SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS

Cristian-Alexandru Staicu
TU Darmstadt
cris.staicu@gmail.com

Michael Pradel
TU Darmstadt
michael@binaervarianz.de

Benjamin Livshits
Imperial College London
livshits@ic.ac.uk

***Abstract*—The Node.js ecosystem has lead to the creation of many modern applications, such as server-side web applications and desktop applications. Unlike client-side JavaScript code, Node.js applications can interact freely with the operating system without the benefits of a security sandbox. As a result, command injection attacks can cause significant harm, which is compounded by the fact that independently developed Node.js modules interact in uncontrolled ways. This paper presents a large-scale study across 235,850 Node.js modules to explore injection vulnerabilities. We show that injection vulnerabilities are prevalent in practice, both due to `eval`, which was previously studied for browser code, and due to the powerful `exec` API introduced in Node.js. Our study suggests that thousands of modules may be vulnerable to command injection attacks and that fixing them takes a long time, even for popular projects. Motivated by these findings, we present Synode, an automatic mitigation technique that combines static analysis and runtime enforcement of security policies to use vulnerable modules in a safe way. The key idea is to statically compute a template of values passed to APIs that are prone to injections, and to synthesize a grammar-based runtime policy from these templates. Our mechanism is easy to deploy: it does not require any modification of the Node.js platform, it is fast (sub-millisecond runtime overhead), and it protects against attacks of vulnerable modules, while inducing very few false positives (less than 10%).**

I. INTRODUCTION

JavaScript is the most widely-used programming language for the client-side of web applications, powering over 90% of today's web sites[1]. Recently, JavaScript has become increasingly popular for platforms beyond the browser: server-side and desktop applications that use NODE.JS, mobile programming (Apache Cordova/PhoneGap); it is even used for writing operating systems, such as Firefox OS. One of the forces behind using JavaScript in other domains is to enable client-side programmers to reuse their skills in other environments.

Unfortunately, this skill transfer also spreads the risk of misusing JavaScript in a way that threatens software security. One example of this is bad programming habits of client-side JavaScript, such as the widespread use of the `eval` construct [32], spreading to the emerging platforms. Additionally, new types of vulnerabilities and attacks become possible, which do not directly map to problems known in the client-side domain. For example, recent work shows that mobile applications written in JavaScript contain injection vulnerabilities [15] and that the impact of attacks in mobile applications is potentially more serious than that of client-side cross-site scripting (XSS). Others have shown how dangerous the use of dangerous and outdated JavaScript APIs can be [17]. Companies like the Node Security Platform[2] and Snyk[3] are maintaining vulnerability data for platforms that include NODE.JS and Ruby Gems, underlining the importance of these issues, but do no provide actionable prevention strategies.

This paper is the first to thoroughly investigate a security issue specific to JavaScript executed on the NODE.JS platform. Specifically, we focus on injection vulnerabilities, i.e., programming errors that enable an attacker to inject and execute malicious code in an unintended way. Injection vulnerabilities on the NODE.JS platform differ from those on other JavaScript platforms in three significant ways.

1) Injection APIs and impact of attacks: NODE.JS provides two families of APIs that may accidentally enable injections. The `eval` API and its variants take a string argument and interpret it as JavaScript code, allowing an attacker to execute arbitrary code in the context of the current application. The `exec` API and its variants take a string argument and interpret it as a shell command, giving the attacker access to system-level commands, beyond the context of the current application. Moreover, attackers may combine both APIs by injecting JavaScript code via `eval`, which then uses `exec` to execute shell commands. Because of these two APIs, and because NODE.JS lacks the security sandbox of the web browser, injection vulnerabilities in NODE.JS can cause significantly more harm than in the browser, e.g., by modifying the local file system or even taking over the entire machine.

[1] JavaScript usage statistics: `http://w3techs.com/technologies/details/cp-javascript/all/all`.

[2] `https://nodesecurity.io/`

[3] `https://snyk.io/`

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23071
www.ndss-symposium.org

2) Developer stance: While it is tempting for researchers to propose an analysis that identifies vulnerabilities as a solution, to have longer-range impact, it helps to understand NODE.JS security more holistically. By analyzing security issues reported in the past and through developer interactions, we observe that, while injection vulnerabilities are indeed an important problem, developers who use and maintain JavaScript libraries are generally reluctant to use analysis tools and are not always willing to fix their code.

To better understand the attitude of NODE.JS module developers toward potential injection flaws, we submitted a sample of 20 bug reports to developers on GitHub. Somewhat to our surprise, only about half the reports were attended to and only a small fraction was fixed (the results of this experiment are detailed in Figure 4). To understand the situation further, we reviewed many cases of the use of `eval` and `exec`, to discover that most (80%) could be easily refactored by hand, eliminating the risk of injections [24]. These observations suggest that even given the right analysis tool, it is *unlikely* that developers will proceed to voluntarily fix potential vulnerabilities.

3) Blame game: A dynamic we have seen developing is a *blame game* between NODE.JS module maintainers and developers who use these modules, where each party tries to claim that the other is responsible for checking untrusted input. Furthermore, while an individual developer can find it tempting to deploy a *local fix* to a vulnerable module, this patch is likely to be made obsolete or will simply be overwritten by the next module update. These observations motivated us to develop an approach that provides a high level of security with a very small amount of developer involvement.

Synode: Given the above observations about the development process around NODE.JS modules, it is crucial to offer a solution that provides complete automation. This paper presents SYNODE, an automatic approach to identify potential injection vulnerabilities and to prevent injection attacks. The basic idea is to check all third-party modules as part of their *installation* and to rewrite them to enable a *safe mode* proposed in this paper. A mix of two strategies is applied as part of rewriting.

- **Static:** we *statically* analyze the values that may be passed to APIs prone to injections. The static analysis also extracts a *template* that describes values passed to these dangerous APIs.
- **Runtime:** for code locations where the static analysis cannot ensure the absence of injections, we offer a *dynamic* enforcement mechanism that stops malicious inputs before passing them to the APIs.

A combination of these techniques is applied to a module at the time of installation via NODE.JS installation hooks, effectively enforcing a safe mode for third-party modules. In principle, while our runtime enforcement may be overly conservative, our evaluation in Section VIII shows that such cases are rare.

Alternative approaches: There are several alternatives to our static-dynamic analysis. One alternative is a sound static analysis that conservatively rejects all NODE.JS modules for which the analysis cannot guarantee the absence of injection vulnerabilities. Unfortunately, due to the dynamic features of JavaScript [2], a reasonably precise static analysis is virtually never sound [2], [47], whereas a fully sound analysis would result in many false positives. Another alternative is a training-based approach that learns from safe executions which values are safe to pass to APIs prone to injections. While this approach works well for client-side JavaScript [37], [25], [29], where a lot of runtime behavior gets triggered by loading the page, relying on training is challenging for NODE.JS code, which often comes without any inputs to execute the code. Finally, security-aware developers could resort to manually analyzing third-party modules for potential vulnerabilities. However, manual inspection does not scale well to the large number of modules available for NODE.JS and suffers from human mistakes. Instead of these alternatives, SYNODE takes a best-effort approach in the spirit of soundiness [21] that guarantees neither to detect all vulnerabilities nor to reject only malicious inputs. Our experimental evaluation shows that, in practice, SYNODE rejects very few benign inputs and detects all malicious inputs used during the evaluation.

SYNODE relates to existing work aimed at preventing code injections in JavaScript [11], [44], [35]. The closest existing approaches, Blueprint [44] and ScriptGard [35], share the idea of restricting runtime behavior based on automatically inferred templates. In contrast to them, SYNODE infers templates statically, i.e., without relying on inputs that drive the execution during template inference. Our work differs from purely static approaches [11] by defering some checks to runtime instead of rejecting potentially benign code. Moreover, all existing work addresses XSS vulnerabilities, whereas we address injection attacks on NODE.JS code.

Contributions:

- **Study**: We present a study of injection vulnerabilities in 235,850 NODE.JS modules, focusing on why and how developers use potentially dangerous APIs and whether developers appear open to using tools to avoid these APIs. (Section III)
- **Static analysis**: We present a static analysis that attempts to infer templates of user inputs used at potentially dangerous sinks. (Section IV)
- **Runtime enforcement**: For cases that cannot be shown safe via static analysis, we present a runtime enforcement achieved through code rewriting. The runtime approach uses partially instantiated abstract syntax trees (ASTs) and ensures that the runtime values do not introduce any unwanted code beyond what is expected. (Section IV)
- **Evaluation**: We apply our static technique to a set of 15,604 NODE.JS modules that contain calls to sinks. We discover that 36.66% of the sink call sites are statically guaranteed to be safe. For a subset of the statically unsafe modules, we create both malicious inputs that exploit the injection vulnerabilities and benign inputs that exercise the advertised functionality of the module. Our runtime mechanism effectively

prevents 100% of the attacks, while being overly conservative for only 8.92% of the benign inputs.

