Fixing Vulnerabilities Automatically with Linters

Willard Rafnsson, Rosario Giustolisi, Mark Kragerup, and Mathias Høyrup

IT University of Copenhagen

Abstract. Static analysis is a tried-and-tested approach to eliminate vulnerabilities in software. However, despite decades of successful use by experts, mainstream programmers often deem static analysis too costly to use. Mainstream programmers do routinely use linters, which are static analysis tools geared towards identifying simple bugs and stylistic issues in software. Can linters serve as a medium for delivering vulnerability detection to mainstream programmers?
We investigate the extent of which linters can be leveraged to help programmers write secure software. We present new rules for ESLint that detect—and automatically fix—certain classes of cross-site scripting, SQL injection, and misconfiguration vulnerabilities in JavaScript. Evaluating our experience, we find that there is enormous potential in using linters to eliminate vulnerabilities in software, due to the relative ease with which linter rules can be implemented and shared to the community. We identify several open challenges, including third-party library dependencies and linter configuration, and propose ways to address them.

1 Introduction

Motivation. JavaScript is the most commonly used programming language today [36]. Popularized by the Web, JavaScript is now ubiquitous, used for implementing all kinds of software, including Web apps and services, desktop apps, mobile apps, and even embedded software. Its appeal is that it is dynamically and weakly typed, and that it is, at its core, quite simple. However, this is a double-edged sword; the type system and full semantics of JavaScript are infamous for their quirks [18] that frequently befuddle programmers, even experienced ones. As a result, bugs routinely make their way to deployed JavaScript code. These can lead to vulnerabilities that get exploited in privacy attacks, such as injection attacks, serialization attacks, and cross-site scripting attacks [30].

Static analysis is a tried-and-tested approach to eliminate bugs prior to deployment [13]. It involves reasoning about the behavior of a program without executing it, to see if it possesses an undesired property, e.g. "has certain bugs". Since a static analysis tool cannot be both free of false-positives and false-negatives (unless the property is trivial [32]), correctness proofs for tools typically focus on no-false-negatives, so e.g. "code deemed bug-free really is bug-free". Static analysis is successfully used by specialists at large companies, in situations where bugs are deemed too costly, e.g. SLAM at Microsoft [7], Infer at Facebook [10], Spark Ada at Boeing [15], and PolySpace at NASA [9]. Tools for enforcing security policies in JavaScript code, e.g. access control [28], information-flow [17], and

others [16], have long existed. However, despite the use of such tools being long advocated [11], mainstream programmers seem reluctant to use static analysis tools in general [22], due to the poor manner in which they present issues, and the overwhelming number of false positives they generate [8,43,33].

Enter linters. A linter is a static analysis tool that scans (i.e. lints) code for a wide range of issues, including bugs, programming errors, suspicious constructs, stylistic errors, and code smells [40]. Linters have been used extensively since the 70s for all major programming languages, famous examples being Lint [24], FindBugs [6] and ESLint [46] for C, Java, and JavaScript respectively. Strikingly, mainstream linters have no correctness proofs. Rather, the focus of linters is on providing usable improvement suggestions and on reducing false positives [6,40]. This is indeed their appeal; for JavaScript, an overwhelming majority of developers use linters, and they do so to catch errors and produce maintainable code by reducing complexity and enforcing style [40]. Linters are even used together with unit tests as a second line of defense against bugs [40]. Finally, modern linters are fully configurable; linter rules can be toggled for different parts of the code base, and new linter rules can be developed in-house or downloaded from a community-maintained repository [46]. As new bugs, guidelines, and libraries arise, linters can immediately be configured to adapt.

With the lack of mainstream success of advanced static analysis tools for security, and with the popularity of linters, perhaps the most impactful way to deliver vulnerability checking to programmers is through a linter.

Contribution. We investigate the extent of which linters can be leveraged to help programmers write secure code. We focus on ESLint, as it is the most commonly used linter for the most commonly used programming language.

First, we present new rules for ESLint for eliminating vulnerabilities. These rules detect—and automatically *fix*—certain classes of cross-site scripting, SQL injection, and misconfiguration vulnerabilities, all of which are on the OWASP Top Ten list of Web application security risks [30]. Some of these rules are library-dependent, checking for vulnerabilities in the use of React [21] and Express [37], the two JavaScript frameworks that make up the popular MERN stack [26]. We demonstrate the practicality of these rules, notably that they have few false-positives, with examples and unit tests.

Next, supported by our experience, we critically evaluate the state-of-the-art in the use of linters to eliminate vulnerabilities. We evaluate the strengths and limitations of ESLint for eliminating vulnerabilities and find that, while ESLint provides useful facilities for analyzing code, there are shortcomings, notably that it only scans one file at a time. This has consequences when considering library dependencies. We survey existing ESLint plugins, and find only a few rules that check for vulnerabilities. The handful of those that see notable use identify only a few vulnerabilities, and focus on reducing false *negatives*, to the point that (we argue) the rule becomes too much of a hassle to use, and thus gets disabled by programmers. We analyze the ESLint guidelines for creating new rules [39], and find that most of the guidelines run counter to the needs of a rule that eliminates vulnerabilities. We also find that configuring ESLint is nontrivial, which is a

well-known barrier to using linters effectively [40], and propose outsourcing the maintenance of linter configuration to community experts.

