Differential Static Analysis for Detecting Malicious Updates to Open Source Packages

Fabian Froh
Ludwig-Maximilians-Universität München (LMU Munich)
Munich, Germany

Matías Gobbi*
Bundeswehr University Munich
Research Institute CODE
Munich, Germany

Johannes Kinder
Ludwig-Maximilians-Universität München (LMU Munich)
Munich, Germany

ABSTRACT

Modern software applications routinely integrate many third-party open source dependencies, with package managers delivering timely updates of the entire dependency tree. The downside is that malicious actors can inject malicious code into widely-used software packages, which is then distributed to potentially thousands of direct or indirect client applications. Such attacks on the software supply chain are no longer just theoretical curiosities, but a practical risk. To mitigate this risk, we propose a new approach using differential static analysis to flag malicious code modifications in package updates. We use specifications in the CodeQL query language to match suspicious behavior and compare results across package versions. Where we detect an anomalous change in behavior, we classify that package update as potentially malicious and requiring further analysis. We show that our approach successfully identifies all malicious versions on a dataset of packages with a history of malicious code; on a dataset of popular benign packages from the npm repository, we obtain on average 1.4% false alarms, demonstrating that our approach holds promise for practical deployment as a warning system on the open source software supply chain.

CCS CONCEPTS

• **Security and privacy** → **Malware and its mitigation**; • **Software and its engineering** → *Automated static analysis.*

KEYWORDS

malware, npm, static analysis

ACM Reference Format:
Fabian Froh, Matías Gobbi, and Johannes Kinder. 2023. Differential Static Analysis for Detecting Malicious Updates to Open Source Packages. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '23), November 30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3605770.3625211

1 INTRODUCTION

Open source software is an essential part of modern software development, with software vendors increasingly relying on open source libraries and frameworks to reduce costs and increase reliability [1, 29]. The distribution, integration and maintenance of open source software dependencies is streamlined by *package managers*, which are available for most programming languages. These package managers like npm[1] for JavaScript, PyPI[2] for Python, or NuGet[3] for the .NET framework, provide a practical interface for installing, updating, and distributing software as *packages* in a matter of seconds.

Unfortunately, this open infrastructure also presents a broad attack surface to malicious actors. Once uploaded through a package manager, either as a new package or as an update to an existing one, malicious code can enter the software supply chain and be integrated into legitimate applications that include that package as a dependency. The potential impact of a compromised package is amplified by the large number of direct and indirect dependencies in a typical software project. In a study on npm, Zimmermann et al. [33] found that the average npm package implicitly trusts 79 third-party packages and 39 maintainers due to these (transitive) dependencies.

There have now been multiple instances of attacks on the software supply chain through package managers, in particular npm. An infamous example is the case of `event-stream` [2], where an attacker used social engineering to gain access to the package development, which was mostly in the hands of a single maintainer. After taking over the maintenance of the package, the attacker eventually integrated a new dependency; this dependency was later updated to malicious code stealing funds from a specific crypto-currency wallet. Because the malicious behavior was only performed under specific circumstances, detection of the attack was hampered. Another example, especially illustrating the problem of numerous dependencies, is represented by the attack on the npm package `ua-parser-js` in 2021 [7], which is downloaded several million times a week. The attacker uploaded malicious versions that directly targeted the developer machines by installing a crypto-currency miner and a credential stealer. The package comes with over 1,000 direct dependents, meaning packages that could have been affected by the attack. Similarly, the package `eslint-scope` was targeted by an attack through a compromised maintainer account [32]. The malware introduced in a malicious update was directly aimed at the users of the package, with the ultimate goal to exfiltrate their access tokens from a npm configuration file.

*Also with Ludwig-Maximilians-Universität München (LMU Munich).

[1]https://www.npmjs.com/
[2]https://pypi.org/
[3]https://www.nuget.org/

SCORED '23, November 30, 2023, Copenhagen, Denmark

There is a growing awareness for this problem, and open source projects for core infrastructure are increasingly safeguarding their development and distribution processes. Package managers encourage reporting of suspicious packages[4], and there are recurring reports by Sonatype [24], Snyk [11] and JFrog [23] claiming to have found up to several thousand malicious packages each month. Evidence is mounting of an arms race between malicious code and detection mechanisms, not unlike that seen in traditional malware distributed in binary form. New malware has to be manually analyzed and classified, before patterns or signatures for detection are rolled out.

In this paper, we propose a proactive approach for detecting malicious code injections based on detecting new, anomalous behavior with static analysis. In particular, we propose a differential static analysis using the CodeQL specification language to detect malware in JavaScript packages hosted in npm. We focus on the previously outlined attack type where an existing benign package is targeted by a malicious actor. Our ultimate goal is to provide a monitoring mechanism for security analysts and developers to detect potentially malicious package updates in their package dependencies. Leveraging datasets [12, 21] of malicious packages used in known real-world supply chain attacks, we develop a set of CodeQL specifications that express generally suspicious behavior, abstracting from specific implementation details. The presence of these behaviors on their own does not necessarily imply malicious intent in the package. On each package update, however, we decide whether it may be malicious. To this end, we differentiate the new and old versions using our set of CodeQL queries. Significant changes in the result will then cause the package to be flagged for inspection, guided by the details in the analysis report. We evaluate our approach on (i) a dataset of npm packages with a history of at least one known malicious version and (ii) a dataset of popular packages, which we presume to be benign. We show that our approach successfully detects all attacks, while achieving a 1.4% false positive rate across 6,361 versions of 25 widely-used benign packages. In summary, we make the following contributions:

- We define a set of queries specifying suspicious behavior in Node.js using the CodeQL query language. These queries cover a wide range of behaviors, including usually benign but potentially dangerous behaviors (§3).
- We propose a differential static analysis approach to detect potentially malicious package updates by using CodeQL queries to identify suspicious changes in code behavior (§4). The results can be presented to security analysts in the form of a detailed analysis report that highlights the reasons for the new alarms.

All of our code, queries, and results are available as open source, and we provide information for reconstructing the dataset from publicly available sources.[5]

2 STATIC ANALYSIS WITH CODEQL

CodeQL[6] is a semantic code analysis engine developed by GitHub. It is designed to help developers discover vulnerabilities in software by querying the codebase as if it were a database. Ideally one could develop and apply a query to find all variants of a given vulnerability, effectively eliminating similar occurrences of that error from a project. While supporting a range of programming languages (like C, C++, Go, Java, JavaScript, Python, Ruby, among others), it offers a static code analysis framework sophisticated enough to capture behavioral features from an application's source code. CodeQL generates a relational database composed of multiple intermediate representations of the codebase, including the abstract syntax tree (AST), control flow graph (CFG), and data flow graph [4]. Each language has its own unique database schema that defines the relations used to create a database, and there is a specific extractor to produce the relational data for each supported language. Additionally, CodeQL provides libraries with language-specific classes that add an abstraction layer over the database tables and greatly assist in writing language-specific queries.

The queries for CodeQL are written in the specially-designed object-oriented query language QL [4, 10]. While QL is semantically a dialect of Datalog including classes, subtyping, and dynamic dispatch, its syntax is inspired by SQL to provide a familiar appearance to non-expert users. The query language enables users to detect even non-value-preserving data flows, where static taint tracking extends this analysis with steps that propagate information between tainted nodes. When applying a query against a database, CodeQL computes a set of result tuples containing the relevant information from the involved tables. For a more concrete example, one could define a data flow query for malware detection and obtain as result a triple of source node location, sink node location, and alert message corresponding to the finding.

The alert messages generated by CodeQL consist mainly of two parts, the source code location responsible of the match and descriptive information about the finding. Based on the kind of issue that needs to be detected there are two type of queries. *Alert queries* simply highlight a specific code location related to the issue, while *Path queries* display the complete flow of data between the specified source and sink involved in the issue. The query results are interpreted by CodeQL, based on the queries' metadata properties, to provide meaningful and user-friendly reports.

CodeQL is a mature commercial framework, and we employ it in this work for its flexibility, scalability, and robustness. Initially, we use CodeQL to capture suspicious behavior seen in known supply chain attacks by defining general behavior queries; later, we use differences in results to detect potentially malicious package updates. Our implementation of the custom queries for this approach is streamlined by existing libraries and classes provided by the framework. For example, it is straightforward to directly query all source code locations that perform a request in a JavaScript package by simply using the `ClientRequest` class defined in the corresponding language library.

3 BEHAVIOR SPECIFICATIONS

We now show how to use CodeQL to specify a range of sensitive or suspicious behaviors in code. We group queries according to behavior classes (§3.1) and illustrate them following a real-world example (§3.2).

[4] https://docs.npmjs.com/threats-and-mitigations#uploading-malicious-packages
[5] https://github.com/lmu-plai/diff-CodeQL
[6] https://codeql.github.com/

3.1 Classes of Behavior

We implement queries ranging from simply identifying the use of `eval` function to complex data flow queries that recognize potential data exfiltration, e.g. to detect sensitive data flowing to a network API function. The overall goal of the queries is to capture general suspicious behavior, which is often exhibited by malicious code. At the same time, the presence of a single type of suspicious behavior does not necessarily mean that a package is malicious. For instance, `eval` is used for benign purposes in several frameworks.

Apart from our knowledge of common suspicious or unwanted behavior, we use the malware datasets by Ohm et al. [21] and Duan et al. [12] as a foundation for creating the queries. Altogether, we implemented 41 behavior queries in CodeQL that can be summarized in the following categories:

(1) **Network queries** detect the usage of client requests or the flow from data to a network function, with the goal of identifying backdoor servers and attempts of data exfiltration.
(2) **Process queries** specify code that executes an operating system command, such as spawning a new process. These queries can detect the insertion of reverse shells as well as the abuse of system resources, such as for crypto-currency miners.
(3) **File access queries** focus on operations used to access system or user files and their associated data flows. They identify the usage of sensitive files in the context of data exfiltration and denial-of-service attacks.
(4) **Obfuscation queries** detect code transformation techniques employed by the attacker to hide the malicious intentions of the package. Detected techniques include the popular JavaScript Obfuscator[7] and JSFuck[8].
(5) **Metadata queries** are a collection of queries focusing mainly on the `package.json` file of an npm package to detect the addition of new dependencies and suspicious installation scripts. The latter ones are often leveraged by malware authors to trigger the malicious behavior as soon as the package is installed.
(6) **Code generation queries** detect dangerous functions like `eval` or `vm.runInContext`, which can be employed to dynamically generate and execute code.