Our implementation and a benchmark suite containing both malicious and benign inputs passed to the vulnerable modules is available for download:

`https://github.com/sola-da/Synode`

II. Background and Example

Node.js and injection APIs: The Node.js platform is the de-facto standard for executing JavaScript outside of browsers. The platform provides two families of APIs that may allow an attacker to inject unexpected code, which we call *injection APIs*. First, `exec` enables command injections if an attacker can influence the string given to `exec`, because this string is interpreted as a shell command. The `exec` API has been introduced by Node.js and is not available in browsers. Second, calling `eval` enables code injections if an attacker can influence the string passed to `eval`, because this string is interpreted as JavaScript code. Since code injected via `eval` may contain calls to `exec`, any code injection vulnerability is also a command injection vulnerability. The latter distinguishes server-side JavaScript from the widely studied client-side problems of `eval` [32] and introduces an additional security threat. In this paper, we focus on `exec` and `eval`, as these are the most prominent members of the two families of APIs. Extending both our study and our mitigation mechanism to more APIs, e.g., `new Function()` or modules, e.g., `shelljs` is straightforward. Moreover, the approach can also be applied with minimal effort to other types of security vulnerabilities, e.g. SQL injections and path traversals.

In contrast to the browser platform, Node.js does not provide a security sandbox that controls how JavaScript code interacts with the underlying operating system. Instead, Node.js code has direct access to the file system, network resources, and any other operating system-level resources provided to processes. As a result, injections are among the most serious security threats on Node.js, as evidenced by the Node Security Platform[4], where, at the time of writing, 20 out of 66 published security advisories address injection vulnerabilities.

Module system: Code for Node.js is distributed and managed via the *npm* module system. A module typically relies on various other modules, which are automatically installed when installing the module. There is no mechanism built into npm to specify or check security properties of third-party modules before installation.

Motivating example: Figure 1 shows a motivating example that we use throughout the paper to illustrate our approach. The function receives two parameters from an unknown source and uses them to copy a file on the local file system. The parameters are intended to represent a file name and a file extension, respectively. To copy the file, lines 2 to 5 construct a string that is passed as a command to `exec` (line 7), which will execute a shell command. The code also logs a message to the console. Line 10 retrieves

[4] `https://nodesecurity.io/advisories/`

```
1 function backupFile(name, ext) {
2   var cmd = [];
3   cmd.push("cp");
4   cmd.push(name + "." + ext);
5   cmd.push("~/.localBackup/");
6
7   exec(cmd.join(" "));
8
9   var kind = (ext === "jpg") ? "pics" : "other";
10  console.log(eval("messages.backup_" + kind));
11 }
```

Fig. 1: Motivating example.

the content of the message by looking up a property of the `messages` object. The property and the message depend on the extension of the backed up file. Implementing a lookup of a dynamically computed property with `eval` is a well-known misuse of `eval` that frequently occurs in practice [32]. For example, suppose the function is called with `backupFile("f", "txt")`. In this case, the command will be `cp f.txt ~/.localBackup` and the logged message will be the message stored in `messages.backup_other`.

The example contains two calls to APIs that may allow for injecting code (lines 7 and 10). As an example for an injection attack, let us consider the following call:

```
backupFile("-help && rm -rf * && echo ", "")
```

The dynamically constructed command will be:

```
cp -help && rm -rf * && echo . ~/.localBackup/
```

Unfortunately, this command does not backup any files but instead it creates space for future backups by deleting all files in the current directory. Such severe consequences distinguish the problem of injections on Node.js from injections known from client-side JavaScript, such as XSS: because Node.js code runs without any sandbox that could prevent malicious code from accessing the underlying system, an attacker is able to inject arbitrary system-level commands.

III. A Study of Injection Vulnerabilities

To better understand how developers of JavaScript for Node.js handle the risk of injections, we conduct a comprehensive empirical study involving 235,850 npm modules. We investigate four research questions (RQs).

A. RQ1: Prevalence

At first, we study whether APIs that are prone to injection vulnerabilities are widely used in practice. We call a module that directly calls an injection API an *injection module*. To assess whether a module uses another module that calls an injection API, we analyze dependences between modules, as specified in their `package.json` file. Given an injection module m_{inj}, we say that another module m_1 has a level-1 (level-2) dependence if it depends on m_{inj} (via another module). Figure 2 shows how many npm modules use injection APIs, either directly or via another module. We find that 7,686 modules and 9,111 modules use `exec` and `eval`, respectively, which corresponds to 3% and 4% of all modules. In total, 15,604 modules use at least one injection API. Furthermore, about 20% of

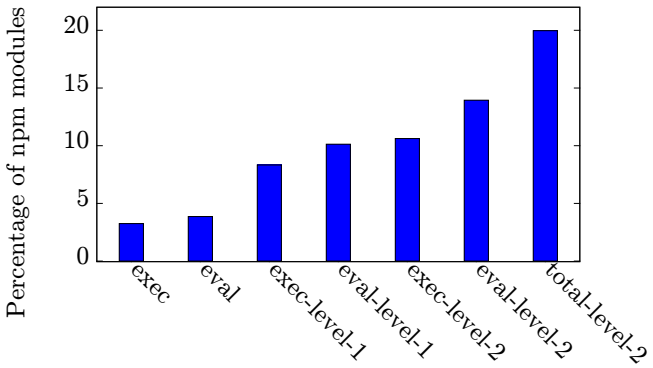


Fig. 2: Prevalence of uses of injection APIs in npm modules.

all modules depend directly or indirectly on at least one injection API.

Estimating the potential effect of protecting vulnerable modules shows that fixing calls to the injection APIs in the most popular 5% of all injection modules will protect almost 90% of all directly dependent modules. While this result is encouraging, it is important to note that 5% of all modules still corresponds to around 780 modules, i.e., many more than would be reasonable to fix manually. Moreover, manually fixing these modules now would be a point-in-time solution that does not ensure the safety of future versions of modules and of new modules.

B. RQ2: Usage

To understand why developers use injection APIs, we identify recurring usage patterns and check whether the usages could be replaced with less vulnerable alternatives. We manually inspect a random sample of 50 uses of `exec` and 100 uses of `eval` and identify the following patterns.

Patterns of `exec` usage: The majority of calls (57%) trigger a single operating system command and pass a sequence of arguments to it. For these calls, the developers could easily switch to `spawn`, which is a safer API to use, equivalent to the well-known `execv` functions in C. The second-most common usage pattern (20%) involves multiple operating system commands combined using Unix-style pipes. For this pattern, we are not aware of a simple way to avoid the vulnerable `exec` call. The above-mentioned `spawn` accepts only one command, i.e., a developer would have to call it multiple times and emulate the shell's piping mechanism. Another interesting pattern is the execution of scripts using relative paths, which accounts for 10% of the analyzed cases. This pattern is frequently used as an ad-hoc parallelization mechanisms, by starting another instance of Node.js, and to interoperate with code written in other programming languages.

Patterns of `eval` usage: Our results of the usage of `eval` mostly match those reported in a study of client-side JavaScript code [32], showing that their findings extend

```
1 function escape(s) {
2   return s.replace(/"/g, '\\"');
3 }
4 exports.open = function open(target, callback) {
5   exec(opener + ' "' + escape(target) + '"');
6 }
7
8 // Possible attack: open("`rm -rf *`");
```

Fig. 3: Regular expression-based sanitization and input that bypasses it.

to Node.js JavaScript code. One usage pattern that was not previously reported is to dynamically create complex functions. This pattern, which we call "higher-order functions", is widely used in server-side JavaScript for creating functions from both static strings and user-provided data. We are not aware of an existing technique to easily refactor this pattern into code that does not use `eval`.

Overall, we find that over 20% of all uses of injection APIs cannot be easily removed. Furthermore, many of the remaining uses are unlikely to be refactored by the developers, e.g., because techniques for removing usages of `eval` [24], [14] are available but not adopted by developers.

C. RQ3: Existing Mitigation

To understand how developers deal with the risk of injections, we study to what extent data gets checked before being passed into injection APIs. Specifically, we analyze two conditions. First, whether a call site of an injection API may be reached by attacker-controlled data, i.e., whether any mitigation is required. We consider data as potentially attacker-controlled if it is passed as an input to the module, e.g., via a network request, or if the data is passed from another module and then propagates to the injection call site. Second, if the call site requires mitigation, we analyze which mitigation technique the developers use. We find that 58% of the inspected call sites are exploitable, i.e., attacker-controlled data may reach the injection API. Among these call sites, the following mitigation techniques are used:

None: A staggering 90% of the call sites do not use any mitigation technique at all. For example, the call to `exec` in the motivating example in Figure 1 falls into this category.

