];return _0x34436c;},_0x599d(
    ↪ _0x102814,_0x3deeb4);}(function(_0x4c79a6,
    ↪ _0x359279){var _0x4483ff=_0x599d,_0x305664
    ↪ =_0x4c79a6();while(!![]){try{var _0x267b65
    ↪ =parseInt(_0x4483ff(0x1d8))/0x1+-parseInt(
    ↪ _0x4483ff(0x1e2))/0x2*(parseInt(_0x4483ff(
    ↪ 0x1e1))/0x3)+-parseInt(_0x4483ff(0x1dc))/0
    ↪ x4+parseInt(_0x4483ff(0x1df))/0x5+parseInt
    ↪ (_0x4483ff(0x1e0))/0x6*(-parseInt(
    ↪ _0x4483ff(0x1da))/0x7)+parseInt(_0x4483ff(
    ↪ 0x1db))/0x8+-parseInt(_0x4483ff(0x1d7))/0
    ↪ x9*(parseInt(_0x4483ff(0x1dd))/0xa);if(
    ↪ _0x267b65===_0x359279)break;else _0x305664
    ↪ ['push'](_0x305664['shift']());}catch(
    ↪ _0x4d466d){_0x305664['push'](_0x305664['
    ↪ shift']());}}}(_0x10ad,0x803fc),console[
    ↪ _0x3b8fdd(0x1de)](_0x3b8fdd(0x1d9)));
```

**Listing 4.** Obfuscated source code produced by applying the JavaScript Obfuscator in its default configuration.

It is no surprise that certain obfuscating transformations will thwart our approach to semantic malware detection, as CodeQL would not be able to model the malicious information flows specified. However, we argue that obfuscated code should be deemed suspicious per se, and that we should detect obfuscation and warn about it. Furthermore, while CodeQL is unable to precisely model information flow in obfuscated code, it is perfectly capable of detecting whether a package is obfuscated. To this end, we develop specialized CodeQL queries to detect artifacts of obfuscated code, specifically those generated by the obfuscations of the widely-used JavaScript Obfuscator. As a result, we treat obfuscated source code as instances of yet another malware family.

We consider the following *obfuscation artifacts* in JavaScript source code: **array** refers to the presence of an array of strings which contains literals and method names from the original source code. **parse** encodes taking specifically generated strings from the array, and trying to parse an integer from each one of them. **rotate** describes the shuffling of elements of the array until a magic number is successfully computed. From these patterns we developed queries against obfuscation and applied them to the entire package repository. We discuss the results in §V-D and the prevalence of obfuscated code on npm in §V-E.

## V. Evaluation

We now evaluate Genie and answer the following five research questions:

- **RQ1** Does our approach find malicious packages in practice, and is the amount of false alarms manageable?
- **RQ2** Does our method scale to the entire registry? How resource-intensive is CodeQL?
- **RQ3** How does our technique fare against a simpler baseline approach?

[3] https://github.com/javascript-obfuscator/javascript-obfuscator

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.

- **RQ4** Are we able to detect obfuscated code statically using CodeQL?
- **RQ5** How prevalent is obfuscated code in npm?

To answer these questions, we conduct a case study to find malware following our methodology (§V-A). While doing so, we measure the performance of the CodeQL engine to create and query databases of JavaScript packages (§V-B). Furthermore, we implemented a baseline approach using syntactic signatures for comparison (§V-C). Lastly, we evaluate the effectiveness of our obfuscation detection queries (§V-D) and assess the prevalence of obfuscated code in the registry (§V-E).

### A. *RQ1: Case Study*

We downloaded the latest version of more than 1.8 million packages from the registry from 23 to 25 May 2022. On 27 June 2022, one month after the initial download, we searched for security placeholders in the package repository and, when found, retrieved the copy we had stored locally to build a dataset of recently removed packages from npm. We collected a total of 91 recently removed packages, which we later transformed into a dataset of 75 malicious packages after an in-depth analysis and classification of each one. Of the excluded samples, 6 contained malicious shell scripts instead of JavaScript, and the remaining 10 we did not find to be malware (e.g., they were non-functional or empty). In Table II, we show characteristics of each documented malicious cluster. These packages were grouped by manually analyzing their source code and following the classification by Ohm et al. [7]. From the numbers in the dataset, we can see that at least three separate malware campaigns seem to have been running at the time of our analysis. Almost all samples execute their malicious behavior during installation, and the most common objective from these attacks appears to be data exfiltration.