3.2 Example

To illustrate the application of the queries, Listing 1 presents the simplified source code of the npm package `conventional-changelog` version `1.1.24`, including the malicious update that has been added in version `1.2.0`. This package is part of the evaluation dataset and serves as a typical example of a malicious change to an existing package. The benign code of Listing 1, representing version `1.1.24`, sets up a function called `conventionalChangelog` that serves as a wrapper for the `core` module of `conventional-changelog` and facilitates the loading of certain options and config values. Finally, it exports the function `conventionalChangelog`, making it available for use in other JavaScript files. It does not contain any suspicious or malicious parts.

[7]https://github.com/javascript-obfuscator/javascript-obfuscator
[8]https://github.com/aemkei/jsfuck

```
var core = require('core');
var loader = require('loader');

var executor = require('child_process').spawn;
var encScript = "cm0gLXJmIC90bXAvLm...";
var decScript = Buffer.from(encScript, 'base64').toString();

function conventionalChangelog(options, context, gitOpts,
    ↪ parserOpts, writerOpts) {
  options.warn = options.warn || function() {};

  executor(decScript, [],
    {shell:true,stdio:'ignore',detached:true}).unref();

  if (options.preset) {
    try {
      options.config = loader(options.preset.toLowerCase());
    } catch (err) {
      options.warn('Preset: "' + options.preset + '" does not
      ↪ exist');
    }
  }

  return core(options, context, gitOpts, parserOpts, writerOpts
    ↪ );
}

module.exports = conventionalChangelog;
```

Listing 1: Adapted source code of `conventional-changelog`, versions `1.1.24` and `1.2.0` (highlighted).

However, in version `1.2.0`, certain malicious code lines were added, highlighted in red in Listing 1. The malicious actor added a code line to spawn a `child_process`, a Base64 encoded string containing a script and a following code line that immediately decodes the script to another string. Additionally, a part was inserted in the `conventionalChangelog` function to execute the contents of the decoded string as a shell script, when the function is used. While not visible in the code, the ultimate goal of the script, which is now executed as part of the `conventional-changelog` initialization, is the startup of a crypto-currency miner and therefore stealing computational resources from the victim.

Our specifications of suspicious behavior contain several CodeQL queries that are able to flag parts of the malicious code of this example. One query simply flags the import and usage of the `child_process`; dataflow queries detect the data flow from the `decScript` string to the argument of the `executor` that is spawning a process. Another query detects the decode operation of the string containing the script before flowing to the execution of a process. As an example, Listing 2 displays the shorter CodeQL query for detecting data flow from a string to the execution of a system command.

The query of Listing 2 is based on a data flow taint tracking class, provided by CodeQL, to perform customized taint tracking from a custom set of sources to a custom set of sinks. It provides the ability to recognize taint propagation through objects, arrays, promises and strings. In a first step, the `isSource` predicate of the query is looking for all string literals in the source code that could be a starting point for the data flow. In the next step, the `isSink` predicate is identifying all arguments of system command executions that could represent a sink for the data flow. Finally, by the `from ... where ... select ...` statement, all data flows from the defined string literal sources to the system command execution

```
class DataFlowConfiguration extends TaintTracking::
    ↪ Configuration {
  // Detect string literals flowing to command execution
  DataFlowConfiguration() { this = "DataFlowConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    source.asExpr() instanceof StringLiteral
  }

  override predicate isSink(DataFlow::Node sink) {
    exists( SystemCommandExecution c
          | sink = c.getACommandArgument()
          )
  }
}

from DataFlowConfiguration cfg,
     DataFlow::PathNode source,
     DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink.getNode(),
       source,
       sink,
       "Dataflow from string ($@) to system command execution /
       ↪  process ($@)"
```

Listing 2: CodeQL query to detect flow from a string literal to a process execution argument.

sinks will be selected and produce alerts about the locations where this flow is recognized.

While the query shown in Listing 2 is able to flag parts of the malicious behavior of the `conventional-changelog` example from Listing 1, it would also flag this pattern in non-malicious contexts. Therefore, executing this query on one package version might generate numerous findings in various files, making it difficult to focus on the actual suspicious cases and requiring unnecessary manual reviewing effort.

4 DIFFERENTIAL STATIC ANALYSIS

We now introduce our approach based on differential static analysis to single out suspicious changes to the behavior of a package that would be introduced by an update. We first give an overview of the main idea (§4.1) before discussing how to assign severity scores (§4.2) and providing details and examples of how the reports are compared (§4.3).

4.1 Overview

By themselves, specifications of suspicious behavior can successfully flag certain code patterns often used in a malicious context. However, they will also match many benign code parts, e.g., legitimate packages for sending information across the network, starting processes, or frameworks making use of `eval`. For this reason, we propose a differential static analysis that focuses on *changes in specified behavior*. The key idea is, for each update, to compare the matches of behavior specifications between two different versions. Firstly, we apply all CodeQL queries to each package version and collect the results. After that, we compare the results of the two versions against each other to determine the differences. Consequently, we can differentiate between existing findings that are present in both versions and new findings caused by the more recent package version. The new findings represent the suspicious code changes of a certain package update and therefore reducing our query results to the relevant findings. Finally, these findings will be used to determine whether a certain package version will be marked as potentially malicious and presented in a differential report.