Regular expressions: For 9% of the call sites, the developers harden their module against injections using regular expression-based checks of input data. An example in our data set is shown in Figure 3. Unfortunately, most regular expressions we inspected are not correctly implemented and cannot protect against all possible injection attacks. For example, the `escape` method in Figure 3 does not remove back ticks, allowing an attacker to deliver a malicious payload using the command substitution syntax, as illustrated in the last line of Figure 3. In general, regular expressions are fraught with danger when used for sanitization [13].

Sanitization modules: To our surprise, none of the modules uses a third-party sanitization module to prevent

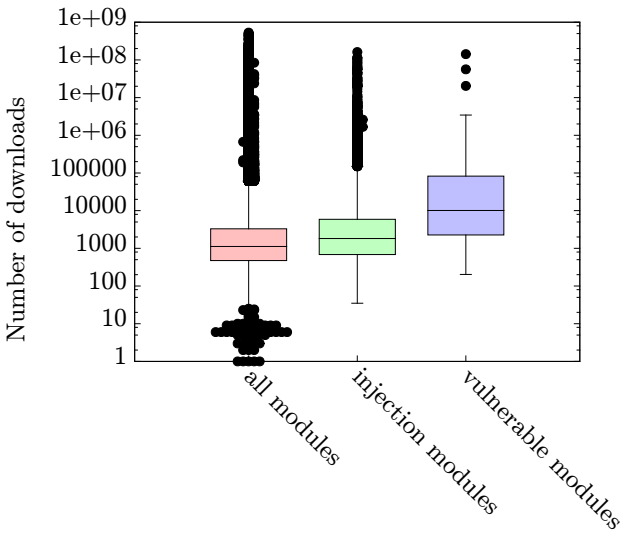


Fig. 4: Comparison of the popularity of all the modules, modules with calls to injection APIs, and modules with reported vulnerabilities. The boxes indicate the lower quartile (25%) and the upper quartile (75%); the horizontal line marks the median; the dots are outliers.

injections. To validate whether any such modules exists, we searched the npm repository and found six modules intended to protect calls to `exec` against command injections: shell-escape, escapeshellarg, command-join, shell-quote, bash, and any-shell-escape. In total, 198 other modules depend on one of these sanitization modules, i.e., only a small fraction of the 19,669 modules that directly or indirectly use `exec`. For `eval`, there is no standard solution for sanitization and the unanimous expert advice is to either not use it at all in combination with untrustworthy input, or to rely on well tested filters that allow only a restricted class of inputs, such as string literals or JSON data.

We conclude from these results that a large percentage of the 15,604 modules that use injection APIs are *potentially vulnerable*, and that standard sanitization techniques are rarely used. Developers are either unaware of the problem in the first place, unwilling to address it, or unable to properly apply existing solutions.

D. RQ4: Maintenance

To understand whether module developers are willing to prevent vulnerabilities, we reported 20 previously unknown command injection vulnerabilities to the developers of the modules that call the injection APIs. For each vulnerability, we describe the problem and provide an example attack. Most of the developers acknowledge the problem. However, in the course of several months, only three of the 20 vulnerabilities have been completely fixed, confirming earlier observations about the difficulty of effectively notifying developers [8], [9], [41].

One may hypothesize that these vulnerabilities are characteristic to unpopular modules that are not expected to be well maintained. We checked this hypothesis by measuring the number of downloads between January 1 and February 17, 2016 for three sets of modules: (i) modules with vulnerabilities reported either by us or by others via the Node Security Platform, (ii) all modules that call an injection API, (iii) all modules in the npm repository. Figure 4 summarizes our results on a logarithmic scale. The boxes are drawn between the lower quartile (25%) and the upper one (75%) and the horizontal line marks the median. The results invalidate the hypothesis that vulnerable modules are unpopular. On the contrary, we observe that various vulnerable modules and injection modules are highly popular, exposing millions of users to the risk of injections.

E. Case Study: The growl Module

To better understand whether developers are aware of possible injection vulnerabilities in modules that they use, we manually analyzed 100 modules that depend on growl. The growl module displays notifications to users by invoking a particular command via `exec`, which is one of the vulnerabilities we reported as part of RQ4. We found that modules depending on growl pass various kinds of data to growl, including error messages and data extracted from web pages. As anticipated in RQ1, vulnerabilities propagate along module dependences. For example, the loggy module exposes the command injection vulnerability in growl to 15 other modules that depend on loggy by sending inputs directly to growl without any check or sanitization.

We found only four modules that sanitize the data before sending it to the vulnerable module: autolint, mqtt-growl, bungle, and chook-growl-reporter. We report these sanitizers in Figure 5. Sadly, we find that all these methods are insufficient: one can easily bypass them, as illustrated by the example input at the end of Figure 5. The input again exploits the command substitution syntax, which is not considered by any of the sanitizers.

Impact of our study: After we published a preliminary version of this paper [39], several providers of Node.js vulnerability databases included findings of the study as vulnerability reports.[5]

IV. Overview

The previous section shows that the risk of injection vulnerabilities is widespread, and that a practical technique to mitigate them must support module maintainers who are not particularly responsive. Motivated by these findings, this section presents Synode, a novel mitigation approach that combines static analysis and runtime enforcement into a fully automatic approach to prevent injection attacks. To the best of our knowledge, our approach is the first to address the problem of injection vulnerabilities in Node.js JavaScript code.

The overall idea of the mitigation technique is to prevent injections at the call sites of injection APIs. Figure 6

[5] `https://snyk.io/vuln/page/2?type=npm`, CWE-94, npm:nd-validator:20160408 and `https://nodesecurity.io/advisories`.

```
1  // sanitization in autolint
2  function escape(text) {
3    return text.replace('$', '\\$');
4  }
5
6  // sanitization in mqtt-growl
7  message = message.replace(/"/g, "\\\"");
8
9  // sanitization in bungle
10 const ansiRx =
11   /[\u001b\u009b][[()#;?]*
12   (?:[0-9]{1,4}(?:;[0-9]{0,4})*)?
13   [0-9A-ORZcf-nqry=><]/g;
14 Growl(message.replace(ansiRx, ''));
15
16 // sanitization in chook-growl-reporter
17 function escapeForGrowl(text) {
18   var escaped = text.replace(/\(/g, '\\(');
19   escaped = escaped.replace(/\)/g, '\\)');
20   escaped = escaped.replace(/\""/g, '\\""');
21   return escaped;
22 }
23
24 // input that bypasses all the sanitizations:
25 // "tst`rm -rf *`";
```

Fig. 5: Broken sanitization in growl's clients.

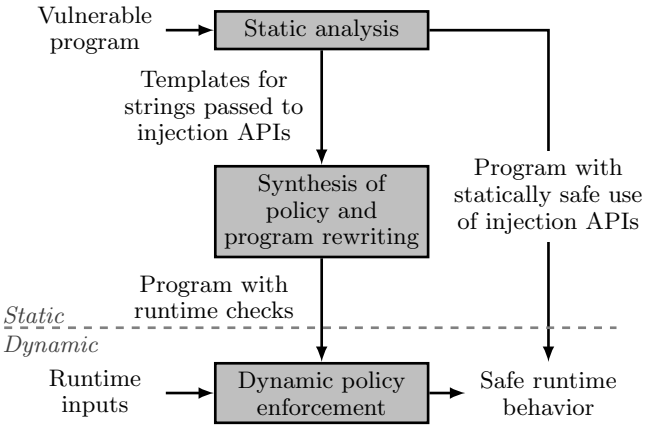
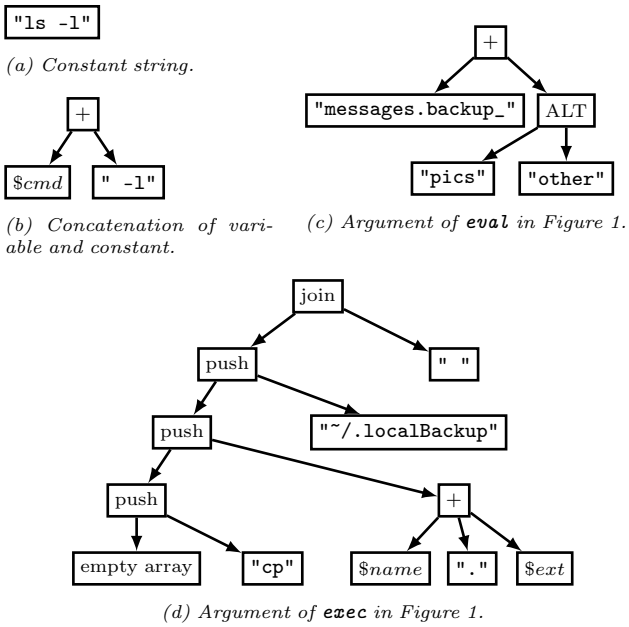


Fig. 6: Architectural diagram of Synode.

shows an overview of the approach. Given a potentially vulnerable JavaScript module, a static analysis summarizes the values that may flow into injection APIs in the form of string templates, or short *templates*. A template is a sequence consisting of string constants and holes to be filled with untrusted runtime data. For call sites where the analysis can statically show that no untrusted data may flow into the injection API, no further action is required to ensure safe runtime behavior. Similar approaches for identifying statically safe call sites are adopted in practice for other languages, e.g., Java[6] and Python[7].