According to our proposed methodology, after identifying twelve different malicious clusters, we defined CodeQL queries to specifically match the samples contained in each one of them. For their development, and further application, we used the `2.9.2` release of the CodeQL command-line toolchain along with the corresponding compatible libraries. The purpose of the queries can be summarized as follows: *dependency-** queries match packages including malware as dependency of the main application, while performing other malicious actions during installation (e.g., stealing the machine's npm cache or making requests to unknown addresses). With CodeQL, we parse the installation scripts of a package to detect these suspicious behaviors. Both clusters try to propagate in the ecosystem with probe samples. *discord-** queries match a variety of malware related to the Discord messaging platform. These range from stealing billing information or private tokens to attaching a malicious payload into the application's source code. With CodeQL, we detect suspicious file system operations (e.g., reading or writing Discord related files) and flows from sensible data to network requests. *theft-** queries match packages that steal personally identifiable information (PII) from the victim and send it to an attacker-controlled domain. The sources are mostly package-related metadata, operating system information and environment variables. The sinks are HTTPS requests, DNS queries, and even `ping` commands. Some of these clusters target *dependency confusion* vulnerabilities and display a security disclaimer in their source code (yet still exfiltrating PII). *other-** queries match packages that attempt to perform suspicious communication with a domain that was no longer available at the time of our analysis. Our CodeQL queries match a request made by a deprecated library and a request made through a spawned shell.

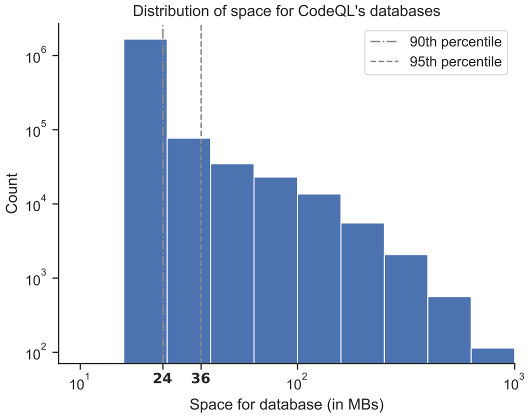

**Figure 1.** Space for CodeQL's databases.

The analysis of the source code and the following development of a CodeQL query to match a given cluster took on average between 30 to 45 minutes. This process was shorter when the source code was straightforward, the malicious behavior could be succinctly described with a custom dataflow configuration, or the malware performed unusual actions. When the samples had complex source code or implemented novel attacks, it took longer. Generally, the query development takes the majority of the analyst time for a given cluster.

We applied GENIE with the finalized queries to our snapshot of the npm registry to find malicious packages similar to those collected in our dataset of removed packages. Table II shows the results we obtained. In total, we found 125 available malicious packages in npm, which we individually confirmed as malicious, with no false positives. After reporting the packages, npm replaced all but one of them with a security placeholder for violating the terms of service. One package implementing user tracking remains on npm but is marked as deprecated and now maintained by npm. As a result, we were able to identify samples from four malware campaigns, with all of them performing data exfiltration (see column #R in Table II). Note that the remaining queries did not find any other samples outside of the collected dataset. This could mean that the removed packages were not part of an attack campaign, but one-off malware packages.

Note that the absence of false alarms is a consequence of the targeted approach we follow with the semantic malware signatures in CodeQL. Clearly we expect to eventually observe

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.

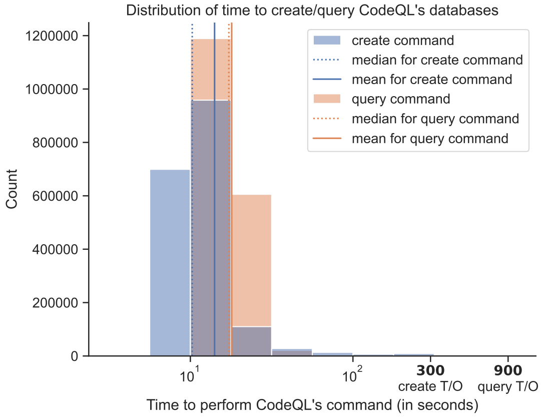

**Figure 2.** Time for CodeQL commands.

false positives, e.g., with borderline behavior; but by design our method is focused on having high precision (reducing the time spent manually reviewing its results), possibly at the expense of lower recall (potentially missing malicious packages in the registry). In summary, we can answer RQ1 affirmatively. Our approach does indeed detect malware in the npm registry. Furthermore, our first application shows promising results, considering the lack of false alarms.