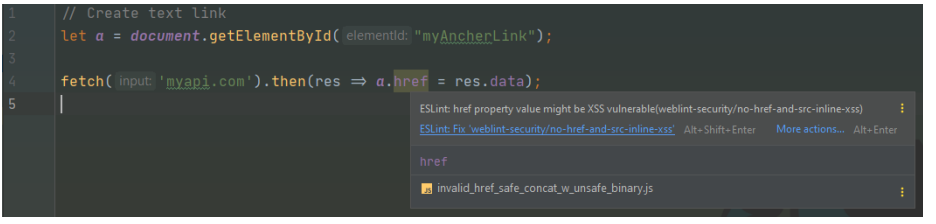
Finally, we make observations as well as recommendations with regards to using linters to eliminate vulnerabilities. We find that there is a great, and unexplored, potential in using linters to eliminate vulnerabilities. This is based on the popularity of ESLint, the rich features that ESLint provides for analyzing code, the scarcity and inadequacy of existing rules for security, and the relative ease with which we successfully implemented rules that are more practical. We find it crucial that linters are fully configurable. Community-maintained linter rules made ESLint successful, and are a necessary feature if programmers are to keep up with the rapidly-evolving security landscape. For the same reason, it is important that linter configuration can be outsourced to experts. For rules to be useful, we find it important that it be clear what kind of vulnerabilities the rule intends to detect. The rule should also detect vulnerabilities as precisely as possible, i.e. few false alarms. Together, this saves time, instills confidence, and reduces frustration, in the programmer. Finally, linters must provide more comprehensive means of scanning source code. Crucially, linters must analyze dependencies on third-party libraries. This is because 93% of the code of a modern Web application is open-source library code [42], 70.5% of Web apps have vulnerabilities from a library, and the vast majority of those vulnerabilities are known or have been patched [41]. A linter could detect these vulnerabilities.

Outline. We describe what we advocate in Section 2. We summarize ESLint in Section 3. We present our linter rules that find and eliminate vulnerabilities in Section 4. We evaluate the state-of-the-art against our experience in Section 5, and provide our recommendations in Section 6. Finally, we contrast our findings with related work in Section 7, and conclude in Section 8.

2 In A Nutshell

Mainstream programmers routinely use linters to eliminate bugs and improve the quality of their code. This is illustrated in Figure 1. This JavaScript code sends a request to `myapi.com` and assigns the data in the response to the `href` on the page. The programmer has integrated ESLint, a linter for JavaScript, into her integrated development environment (IDE). ESLint reports, in Figure 1a, that this code may be vulnerable to a cross-site scripting attack, since the data received from `myapi.com`, and thus the URL in the `href`, could be controlled by an attacker (e.g. if the API reads from a database that the attacker can inject code into). This way, the attacker injects JavaScript into the response, which would get executed on the client upon pressing the URL. More importantly, ESLint also presents the option to *automatically fix* the vulnerability. Picking this option yields the code in Figure 1b, which fixes the vulnerability by sanitizing the response from `myapi.com`.

This is `no-href-and-src-inline-xss`, one of the ESLint rules that we propose in this paper, at work. It is this kind of automatic finding and fixing of vulnerabilities that we desire and advocate. Automatic fixing of vulnerabilities



(a) finding a vulnerability

(b) fixing a vulnerability

Fig. 1: Using a linter in an IDE to find & fix a vulnerability.

is incredibly valuable to mainstream programmers, as some vulnerabilities are notoriously difficult to debug, requiring expertise and deep knowledge about the whole code base. Our goal in this paper is to investigate the extent of which linters can deliver on this. We do this in the context of JavaScript, and its most popular linter, ESLint.

3 ESLint

ESLint [46] is a linter for JavaScript. Initially released in 2013, ESLint has since become a *de facto* standard tool for JavaScript development, used by the vast majority of JavaScript developers [40]. ESLint is actively developed, supporting both current and upcoming standards of ECMAScript.

A description of how ESLint and its rules work can be found in the appendix. We now briefly list the defining features of ESLint.

3.1 Features

ESLint has several features that distinguish it from other linters.

Automatic. ESLint can fix problems automatically, thus freeing the developer from coming up with a fix.

Customization. ESLint rules can be turned on and off, new rules can be added by downloading plugins from a community-maintained repository, and developers can maintain their own custom rules in-house.

Integration. ESLint builds into mainstream text editors and IDEs and can be run as part of a continuous integration and deployment (CI/CD) pipeline.

ESLint is a community-effort; it is open-source (thus freely available), and actively scrutinized by its community. The ESLint developers emphasize the importance of clear documentation and communication; rules should be well documented, provide useful improvement suggestions, and have few false-positives.

4 ESLint Rules for Fixing Vulnerabilities

We implement four new ESLint rules for finding and fixing vulnerabilities. Our rules target well-defined instances of today's most critical Web security risks from the OWASP Top Ten [30]: cross-site scripting, server misconfiguration, and SQL injection. Three of our rules not only detect, but *automatically fix*, the vulnerabilities. We demonstrate that the rules have few false positives with unit tests. Together, this shows that linters can help programmers write secure code.

For each rule, we describe the vulnerability it targets, how existing ESLint rules fall short, our rule implementation, and automatic code fixing.

Testing. Our rules sections have multiple unit test files corresponding to the different use cases described for each rule. The tests utilize the built-in ESLint rule tester to validate code cases for the absence of false alarms, as well as for validating the automatic fixer functionality when applicable.

4.1 Cross-Site Scripting

Vulnerability. Cross-site scripting (XSS) is a code injection vulnerability where a victim, while navigating a benign (yet vulnerable) Web page, unwittingly executes attacker-controlled data in the browser. This lets an attacker obtain private information (e.g. access cookies and session tokens) from the victim's browser, redirect the victim to a malicious website, and more, all while the victims believe they are interacting with the original benign Web page. Our rule targets specific kinds of DOM XSS vulnerabilities, i.e. where a page calls a JavaScript API and uses the response to modify a DOM element in the victim's browser; an attacker can exploit a vulnerable API to ship code to the victim's browser. Concretely, we focus on modifications of `src` attributes, and of the `href` attribute of anchor tags. An example of such a vulnerability is the following.

```
1 fetch("myapi.com").then(res => a.href = res.data);
```

This is the scenario presented in Section 2. The result of the API call may contain unsanitized attacker-controlled data. Such data might contain a malicious script that will be executed in the victim's browser upon link activation.