For the remaining call sites, the approach synthesizes a runtime check and statically rewrites the source code to perform this check before untrusted data reaches the injection API. When executing the module, the rewritten code enforces a security policy that checks the runtime values to be filled into the holes of the template against

[6]https://www.youtube.com/watch?v=ccfEu-Jj0as
[7]https://github.com/dropbox/python-invariant



(a) Constant string.

(b) Concatenation of variable and constant.

(c) Argument of `eval` *in Figure 1.*

(d) Argument of `exec` *in Figure 1.*

Fig. 7: Examples of template trees.

the statically extracted template. If this check fails, the program is terminated to prevent an injection attack.

The Synode approach is conservative in the sense that it prevents potential vulnerabilities without certain knowledge of whether a vulnerability gets exploited by an attacker. The reason for this design decision is twofold. First, most Node.js modules are used in combination with other modules, i.e., we cannot reason about the entire program. Second, there is no trust model that specifies which module should sanitize untrusted inputs or even which inputs are untrusted. Our assumption is that user inputs and inputs from other modules are potentially attacker-controlled, an assumption shared by the vulnerabilities published at the Node Security Platform. Given these constraints and assumptions, Synode protects users of potentially vulnerable modules in an automatic way.

V. Static Analysis

We present a static analysis of values passed to injection APIs. For each call site of such an API, the analysis summarizes all values that may be passed to the called function into a tree representation (Section V-A). Then, the analysis statically evaluates the tree to obtain a set of templates, which represent the statically known and unknown parts of the possible string values passed to the function (Section V-B). Finally, based on the templates, the analysis decides for each call site of an injection API whether it is statically safe or whether to insert a runtime check that prevents malicious strings from reaching the API (Section V-C).

A. Extracting Template Trees

The analysis is a flow-sensitive, path-insensitive, intra-procedural, backward data flow analysis. Starting from a

Example	Templates
(a)	t_{a1} = ["`ls -l`"]
(b)	t_{b1} = [$\$cmd$, "` -l`"]
(c)	t_{c1} = ["`messages.backup_pics`"]
	t_{c2} = ["`messages.backup_other`"]
(d)	t_{d1} = ["`cp `", $\$name$, "`.`", $\$ext$, , "` ~/.localBackup`"]

Fig. 8: Evaluated templates for the examples in Figure 7.

call site of an injection API, the analysis propagates information about the possible values of the string argument passed to the API call along inverse control flow edges. The propagated information is a tree that represents the current knowledge of the analysis about the value:

Definition 1 (Template tree)**.** *A template tree is an acyclic, connected, directed graph* $(\mathcal{N}, \mathcal{E})$ *where*

- *a node* $n \in \mathcal{N}$ *represents a string constant, a symbolic variable, an operator, or an alternative, and*
- *an edge* $e \in \mathcal{E}$ *represents a nesting relationship between two nodes.*

Figure 7 shows several examples of template trees:

- Example (a) represents a value known to be a string constant "`ls -l`". The template tree consist of a single node labeled with this string.
- In example (b), the analysis knows that the value is the result of concatenating the value of a symbolic variable $\$cmd$ and the string constant "` -l`". The root node of the template tree is a concatenation operator, which has two child nodes: a symbolic variable node and a string constant node.
- Example (c) shows the tree that the analysis extracts for the values that may be passed to `eval` at line 10 of Figure 1. Because the value depends on the condition checked at line 9, the tree has an alternative node with children that represent the two possible string values.
- Finally, example (d) is the tree extracted for the value passed to `exec` at line 7 of Figure 1. This tree contains several operation nodes that represent the push operations and the string concatenation that are used to construct the string value, as well as several symbolic variable nodes and string constant nodes.

To extract such templates trees automatically, we use a data flow analysis [26], [1], which propagates template trees through the program. Starting at a call site of an injection API with an empty tree, the analysis applies the following transfer functions:

- *Constants.* Reading a string constant yields a node that represents the value of the constant.
- *Variables.* A read of a local variable or a function parameter yields a node that represents a symbolic variable.
- *Operations.* Applying an operation, such as concatenating two strings with `+`, yields a tree where the root node represents the operator and its children represent the operands.
- *Calls.* A call of a function yields a tree where the root node represents the called function and its children represent the base object and arguments of the call.
- *Assignments.* An assignment of the form $lhs = rhs$ transforms the current tree by replacing any occurrence of the symbolic variable that corresponds to lhs by the tree that results from applying the transition function to rhs.

Whenever the backward control flow merges, the analysis merges the two template trees of the merged branches. The merge operation inserts an alternative node that has the two merged trees as its children. To avoid duplicating subtrees with identical information, the analysis traverses the two given trees t_1 and t_2 to find the smallest pair of subtrees t_1' and t_2' that contain all differences between t_1 and t_2, and then inserts the alternative node as the parent of t_1' and t_2'.

Template tree construction example. For example, consider the call site of `eval` at line 10 of Figure 1. Starting from an empty tree, the analysis replaces the empty tree with a tree that represents the string concatenation at line 10. One child of this tree is a variable node that represents the variable `kind`, which has an unknown value at this point. Then, the analysis reasons backwards and follows the two control flow paths that assign "`pics`" and "`other`" to the variable `kind`, respectively. For each path, the analysis updates the respective tree by replacing the variable node for `kind` with the now known string constant. Finally, the variable reaches the merge point of the backward control flow and merges the two trees by inserting an alternative node, which yields the tree in Figure 7c.

B. Evaluating Template Trees

Based on the template trees extracted by the backward data flow analysis, the second step of the static analysis is to evaluate the tree for each call site of an injection API. The result of this evaluation process is a set of templates:

Definition 2 (Template)**.** *A template is a sequence* $t = [c_1, \ldots, c_k]$ *where each* c_i *represents either a constant string or an unknown value (hole).*

For example, the template trees in Figure 7 are evaluated to the templates in Figure 8. To obtain the templates for a given tree, the analysis iteratively evaluates subtrees in a bottom-up way until reaching the root node. The evaluation replaces operation nodes that have a known semantics with the result of the operation. Our implementation currently models the semantics of string concatenation, `Array.push`, `Array.join`, and `String.replace` where the arguments are constant strings. These operations cover most templates trees that the analysis extracts from real-world JavaScript code (Section VIII-A). For alternative nodes, the evaluation considers both cases separately, duplicating the number of template trees that result from the evaluation.

Finally, the analysis transforms each evaluated tree into a template by joining continuous sequences of characters into constant strings and by representing all symbolic values and unknown operations between these constants as unknown values.

C. Identifying Statically Safe Calls

After evaluating template trees, the analysis knows for each call site of an injection API the set of templates that represent the string values passed to the call. If all templates for a particular call site are constant strings, i.e., there are no unknown parts in the template, then the analysis concludes that the call site is statically safe. For such statically safe call sites, no runtime checking is required. In contrast, the analysis cannot statically ensure the absence of injections if the templates for the call site contain unknown values. In this case, checking is deferred to runtime, as explained in Section VI.

For our running example, the analysis determines that the `eval` call site at line 10 of Figure 1 is statically safe because both possible values passed to the function are known. In contrast, parts of the strings that may be passed to `exec` at line 7 are unknown and therefore the check whether an injection happens is deferred to runtime.

VI. Dynamic Enforcement

For call sites where the values passed to the injection API cannot be statically determined, we provide a dynamic enforcement mechanism. The goal of this mechanism is to reject values found to be dangerous according to a policy. Intuitively, we want to prevent values that expand the template computed for the call site in a way that is likely to be unforeseen by the developer. Our approach achieves this goal in two steps:

1) Before executing the module, the approach transforms the statically extracted set of templates for a call site into a set of partial abstract syntax trees (PAST) that represents the expected structure of benign values. The trees are partial because the unknown parts of the template are represented as unknown subtrees.
2) While executing the module, the approach parses the runtime value passed to an injection API into an AST and compares the PASTs from step 1 against the AST. The runtime mechanism enforces a policy that ensures that the runtime AST is (i) derivable from at least one of the PASTs by expanding the unknown subtrees and (ii) these expansions remain within an allowed subset of all possible AST nodes.