### *B. RQ2: Performance of CodeQL*

Taking into account that we want to monitor the registry following our methodology, the performance of CodeQL when analyzing a JavaScript project is vital. We evaluate the computational feasibility of our proposed approach by measuring the size of the analyzed packages and their corresponding databases, and computing the time required to create each database and to apply all of our queries to it. The case study was conducted on five Debian servers with two AMD EPYC 7763 64-Core processors and 1 TB of RAM each. Our snapshot of the registry consists of, in total, more than 1.8 million packages.

First, we study the relationship between the size of an npm package and its corresponding CodeQL database. We computed the Pearson correlation coefficient between both variables and obtained a value of 0.64. This means that the size of the source code in a package loosely determines the size of the produced database. In Figure 1 we show the size distribution of CodeQL databases. The smallest databases need 18 MB of space, which is also the value of the median, where the mean is close to 22 MB. The biggest database, corresponding to the package `wyraz` requires almost 1 GB. In total, CodeQL needs close to 41 TB of space to store the databases of all the packages in the snapshot. Our results show that, even for small packages, a CodeQL database needs a considerable, but rather stable, amount of space for storage.

Second, we study the distribution of the time required to apply both CodeQL operations, creating and querying a database, shown in Figure 2. Initially, we built the databases for all the packages in our dataset. Setting a timeout of 300 seconds for the `database create` command, 95% of the databases where created in less than 22 seconds, with 0.3% timing out. From all the packages in the snapshot, 1% failed during the creation of their database for containing syntax errors. In our setup, the creation of databases took approximately 47 hours of wall-clock time.

Then, all of our twelve developed CodeQL queries were applied concurrently on each database. Setting a timeout of 900 seconds for the `database query` command, 95% of the databases were queried in less than 23 seconds, with only the package `@accordproject/ergo-compiler` timing out. From all the databases, less than 0.01% ran out of memory during querying. In our setup, the querying of databases took approximately 123 hours of wall-clock time. Note that the entire process is highly parallelizable, since both commands can be applied to multiple packages (or databases) at the same time. In our case study, we had 32 and 16 processes running simultaneously per server for the creation and querying of the databases, respectively. The command `database create` built a database every 0.09 seconds while `database query` applied all queries to a database every 0.24 seconds, both on average.

We can give a generally positive answer to RQ2, as our case study was successfully deployed. The resource requirements to deploy CodeQL at registry scale are considerable, especially given the number of package submissions to npm. A possible avenue for future work would be to investigate incremental analysis for large packages to reduce the cost of reanalyzing large code bases.

### *C. RQ3: Comparison against Baseline*

Considering our method requires human intervention to develop signatures, we would like to contrast it against more automated approaches. We chose to compare against a simple baseline using file hashes to detect exact copies of malware. Making use of our collected dataset of recently removed malicious packages, we measured the effectiveness of a technique capable of searching for exact copies of malware found in the registry, and assessed our approach against it.

The general idea of the baseline technique is to compute a single hash for a given package and see if it matches with the hash from a recently removed malicious package. Since the `package.json` file of a given npm package contains its metadata (like its unique package name), and there was no non-JavaScript-based malware in our dataset, we decided to only include `.js` and `.ts` files when calculating the hash of a given package. We computed the hash for each package in the collected dataset, and then searched for matches in our snapshot of the registry.

Clearly, this type of approach has the advantages of being simple to deploy, extremely fast (an average of 0.05 seconds per package), and space efficient (5 GB for all hashes vs. 50 TB for the CodeQL databases). However, the baseline approach did not find a single additional match for any malicious package, which suggests that either no duplicates were ever uploaded, or, more likely, that npm already implements a

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.

| Query | #npm | #sample | P | R | $F_1$ | TP | FP | TN | FN |
|---|---|---|---|---|---|---|---|---|---|
| array-parse&rotate | 376 | 163 | **1.00** | 0.77 | 0.87 | **163** | **0** | 20 | 47 |
| array-parse | 412 | 183 | **0.93** | 0.81 | 0.87 | **172** | **11** | 9 | 38 |
| array-rotate | 459 | 192 | **0.98** | 0.90 | 0.94 | **190** | **2** | 18 | 20 |
| array | 568 | 230 | **0.91** | 1.00 | 0.95 | **210** | **20** | 0 | 0 |

**Table III.** Metrics calculated over a sample of matched packages for our characterization of obfuscation.

form of de-duplication to protect against malware and spam. With respect to RQ3, we can conclude that GENIE clearly outperforms a simple baseline approach and is capable of finding malware that evaded the security mechanisms set up in the registry.