Existing Rules. The most notable plugin which includes rules for detecting XSS is `eslint-plugin-no-unsanitized`. It includes rules that detect unsanitized data used for manipulating the HTML content of DOM elements. However, the plugin considers only the HTML content of elements, and is not capable of detecting vulnerabilities that occur when manipulating `href` and `src` attributes. Furthermore, the rules result in a lot of false positives; as existing user studies have pointed out [8,43,33], this would discourage programmers from activating the rule. Upon investigation, we found that the main issue is that the rule only considers explicit strings in the DOM assignments as safe, and raises a flag in any other case. We observe that there are, in fact, many valid and safe cases of DOM assignments, which could be ruled out as a possible vulnerability. One example of this would be the use of a variable holding an explicit string in the

assignment. Additionally, the rules do not support web development frameworks that render HTML content dynamically, such as the popular framework React.

Our Approach. Our approach targets XSS attacks through modification of `href` and `src` attributes. This has not been addressed by ESLint plugins to date.

To achieve this, we maintain a set of variable identifiers that are in a safe state at any point while traversing the AST. We determine whether a variable is in a safe state by tracking and evaluating the values of all variable initializations and assignments in the code. We consider explicit strings as well as constructs that implicitly form string expressions from other explicit strings safe. This reduces false positives without affecting correctness. The benefit of this is clear when using template strings and string concatenations that involve variables that contain e.g. explicit strings.

When our `no-href-and-src-inline-xss` rule is run on the example above, it outputs the message *"href property value might be XSS vulnerable"*, and highlights the `".href ="` part of the code. Our rule documentation, accessible within ESLint, provides further information about the vulnerability to the programmer.

This approach generalizes to the use of libraries and frameworks. Our rule `no-href-and-src-inline-xss-react` shows this; it detects the same vulnerability in the use of Facebook's React [21] framework. Here, in addition to tracking the safety of variables, we need to track the safety of values held in special React states. React uses JSX (see React documentation) to dynamically render content and automatically modify the DOM in the user's browser. This syntax also applies to the `src` and `href` attributes, and the rule therefore considers this feature specifically. While the previous rule is part of the `"recommended"` rule set of the plugin, this rule is part of the `"react"` rule set. The two rules are specifically implemented in such a way that they do not conflict or overlap. The rule set of an ESLint configuration is normally extended first by recommended rules, and then by any additional rules, such as React-specific rules for full framework support.

Automatic fix. We take advantage of the *fixer* functionality of the ESLint interface to enable the programmers to *automatically fix* the vulnerability identified by these rules. When selected by the programmer, the rule applies our suggested code change to the file directly.

When a value might contain malicious code, it is recommended to sanitize the value before using it. Unfortunately, while some libraries, such as Angular and React, sanitize some strings behind-the-scenes, their sanitization is not complete, and only focuses on `<script>` tags. Furthermore, SQL libraries have sanitization functions to protect against SQL injection attacks, which do not trivially port to our scenario. We propose a novel way of escaping executable code, using built-in Javascript string functions to sanitize the `javascript:` prefix.

```
1 val.toLowerCase().replace('javascript:','/javascript/:/'))
```

Applying the automatic fixer to the above example results in a replacement of the right-hand side of the assignment of the `href` attribute with escaping applied.

```
1 fetch("myapi.com").then( res => a.href =
      (res.data).toLowerCase().replace('javascript:','/javascript/:/'));
```

Limitations. Our rules do have some false positives. Our rules flags any assignment of any function application regardless of its safety, due to the concern that the function calls an API which retrieves attacker-controlled data. For example, `"my".concat("string");` is safe, yet is flagged as unsafe. Furthermore, we have not tested the sanitization function in our fixer; it could potentially be circumvented through the use of filter evasion. There may also be tried-and-tested sanitizers that we can, and should use, such as the `xss-filters` package in `npm`.

4.2 Security Misconfiguration

Vulnerability. Security Misconfiguration is a broad security problem with equally broad ramifications. These can arise e.g. from default configurations, default credentials, unnecessary features being enabled (e.g. ports) or installed, and unpatched flaws in the server software/hardware stack. Our rule targets misconfiguration of HTTP response headers in Node.js backend applications, specifically ones built on the Express [37] Web application framework. The aim of the rule is to eliminate certain clickjacking, MIME-sniffing, and XSS attacks, by recommending the use of the Helmet [19] library for Express, which automatically configures some important HTTP headers in a safe manner. An example is

```
1 const myApp = require('express');
2 myApp.listen(8080);
```

This minimal Express application launches a Web server that accepts network traffic on port 8080. However, HTTP response headers have not been configured.

Existing Rules. The `eslint-plugin-security-node` plugin has a rule called `detect-helmet-without-nocache`. The rule flags code that uses Helmet for configuring HTTP response headers without the `noCache` setting enabled. However, we observe that there are much more impactful security settings available in Helmet; Helmet provides a set of default configurations for security [19] which can be enabled by invoking the Helmet object. No available ESLint rule encourages the use of these defaults. Furthermore, the existing rule encouraging the use of `noCache` is very limited in the cases that it covers. For example, it only considers programs where the variable holding the Express object is named `app`; if any other name is used, the rule would flag the concerned code, regardless of whether the code is vulnerable or not. Finally, `noCache` is deprecated. To the best of our knowledge, there are no other ESLint plugins that detect HTTPS response header vulnerabilities.

Our Approach. Our approach targets the use of Express without Helmet. It keeps track of the correct usage of Helmet with the recommended defaults and, if the recommended defaults are not correctly enabled, flags the line of code that launches the application.

To achieve this, we track whether Express and Helmet are enabled, and track identifier names holding the corresponding object instances. If the application uses Express but does not import Helmet at all, we flag the code as unsafe. If Helmet is imported, but the recommended defaults are not invoked, we also flag

the code as unsafe. Since we cover all cases, our rule has few false alarms. Our rule is more general than the previous ones since it considers all uses of Express, and not just uses of Express, in an object named `app`, without `noCache`.

When our `detect-missing-helmet` rule is run on the example above, it outputs the message *"Use the Helmet.js module for enhanced security on HTTP response headers in your Express application."* with a link to the setup documentation. Our rule also suggests the use of the `expectCT` Helmet setting for information purposes (without enforcing it), which can help prevent certificate abuse.