The following two subsections present the two steps of the dynamic enforcement mechanism in detail.

A. Synthesizing a Tree-based Policy

The goal of the first step is to synthesize for each call site a set of trees that represents the benign values that may be passed to the injection API. Formally, we define these trees as follows:

Definition 3 (Partial AST)**.** *The partial AST (PAST) for a template of an injection API call site is an acyclic, connected, directed graph* $(\mathcal{N}, \mathcal{E})$ *where*

- $\mathcal{N}_{sub} \subseteq \mathcal{N}$ *is a set of nodes that each represent a subtree of which only the root node* $n_{sub} \in \mathcal{N}_{sub}$ *is known, and*
- $(\mathcal{N}, \mathcal{E})$ *is a tree that can be expanded into a valid AST of the language accepted by the API.*

For example, Figure 9a shows the PAST for the template t_{d1} from Figure 8. For this partial tree, $\mathcal{N}_{sub} = \{\text{HOLE}\}$, i.e., the hole node can be further expanded, but all the other nodes are fixed.

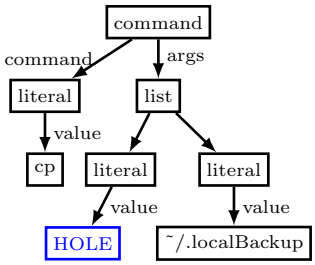
To synthesize the PAST for a template, the approach performs the following steps. At first, it instantiates the template by filling its unknown parts with simple string values known to be benign. The set of *known benign values* must be defined only once for each injection API. Figure 10 shows the set of values we use for `exec` and `eval`, respectively. The approach exhaustively tries all possible assignments of these values to the unknown parts of a template. Then, each of the resulting strings is given to a parser of the language, e.g., a JavaScript or Bash parser. If and only if the string is accepted as a legal member of the language, then the approach stores the resulting AST into a set of *legal example ASTs*.

Given the legal example ASTs for a template, the next step is to merge all of them into a single PAST. To this end, the approach identifies the least common nodes of all ASTs, i.e., nodes that are shared by all ASTs but that have a subtree that differs across the ASTs. At first, the given ASTs are aligned by their root nodes, which must match because all ASTs belong to the same language. Then, the approach simultaneously traverses all ASTs in a depth-first manner and searches for nodes n_{lc} with children that differ across at least two of the ASTs. Each such node n_{lc} is a least common node. Finally, the approach creates a single PAST that contains the common parts of all ASTs and where the least common nodes remain unexpanded and form the set $\mathcal{N}_{sub}$ (Definition 3). Note that $\mathcal{N}_{sub}$ is effectively an under-approximation of the possible valid inputs, given that we construct it using a small number of known benign inputs. However, in practice we do not observe any downsides to this approach, as discussed in Section VIII.

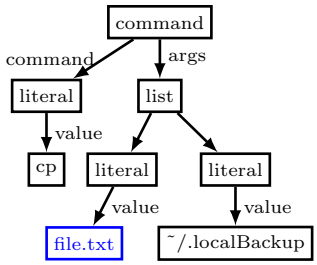
Policy synthesizing example. For example, for the template t_{d1} and the known benign inputs for Bash in Figure 10, the first argument passed to `cp` will be expanded to the values `./file.txt.ls`, `ls.ls`, `./file.txt../file.txt` and `ls../file.txt`. All these values are valid literals according to the Bash grammar, i.e., we obtain four legal example ASTs. By merging these ASTs, the approach obtains the PAST in Figure 9a because the only variations observed across the four ASTs are in the value of the literal.

B. Checking Runtime Values Against the Policy

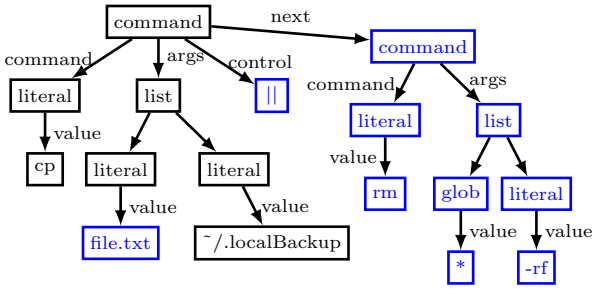
The set of PASTs synthesized for a call site is the basis of a policy that our mechanism enforces for each string



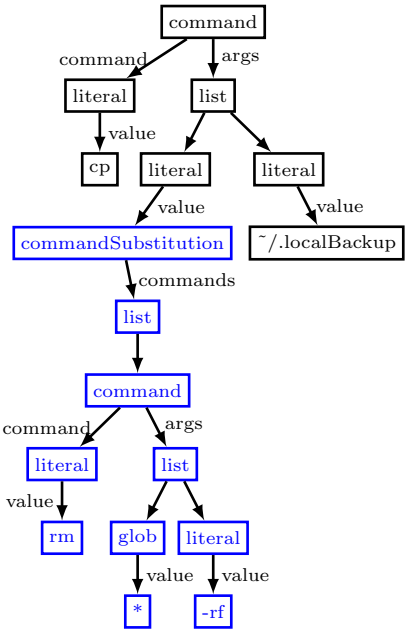
(a) Partial AST for the template in Figure 8.



(b) AST accepted by the policy derived from the partial AST.



(c) AST rejected by the policy derived from the partial AST.



(d) AST rejected by the policy derived from the partial AST.

Fig. 9: A partial AST and three ASTs compared against it. The blue nodes are holes and runtime values filled into the holes at runtime.

API	Language	Known benign values
`exec`	Bash	`"./file.txt"`, `"ls"`
`eval`	JavaScript	`x`, `y`, `"x"`, `x.p`, `{x:23}`, `23`

Fig. 10: Known benign values used to synthesize PASTs.

passed at the call site. We implement this enforcement by rewriting the underlying JavaScript code at the call site. When a runtime value reaches the rewritten call site, then the runtime mechanism parses it into an AST and compares it with the PASTs of the call site. During this comparison, the policy enforces two properties:

- **P1**: The runtime value must be a syntactically valid expansion of any of the available PASTs. Such an expansion assigns to each node $n_{sub} \in \mathcal{N}_{sub}$ a subtree so that the resulting tree (i) is legal according to the language and (ii) structurally matches the runtime value's AST.
- **P2**: The expansion of a node n_{sub} of the PAST is restricted to contain only AST nodes from a predefined set of *safe node types*. The set of safe node types is defined once per language, i.e., it is independent of the specific call site and its PASTs. For shell commands passed to `exec`, we consider only nodes that represent literals as safe. For JavaScript code passed to `eval`, we allow all AST node types that occur in JSON code, i.e., literals, identifiers, properties, array expressions, object expressions, member expressions, and expression statements. The rationale for choosing safe node types is to prevent an attacker from injecting code that has side effects. With the above safe node types, an attacker can neither call or define functions, nor update the values of properties or variables. As noted in previous work [30], such a restrictive mechanism may cause false positives, which we find to be manageable in practice though.

Policy checking example. To illustrate these properties, suppose that the three example inputs in Figure 11 are given to the `backupFile` function in Figure 1. Input 1 uses the function as expected by the developer. In contrast, inputs 2 and 3 exploit the vulnerability in the call to `exec` by passing data that will cause an additional command to be executed. Figure 9 shows the PAST derived (only one because there is only one template available for this call site) for the vulnerable call site and the ASTs of the three example inputs. Input 1 fulfills both P1 and P2 and the value is accepted. In contrast, the policy rejects input 2 because it does not fulfill P1. The reason is that the AST of the input (Figure 9c) does not structurally match the PAST. Likewise, the policy rejects input 3 because it fulfills P1 but not P2. The reason for not fulfilling P2 is that the expanded subtree (i.e., the highlighted nodes in Figure 9d) contain nodes that are not in the set of safe node types.

To summarize, the enforced policy can be formalized as follows:

Definition 4 (Security Policy)**.** *Given a runtime value v, a set of PASTs $\mathcal{T}$, and a set $\mathcal{N}_{safe}$ of safe node types, v is* rejected *unless there exists an expansion t' of some $t \in \mathcal{T}$,*

ID	name	ext	Property	
			P1	P2
1	file	txt	✓	✓
2	file	txt || rm * -rf	✗	–
3	file	$(rm * -rf)	✓	✗

Fig. 11: Inputs compared against the partial AST in Figure 9a.

where

- *t' is isomorphic to the AST of v, and*
- *let $\mathcal{N}_{input}$ be the set of nodes that belong to a subtree in the AST of v that matches a node in $\mathcal{N}_{sub} \in t$, then the node type of all $n \in \mathcal{N}_{input}$ is in $\mathcal{N}_{safe}$.*

Our runtime enforcement approach can be applied to any kind of injection API that expects string values specified by a context-free grammar. The effectiveness of the enforcement depends on two language-specific ingredients: the set of benign example inputs and the set of safe AST node types. Given that we are primarily interested in `eval` and `exec` sinks, we have created these ingredients for JavaScript and Bash, and Section VIII-B shows both to be effective for real-world Node.js code.

