Bilingual Problems: Studying the Security Risks Incurred by Native Extensions in Scripting Languages

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security

Sazzadur Rahaman
University of Arizona

Ágnes Kiss
CISPA Helmholtz Center for Information Security

Michael Backes
CISPA Helmholtz Center for Information Security

Abstract

Scripting languages are continuously gaining popularity due to their ease of use and the flourishing software ecosystems surrounding them. These languages offer crash and memory safety by design. Thus, developers do not need to understand and prevent low-level security issues like the ones plaguing the C code. However, scripting languages often allow *native extensions*, a way for custom C/C++ code to be invoked directly from the high-level language. While this feature promises several benefits, such as increased performance or the reuse of legacy code, it can also break the language's guarantees, e.g., crash safety.

In this work, we first provide a comparative analysis of the security risks of native extension APIs in three popular scripting languages. Additionally, we discuss a novel methodology for studying the misuse of the native extension API. We then perform an in-depth study of npm, an ecosystem that is most exposed to threats introduced by native extensions. We show that vulnerabilities in extensions can be exploited in their embedding library by producing reads of uninitialized memory, hard crashes, or memory leaks in 33 npm packages simply by invoking their API with well-crafted inputs. Moreover, we identify six open-source web applications in which a weak adversary can deploy such exploits remotely. Finally, we were assigned seven security advisories for the work presented in this paper, most labeled as high severity.

1 Introduction

Originally, the primary use case for modern scripting languages [57] like Python, Ruby, or JavaScript was the development of web applications. Recently, though, they became tremendously popular, general-purpose programming languages with powerful emerging use cases like TensorFlow for machine learning in Python or Electron.js for portable desktop applications in JavaScript. This development is supported by massive open-source ecosystems such as npm, PyPI and RubyGems. There is a large body of work identifying a plethora of security risks that affect these software repositories [6, 24, 27, 30, 66, 76], which in turn, can impact real-world websites [65]. However, prior work only considers security risks present in the scripting code, thus, ignoring the important cross-language interactions in these ecosystems.

Native extensions are a convenient way to allow low-level functionality to be directly invoked from a scripting language. Package managers like `npm`, `pip` or `gem` enable smooth usage of such extensions by compiling at install time the extension's binary [3–5]. The binary is loaded on-demand at runtime in the scripting code's process, unlocking cross-language cooperation. In this way, developers can expose hardware capabilities that were originally beyond the reach of the scripting language. Native extensions also enable the reuse of mature, legacy code written in low-level languages. Databases like SQLite[1] or cryptographic libraries like OpenSSL[2] are often exposed using native extensions. Finally, native extensions aid the development of performance-critical code in low-level languages. For instance, a non-negligible part of TensorFlow is written in C++ and exposed to Python through bindings.

One can think of native extensions as the democratization of the binding layer, which glues the language engines with their surrounding environment. Previous work [19, 29] discusses the security risks incurred by this layer and provides evidence that vulnerabilities are prevalent even in binding code of popular runtimes like Node.js. This type of code is usually developed by highly-skilled developers, whereas native extensions can be written by anyone. As one may expect, writing reliable native extensions is difficult since subtle bugs may arise at the language boundary. The main culprit for this are the fundamental differences in representing data in the two languages, e.g., weak dynamic typing vs. strong static typing. Moreover, a mistake in an extension may propagate in the ecosystem, affecting several libraries that depend on it or even compromising production-ready applications.

Let us consider the example in Figure 1 to illustrate how bugs may arise when using native extensions. The

[1] https://www.sqlite.org
[2] https://www.openssl.org/

(a) JavaScript client of the package `nativepad`

```
1 let nlib = require('nativepad');
2 nlib('foo'); //returns "foopad"
3 nlib('foo\0bar'); //returns "foo" followed by
      three uninitialized bytes
4 nlib(true); //returns 4 uninitialized bytes
5 nlib({toString: 42}); //hard crash (segfault)
```

(b) JavaScript code for the package `nativepad`

```
1 let addon=require('bindings')('addon.node');
2 module.exports = (str) => {
3     if (!str)
4         throw 'Invalid string';
5     return addon.Pad(str);
6 }
```

(c) C++ code for the native extension

```
napi_value Pad(napi_env env, napi_callback_info info) {      1
   napi_status status;                                       2
   size_t argc = 1, strSize;                                 3
   napi_value args[1], result;                               4
   status = napi_get_cb_info(env, info, &argc, args, NULL    5
       , NULL);
   assert(status == napi_ok);                                6
   napi_get_value_string_utf8(env, args[0], NULL, NULL, &    7
       strSize);
   strSize = strSize + 4;                                    8
   char myStr[strSize];                                      9
   napi_get_value_string_utf8(env, args[0], myStr,           10
       strSize, NULL);
   strcat(myStr, "pad");                                     11
   napi_create_string_utf8(env,myStr,strSize,&result);       12
   return result;                                            13
}                                                            14
```

Figure 1: Example of a hypothetical npm package called `nativepad` (b), its native extension (c) and a client invoking it (a). The dashed arrows show the data flows between the three components.