Automatic fix. Again, we empower the programmer to fix this vulnerability, by implementing an ESLint fixer.

We utilize the stored identifier names to provide correct adaptations of the vulnerable code, which involves simply inserting lines that import Helmet and invoke it with its defaults. Applying our fixer on the above example thus yields the following modified code.

```
const myApp = require('express');
const helmet = require('helmet');
myApp.use(helmet());
myApp.listen(8080);
```

4.3 SQL Injection

Vulnerability. SQL Injection is a code injection vulnerability where an attacker can, by carefully crafting input data to the Web application frontend, inject his own SQL queries into the database in the backend. This can enable the attacker to bypass login authentication checks, read or modify database records, execute administrative operations on the database, and, in some cases, even issue commands to the underlying operating system. Our rule targets the occurrence of variables in the construction of SQL query strings in Node.js backend applications. An example of this follows.

```
let phone = readline.question("Your phone number?\n");
const sql = 'SELECT * FROM users WHERE tlf = ' + phone;
dbConnection.query(sql, (err, result) => console.log(result));
```

This minimal command-line Node.js application queries a database of users for a record matching a phone number. However, the phone number is supplied as a variable; an attacker can obtain the whole table by providing the phone number `"' OR 1=1"` as input, or delete the `users` table by providing the phone number `"'; DROP TABLE users"`.

Existing Rules. The previously mentioned plugin `eslint-plugin-security-node` has a rule for detecting SQL injection vulnerabilities: `detect-sql-injection`. This rule flags all queries that provide anything other than an explicit string as its first parameter. This leads to an explosion in the number of false positives. For example, the rule flags as vulnerable when an explicit string is stored in a variable, or when two explicit strings are concatenated into the parameter. Furthermore, the most common SQL queries for Web application depend in some

way on user input, such as queries for login credentials. Alarmingly, the rule only considers cases where the database connection is stored in a variable named either `connection`, `connect`, or `conn`. Using a different name causes the rule to disregard the SQL queries, leading to potential false negatives. This limitation is not present in the plugin documentation.

Another plugin is `eslint-plugin-sql-injection`, which just includes a single rule for detecting SQL injections. The approach here is to check if a query call is using string concatenations where at least one value, in the concatenation, is not an explicit string. While this does reduce false positives (compared to the previous rule), it does dramatically increase false negatives. For instance, this rule does not flag a variable if it is not used in a concatenation or a template string. The rule also requires the programmer to manually specify the name of the function which queries the database in the ESLint configuration file. If the user does not provide such a configuration, the rule does not flag any vulnerabilities, leading to false negatives.

Our Approach. Our approach targets the occurrence of variables in queries, requiring instead the use of a prepared statement. A prepared statement fixes the structure of the query before values are inserted into the query, thus preventing such values from modifying the structure of the query.

To achieve this, we follow a similar approach as in our rules for XSS vulnerabilities. We maintain a set of safe variables throughout the analysis process. We flag only code that uses unsafe variables or unsafe values in the SQL query execution. This reduces false positives, as we do not impose restrictions on variable names. Furthermore, our approach considers any function call named `query` which takes parameters. As a result, our approach is not tailored to any specific database driver, which means that it can detect vulnerabilities in queries for e.g. MySQL, PostgreSQL, etc., since they all export a function named `query`.

When our `detect-sql-injection` rule is run on the example above, it outputs the message *"Parameterize the input for the query, to avoid SQL Injection vulnerabilities. See more at: https://www.npmjs.com/package/mysql#escaping-query-values"*. The warning highlights the query call for visual guidance in IDEs.

Fix. To fix the vulnerability, the programmer can, by looking at the information presented to him by the rule, turn the query into a prepared statement. For our example, the following fixed version is accepted by our rule.

```
let phone = readline.question("Your phone number?\n");
const sql = 'SELECT * FROM users WHERE tlf = ?';
dbConnection.query(sql, [phone], (err, result) => console.log(result));
```

Neither of the existing rules would flag neither the original nor fixed version of this example, thus implying false negatives. However, if `dbConnection` is renamed to a name that the rules recognize, then both rules would reject both examples, thus implying false positives.

For this rule, we do not include the automatic code fixing implementation as in the previously proposed rules. This is a highly desirable feature; we will experiment with this in the future (hopefully by the camera-ready deadline). Implementing this is challenging because this requires changing the code on a

previous line (`const sql`) while analyzing the node corresponding to the later line that performs the actual query. This also requires manipulating query strings and constructing arrays of parameters in a way which supports all possible ways of constructing such a string.

Limitations. Our rule does have some false positives, notably since it flags calls of functions named `query`. While this catches queries for all common SQL APIs, it at the same time catches same-named functions on any object.

5 Analysis

Supported by our experience with creating linter rules, we critically evaluate the state-of-the-art in the use of linters to eliminate vulnerabilities in software. We analyze ESLint as a tool, existing plugins for ESLint, existing guidelines for writing rules, and the challenge of maintaining a linter configuration. We make four important observations on the prospect of using linter rules for security.

5.1 ESLint Strengths and Limitations

Based on our experience, we evaluate the strengths and limitations of using ESLint to find and fix vulnerabilities.

Strengths. ESLint has several strengths which make it well-suited for helping programmers find and fix vulnerabilities.

First, and most important, is its *rules customization*. ESLint comes shipped with a host of rules, which can be turned on and off for different parts of the code base. More importantly, programmers can create their own custom rules in-house. ESLint provides a rich API for this; rules can maintain their own state, traverse the AST freely when invoked, and precisely assign blame to sections of the code that are at fault. Crucially, ESLint provides an API for automatically fixing a vulnerability. This is immensely beneficial to programmers, as debugging a vulnerability can be a complex, arduous, and time-consuming task. New rules can be shared with the community on `npm`, which already contains at least tens of thousands of linting rules. As soon as a new vulnerability and fix are discovered, they can be shared with everyone.