VII. Implementation

Static analysis: We implement the static analysis in Java, building upon the Google Closure Compiler[8]. The analysis is an intraprocedural, backward data flow analysis, as described in Section V-A. The states propagated along the control flow edges are sets of template trees and the join operation is the union of these sets. To handle loops and recursion, the static data flow analysis limits the number of times a statement is revisited while computing a particular data flow fact to ten. When applying the static analysis to a module, we impose a one minute timeout per module. Considering the deployment strategy we propose for Synode later in this section, we believe that an analysis that takes longer would be of little practical use. We show in the evaluation that the cases in which the timeout expires are rare and therefore for these cases, Synode alerts the user that a manual inspection of the module is needed. As described in Section V-B, after finishing the data flow analysis of a module, the implementation transforms the collected template trees into templates. Lastly, the analysis writes the set of templates for each call site into a text file to be used by the dynamic analysis.

Runtime analysis: We implement the dynamic analysis in JavaScript. Before executing the module, the analysis pre-computes the PASTs for each call site based on the templates gathered by the static analysis. While executing a module, the analysis intercepts all calls to `exec` and `eval` and extracts the strings passed to these function to be checked against our policy. To parse strings given to `exec` and `eval`, we build upon the esprima[9] and shell-parse[10] modules.

[8] `https://developers.google.com/closure/`

Kind of template tree	Call sites	
	exec	eval
Evaluates to constant string without holes	31.05%	39.29%
Holes due to symbolic variables only	49.02%	34.52%
Holes due to unsupported operations	19.93%	26.19%

Fig. 12: Template trees extracted by the static analysis.

Automatic deployment: As shown by our study (Section III), the practical benefits of a technique to prevent injection attacks depend on how seamlessly the technique can be deployed. A particular challenge is how to apply a mitigation technique to code written by third parties that may not be willing to modify their code. To make the deployment of Synode as easy as possible without relying on the cooperation of third-party code providers, we advocate an approach in which a module developer or a system administrator adds a post-installation script[11] to the application packaged as an npm module.

The script runs on each explicitly declared third-party dependent module and, if necessary, performs the code rewriting step that adds dynamic enforcement at each statically unsafe call site of an injection API. As a result, our technique to prevent injection attacks can be deployed with very little effort and without requiring any knowledge about third-party code.

VIII. Evaluation

We evaluate our mitigation technique by applying it to all 235,850 Node.js modules. To avoid analyzing modules without any injection call sites, we filter modules by searching for call sites of these methods and include all 15,604 modules with at least one such call site in our evaluation. We apply our static analysis for each module separately to decide whether the sink call sites are statically safe or runtime protection is needed for that module. Since evaluating the runtime mechanism requires inputs that exercise the modules, we consider a subset of the modules, with known vulnerabilities, found by others or by us during the study (Section III).

We perform all our measurements on a Lenovo ThinkPad T440s laptop with an Intel Core i7 CPU (2.10GHz) and 12 GB of memory, running Ubunu 14.04.

A. Static Analysis

Statically safe call sites: The static analysis finds 18,924 of all 51,627 call sites (36.66%) of injection APIs to be statically safe. That is, the values that are possibly passed to each of these call sites are statically known, and an attacker cannot modify them. To further illustrate this point, Figure 12 shows to what extent the analysis can

[9] `http://esprima.org/`

[10] `https://www.npmjs.com/package/shell-parse`

[11] `https://docs.npmjs.com/misc/scripts`

evaluate trees into templates. For 31.05% and 39.29% of all call sites of `exec` and `eval`, respectively, the template tree contains only constant nodes, operators supported by the analysis, and alternative nodes, which yield constant strings after evaluating the tree.

The remaining template trees also contain symbolic variable nodes. Most of these trees (49.02% and 34.52%) are fully evaluated by the analysis, i.e., they contain no unsupported operators. It is important to note that the static analysis may provide a useful template even if the template tree contains an unsupported operation. The reason is that the other nodes in the tree often provide enough context around the unknown part created by the unsupported operation.

Context encoded in templates: To better understand the templates extracted by the static analysis, we measure how much context about the passed string the static analysis extracts. First, we measure for each call site how many known characters are present per template, on average. The majority of call sites contain at least 10 known characters and for 10,967 call sites (21.24%), there is no known character, i.e., our approach relies entirely on dynamic information. Second, we measure how many unknown parts the extracted templates contain. As shown in Figure 13a, the templates for the vast majority of call sites has at most one hole, and very few templates contain more than five holes.

The main reason for templates with a relatively large number of holes is that the string passed to injection API is constructed in a loop that appends unknown values to the final string. The static analysis unrolls such loops a finite number of times, creating a relatively large number of unknown parts.

Third, we measure how many templates the analysis extracts per call site. Because different executed paths may cause different string values to be passed at a particular call site of an injection API, the analysis may yield multiple templates for a single call site. Figure 13b shows that for most call sites, a single template is extracted.

Reasons for imprecision: To better understand the reasons for imprecision of the static analysis, we measure how frequent particular kinds of nodes in template trees are. We find that 17.48% of all call sites have a template tree with at least one node that represent a function parameter. This result suggests that an inter-procedural static analysis might collect even more context than our current analysis. To check whether the static analysis may miss some sanitization-related operations, we measure how many of the nodes correspond to string operations that are not modelled by the analysis and to calls of functions whose name contains "escape", "quote", or "sanitize". We find that these nodes appear at only 3.03% of all call sites. The low prevalence of such nodes, reiterates the observation we made during our study: An npm module that uses sanitization when calling an injection API is the exception, rather than the rule.

Analysis running time: Our analysis successfully completes for 96.27% of the 15,604 modules without hitting the one-minute timeout after which we stop the analysis of a module. The average analysis time for these modules is 4.38 seconds, showing the ability of our approach to analyze real-world code at a very low cost.

We conclude from these results that the static analysis is effective for the large majority of call sites of injection APIs. Either the analysis successfully shows a call site to receive only statically known values, or it finds enough context to yield a meaningful security policy to be checked at runtime. This finding confirms our design decision to use a scalable, intra-procedural analysis. The main reason why this approach works well is because most strings passed to injection APIs are constructed locally and without any input-dependent path decisions.

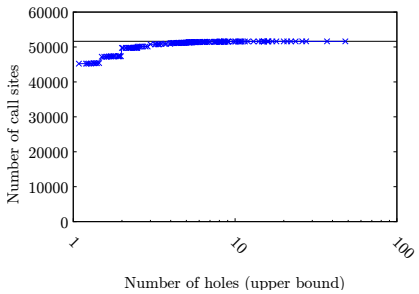
B. Runtime Mechanism

For evaluating the runtime mechanism we consider a set of 24 vulnerable modules listed in Figure 14. The set includes modules reported as vulnerable on the Node Security Platform, modules with vulnerabilities found during our study, and clients of known vulnerable modules. We exercise each module both with benign and with malicious inputs that propagate to the call sites of injection APIs.[12] As benign inputs, we derive example usages from the documentation of a module. As malicious inputs, we manually craft payloads that accomplish a specific goal. The goal for `eval` is to add a particular property to the globally available `console` object. For `exec`, the goal is to create a file in the file system. Figure 14 lists the modules and the type of injection vector we use for the malicious inputs. "(I)nterface" means that we call the module via one of its exported APIs, "(N)etwork" means that we pass data to the module via a network request, "(F)ile system" means that the module reads input data from a file, and "(C)ommand line" means that we pass data as a command line argument to the module. In total, we use 56 benign inputs and 65 malicious inputs.

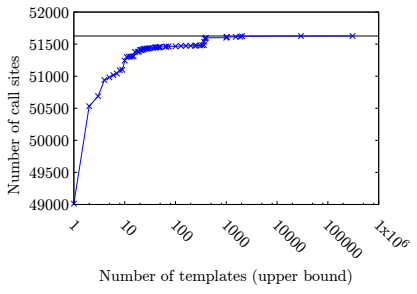
False positives: Across the 56 benign inputs, we observe five false positives, i.e., a false positive rate of only 8.92%. Three false positives are caused by limitations of our static analysis. For example, Figure 15 contains code that constructs a command passed to `exec` by transforming an array `keys` of strings using `Array.map`. Because our static analysis does not model `Array.map`, it assumes that the second part of `cmd` is unknown, leading to a PAST with a single unknown subtree. Our runtime policy allows filling this subtree with only a single argument, and therefore rejects benign values of `dmenuArgs` that contain two arguments. Further improvements of our static analysis, e.g., by modeling built-in functions, will reduce this kind of false positive.