`nativepad` package uses a native extension to pad a given string to the right with the literal `"pad"`. Its native extension in Figure 1c employs four calls to the extension API: one at line 5 to retrieve the arguments, one at line 7 to get the length of the first argument, one at line 10 for converting the JavaScript string into a C one, and finally one call at line 12 to convert the C string back into a JavaScript one. Additionally, the extension allocates the memory to store the padded string and performs the string concatenation using `strcat`. The JavaScript code of the `nativepad` package in Figure 1b is trivial, performing a simple null check on the input and invoking the `Pad` function of the native extension. Now, let us consider a client in Figure 1a that invokes the exported function with different arguments. Note that the client is oblivious to the use of native extensions, i.e., the `require` statement in line 1 would be exactly the same for loading a "pure" JavaScript package. When invoking `nativepad` with a well-behaved string, the padding is performed as expected. However, when the null terminator (`\0`) is present in the string, the native extension exposes uninitialized memory. Though it leads to string termination in C, the JavaScript runtime treats this character as any other character, i.e., counts it towards the string length. Even more surprising behavior emerges, e.g., a hard crash of the Node.js process, if unexpected values (e.g., Booleans or certain object literals) are provided. Such outcomes may surprise users, potentially leading to security incidents, e.g., denial of service.

While the considered example does not follow the best practices of the native extension API, e.g., checking the argument type or the return value, we believe that the runtime should be robust enough to protect against such a *misuse*. We notice that there is a large design space for a native extension API and that different design decisions make programming with the obtained API more dangerous than others. To explore this design space, we study the native extension API in three popular scripting languages and show that misuse is possible in each of them. However, there are significant differences across languages. The Node.js API is by far the most permissive, allowing several types of misuse, such as calling a native extension with insufficient arguments or integer overflow for numeric values exchanged across the language boundary.

To study the security implications of using native extensions, our methodology first identifies misuse in open-source libraries. To that end, we perform both intra-procedural and cross-language static analysis. We propose a simple yet effective way of constructing cross-language graphs that combines the two functions closest to the language boundary. We then perform demand-driven data-flow analysis on open-source web applications to study the impact of the library-level problems at the application level.

In our evaluation, we first perform an empirical study of the prevalence of misuse in 6,432 npm packages with native extensions. We show that even popular packages are prone to misuse, and we provide evidence that an attacker can cause real harm to web applications by leveraging the bugs introduced by API misuse. Concretely, we provide proof-of-concept exploits for 33 npm packages, showing that attackers can break the language guarantees by manipulating the inputs to these packages (strong attacker model). Moreover, we identify six open-source web applications in which hard crashes can be caused remotely (weak attacker model). We were assigned seven CVEs for our findings, most labeled as high severity.

In summary, we provide the following novel contributions:

- We are the first to thoroughly analyze the security risks of native extensions in scripting languages. Several design

decisions enable vulnerabilities and burden the developer with the task of using the API in a secure way.
- We present a novel methodology that enables the study of vulnerabilities caused by misuse of the native extension API. We show how cross-language static analysis can be used for automatic vulnerability detection.
- We provide evidence that vulnerabilities caused by native extensions are present in open-source software packages and that they also affect web applications using them.

2 Threat Model

We assume that attackers have neither control over the native extension's code nor the privilege to execute arbitrary scripting language code. Developers of extensions are not malicious, but they might inadvertently introduce vulnerabilities in their code. We consider a native extension vulnerable if it can be used in a way that breaks the guarantees of the scripting language. For example, if it can crash the process, read/write to unintended locations, or execute arbitrary code. Note that native extensions may also be used to hide malicious payloads in supply chain [30] or protestware [2] attacks. Our goal is to demonstrate the dangers of native extension vulnerabilities, even if the developers are honest. Therefore, detecting supply chain abuse attacks are out of scope for this work.