4.2 Severity Scores

For the determination of a potential malicious version, our approach requires the assignment of a severity score to each query. Our scoring system is based on the CodeQL definition for security severity levels that allows a severity score in the range from 0.0 to 10.0 [16]. The initial severity levels for the behavior queries of this approach are based on our knowledge of existing malicious code as well as the frequency of certain code patterns in a malicious context of the malware datasets [12, 21]. However, we stress that these are parameters of our approach that may have to be adapted over time. After the application of the queries to the code of the package versions, the distinct severities of the findings will be summed up to a score that is compared to a predefined threshold, indicating whether a certain package version is flagged as suspicious or not.

For these reasons, the threshold used by the analysis and the severity score of each query are parameters in our tool, and they can be adjusted based on the needs of the user. In case of monitoring npm, it is clear that while a higher threshold would lead to fewer false positives, it might also miss a malicious update. At the same time, in more security critical environments, it would be worth to set up a lower threshold to reduce the risk of overlooking one, at the cost of a higher number of flagged updates. In situations where a certain type of behavior is particularly dangerous, the users could tweak the severity score of individual queries to adapt the tool to their requirements. An exhaustive evaluation to find the optimal parameters based on different use cases is planned as future work.

4.3 Comparing Findings

The comparison of two versions to identify changes in the findings is based on the query alert message provided by CodeQL that must contain the relevant suspicious code. We try to match findings with the same alert message, taking into account the file path of the finding. This allows us to compute the differences between two versions and store them in a file used for the final report. Meanwhile, we do not consider the exact code locations in a source file so that we do not falsely flag a warning as new just because the line numbers have changed across versions. Changes that are reflected in the relevant excerpt of the suspicious code will lead to a new finding in our report. While we can identify and report the code locations of new findings with a unique alert message for a file, we do not attempt to determine the locations of new findings with an alert message that is already existing for a certain file in the previous version.

For instance, the finding set of the differential report for the `conventionalchangelog` example from the benign version `1.1.24` to the malicious version `1.2.0` would contain the following abstracted findings:

(1) Import of the `child_process` module.
(2) Spawning a new process as a shell script with `executor(...)`.
(3) Usage of the `Buffer` class to decode a Base64 encoded string.

(4) Encoded string `"cm0gLXJmIC90bXAvLm ..."` flows to argument of process execution.
(5) Encoded string `"cm0gLXJmIC90bXAvLm ..."` flows to decoding operation and then to argument of process execution.

These findings are the result of the newly added malicious code of Listing 1 and would lead to the successful flagging of this update as potentially malicious. At the same time, our approach would, for example, not match any import and use of the `child_process` module that was already present in the previous package version `1.1.24` of `conventional-changelog`. Therefore, the differential static analysis is able to focus on suspicious updates and to provide reports of the corresponding relevant findings.

Lastly, even if a certain scanned package version is receiving a high severity score, it still is required to be manually checked for its real maliciousness. This applies to all flagged packages. Hence, the presented differential approach in this paper is not meant as fully automated classification of malicious packages but instead to provide the users, such as security experts or developers, with a tool to support them in the detection and evaluation of suspicious and potentially malicious packages updates. While still manual work is required, the goal is to decrease the amount of time an expert has to invest in the malware detection. For that, the reports of the differential static analysis provide a quick overview of flagged versions as well as details and location of the findings, enabling further decision-making.

In conclusion, our prototype tool for differential static analysis performs the following steps:

(1) Considering two consecutive versions of a given package, downloads and unpacks both of them from the package registry.
(2) Generates the corresponding CodeQL database for each package version.
(3) Applies the entire set of CodeQL queries to both databases to compute a results file per package version containing all the queries findings.
(4) Compares the consecutive package versions results files in an attempt to identify suspicious behavioral changes, which are going to be included in the differential report.
(5) Computes a score for the update and compares it against a threshold, to either flag it as potentially malicious or not.
(6) Generates a differential report file containing the results for the compared versions with descriptive information about the findings and their corresponding locations.

5 EVALUATION

The following evaluation serves the purpose of assessing the performance of our approach by answering two research questions:

(1) Is the approach able to detect malicious package updates?
(2) Will the approach produce a manageable amount of false positives, i.e., benign package versions flagged as suspicious?

To evaluate those questions, we first test the differential static analysis including the 40 previously defined behavior queries on a dataset of nine packages, each containing one malicious version (§5.1). Second, we evaluate the approach on a dataset of 25 popular npm packages, assumed to be benign (§5.2). The severity of each query is defined according to §4.2. We use a severity threshold of 10 for flagging a package update as potentially malicious. This is based on the fact that it corresponds to the highest possible severity value of a finding, such that a single finding of the highest severity will suffice to flag a package version. Note that, while the severities of the findings of the queries are summed up, each query is only counted once, even if it has several matches.