Second, ESLint provides advanced *linter customization* options. ESLint enables a programmer to use, or write, a custom parser, to provide additional capabilities to linter rules. This would prove useful for writing more advanced linter rules or for linting syntactic extensions to ESLint. In fact, this is done to enable linting of TypeScript [20]. Furthermore, a programmer can use, or write, a custom "processor", which pre-processes a non-JavaScript file before parsing it. This is done to enable linting of non-JavaScript files that contain JavaScript, e.g. HTML source files. In addition, a programmer can use, or write, a custom formatter, to change how ESLint displays linter results.

Third, and last, is *integration*. ESLint builds into mainstream text editors and IDEs, as we have seen in Section 2, making it easy to adopt by mainstream developers. Furthermore, as is clear from its customization options and since

it can be run from the command-line, ESLint can be run as part of a CD/CI pipeline, as part of the building, testing, or deployment process.

Limitations. ESLint is not without its limitations, however.

First, like most linters, ESLint has *no correctness proof*. Not only need the programmer trust the claim of the maintainers of ESLint that ESLint does what it claims to do; the programmer also needs to trust the rule creators that their rules do what they specify. This is mitigated somewhat by the fact that ESLint, and the rules published on npm are publicly available, meaning that anyone can scrutinize them for correctness. However, the reason we have correctness proofs is that no amount of testing can guarantee the absence of bugs, and some bugs in open-source software are so subtle that they pass human scrutiny for decades [23]. Until we have formally-verified linters, this is the best we have got.

Second, and last, ESLint only scans *one file at a time*. This poses two problems for using ESLint for security. One is that ESLint will not scan dependencies. This is an issue since 93% of the code of modern Web apps is open-source library code [42], and 70.5% of Web apps have vulnerabilities from a library. This can be mitigated by making overapproximations on calls to libraries, although this would produce false positives. The other, more serious, problem is compositional reasoning. Even if all modules that make a modern Web app are scanned separately and found to be free of vulnerabilities, the way in which these modules interact may introduce vulnerabilities [29,45]. This can be mitigated by producing, from all JavaScript files that constitute a given Web app, one large JavaScript file, and then running ESLint on it. However, if ESLint finds a vulnerability in that composite file, assigning blame to the original source files would be difficult.

Evaluation. As demonstrated in Section 4, despite these limitations, we were quite successful in implementing practical rules that help programmers detect and eliminate vulnerabilities in software. Given how easy it is to share rules with the community, the potential impact of doing this is high. In fact, after being on npm for 1 week, our plugin has over 1.000 weekly downloads. This is without any promotion of the plugin. As awareness of this plugin increases, we imagine that its popularity will increase.

Observation 1 *Linters can (and should) be used to detect and eliminate vulnerabilities in software.*

Observation 2 *The potential impact of creating and sharing linter rules for security is high.*

5.2 ESLint Security Plugins

Given the above observations, the popularity of linters, our success with creating rules for vulnerabilities, and the prevalence of vulnerabilities in software, it stands to reason that ESLint would have rules for finding and fixing vulnerabilities in quantity and quality. Surprisingly, we find that this is not the case.

We surveyed the security relevance of more than 250 ESLint plugins in npm, in descending order of popularity, by briefly looking at their descriptions and

then investigating the implementations of those that contain security-related rules. We also specifically investigated a handful of less popular security-specific plugins. We summarize our findings for the most relevant plugins in the following.

eslint-plugin-security. (135.000 weekly downloads). This plugin informs programmers of a wide range of vulnerabilities in Node.js applications. It is the most popular ESLint security plugin. The popularity of the plugin is consistent over time. However, the maintenance of the code is not, with more than three years since the last update. Furthermore, as stated on the `npm` page of the plugin, the rules have a lot of false positives. Finally, the rules target narrow situations, none of which fall under the vulnerabilities that we target with our rules.

eslint-plugin-security-node. (900 weekly downloads). This plugin also covers a wide range of vulnerabilities for Node.js applications. It has considerable overlap with the previous ones but does attempt to cover some different vulnerabilities. No documentation exists for many of the rules, and the quality of the rule implementations is very low. Some of the rules have considerable false *negatives*. This is illustrated in rules `detect-sql-injection` and `detect-helmet-without-nocache`, which we discussed in Sections 4.2 and 4.3; if variables do not have the names that the rules expect, the rules completely disregard the code, thus possibly accepting vulnerable code. Despite the plugin being last updated 4 months ago, one of the rules throws an exception if the plugin is installed in projects using Node.js version 14 or above.

eslint-plugin-no-unsanitized. (25.000 weekly downloads). This plugin discourages developers from using unsafe manipulation of the DOM with methods such as `document.write` and `.innerHTML`. The goal is to prevent XSS attacks. The plugin is consistently maintained by Mozilla. However, the rules have several shortcomings, discussed in detail in Section 4.1: The rules in the plugin generate many false positives; they only allow explicit strings in DOM assignments. Furthermore, the rules do not consider the manipulation of `href` and `src`. Finally, the rules do not support Web frameworks that render HTML dynamically, such as the popular React.

eslint-plugin-no-unsafe-innerhtml. (12.000 downloads per week). This plugin is the same as the previous one, except with a smaller scope of considering `.innerHTML` assignment. It was last updated 3 years ago.

eslint-plugin-no-secrets. (5.000 weekly downloads). This plugin discourages programmers from having different kinds of secrets in the source code. The rule uses regex to find patterns of potentially secret values in the code. However, the rule has false negatives, in the form of secrets that do not match any of the given patterns. The rule does include, as an option, to ignore specific secret identifiers, which can help decrease the number of false positives that the rule generates. The plugin was updated within the last 3 months.

eslint-plugin-sql-injection. (2 weekly downloads). This plugin discourages programmers from using string concatenation in SQL query execution. As discussed in detail in Section 4.3, this rule has a lot of false negatives, since it ignores all variables that are not specified in the rule configuration, and even then, does not consider e.g. template strings. It has not been updated in 2 years.

Evaluation. There are surprisingly few plugins that target vulnerabilities. The handful of security rules that see notable use identify only a few vulnerabilities in total. Alarmingly, they focus overwhelmingly on reducing false *negatives*, while still having considerably many false positives, to the point that the rules might be too inconvenient use, and thus gets disabled by programmers. Finally, the checks that the rules are making are rather simple. As demonstrated in Section 4, we were able to implement security rules as a proof-of-concept with relative ease.