The remaining two false positives are caused by the set of safe node types in our runtime mechanism. For example, the mol-proto module uses `eval` to let a user define arbitrary function bodies, which may include AST nodes beyond our set of safe node types. Arguably, such code should be

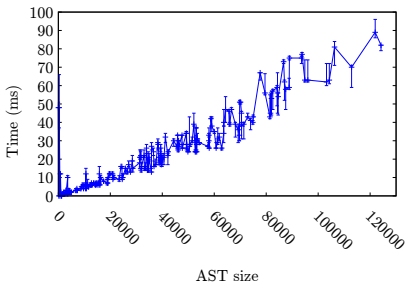
[12]The modules and inputs are available as benchmarks for future research: *URL removed for double-blind review.*



(a) CDF for the average number of holes per call site. Note the logarithmic horizontal axis.



(b) CDF for the number of inferred templates per call site. Note the logarithmic horizontal axis.



(c) Overhead of runtime checks depending on input size.

Fig. 13: Details on static analysis and overhead of runtime checks.

Type	Benchmark Module	Injection vector	Inputs benign	Inputs malicious	False negatives	False positives	Average overhead (ms)
Advisories (Sect. II)	gm	I	1	2	0	0	0.41
	libnotify	I	4	2	0	1	0.19
	codem-transcode	N	1	4	0	0	0.80
	printer	I	1	4	0	0	0.28
Reported by us (Sect. III-D)	mixin-pro	I	2	4	0	0	0.16
	modulify	I	1	2	0	1	0.04
	mol-proto	I	1	2	0	1	0.07
	mongoosify	I	1	2	0	0	0.04
	mobile-icon-resizer	FS	1	5	0	0	0.39
	m-log	I	11	1	0	0	0.05
	mongo-parse	I	1	2	0	0	0.11
	mongoosemask	I	1	1	0	0	0.04
	mongui	N	1	2	0	0	0.05
	mongo-edit	N	1	1	0	0	0.04
	mock2easy	N	1	2	0	0	0.03
Case study (Sect. III-E)	growl	I	1	2	0	0	2.72
	autolint	FS	4	4	0	0	1.59
	mqtt-growl	N	1	2	0	0	3.19
	chook-growl-reporter	I	1	1	0	0	1.60
	bungle	FS	14	4	0	0	1.99
Other exec (Sect. III-B)	fish	I	1	4	0	0	0.21
	git2json	I	1	4	0	1	0.37
	kerb_request	I	3	4	0	0	0.25
	keepass-dmenu	CL	1	4	0	1	0.52
	Total		56	65	0	5	
	Average						0.74

Fig. 14: Results for runtime enforcement. Used injection vectors: module's interface (I), network (N), file system (FS), command line (CL).

refactored for enhanced security. Alternatively, if a user of a module trust that module, she can whitelist either specific call site of the injection API or the entire module.

Overall, we conclude that the approach is effective at preventing injections while having a false positive rate

```
1 var keys = Object.keys(dmenuOpts);
2 var dmenuArgs = keys.map(function (flag) {
3   return '-' + flag + ' "' + dmenuOpts[flag] + '"';
4 }).join(' ');
5 var cmd = 'echo | dmenu -p "Password:" ' + dmenuArgs;
6 exec(cmd);
```

Fig. 15: Example of a false positive.

that is reasonably low, in particular for a fully-automated technique.

False negatives: SYNODE prevents all attempted injections during our evaluation, i.e., there are not false negatives. In general, however, there are multiple reasons that might cause false negatives. First, our static analysis fails to identify highly dynamic sink calls:

```
global["ev"+"al"](userInput);
```

Because the code to construct call targets can be arbitrarily complex, no static analysis can guarantee to detect all sink calls. As SYNODE targets code that is vulnerable by accident, and not malicious on purpose, we consider the problem of hidden sink calls to be negligible in practice. Second, SYNODE prevents the addition of new commands to the templates, but it does not defend against data only attacks. For example, sometimes it is insufficient to ensure that the input is a literal:

```
exec("rm " + userInput)
```

Computational cost: Our static analysis identifies a total of 1,560 templates for the injection APIs in the considered modules. For each of them, we construct a PAST with a median computation time of 2 milliseconds per module. We note that for some modules this number is significantly higher due to our simple PAST construction algorithm and due to the high number of templates per module.

The last column of Figure 14 shows the average runtime overhead per call of an injection API that is imposed by the runtime mechanism (in milliseconds). We report absolute times because the absolute overhead is more meaningful than normalizing it by a fairly arbitrary workload. Our enforcement mechanism costs 0.74 milliseconds per call, on

average over 100 runs of the modules using all the inputs. This result demonstrates that the overhead of enforcement is generally negligible in practice.

To demonstrate the scalability of our runtime enforcement, we consider input data of different size and complexity and pass it to the injection APIs. Here, we focus on `eval` call sites from Figure 14 only. As inputs, we use a diverse sample of 200 JavaScript programs taken from a corpus of real-world code[13]. For every call to `eval`, we pass all 200 JavaScript programs 100 times each and measure the variance in enforcement times. Figure 13c shows the enforcement time, in milliseconds, depending on the size of the JavaScript program, measured as the number of AST nodes. For each input size, the figure shows the 25% percentile, the median value, and the 75% percentile. We find that the enforcement time scales linearly. The reason is that all steps of the runtime enforcement, i.e., parsing the input, matching the AST with the PASTs, and checking whether nodes are on a whitelist, are of linear complexity.

IX. Related Work

Analysis of Node.js code: Injections into Node.js code are known to be exploitable [43] and there is a community-driven effort[14] to identify such problems. Ojamaa and Düüna [28] identify denial of service as one of the main threats for the Node.js platform and also mention `eval` as a security risk. We take these observation further by presenting an in-depth study of injection vulnerabilities on Node.js and presenting a technique to prevent injections. NodeSentry [7] is a security architecture for least-privilege integration of Node.js modules. Using their terminology, Synode's runtime enforcement is a lower-bound policy check on `exec` and `eval`. Our mechanism is more powerful since it uses a static analysis to perform fine-grained policy enforcement. Madsen et al. [22] enhance the call graph construction for Node.js applications with event-based dependencies. Our static analysis is intra-procedural but could benefit from integrating an inter-procedural approach, which may further reduce the false positive rate.

Staicu and Pradel show that many real-world web sites suffer from ReDoS vulnerabilities [38], a form of algorithmic complexity attack that may cause web sites to be unreachable. Their work underlines the importance of fixing vulnerabilities in popular Node.js modules, but addresses a different kind of vulnerability than this paper.

Program analysis for JavaScript: Studies of client-side JavaScript [49], [32] show that `eval` is prevalent but often unnecessary. We extend these findings to server-side JavaScript, add a new category of `eval` uses, and categorize uses of `exec`, an API not studied by existing work. Other studies focus on inclusions of third-party libraries [27], the `postMessage` API [36], injection attacks on JavaScript-based mobile applications [15], and the use of outdated libraries [17]. In contrast, we study Node.js code and address vulnerabilities specific to this platform.

[13] `http://learnbigcode.github.io/datasets/`

[14] `https://nodesecurity.io/advisories`

Blueprint [44] prevents XSS attacks by enforcing that the client-side DOM resembles a parse tree learned at the server-side. Their work shares the idea of comparing data prone to injections to a tree-based template. Our work differs by learning templates statically and by focusing on command injections in Node.js code. Stock et al. [40] study DOM-based XSS injections and propose dynamic taint tracking to prevent them. Similar to Synode, their prevention policy is grammar-based. However, their strict policy to reject any tainted data that influences JavaScript code except for literals and JSON would break many uses of `eval` found during our study. Another difference is that we avoid taint tracking by statically computing sink-specific templates.

Defenses against XSS attacks [37], [25] use signature-based whitelisting to reject scripts not originating from the website creator. SICILIAN [37] uses an AST-based signature; nsign [25] creates signatures from script-dependent elements and context-based information. Both rely on a training phase to discover valid signatures. Our work also uses templates as a white-listing mechanism. However, we do not rely on testing to collect these templates but compute them statically. As we show in our evaluation, there may be hundreds or even thousands of paths that reach an injection API call site, i.e., constructing valid signatures for every path is infeasible.

CSPAutoGen [29] presents an automatic way to generate CSP policies on the server-side in order to protect against illegitimate script execution on the client-side. It uses gASTs, partial ASTs similar in structure with ours, but different in many ways. First of all, gASTs are created during a training session, which limits the approach to behavior observed during the training phase. gASTs also differ from our partial ASTs in the way they are enforced: gASTs are synthesized into a JavaScript function that replaces the actual call to the sink. Such a step cannot be easily implemented for sinks other than `eval` since these sinks call outside the JavaScript world. For example, to refactor a call to `exec` that uses the `awk` system utility on Linux, we would need to completely rewrite `awk` in JavaScript.