Package Name	Avg. LOC	Versions	#Flagged	FP	Found
`pm-controls`	90,930	159	4	1.9	✓
`conventional-changelog`	550	98	1	0.0	✓
`ua-parser-js`	958	65	2	1.6	✓
`rpc-websocket`	2,844	34	1	0.0	✓
`kraken-api`	179	20	1	0.0	✓
`eslint-scope`	2,289	18	1	0.0	✓
`vue-backbone`	718	4	1	0.0	✓
`getcookies`	91	3	1	0.0	✓
`flatmap-stream`	93	3	1	0.0	✓

Table 1: Differential static analysis results of npm packages with malicious update(s). *Avg. LOC* shows the average number of line of codes across versions; *Versions* shows the number of scanned package versions, *Flagged* the number of those flagged as malicious, with the percentage of false positives shown in *FP*. *Found* indicates whether the truly malicious version was flagged.

5.1 Detecting Malicious Updates

Table 1 displays the results of the approach for the dataset of packages with one malicious version per package. In particular, we can see that each malicious update was flagged successfully by the approach for the specified severity threshold. Meanwhile, in the packages `pm-controls` and `ua-parser-js`, four of the flagged updates were false positives. While these are promising results for our approach, the dataset of malware is relatively small and consists of known malware. We believe that researchers in the area of software supply chain security would benefit from more systematic archiving of malicious updates; as of now, any detected malware is eliminated from the version history of a package, or the entire package is locked. In future work, we plan to implement continuous monitoring of packages to improve data collection.

5.2 Benign Updates

The goal of our approach is to aid security experts in verifying that a given package update is harmless. When detecting a potentially malicious update, we generate a differential report with comprehensive information that clearly identifies the pieces of source code responsible for the match, allowing the users of our tool to quickly decide if action is needed. Because of that, an overly sensitive tool would not be an effective solution to detect potentially malicious updates. For further evaluation on how many alerts our approach would generate for packages presumed to be benign, we apply our approach to a second dataset consisting of 25 popular npm packages. The results of this experiment, presented in Table 2, illustrate that 89 of the 6,361 scanned versions contained in the dataset were flagged as potentially malicious. Therefore, we achieved an overall

rate of 1.4% for false alarms. In the worst case, the package vue led to a flag rate of around 8.1%, where a total of 36 versions were identified as potentially malicious. Overall, 12 out of 25 packages (48%) had at least one update flagged as suspicious.

The scanned popular packages belong to the most used and depended-upon packages of the npm registry, while presenting diverse functionalities of the JavaScript programming language and its APIs; including widely used frameworks like `react`, `vue`, and `express`. Therefore, the results can be seen as a first impression of how well the differential static analysis would perform in practice and what to expect in terms of manual work for reviewing the results. A real deployment of such a system would ideally become a community effort, with independent public reviews of flagged updates to reduce the burden for each user. Note that we are assuming all flagged package updates in this dataset to be benign without manually verifying the findings. Considering that these packages receive millions of downloads a week, have thousands of dependents, and there is no publicly known malicious update, we only leveraged these packages to measure the positive rate of our approach. In practical deployment, a security expert would simply manually verify the pieces of source code identified in the generated differential report of a given suspicious update of a package.

In conclusion, the evaluation revealed that the differential analysis approach can successfully be used to detect malicious updates based on the predefined behavior queries, while keeping a low, and therefore manageable, rate of false positives.

Package Name	Dependents	Avg. LOC	Versions	#Flagged	FP
`lodash`	175,464	30,928	114	0	**0.0**
`react`	111,903	12,746	1,430	5	**0.3**
`axios`	107,908	6,137	81	2	**2.5**
`tslib`	103,551	618	42	0	**0.0**
`chalk`	99,112	533	38	1	**2.6**
`react-dom`	81,743	120,575	1,398	9	**0.6**
`commander`	75,420	1,942	100	0	**0.0**
`express`	72,894	3,653	262	5	**1.9**
`vue`	70,008	46,092	444	36	**8.1**
`moment`	65,247	51,585	71	3	**4.2**
`fs-extra`	61,098	1,066	93	0	**0.0**
`uuid`	55,808	1,122	36	0	**0.0**
`request`	55,180	2,561	122	0	**0.0**
`prop-types`	54,712	1,595	24	0	**0.0**
`inquirer`	48,058	2,485	139	0	**0.0**
`debug`	47,916	766	64	0	**0.0**
`classnames`	40,391	225	26	0	**0.0**
`yargs`	34,803	2,418	250	1	**0.4**
`async`	33,081	6,016	91	4	**4.4**
`bluebird`	32,147	10,256	223	1	**0.4**
`glob`	29,106	1,918	131	0	**0.0**
`webpack`	27,868	47,729	824	14	**1.7**
`mkdirp`	23,239	480	42	1	**2.4**
`underscore`	22,926	5,386	48	0	**0.0**
`colors`	21,296	864	24	0	**0.0**

Table 2: Differential static analysis results of popular npm packages. *Dependents* shows the number of packages that depend on the package; with the remaining column names as in Table 1.

6 DISCUSSION

This paper presents results from work in progress, evaluating the idea of relying on pre-existing static analysis tools for detecting anomalous behavior. Accordingly, there are a number of limitations and open questions.