Observation 3 *The effort needed to create & share linter rules for security is low.*

Observation 4 *Linting for vulnerabilities is underexplored.*

5.3 ESLint Rule Guidelines

Pondering the reason for the state of ESLint plugins for security, we analyze the ESLint guidelines for creating new core rules [39], to assess whether the guidelines are a good fit for writing rules for security. We find that most of the guidelines run counter to the needs of a rule for security. The following is a summary of the guidelines:

Widely applicable. The rule should be of importance to a large number of developers; no individual preferences.
Generic. The rule must not be so specific that it is hard to know when to use it; at most two "and"s.
Atomic. The rule must work on its own, and be oblivious to other rules.
Unique. The rule must not produce same warnings as existing rules (no overlap) as that confuses the programmer.
Library-agnostic. The rule must not be based on specific libraries or frameworks (except Node.js).
No conflicts. The rule must not conflict with other rules.

For a rule to be incorporated into the core rules set, the rule *had*[1] to follow these guidelines. However, "widely applicable" and "generic" runs counter to the fact that vulnerabilities arise under highly specific circumstances, and "library-agnostic" runs counter to the fact that most vulnerabilities arise from libraries and their use, as explained under limitations in Section 5.1. While a rule author does not *have* to follow these guidelines, she might want to, in the hope that the rule makes its way to core, and thus has more impact. Finally, these are the only guidelines for writing rules; there are no guidelines for writing rules for security.

5.4 ESLint Configuration

Configuring ESLint is a nontrivial task, which is a well-known barrier to using linters effectively. A study conducted by Delft University of Technology in 2019, shows that 38,3% of the study participants (all JavaScript developers) agreed that *"creating or maintaining configurations was a challenging part of using a linter."* [40] This validates our perception that the domain of linters contains problem areas and difficulties worth exploring and improving.

[1] ESLint no longer accepts new rules into the core rule set, as of 2020.

6 Recommendations

We formulate four recommendations on the use of linters for security.

Recommendation 1 *Linters should be configurable*

We find that there is no linter that comes with a set of fixed rules that addresses all critical security vulnerabilities. This observation is corroborated by our new rules on cross-site scripting, which demonstrate that the one-rule-fix-all approach is insufficient to tackle all facets of cross-site scripting attacks. Hence, linters should be configurable and pluggable so that users can extend them with either in-house rules or with community rules, which the users can enable as needed.

Recommendation 2 *Linters should scan code differently to enable the finding of more security vulnerabilities*

We have found that linters are precluded from finding some vulnerabilities because files are evaluated individually and in an arbitrary order. However, files often depend from other files, and vulnerabilities may arise by looking on how such files depend on each other (e.g. library dependencies). We observe that this can be potentially fixed by scanning multiple files, with the evaluation order provided by the user or automatically suggested. A linter could even report issues in libraries to the library authors, along with instructions on how to patch the vulnerability. With more general ways of scanning code in place, a linter can be viewed as a *framework* for implementing static analysis tools.

Recommendation 3 *Linter security rules should have proper descriptions*

We have found that existing rules do not precisely describe what kind of vulnerabilities they intend to detect. Rules should instead provide the user with a precise explanation of the potential issue flagged in a piece of code. Failing in doing so may affect the user understanding of the possible issues with their code and, even worse, may lead the user to ignore the output of the rule. We believe that providing proper descriptions is a key enabler to retain linter popularity also for finding security vulnerabilities.

Recommendation 4 *Linter security rules should maximize case coverage and reduce false positives*

A rule should detect as many instances of a vulnerability as possible, and it should do so as precisely as possible. The value that a rule provides to the programmer scales with its ability to correctly flag as many possible vulnerable code cases as possible. However, a rule should strive to eliminate false positives as these are a significant challenge when using linters [40]. Intuitively, maximizing case coverage and reducing false positive might be seen as two contrasting requirements: attempts at maximizing case coverage may introduce new false positives. Moreover, from a security perspective, much focus is put on avoiding

missing vulnerabilities hence avoiding false negatives. However, we observe that the number of false positives affect considerably the development flow of the user, who has to investigate each flagged case to determine whether it is a false negative or not, which can lead the user to decide to disable the rule.

7 Related Work

A comprehensive list of open source and commercial static analysis tools is available in [31]. Here, we focus on prior works on linters and linter-like tools for JavaScript and other programming languages, putting emphasis on how they address security vulnerabilities.

JSLint [12] and JSHint [27] are two popular linters for JavaScript. The former being more dogmatic in linting, the latter being more flexible. None of them focuses on linting for security vulnerabilities nor allows for user-developed plugins. Sonarlint [35] is a linter that supports several programming languages, including JavaScript. It notably categorizes security-related rules in vulnerabilities and security hotspots. A vulnerability has a higher security impact than a security hotspot. For example, the use of a Web SQL database is considered a vulnerability, while hardcoding credentials is considered a security hotspot. Sonarlint cannot be extended via plugins and, at the time of writing this paper, it counts 3 non-deprecated vulnerability rules and 15 security hotspot rules for JavaScript. Similarly to our detect-sql-injection rule, Sonarlint has a rule that flags the execution of SQL queries that are built using formatting of strings. However, differently from our approach, Sonarlint has limited coverage on Node.js APIs and does not cover cross-site scripting attacks.

Several linters for other programming languages than JavaScript have considered vulnerability rules. Splint [14] is probably the first extensible linter for security vulnerabilities. It parses C code to find potential buffer overflows and provides a general language that enables users to define their own rules. However, to the best of our knowledge, there are no community-driven rules available today, hence, making a proper comparison against our proposed rules is impossible. Flawfinder [44] is a simpler tool that instead does a lexical analysis of C/C++ code in order to find security weaknesses. It has been recently found that Flawfinder detects more types of vulnerabilities than its competitors [25]. Cpplint[1] is an automated checker to make sure that a C++ program follows Google's style guide, which includes very few security tests. Neither Flawfinder nor Cpplint supports plugins.