### D. RQ4: Detecting Obfuscation

Table III shows the results of applying our queries for detecting obfuscation to the entire package repository. We evaluate four different configurations of queries over obfuscation artifacts. Going from the least general query, which captures every described artifact, to the most general query, which detects heavy use of array values to compute keys for property accesses, we can partially order the sets of matched packages by inclusion. Any package flagged by a given query is also going to be flagged by all queries that have strictly fewer constraints.

To measure the performance of our queries, we manually analyzed the matched packages to verify that they were indeed obfuscated. Given the number of packages, we reviewed a sample for computing the metrics. Considering that the population size of matched packages was 568, we took a random sample of 230 to guarantee a confidence level of 95% that the real precision of the queries was within ±5% of the measured value. The metrics computed over the sample are displayed in Table III. It is worth noting that we only consider negatives from the selected sample and not from the entire registry, meaning that the true negatives (TN), the false negatives (FN), and the derived metrics, Recall (R) and $F_1$, just provide an intuition for the actual values. For deploying the obfuscation queries without introducing false positives, the first query specifying all artifacts is most suited. In contrast, the overall best $F_1$ performance is achieved by the most general query. On RQ4, we can conclude that it is indeed feasible to detect common code obfuscations, circumventing one of the major weaknesses of a static analysis approach.

### E. RQ5: Use of Obfuscation

We use our queries to detect obfuscated packages on npm to measure their popularity. To this end, we used an open-source utility to get download counts in npm[4], from the start of June 2022 to the end of May 2023, for all packages in our snapshot and focused on those matched by our most precise query.

The downloads of obfuscated packages only account for less than 0.0001% of the total number of downloads in the entire registry. More than 60% (236) of obfuscated packages had, on average, less than 1 daily download in the last year, while close to 85% (313) had less than 5 daily downloads. When manually studying some of the obfuscated packages, we noticed that most of them were undocumented basic utilities developed by individual maintainers, which used obfuscation in an attempt to protect their code from being reused without their permission.

Nevertheless, there are also some popular obfuscated packages in the registry. About 5% (20) had more than 25 daily downloads on average in the last year, and close to 1% (6) had more than 100 daily downloads. Some of the most popular obfuscated packages, including `lightrun`, `discord-dashboard`, or those from the `@salesforce/` scope, use obfuscation to protect a paid product or service associated with them. Even considering these exceptions, the reasoning behind the use of these techniques is unclear for most of the samples found in the registry. Packages without documentation nor repository, like `adzone`, or listing a repository which does not actually host its source code, like `raganork-bot`, end up being indistinguishable from obfuscated malware such as `test-pls`, which steals sensitive information when used.

Currently, there is no policy constraining the upload of obfuscated packages in the registry. Although an obfuscated package is not intrinsically malicious, neither humans nor static analysis-based tools can understand and vet its source code, effectively disabling basic control mechanisms of the open source community. Hence, such packages become particularly dangerous to rely on. Even though there are obfuscated packages on npm, most of them tend to be unpopular. We recommend that npm restrict developers from uploading packages with obfuscated source code, following policies such as the one of the Chrome Web Store [41]. Leveraging CodeQL as a detector of these techniques, it should be possible to automatically stop the upload of new obscured packages to the registry. Although banning obfuscation entirely would be preferable, it would be possible to allow exceptions for legitimate use cases with alternative closed-source vetting and certification, with appropriately posted warning messages.

## VI. DISCUSSION AND LIMITATIONS

We now report the lessons learned from our work and identify some conceptual and technical limitations.

Our static analysis technique leverages recently discovered malicious packages to develop highly specific CodeQL queries designed to match similar malware targeting package repositories, like npm. An application of the approach produced no

[4] https://github.com/kgryte/npm-package-download-counts

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.

false alarms, while detecting a total of 125 malicious packages. We find that the upfront investment of developing semantic signatures in comparison to automatically learning classifiers pays off with the increased reliability of results by saving on manual effort for validation and relabeling. In addition, a single report can be used to take down entire families of similar malware where the developed queries can be used to stop further similar attacks. Despite this, the approach has some limitations that we discuss in the following.

### A. Analysis Scope

The development of CodeQL queries is a manual process, which requires a level of expertise from the query developer similar of a malware analyst in the traditional process of generating static signatures or rules. Furthermore, we are not able to detect completely novel supply chain attacks, due to the fact that we require the detection and removal of an initial malicious package to use as template for the generation of a semantic signature. We trade recall for precision to minimize the time spent on manual vetting of false positives. Thus, GENIE can be seen as a first line of defense for package repositories rather than a comprehensive protection mechanism that detects all malware.