We propose two attacker models. First, for analyzing packages with native extensions in isolation, we assume a **strong attacker model**, in which attackers are able to control any argument passed to a library with native extensions, as well as their number. However, only objects that can be serialized as JSON are allowed as arguments, and no modifications of the builtins are permitted. For example, our setting does not allow attacker-defined functions or modifications of `Object.prototype`. Thus, even this strong attacker model is much weaker than the one used in prior work on JavaScript bindings [19, 29], where the authors assumed that attackers could inject arbitrary code in the engine. Ultimately, we aim to identify vulnerabilities within native extensions that can be triggered remotely under a **weak attacker model**, where we assume a web attacker that can only provide inputs to a web application through its HTTP interface. These inputs may propagate to a native extension and trigger a vulnerability in its implementation.

Package managers for scripting languages exhibit a significant amplification effect for vulnerabilities [32, 54, 76]. We note that this effect also increases the significance of native extension vulnerabilities. Since package managers do not have any privilege restriction in place or a transparency mechanism to warn about the usage of security-sensitive APIs, they may transitively depend on libraries with native extensions without being aware of this fact. Attackers, thus, can exploit vulnerabilities caused by native extensions in client packages.

3 Misuse in Different Languages

To shed light on the pitfalls of existing native extension APIs, we build several simple extensions in three different scripting languages. These extensions are deliberately vulnerable, attempting to stress the corner cases of the API, e.g., by omitting type checks on values coming from the scripting language. We then attempt to break the safety of the scripting language by providing well-crafted values to the vulnerable extension's methods, i.e., we assume a strong attacker model in this section. Finally, we observe whether the API actively tries to prevent the exploitation and if so, in which way. For creating the list of misuses, we draw inspiration from the work of Brown et al. [19] for JavaScript bindings. However, we also add several misuses specific to native extensions, e.g., read-write local variables. For the study, we use Node.js 15.4.0, Python 3.8.5, and Ruby 2.7.0p0. For Node.js we consider two different native extension APIs, i.e., Nan[3] and N-API[4], due to their prevalence in open-source projects.

In Table 1, we provide an overview of our findings. We mark each misuse type with a unique identifier (M_i) and will use these throughout the paper referring to them. One can see that there is a lot of variation among the considered languages, i.e., while some prevent most of the misuses by construction, others put the burden of using the API in a safe way on the developer. Nevertheless, none of the languages prevents all misuse. For example, a crash in the native extension compromises the availability of the application relying on it in all considered APIs. Below, we discuss in detail each misuse class and how they are handled by different APIs.

Error containment. As mentioned earlier, scripting languages follow a no-crash philosophy. For example, in the case of division by zero, Ruby and Python produce an exception that can be gracefully handled in a `try-catch` block, while JavaScript simply outputs the `Infinity` value. Moreover, in Node.js, developers often rely on a process-level exception handler that prevents any unexpected exception from crashing the application. We believe that this crash avoidance mentality has to do with the main use case of scripting languages, i.e., writing web applications, for which availability is one of the most important requirements. Native extensions can violate this no-crash philosophy in two ways: by producing low-level crashes (M_2) that terminate the whole process or by leaking low-level exceptions (M_1) that cannot be handled by a `try-catch` block in the scripting language. Let us consider the `int64-napi` npm package that wraps the `int64` C type. It provides a `divide` method that can be invoked as follows:

```
const int64 = require('int64-napi');
const Int64 = int64.Int64;
try {
 int64.divide(10, 0); // hard crash of Node.js
} catch(e) { } // never invoked
```

[3] https://www.npmjs.com/package/nan
[4] https://nodejs.org/api/n-api.html

Table 1: Different misuses of the native extension API and their prevalence in the considered languages. ● means that the API allows the misuse, ◐ that the API partially allows it, and ○ that the API prevents the misuse. We estimate the severity based on the impact a given misuse might have on the security, privacy, or availability of a web application.

Type	Id	Misuse	Node.js-N-API	Node.js-Nan	Python	Ruby	Severity
Errors	M_1	Not catching C++ exceptions	●	●	●*	N/a	Low
	M_2	Not handling runtime errors in C/C++	●	●	●	●	Medium
Arguments	M_3	Passing arguments with a wrong type	●	●	○	○	High
	M_4	Passing wrong number of arguments	●	◐	○	○	High
	M_5	Not accounting for different semantics of \0	●	●	○	○	High
	M_6	Passing arguments that overflow numeric types	●	●	○	○	High
Ret.	M_7	Missing return statement	●	○	◐	●	Low
	M_8	Declaring interface methods that return void	○	○	◐	○	Low
Mem.	M_9	Returning uninitialized memory values	●	○	●	○	Medium
	M_{10}	Mismanagement of cross-language pointers	●	○	●	○	Low
High-level	M_{11}	Producing unexpected side-effects in the runtime	○	○	○	●	High
	M_{12}	Blocking the runtime with slow cross-language calls	●	●	●	●	Medium
Low-level	M_{13}	Reading outside of an allocated buffer	●	○	○	○	High
	M_{14}	Using a pointer after it was freed	●	●	●	◐	High
	M_{15}	Freeing a pointer twice	◐	◐	◐	◐	High
	M_{16}	Failing to deallocate unused memory	●	●	●	●	Low
	M_{17}	Interpreting user input as format string	◐	◐	○	○	High