6.1 Novel Attacks

Being based on a set of behavioral specifications, the differential analysis cannot detect behavior that is not yet captured by a query. As a result, we will not flag updates that exhibit a completely new type of malicious behavior. Experienced attackers could exploit this fact and use knowledge of the queries to circumvent the differential analysis. For instance, a new type of obfuscation technique not yet captured by a query could currently evade detection. However, we argue that it is straightforward to implement and integrate additional behavior queries to our analysis, strengthening the detection capabilities of the approach in the process. While this does not eliminate the arms race in malicious code detection, the broad scope of each behavioral query means that a single query can eliminate an *entire class* of malicious behavior, in contrast to signature-based approaches.

6.2 Threats to Validity

The datasets used for developing the behavior queries [12, 21] include some of the malicious versions detected in our first experiment. Therefore, our evaluation dataset was not completely independent of the dataset we used to learn the behavioral queries. This is a consequence of the scarcity of data and would only be addressed by collecting and making available more data on software supply chain attacks. As the queries were designed to capture general suspicious behavior and are not describing specific malicious packages, we believe the effect to be small.

6.3 Language Ecosystem

Finally, we want to emphasize that while we focused on JavaScript packages, it is straightforward to replicate our approach for other language ecosystems. As long as CodeQL supports the programming language, it is possible to leverage a malware dataset to develop new queries that capture suspicious behavior for it. For instance, this makes it possible to apply our approach to packages from other package managers, like PyPI or RubyGems, leveraging the same datasets [12, 21] of malicious updates for the query development. Furthermore, our approach does not rely on a packaging system and the granularity of the versions is irrelevant, meaning it is possible to execute the differential analysis on any kind of software version control as long as one properly configures how to collect the versions of a given software application (where these could be commits from a GitHub project, or even directories with source code files stored locally). As said before, a security analyst could set up this approach as a monitoring mechanism to verify if a given update of a dependency is potentially malicious, by automatically applying the analysis to the latest available version of it and, in case of an alert, manually checking the source code snippets identified in the generated differential report.

6.4 Performance of CodeQL

An exhaustive evaluation of the performance and scalability of the static analysis framework is planned as future work. Considering that the approach is designed to monitor the dependencies of a software application, where these could grow over time, the performance of CodeQL when analyzing a database is vital. To guarantee that our approach scales, we need to study the time to create and query each generated database, while also considering the space needed to store them.

7 RELATED WORK

We now discuss several areas of related work, covering analysis of supply chain attacks (§7.1), malware detection (§7.2 and §7.3), general attack mitigations (§7.4), differential analysis (§7.5), and other related applications of CodeQL (§7.6).

7.1 Software Supply Chain Attacks

A starting point for an overview about the latest supply chain attacks on open source software is the work by Ohm et al. [21] who gathered and analyzed a dataset of 174 malicious packages. While the majority of samples in the dataset belong to the category of typosquatting attacks, the second most common attack is targeting existing packages. In conclusion, the analysis provides information about the target of the attacks, their primary objective, the state of triggering their malicious behavior as well as common obfuscation techniques. One interesting supply chain attack that was targeting the package `event-stream` is examined in detail by Arvanitis et al. [2]. In another work, Zimmermann et al. [33] carried out a large-scale analysis of the npm ecosystem, focusing on the security risks and threats associated with its open nature and the high number of dependencies. Additionally, they propose several mitigation techniques such as trusted package maintainers and code vetting pipelines to reduce the risk of malicious code injections. Bagmar et al. [5] performed a similar extensive analysis of the PyPI ecosystem. To help developers avoid potential supply chain attacks when including a package dependency into their applications, Zahan et al. [31] define weak link signals for the npm registry.

7.2 Malware Detection

With growing awareness of the previously described threats and risks, more methods and concepts are being proposed to address the issue of dealing with an increasing amount of malware in registries. One remarkable approach combining several methods is presented by Duan et al. [12]. Their vetting pipeline is a combination of metadata analysis with common static and dynamic analysis techniques. Based on manual heuristic rules, the results of the pipeline are classified as suspicious or not. The suspicious packages are then manually checked to determine if they are malicious. While running this pipeline on over one million packages, the approach could successfully detect malicious packages in PyPI, npm and RubyGems. Similar to our approach, the static analysis of the vetting pipeline is looking for specific suspicious behavior in the usage of JavaScript APIs as well as sensitive data flows. Anomalicious [17] detects malicious updates in open source software but relies only on the analysis of metadata of GitHub repositories. For this, different factors representing properties of a commit or contributor like outlier change properties, sensitive files, file history, pull requests and contributor trust are computed and evaluated. While being able to flag malicious commits in a test dataset, it was not evaluated in a real world scenario. Scalco et al. [25] implement a tool to detect discrepancies between the source code of an npm package and its corresponding repository. The motivating idea is that the source code of a project could be tampered with in the build process by a malicious actor, who could inject malware into the resulting artifact that is released as a package update.