### B. Static Analysis

Given our technique is based on CodeQL, we are tied to the capabilities of its underlying analysis engine. Meaning that a limitation in CodeQL could lead to false negatives, as GENIE might not be able to detect certain types of malicious behavior. However, the tool is actively developed and maintained, so we expect that the prospect of our approach will improve as the tool evolves. As CodeQL is a static analysis framework, our proposed technique relies entirely on static analysis, inheriting all of its limitations. The dynamic nature of JavaScript is a particularly well-known challenge for static analysis, and there are unavoidable issues arising from runtime code generation and reflection, such as the infamous `eval`, which are also leveraged by multi-stage payload malware. Regardless of the programming language, the use of obfuscation techniques in source code, like string encryption or control flow flattening, hinders this analysis in particular, although we describe how to detect and prevent such obfuscations in §IV-F.

### C. Languages and Dependencies

We could in principle analyze code in all languages also supported by CodeQL. In packages where the malicious code is written in an unsupported language, such as a shell script, we are unable to model its behavior in a query to match it. A technical limitation of CodeQL is that when building a database from a given codebase, only code from a single language is considered; one exception is the combination of HTML and JavaScript, where CodeQL supports cross-language data flows due to this particular combination's importance. Related to this, we only consider the source code of the main package when building the corresponding database, which means that we are so far unable to model malware where the relevant behavior is split across packages. While this restriction is in principle easy to lift, including the source code of all dependencies in full will incur a significant cost. Summarization techniques could help to improve the analysis for such cases [42].

## VII. RELATED WORK

In this section, we discuss related work from three research areas: open source ecosystem studies, software supply chain security, and CodeQL and similar analysis frameworks. While there is a wide variety of work that focuses in the npm registry, to the best of our knowledge, we are the first to leverage CodeQL as a malware detection tool for it.

### A. Analysis

Studies of open source ecosystems raise awareness of exploitable vulnerabilities in package repositories, while providing insight into the taxonomy of malicious packages. Zimmermann et al. [3] perform a large-scale study where they discuss several mitigation strategies to deal with security threats present in the npm ecosystem. Ohm et al. [7] present a dataset of malicious packages used in real-world attacks on open source software supply chains, which were distributed in popular package repositories. Pinckney et al. [43] analyze how developers use semantic versioning in npm to study potential associated security vulnerabilities. Abdalkareem et al. [2] conducted a survey, followed by an empirical study of Node.js applications, to examine the reasons and drawbacks of using trivial packages. Zahan et al. [44] propose weak link signals for the npm registry, in an attempt to help developers make more educated decisions when including a package dependency into their applications.

### B. Security

Duan et al. [14] present a large scale vetting pipeline for malware detection in registries, which leverages program analysis techniques such as metadata, static, and dynamic analysis. Given that they do not claim to be the state of the art, but rather aim to reveal the severity of the security related problems from registries, they do not discuss the performance of the combined pipeline.

Sejfia and Schäfer [15] develop AMALFI, a machine-learning based approach for automatic malware detection in npm, where the feature set includes information about the capabilities used by a package and how these change between versions. The dataset used to train the models was the archive of malicious package versions provided by npm, extended with the corresponding benign versions of each package. After observing high initial false positive rates, the authors retrained with manually validated and re-labeled results. So, while training the classifier inherently does not require human intervention, the machine learning-based approach also requires the human in the loop for validation and re-labeling. In contrast, GENIE requires a higher upfront cost to develop queries, but compensates the effort with keeping false positives low. According to communication with the authors, AMALFI

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore.  Restrictions apply.

is currently deployed by npm, where the original dataset used for training covered the archive of malicious package versions provided by the registry maintainers. Unfortunately, it is not possible to compare GENIE to AMALFI, since the source code to apply AMALFI on a different dataset, or even just replicate their results, has not been released. The only anecdotal point of comparison is a single package, `bipiy74902-wx1`, which was flagged by GENIE in our experiments. According to the data released with AMALFI, this package was analyzed by their models during evaluation but not detected as malicious.