DLint [10] is a dynamic checker for violations of code quality rules, including uses of `eval` missed by static analysis. Dynamic [12] and hybrid (static and dynamic) [6] information flow analyses for JavaScript track how values at some program location influence values at another program location. The FLAX system [34] and a system proposed by Lekies et al. [18] use dynamic taint analysis to detect vulnerabilities caused by missing input validation and then generate inputs that exploit these vulnerabilities. Jin et al. [15] use a static taint analysis to detect code injection vulnerabilities. In contrast to these approaches, we do not require an information flow (or taint) analysis, performing lightweight runtime checks at possible injection locations, without tracking the values that flow to these locations.

Several approaches rewrite JavaScript code to enforce security policies. Yu et al. [48] propose a rewriting technique based on edit automata that replaces or modifies particular calls. Gatekeeper [11] is a static analysis for

a JavaScript subset that enforces security and reliability policies. Instead of conservatively preventing all possibly insecure behavior, our approach defers checks to runtime when hitting limitations of purely static analysis. Other techniques [14], [24] replace `eval` calls with simpler, faster, and safer alternatives. Their main goal is to enable more precise static analysis; our focus is on preventing injections at runtime.

Program analysis for other languages: CSAS [33] uses a type system to insert runtime checks that prevent injections into template-based code generators. Livshits et al. [20] propose to automatically place sanitizers into .NET server applications. Similar to our work, these approaches at first statically address some code locations and use runtime mechanisms only for the remaining ones. CSAS differs from our work by checking code generators instead of final code. The approach in [20] addresses the problem of placing generic sanitizers, whereas we insert runtime checks specific to injection call sites.

There are several purely dynamic approaches to prevent injections. XSS-Guard [4] modifies server applications to compute a shadow response along each actual response and compares both responses to detect unexpected, injected content. Instead of comparing two strings with each other, our runtime mechanism compares runtime strings against statically extracted templates. ScriptGard [35] learns during a training phase which sanitizers to use for particular program paths and detects incorrect sanitization by comparing executions against the behavior seen during training. Their approach is limited by the executions observed during training and needs to check all execution paths, whereas Synode statically identifies some locations as safe.

Su and Wassermann [42] formalize the problem of command injection attacks and propose grammar-based runtime prevention. Their work shares the idea to reject runtime values based on a grammar that defines which parts of a string may be influenced by attacker-controlled values. Their analysis tracks input data with special marker characters, which may get lost on string operations, such as `substring`, leading to missed injections. Instead, our analysis does not need to track input values through the program. Buehrer et al. [5] take a similar approach to mitigate SQL injections. They construct two parse trees at runtime, one representing the developers intentions only and one including the user input. They use these trees to ensure that the user input contains only literals. Their approach is purely dynamic and employs markers for tracking user input, similar to Su and Wassermann [42]. Ray and Ligatti [30] propose a novel formulation of command injections that requires dynamic taint tracking and a set of trusted inputs. For Node.js libraries, example inputs are rarely available.

Constraint-based static string analysis, e.g., Z3-str [50] is a more heavy-weighted alternative to our static analysis. Even though such techniques have the potential of producing more precise templates, we opted for efficiency, enabling us to apply the analysis easily to thousands of npm modules. Wassermann et al. address the problem of finding inputs that trigger SQL injections [46] and XSS vulnerabilities [45] in PHP code. Ardilla [16] finds and exploits injection vulnerabilities in PHP through a combination of taint analysis and test generation. Instead of triggering attacks, our work addresses the problem of preventing attacks. Similar to our preliminary study of dependences on injection APIs, a recent work analyzes the usage of the unsafe API in Java [23]. Existing analyses for Java [31] and Android applications [3], [19] focus on security risks due to libraries, which shares with Synode the idea to consider third-party code as a potential security threat.

X. Conclusions

This paper studies injection vulnerabilities in Node.js and shows that the problem is widespread and not yet adequately addressed. We present Synode, an automated technique for mitigating injection vulnerabilities in Node.js applications. At the same time, the approach effectively prevents a range of attacks while causing very few false positives and while imposing sub-millisecond overheads. To aid with its adoption, our technique requires virtually no involvement on the part of the developer. Instead, Synode can be deployed automatically as part of module installation.

In a broader scope, this work shows the urgent need for security tools targeted at Node.js. The technique presented in this paper is an important first step toward securing the increasingly important class of Node.js applications, and we hope it will inspire future work in this space.

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the Emmy Noether project "ConcSys", and by the Royal Society Wolfson Merit Award.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Computing Surveys*, 2017.

[3] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in android and its security applications. In *CCS*, pages 356–367, 2016.

[4] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *DIMVA*, pages 23–43, 2008.

[5] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Workshop on Software Eng. and Middleware*, pages 106–113, 2005.

[6] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *PLDI*, pages 50–62. ACM, 2009.

[7] W. De Groef, F. Massacci, and F. Piessens. NodeSentry: least-privilege library integration for server-side JavaScript. In *ACSAC*, pages 446–455, 2014.

[8] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. Fear the EAR: discovering and mitigating execution after redirect vulnerabilities. In *CCS*, pages 251–262, 2011.

[9] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *IMC*, pages 475–488, 2014.

[10] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *ISSTA*, pages 94–105, 2015.

[11] S. Guarnieri and B. Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security*, pages 151–168, 2009.

[12] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC*, pages 1663–1671, 2014.

[13] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security*, pages 1–16, Aug. 2011.

[14] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *ISSTA*, pages 34–44, 2012.

[15] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In *Conference on Computer and Communications Security*, pages 66–77, 2014.

[16] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE*, pages 199–209, 2009.

[17] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. 2017.

[18] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*, pages 1193–1204, 2013.

[19] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. Libd: Scalable and precise third-party library detection in android markets. In *ICSE*, 2017.

[20] B. Livshits and S. Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *POPL*, pages 385–398, 2013.

[21] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015.

[22] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven Node.js JavaScript applications. In *OOPSLA*, pages 505–519, 2015.

[23] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: the Java unsafe API in the wild. In *OOPSLA*, pages 695–710, 2015.

[24] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: semi-automated removal of eval from JavaScript programs. In *OOPSLA*, pages 607–620, 2012.

[25] D. Mitropoulos, K. Stroggylos, D. Spinellis, and A. D. Keromytis. How to train your browser: Preventing XSS attacks using contextual script fingerprints. *Trans. Priv. and Sec.*, 19(1):2, 2016.

[26] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, second edition edition, 2005.

[27] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *ACM Conference on Computer and Communications Security*, pages 736–747, 2012.

[28] A. Ojamaa and K. Düüna. Assessing the security of Node.js platform. In *Intl. Conf. f. Internet Techn. and Secured Transactions*, pages 348–355, 2012.

[29] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. CSPAutoGen: Black-box enforcement of content security policy upon real-world websites. In *CCS*, pages 653–665, 2016.

[30] D. Ray and J. Ligatti. Defining code-injection attacks. In *POPL*, pages 179–190, 2012.

[31] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini. Call graph construction for java libraries. In *FSE*, pages 474–486, 2016.

[32] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - A large-scale study of the use of eval in JavaScript applications. In *ECOOP*, pages 52–78, 2011.

[33] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *CCS*, pages 587–600, 2011.

[34] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[35] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In *CCS*, pages 601–614, 2011.

[36] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postmessage in HTML5 websites. In *NDSS*, 2013.

[37] P. Soni, E. Budianto, and P. Saxena. The SICILIAN defense: Signature-based whitelisting of web JavaScript. In *CCS*, pages 1542–1557, 2015.

[38] C.-A. Staicu and M. Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. Technical Report TUD-CS-2017-0305, TU Darmstadt, 2017.

[39] C.-A. Staicu, M. Pradel, and B. Livshits. Understanding and automatically preventing injection attacks on node.js. Technical Report TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science, 2016.

[40] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise client-side protection against DOM-based cross-site scripting. In *USENIX Security*, pages 655–670, 2014.

[41] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In *USENIX Security*, pages 1015–1032, 2016.

[42] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL*, pages 372–382, 2006.

[43] B. Sullivan. Server-side JavaScript injection. *Black Hat USA*, 2011.

[44] M. Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Sec. and Privacy*, pages 331–346, 2009.

[45] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171–180, 2008.

[46] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA*, pages 249–260, 2008.

[47] S. Wei. Practical analysis of the dynamic characteristics of JavaScript, 2015.

[48] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *POPL*, pages 237–249, 2007.

[49] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *WWW*, pages 961–970, 2009.

[50] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a Z3-based string solver for web application analysis. In *ESEC/FSE*, pages 114–124, 2013.