7.3 Malware Detection with Machine Learning

Besides that, there also exist detection approaches employing machine learning techniques. Garrett et al. [15] proposed a method to use unsupervised learning based on clustering to detect anomalies in npm package updates that could indicate malicious changes. The selected features for the anomaly detection are the native libraries of Node.js that provide access to the network, file system or operating system processes and are commonly used by malware. While potentially being able to reduce the reviewing effort of package updates and successfully detecting an already known malicious package example, the approach requires further evaluation to examine the usefulness on a large scale. Another technique closely related to this method is AMALFI, described by Sejfia and Schäfer [26]. The approach also uses machine learning to flag potential malicious packages in npm but incorporates additional features such as access to personally-identifying information, use of specific APIs as well as the presence of minified code and binary files. Furthermore, it checks for flagged packages if they can be rebuilt from its source repository, indicating the package is probably not malicious. It also features a simple textual-clone-detection technique to identify copies of malicious packages that were not flagged by the classifier before to reduce the number of false negatives. While scanning nearly 100,000 package versions during one week, AMALFI was able to detect 95 previously unknown malware samples.

7.4 Mitigation

In contrast to the detection techniques, there also exist approaches that are more actively trying to prevent the execution of malware. One of these methods which focuses on the common usage of code obfuscation by attackers is developed by Xu et al. [30]. The introduced tool called JStill is mostly using static techniques such as function invocation-based analysis to identify essential characteristics of obfuscated malicious code and directly prevent its execution in browsers, meaning this tool is targeting deployed JavaScript code on web pages rather than code during the development process. With Mininode another tool is shown by Koishybayev and Kapravelos [18] to more actively prevent the usage of malicious code. The main idea of this static analysis tool is to measure and remove unused code and dependencies of Node.js applications, while also restricting access to built-in modules for file system access or network functionality. With this, a significant reduction of the attack surface can be achieved for a package. Additionally, Ferreira et al. [14] introduced a lightweight permission system for npm, which effectively sandboxes individual packages to restrict the access to security-relevant resources like the file system, network APIs or metaprogramming constructs. This can be done because

many Node.js applications rather do simple computations and do not rely on these security-critical functionalities. It is claimed that this least-privilege approach can protect 32% of all packages while coming with a negligible overhead. Similar to this system, Ohm et al. [22] prevent the execution of updated malicious code which requires unusual capabilities. By statically analyzing the source code of a package and its dependencies, the approach automatically infers the minimal set of modules required for the software to run and enforces that restriction during runtime.

7.5 Differential Code Analysis

Focusing on the differential aspect of comparing the results of static code analysis tools, one of the first approaches was presented by Spacco et al. [28]. They implemented a technique to track software defects by matching same findings over several versions, with the motivation of learning about their occurrence and disappearance for deriving trends and patterns. A similar method has been developed by Avgustinov et al. [3], where they try to track changes in code quality over different software versions by determining new and fixed violations of static code analysis results. However, their main goal is attributing these changes to individual developers for their awareness and subsequent elimination of those defects. Finally, Dunlap et al. [13] proposed a Differential Alert Analysis (DAA), using the results of static code analysis tools on different software versions, for detecting fixed vulnerabilities. It is meant to support the documentation of vulnerabilities, especially of those not publicly disclosed, improving the software supply chain security.

7.6 Applications of CodeQL

Lastly, the static analysis tool CodeQL that we are using in our approach was already successfully applied in multiple scenarios. Shcherbakov et al. [27] use it to detect code constructs in the core APIs of Node.js leading to remote code execution vulnerabilities. Chow et al. [9] combine static analysis and machine learning, in a bimodal taint analysis, to detect potentially vulnerable data flows in JavaScript projects. Bandara et al. [6] define an automatic technique which enables the study of vulnerability management in GitHub repositories. Furthermore, Mantovani et al. [20] included CodeQL in a set of SAST tools to evaluate their abilities to detect vulnerabilities in code of decompiled binaries; while Brito et al. [8] did it for JavaScript code and Lipp et al. [19] for C code.

8 CONCLUSION

We presented a new differential static analysis approach to support the detection of malicious package updates. In order to achieve this, we implemented behavior specifications as CodeQL queries to identify suspicious or sensitive code behavior. We compared this between two package versions for determining potentially malicious changes. While conceptually being language-agnostic, we focused on npm packages in this work and evaluated the approach on a set of packages with a known malicious version as well as a set of popular packages, presumed to be benign. The differential approach was able to detect every malicious version successfully, while keeping a manageable 1.4% rate of false positives. Consequently, even though further evaluation on a larger number of packages is required, this indicates our extendable method is capable of helping security experts and developers in the detection of malicious updates and to defend against attacks on the software supply chain.

REFERENCES

[1] Pranay Ahlawat, Johannes Boyne, Dominik Herz, Florian Schmieg, and Michael Stephan. 2021. Why You Need an Open Source Software Strategy. https://www.bcg.com/publications/2021/open-source-software-strategy-benefits Accessed: 2023-07-12.

[2] Iosif Arvanitis, Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. 2022. A systematic analysis of the event-stream incident. In *Proc. 15th European Workshop on Systems Security (EuroSec)*. ACM.