Ohm et al. [16] leverage a dataset of clustered malware used in software supply chain attacks to automatically generate syntactic signatures for successive detection of similar malicious packages in npm. Considering the high false positive rate obtained, the tool is geared towards aiding analysts by notifying them of suspicious packages that require further assessment while providing hints about possible similar attacks. Ohm et al. [45] evaluate several classifiers on 15k packages from npm but report high false positive rates, which matches our discussion above. Apart from overall malware detection in registries, there is also security relevant work with a narrower scope. In the literature, there are solutions for capturing potentially malicious updates [19, 20, 39, 21], detecting typosquatting attacks [17, 18], and hardening the ecosystem [22, 23, 24, 25].

### C. CodeQL

Our proposed approach consists in leveraging this semantic code analysis engine to find malicious packages in npm, whereas it is mostly used for vulnerability detection. Bandara et al. [46] define an automatic technique which allows to analyze the vulnerability management in GitHub repositories. Shcherbakov et al. [47] implement a framework to identify prototype pollution vulnerabilities which could lead to remote code execution attacks in Node.js applications. Chow et al. [48] combine static analysis, specifically taint analysis, and machine learning to detect potentially vulnerable data flows in JavaScript projects. Froh et al. [21] leverage differential static analysis to detect anomalous changes of behavior in package updates. Despite the approach showing promising results, it is designed to aid developers in vetting their package dependencies, instead of being an overall malware detector. Ideally, GENIE would be combined with such tools to harden the open source software supply chain.

There is also work that, despite not using CodeQL, presents solutions which could be adapted to our problem. Kang et al. [49] design a signature-based static analysis for detecting vulnerabilities that are semantically similar to previously seen vulnerable code. Feng et al. [50] present an approach for identifying malicious apps in Android, incorporating a high-level specification Datalog-based language which allows them to describe semantic characteristics of known malware families, and a static analysis for checking if an application matches a developed signature. Feng et al. [51] improve on the above-mentioned approach by inferring the semantic malware signatures automatically, besides using an approximation matching algorithm. Staicu et al. [42] propose a technique, based on dynamic analysis, for automatically extracting taint specifications of JavaScript libraries, to be later utilized in static analysis. CodeQL has been evaluated and compared against other static analysis tools, in terms of their effectiveness to detect vulnerabilities for JavaScript [52], C [53], and binaries [54].

## VIII. CONCLUSION AND FUTURE WORK

We have presented GENIE, an approach built around the CodeQL static code analysis framework to mitigate malware campaigns threatening the open source software supply chain. By monitoring the registry for recently discovered malicious packages, we designed effective CodeQL queries to match similar malware on the repository.

In our study, we developed 12 queries that allowed us to successfully scan the entire npm repository and flag a total of 125 previously undetected malicious packages from four different campaigns, while producing no false alarms. We also successfully used CodeQL to describe obfuscations, allowing us to flag obfuscated packages trying to evade static analysis. Overall, our evaluation shows that our method is effective and can improve the security of a real-world package repository.

A possible avenue for future work would be to automate the development of CodeQL queries, by leveraging machine learning techniques or by adapting existing approaches for mining malware specifications [55, 56, 57]. Another interesting direction would be to extend our framework to work with other package repositories, such as PyPi and RubyGems, where the same methodology could be applied to detect malware.

## REFERENCES

[1] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proc. of the 14th Int. Conf. Mining Software Repositories MSR*. IEEE Computer Society, 2017.

[2] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proc. 11th Joint Meeting on Foundations of Software Engineering ESEC/FSE*. ACM, 2017.

[3] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *Proc. 28th USENIX Security Symposium*. USENIX Association, 2019.

[4] A. Sparling, "Github issue 116: I don't know what to say." 2018, accessed: 2024-02-10. [Online]. Available: https://github.com/dominictarr/event-stream/issues/116

[5] A. Mihajlov, "Github issue 39: Virus in eslint-scope?" 2018, accessed: 2024-02-10. [Online]. Available: https://github.com/eslint/eslint-scope/issues/39

[6] H. Garrood, "Malicious code in the purescript npm installer," 2019, accessed: 2024-02-10. [Online]. Available: https://harry.garrood.me/blog/malicious-code-in-purescript-npm-installer/

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.

[7] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *17th Int. Conf. Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Springer, 2020.