Bandit [4] is a linter for finding security issues in Python code. Similarly to ESLint, Bandit allows one to write a plugin that can extend the tool with additional security tests. To the best of our knowledge, Bandit has not been as successful as ESLint in attracting new plugins. Differently from the rules proposed in our work, most of the preinstalled tests in Bandit concern hardcoded strings such as passwords, and command injections vulnerabilities such as attacks due to invoking external executables.

FindBugs [6] is a popular linter for Java with over 30 rules for security. It is not extensible. SpotBugs [2] is deemed as the successor or FindBugs and is extensible; Find Security Bugs [5] is the SpotBugs plugin for security vulnerabilities. However, SpotBugs does not provide a repository of plugins. Neither FindBugs nor SpotBugs facilitate automatic bug fixing.

GoCritic, Revive, and Ruleguard are linters for Go that allow external rules. A comparison of the tools is provided in [34]. Gosec [3] is a security linter for Go. It has a limited and not user-extensible number of available rules. It includes rules that raise issues in case of SQL query constructions using format string or string concatenation. However, it does not cover cross-site scripting attacks.

8 Conclusion

Linters are static analysis tools that have great potential to automatically detect and eliminate vulnerabilities in software. This topic is relatively unexplored, and in this work, we have investigated to which extent linters can help mainstream programmers to deliver vulnerability detection, with a specific focus on how a pluggable linter can help JavaScript programmers to write secure code.

We have found that the effort required to create ESLint rules for security vulnerabilities is rather low, while the impact of creating such rules is potentially high. This requires, however, the rules being well documented and being engineered to minimize false positives.

We also observe that linters should be easily configurable to adapt to the new security vulnerabilities facets that may arise in libraries, fostering the rapid development of rules that address such vulnerabilities. Similarly, linters should facilitate the scanning of dependencies since vulnerabilities are likely to come from invoked libraries [42].

We have demonstrated that automatic vulnerability fixing is effective, and believe that linters have the potential to enable detection and correction of vulnerabilities in libraries even before the library developers produce a patch for the vulnerable code: a library developer may add a rule for ESLint while working on a long-term fix for the vulnerability. We also note that library developers who do not use linters may introduce well-known vulnerabilities in their code. Linters that already have community-driven rules addressing such well-known vulnerabilities can thus facilitate the report of them to library developers.

Obviously, linters are not the sole tool for effective vulnerability detection and fixing. Other approaches might achieve a better balance in terms of security and tool popularity. With this work, we aim at broadening the view and at stimulating discussion within the program analysis community towards novel and practical ways to tackle vulnerability detection and fixing in software.

References

1. Cpplint (2009), https://github.com/cpplint/cpplint/
2. Spotbugs (2017), https://spotbugs.github.io/
3. Gosec - golang security checker (2018), https://github.com/securego/gosec
4. Bandit (2019), https://github.com/PyCQA/bandit
5. Arteau, P.: Find security bugs (2012), https://find-sec-bugs.github.io
6. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W.: Using static analysis to find bugs. IEEE Softw. **25**(5), 22–29 (2008)
7. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006. pp. 73–85. ACM (2006)
8. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Gros, C., Kamsky, A., McPeak, S., Engler, D.R.: A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM **53**(2), 66–75 (2010)
9. Brat, G., Klemm, R.: Static analysis of the mars exploration rover flight software. Proceedings of the First International Space Mission Challenges for Information Technology pp. 321–326 (2003)
10. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. pp. 289–300. ACM (2009)
11. Chess, B., McGraw, G.: Static Analysis for Security. IEEE Secur. Priv. **2**(6), 76–79 (2004)
12. Crockford, D.: Jslint (2002), https://www.jslint.com/
13. Ernst, M.D.: Invited talk: Static and dynamic analysis: synergy and duality. In: Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004. p. 35. ACM (2004)
14. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Software **19**(1), 42–51 (2002)
15. Feldman, M.B.: Who's using ada? real-world projects powered by the ada programming language november 2014 (2014), https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html
16. Guarnieri, S., Livshits, V.B.: GATEKEEPER: mostly static enforcement of security and reliability policies for javascript code. In: 18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings. pp. 151–168. USENIX Association (2009)
17. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., Berg, R.: Saving the world wide web from vulnerable javascript. In: Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011. pp. 177–187. ACM (2011)
18. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of javascript. In: ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6183, pp. 126–150. Springer (2010)
19. Hahn, E.: Helmet (2012), https://helmetjs.github.io/
20. Henry, J.: Typescript eslint parser (2019), https://www.npmjs.com/package/@typescript-eslint/parser