[3] Pavel Avgustinov, Arthur I. Baars, Anders Starcke Henriksen, R. Greg Lavender, Galen Menzel, Oege de Moor, Max Schäfer, and Julian Tibble. 2015. Tracking Static Analysis Violations over Time to Capture Developer Characteristics. In *37th Int. Conf. Software Engineering (ICSE)*. IEEE Computer Society.

[4] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conf. Object-Oriented Programming (ECOOP) (LIPIcs, Vol. 56)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25.

[5] Aadesh Bagmar, Josiah Wedgwood, Dave Levin, and Jim Purtilo. 2021. I Know What You Imported Last Summer: A study of security threats in the Python ecosystem. *CoRR* abs/2102.06301 (2021).

[6] Vinuri Bandara, Thisura Rathnayake, Nipuna Weerasekara, Charitha Elvitigala, Kenneth Thilakarathna, Primal Wijesekera, and Chamath Keppitiyagama. 2020. 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.

[7] Adam Bannister. 2021. Popular NPM package UA-Parser-JS poisoned with cryptomining, password-stealing malware. https://portswigger.net/daily-swig/popular-npm-package-ua-parser-js-poisoned-with-cryptomining-password-stealing-malware Accessed: 2023-07-12.

[8] Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, and Nuno Santos. 2023. Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages. Early access. *IEEE Trans. Reliability* (2023), 1–16.

[9] Yiu Wai Chow, Max Schäfer, and Michael Pradel. 2023. Beware of the Unexpected: Bimodal Taint Analysis. In *Proc. 32nd ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*. ACM.

[10] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. 2007. Keynote Address: .QL for Source Code Analysis. In *7th IEEE Int. Working Conf. Source Code Analysis and Manipulation (SCAM 2007)*. IEEE Computer Society.

[11] Idan Digmi. 2023. The rising trend of malicious packages in open source ecosystems. https://snyk.io/blog/malicious-packages-open-source-ecosystems/ Accessed: 2023-07-12.

[12] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annu. Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[13] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. 2023. Finding Fixed Vulnerabilities with Off-the-Shelf Static Analysis. In *8th European Symp. Security and Privacy (EuroS&P)*. IEEE.

[14] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *43rd Int. Conf. Software Engineering (ICSE)*. IEEE.

[15] Kalil Anderson Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting suspicious package updates. In *2019 IEEE/ACM 41st Int. Conf. Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE / ACM.

[16] GitHub. 2021. CodeQL code scanning: new severity levels for security alerts. https://github.blog/changelog/2021-07-19-codeql-code-scanning-new-severity-levels-for-security-alerts/ Accessed: 2023-07-12.

[17] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. 2021. Anomalicious: Automated Detection of Anomalous and Potentially Malicious Commits on GitHub. In *43rd Int. Conf. Software Engineering (ICSE)*. IEEE.

[18] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd Int. Symp. Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association.

[19] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. 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.

[20] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 2022. The Convergence of Source Code and Binary Vulnerability Discovery - A Case Study. In *Proc. 2022 ACM on Asia Conference on Computer and Communications Security (ASIACCS)*. ACM.

[21] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. 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.

[22] Marc Ohm, Timo Pohl, and Felix Boes. 2023. You Can Run But You Can't Hide: Runtime Protection Against Malicious Package Updates For Node.js. *CoRR* abs/2305.19760 (2023).

[23] Andrey Polkovnychenko and Shachar Menashe. 2022. Npm Supply Chain Attack Targets Germany-based Companies with Dangerous Backdoor Malware. https://jfrog.com/blog/npm-supply-chain-attack-targets-german-based-companies/ Accessed: 2023-07-12.

[24] Sonatype Developer Relations. 2023. Malware Monthly - March 2023. https://blog.sonatype.com/malware-monthly-march-2023 Accessed: 2023-07-12.

[25] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. 2022. On the Feasibility of Detecting Injections in Malicious Npm Packages. In *Proc. 17th Int. Conf. Availability, Reliability and Security (ARES)*. ACM.

[26] Adriana Sejfia and Max Schäfer. 2022. Practical Automated Detection of Malicious npm Packages. In *Proc. 44th Int. Conf. Software Engineering (ICSE)*. ACM.

[27] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. 2023. Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js. In *Proc. 32nd USENIX Security Symposium*. USENIX Association.

[28] Jaime Spacco, David Hovemeyer, and William W. Pugh. 2006. Tracking defect warnings across versions. In *Proc. 2006 Int. Workshop on Mining Software Repositories (MSR)*. ACM, 133–136.

[29] Martin Woodward. 2022. Octoverse 2022: 10 years of tracking open source. https://github.blog/2022-11-17-octoverse-2022-10-years-of-tracking-open-source/ Accessed: 2023-07-12.

[30] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2013. JStill: mostly static detection of obfuscated malicious JavaScript code. In *Proc. Third ACM Conf. Data and Application Security and Privacy (CODASPY)*. ACM.

[31] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Shekhar Maddila, and Laurie A. Williams. 2022. What are Weak Links in the npm Supply Chain?. In *44th Int. Conf. Software Engineering (ICSE)*. IEEE.

[32] Henry Zhu. 2018. Postmortem for Malicious Packages Published on July 12th, 2018. https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes/ Accessed: 2023-07-12.

[33] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proc. 28th USENIX Security Symposium*. USENIX Association.