[8] C. Silverio, "'crossenv' malware on the npm registry," 2017, accessed: 2024-02-10. [Online]. Available: https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry

[9] A. Birsan, "Dependency confusion: How I hacked into Apple, Microsoft and dozens of other companies," 2021, accessed: 2024-02-10. [Online]. Available: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

[10] M. Hanley, "Github's commitment to npm ecosystem security," 2021, accessed: 2024-02-10. [Online]. Available: https://github.blog/2021-11-15-githubs-commitment-to-npm-ecosystem-security/

[11] A. Polkovnychenko and S. Menashe, "Malware civil war - malicious npm packages targeting malware authors," 2022, accessed: 2024-02-10. [Online]. Available: https://jfrog.com/blog/malware-civil-war-malicious-npm-packages-targeting-malware-authors/

[12] K. Efimov, "Snyk finds 200+ malicious npm packages, including cobalt strike dependency confusion attacks," 2022, accessed: 2024-02-10. [Online]. Available: https://snyk.io/blog/snyk-200-malicious-npm-packages-cobalt-strike-dependency-confusion-attacks/

[13] A. Sharma, "86 malicious npm packages named after popular nodejs functions," 2022, accessed: 2024-02-10. [Online]. Available: https://blog.sonatype.com/86-malicious-npm-packages-named-after-popular-nodejs-function-names

[14] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *28th Annu. Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2021.

[15] A. Sejfia and M. Schäfer, "Practical automated detection of malicious npm packages," in *44th Int. Conf. Software Engineering (ICSE)*. ACM, 2022.

[16] M. Ohm, L. Kempf, F. Boes, and M. Meier, "If you've seen one, you've seen them all: Leveraging AST clustering using MCL to mimic expertise to detect software supply chain attacks," *CoRR*, vol. abs/2011.02235, 2020.

[17] M. Taylor, R. K. Vaidya, D. Davidson, L. D. Carli, and V. Rastogi, "Spellbound: Defending against package typosquatting," *CoRR*, vol. abs/2003.03471, 2020.

[18] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *IEEE European Symp. Security and Privacy Workshops (EuroS&P)*. IEEE, 2020.

[19] S. Scalco, R. Paramitha, D.-L. Vu, and F. Massacci, "On the feasibility of detecting injections in malicious npm packages," in *Proc. 17th Int. Conf. Availability, Reliability and Security*. ACM, 2022.

[20] K. A. Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Detecting suspicious package updates," in *Proc. 41st Int. Conf. Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE / ACM, 2019.

[21] F. Froh, M. Gobbi, and J. Kinder, "Differential static analysis for detecting malicious updates to open source packages," in *Proc. ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED)*. ACM, 2023.

[22] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node.js applications," in *23rd Int. Symp. Research in Attacks, Intrusions, and Defenses (RAID)*. USENIX Association, 2020.

[23] N. Vasilakis, A. Benetopoulos, S. Handa, A. Schoen, J. Shen, and M. C. Rinard, "Supply-chain vulnerability elimination via active learning and regeneration," in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*. ACM, 2021.

[24] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Containing malicious package updates in npm with a lightweight permission system," in *43rd Int. Conf. Software Engineering (ICSE)*. IEEE, 2021.

[25] E. Wyss, A. Wittman, D. Davidson, and L. D. Carli, "Wolf at the door: Preventing install-time attacks in npm with latch," in *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security*. ACM, 2022.

[26] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *USENIX Security Symp.* USENIX, 2003.

[27] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant, "Semantics-aware malware detection," in *IEEE Symp. Security and Privacy (S&P)*. IEEE Computer Society, 2005.

[28] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *2nd Int. Conf. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, ser. LNCS, vol. 3548. Springer, 2005.

[29] M. Dalla Preda, M. Christodorescu, S. Jha, and S. K. Debray, "A semantics-based approach to malware detection," in *34th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL)*. ACM, 2007.

[30] A. Holzer, J. Kinder, and H. Veith, "Using verification technology to specify and detect malware," in *Proc. 11th Int. Conf. Computer Aided Systems Theory (EUROCAST)*, ser. LNCS, vol. 4739. Springer, 2007, pp. 497–504.

[31] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Proactive detection of computer worms using model checking," *IEEE Trans. Dependable Sec. Comput.*, vol. 7, no. 4, 2010.

[32] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *23rd Annu. Computer Security Applications Conference (ACSAC)*. IEEE Com-

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.

puter Society, 2007.

[33] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *Proc. 28th USENIX Security Symposium*. USENIX Association, 2019, pp. 729–746.

[34] L. Cavallaro, J. Kinder, F. Pendlebury, and F. Pierazzi, "Are machine learning models for malware detection ready for prime time?" *IEEE Secur. Priv.*, vol. 21, no. 2, pp. 53–56, 2023. [Online]. Available: https://doi.org/10.1109/MSEC.2023.3236543

[35] O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble, "Keynote address: .ql for source code analysis," in *7th IEEE Int. Workshop on Source Code Analysis and Manipulation SCAM*. IEEE Computer Society, 2007.