21. Inc., F.: React (2013), https://reactjs.org/
22. Johnson, B., Song, Y., Murphy-Hill, E.R., Bowdidge, R.W.: Why don't software developers use static analysis tools to find bugs? In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. pp. 672–681. IEEE Computer Society (2013)
23. Johnson, P.: 11 software bugs that took way too long to meet their maker. https://www.csoonline.com/article/3404334/11-software-bugs-that-took-way-too-long-to-meet-their-maker.html (2015), CSO, From IDG Communications
24. Johnson, S.C.: Lint, a C program checker. Bell Telephone Laboratories (1977)
25. Kaur, A., Nayyar, R.: A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. Procedia Computer Science **171**, 2023 – 2029 (2020)
26. Keinänen, M.: Creation of a web service using the MERN stack (2018)
27. Kovalyov, A.: Jshint (2011), https://www.jshint.com/, accessed: 2020-06-25
28. Meyerovich, L.A., Livshits, V.B.: Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In: 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA. pp. 481–496. IEEE Computer Society (2010)
29. Mitchell, J.C.: Programming language methods in computer security. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2001, London, UK, January 17-19, 2001. ACM (2001)
30. OWASP Foundation: OWASP Top Ten (2017)
31. OWASP Foundation: Source Code Analysis Tools (2020), https://owasp.org/www-community/Source_Code_Analysis_Tools
32. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society **74**(2), 358–366 (1953)
33. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at google. Commun. ACM **61**(4), 58–66 (2018)
34. Sharipov, I.: ruleguard: dynamic inspection rules for Go (2020), https://quasilyte.dev/blog/post/ruleguard/
35. SonarSource: Sonarlint (2008), https://www.sonarlint.org/
36. Stack Exchange Inc.: Stack Overflow Developer Survey 2020
37. StrongLoop: Express (2010), https://expressjs.com/
38. Team, E.: Espree (2014), https://github.com/eslint/espree
39. Team, E.: Eslint: Contributing new rules (2020), https://eslint.org/docs/developer-guide/contributing/new-rules
40. Tómasdóttir, K.F., Aniche, M., Van Deursen, A.: The Adoption of JavaScript Linters in Practice: A Case Study on ESLint. IEEE Trans. Softw. Eng. (2018)
41. VeraCode: State of software security: Open source edition (2020)
42. Voss, L.: npm and the future of javascript (2018), https://slides.com/seldo/npm-and-the-future-of-javascript/, invited talk at JSConf US 2018
43. Wedyan, F., Alrmuny, D., Bieman, J.M.: The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009. pp. 141–150. IEEE Computer Society (2009)
44. Wheeler, D.: Flawfinder (2001), https://dwheeler.com/flawfinder/
45. Wing, J.M.: A Call to Action: Look Beyond the Horizon. IEEE Secur. Priv. **1**(6), 62–67 (2003)
46. Zakas, N.C.: Eslint (2013), https://eslint.org/

A ESLint

A.1 How ESLint Works

ESLint builds on Node.js and can be installed through `npm`. After that a configuration file is created, ESLint can be run on source files, e.g. from the command line or within an IDE [46]. ESLint takes as parameters which source file to lint and a configuration which e.g. specifies which rules to use.

First, ESLint parses the source file to render an abstract syntax tree (AST) from it. Each node in the resulting AST is a record which contains, amongst others, the type of the syntactic element it represents (e.g. `VariableDeclaration`, `Identifier`, `FunctionDeclaration`, etc.), and information about where in the source file the syntactic element is located (for blame assignment). In case the information in this AST is insufficient (e.g. when writing advanced rules), ESLint lets one specify a different parser from the default one (i.e. Espree [38]), to construct an AST that stores additional information.

Next, ESLint traverses this AST to check that all rules are upheld. Each rule in ESLint is represented by an object. A rule object maintains its own state and exports methods which ESLint calls while traversing the AST. At each node, both while going down and up the AST, ESLint invokes, on each rule object, a method representing the type of the node (`VariableDeclaration`, etc.; see above) and code paths (`onCodePathStart`, `onCodePathSegmentLoop`, etc.). If a rule detects an issue, then the rule reports the issue to a `context`, which ESLint passes as a parameter when it creates the rule object.

A.2 Rules

Each rule in ESLint consists of three files[3]: a source file, a test file, and a documentation file. Source files[2],(e.g. Figure 2), are stored in `lib/rules`. They have the following format[3]. A source file exports an object with two properties.

```
1 module.exports = { meta:meta, create:create }
```

The objects *meta* and *docs* have four properties.

```
1 meta = { docs:docs, type:string, fixable:boolean, schema:schema }
2 docs = { description:string,url:string,recommended:boolean,category:string}
```

In *docs*, `description` is a description of what the rule checks. `url` is the URL to the rule's documentation. `recommended` specifies whether this rule should be added to the list of recommended ESLint rules (which can all be turned on with a single option in the configuration). `category` specifies where this rule should appear in the rules index; valid values include `"Possible Errors"`, `"Best Practices"`, `"Strict Mode"`, & `"Stylistic Issues"`. In *meta*, `type` specifies the type of the rule; valid values are `"problem"` for a rule that identifies bad behavior, `"suggestion"` for a rule that provides improvement suggestions, and

[2] This is a slightly simplified version of the original, from ESLint core.
[3] Rules are not strictly required to follow this format; some deviate from it.

```
module.exports = {
  meta: {
    docs: {
      description: "no returning value from constructor",
      url: "https://eslint[...]/no-constructor-return",
      category: "Best Practices",
      recommended: false
    },
    type: "problem"
  },
  create: function(context){
    const message = "Unexpected return in constructor."
    const stack = [];
    return {
      onCodePathStart: function(_,node){stack.push(node)},
      onCodePathEnd:   function()      {stack.pop()},
      ReturnStatement: function(node){
        const last = stack[stack.length - 1]
        if (!last.parent) { return }
        if ( last.parent.type === "MethodDefinition" &&
             last.parent.kind === "constructor" &&
             node.parent.parent === last || node.argument
        ){ context.report({ node, message }) } }}}}
```

Fig. 2: `no-constructor-return` source file.

`"layout"` for a rule that provides stylistic tips. If a rule does not automatically fix an issue, then the `fixable` property should be omitted. Otherwise, `fixable` should be set to `"whitespace"` if it only affects whitespace, and `"code"` otherwise. *schema* specifies which configuration options the rule accepts and should be omitted if the rule accepts no such options. *create*, called when the rule object is created, returns an object which contains the functions that ESLint calls while traversing the AST (see above). The rule in Figure 2 gives an example of how a rule can maintain state and traverse the AST. Its state is a stack of code paths, which it uses, upon encountering a `return` statement, to examine the AST to see if the statement is occurring within a constructor.

Test files are stored in `tests/lib/rules`. The test file contains sample inputs, along with the expected result of applying the rule on said input (valid, invalid). The unit test can then be run using the testing facility built in ESLint. Documentation files, stored in `docs/rules`, are written in Markdown syntax, and provide a description of what rules check, and how to configure them.

Plugins. Additional rules can be added to ESLint by downloading ESLint plugins. A plugin is a collection of ESLint rules. Plugins are routinely created by individuals and organizations, and shared as packages on `npm`. At present, `npm` contains thousands of ESLint plugins, each of which often contains tens of rules.