[36] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "QL: object-oriented queries on relational data," in *30th European Conf. Object-Oriented Programming ECOOP*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[37] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Security and Privacy (S&P)*. IEEE Computer Society, 2014, pp. 590–604.

[38] X. Zhou, Y. Zhang, W. Niu, J. Liu, H. Wang, and Q. Li, "OSS malicious package analysis in the wild," *CoRR*, vol. abs/2404.04991, 2024.

[39] D. Gonzalez, T. Zimmermann, P. Godefroid, and M. Schäfer, "Anomalicious: Automated detection of anomalous and potentially malicious commits on github," in *43rd Int. Conf. Software Engineering ICSE*. IEEE, 2021.

[40] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys*, vol. 49, no. 1, 2016.

[41] J. Wagner, "Trustworthy chrome extensions, by default," 2018, accessed: 2024-02-10. [Online]. Available: https://blog.chromium.org/2018/10/trustworthy-chrome-extensions-by-default.html

[42] C. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting taint specifications for JavaScript libraries," in *ICSE '20: 42nd Int. Conf. Software Engineering*. ACM, 2020.

[43] D. Pinckney, F. Cassano, A. Guha, and J. Bell, "A large scale analysis of semantic versioning in NPM," in *20th IEEE/ACM Int. Conf. Mining Software Repositories MSR*. IEEE, 2023.

[44] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. S. Maddila, and L. A. Williams, "What are weak links in the npm supply chain?" in *44th Int. Conf. Software Engineering (ICSE)*. IEEE, 2022.

[45] M. Ohm, F. Boes, C. Bungartz, and M. Meier, "On the feasibility of supervised machine learning for the detection of malicious software packages," in *17th Int. Conf. Availability, Reliability and Security ARES*. ACM, 2022.

[46] V. Bandara, T. Rathnayake, N. Weerasekara, C. Elvitigala, K. Thilakarathna, P. Wijesekera, and C. Keppitiyagama, "Fix that fix commit: A real-world remediation analysis of JavaScript projects," in *20th IEEE Int. Working Conf. Source Code Analysis and Manipulation SCAM*. IEEE, 2020.

[47] M. Shcherbakov, M. Balliu, and C. Staicu, "Silent spring: Prototype pollution leads to remote code execution in node.js," in *USENIX Security Symp.*, 2023.

[48] Y. W. Chow, M. Schäfer, and M. Pradel, "Beware of the unexpected: Bimodal taint analysis," in *Proc. of the 32nd ACM SIGSOFT Int. Symp. Software Testing and Analysis ISSTA*. ACM, 2023.

[49] W. Kang, B. Son, and K. Heo, "TRACER: signature-based static analysis for detecting recurring vulnerabilities," in *Proc. of the 2022 ACM SIGSAC Conf. Computer and Communications Security CCS*. ACM, 2022.

[50] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: semantics-based detection of android malware through static analysis," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE)*. ACM, 2014.

[51] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, "Automated synthesis of semantic malware signatures using maximum satisfiability," in *24th Annu. Network and Distributed System Security Symp. NDSS*. The Internet Society, 2017.

[52] T. Brito, M. Ferreira, M. Monteiro, P. Lopes, M. Barros, J. F. Santos, and N. Santos, "Study of JavaScript static analysis tools for vulnerability detection in node.js packages," *CoRR*, vol. abs/2301.05097, 2023.

[53] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static C code analyzers for vulnerability detection," in *31st ACM SIGSOFT Int. Symp. Software Testing and Analysis ISSTA*. ACM, 2022.

[54] A. Mantovani, L. Compagna, Y. Shoshitaishvili, and D. Balzarotti, "The convergence of source code and binary vulnerability discovery - A case study," in *Proc. '22 ACM Asia Conf. on Computer and Communications Security*. ACM, 2022.

[55] H. D. Macedo and T. Touili, "Mining malware specifications through static reachability analysis," in *Proc. 18th European Symp. Research in Computer Security (ESORICS)*. Springer, 2013.

[56] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proc. 1st Annu. India Software Engineering Conference (ISEC)*. ACM, 2008.

[57] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *31st IEEE Symp. Security and Privacy (S&P)*. IEEE Computer Society, 2010.

Authorized licensed use limited to: Queensland University of Technology. Downloaded on January 06,2025 at 02:50:01 UTC from IEEE Xplore. Restrictions apply.