* Python's extension support is mainly intended to be used with C. Nonetheless, we used the `distutils` builtin package to compile a C++ extension and simulate an unhandled C++ exception.

This code snippet produces a hard crash that cannot be handled in the corresponding `try-catch`. Such an outcome may surprise users that consider a catch clause as a universal safety net. Similarly, if there are C++ exceptions that are not properly handled by the native extension, there is no way for the scripting language to catch them. We saw this behavior in all the languages that support C++ native extensions.

Argument translation. Since the analyzed scripting languages are weakly-, dynamically-typed, while C/C++ is strongly-, statically-typed, the native extension API has to assist the user in translating between these two type systems. In Ruby, one needs to specify the number of arguments at extension declaration time. In contrast, in Python, the API for retrieving the arguments mandates that the user specifies the number of arguments (M_4) and their type (M_3). Any violation of these specifications would result in aborting the current method invocation. By contrast, in Node.js, both considered APIs specify that the users should voluntarily check the arguments' types and their number and decide when to proceed. As seen in Figure 1, this may lead to serious problems such as processing strings with negative length or, even worse, user-provided values considered as object pointers. We direct the reader to [19] for an extensive discussion about the implications of breaking type safety in V8-based runtimes. We further stress the fundamental differences in the way errors are signaled in the different scripting languages. Whenever a mismatch is detected between the requested type for a value and its dynamic type, Python and Ruby stop immediately with an exception. One of the two considered native extension APIs for Node.js, N-API, returns a non-empty status code, while the other, Nan, does not detect the mismatch. Even when the types are correctly aligned, problems are still caused by the different ways a given type is represented in the two languages. While the null terminator `\0` can appear in valid strings of the considered scripting languages, in C, it marks the end of a string. Hence, if such characters are allowed to freely cross the language boundary (M_5), as is the case in Node.js, they may allow attackers to strip important information from a value or to cause confusion about the string length as illustrated in Figure 1. Ruby and Python refuse to continue with the invocation when such characters are detected. A similar issue appears when a numeric value overflows (M_6) due to a mismatch in the types' capacity. This case is prevented again by Ruby and Python but allowed in Node.js. An integer overflow may invalidate important checks performed in the scripting language, e.g., `val>0`, since the invariant may not hold anymore for the translated value.

Missing return. To our surprise, there are also subtle bugs involving the return value of a function. A missing return statement (M_7) causes a hard crash when reading the return value in Python, Ruby, and N-API. This may surprise developers who expose the native extension directly to their clients

and never test for such corner cases. Returning null values from the extension does not cause problems in the analyzed languages, but declaring the return value as `void` (M_8) causes a hard crash on method invocation in Python.

Memory issues. Similarly to the example in Figure 1, native extensions may expose non-initialized memory areas to the scripting language (M_9). Such memory locations may contain sensitive user information available in the process. In N-API and Python, one can expose both uninitialized string values and buffers. On the contrary, Ruby and Nan seem to initialize such memory areas with null bytes proactively. Memory issues may also appear due to the garbage collector not freeing pointers to interface objects exchanged across the language boundary (M_{10}). For primitives, all considered APIs prevent this by default. Python, however, makes it easy to overwrite this behavior by claiming ownership of certain pointers. While this is not a problem by itself, carelessly using this feature may compromise the availability of the entire application. Another interesting case is classes being exposed from C/C++ to the scripting language. When using Node.js (N-API), the garbage collector does not free references that are declared using `Napi::Persistent` API. As discussed further in Section 5.2, we identify several real-world libraries that are misusing this API, causing memory leaks.

High-level issues. Most of the considered APIs expose only opaque pointers to the C/C++ world. That is, the native extensions cannot directly access the exact memory location of an object, nor can they modify it without the aid of the API. In Ruby, however, one can obtain a raw pointer that allows the modification not only of the argument passed to the extension, but also of other variables defined in the same memory region (M_{11}). In this way, a problematic extension may access or even alter encapsulated values. Considering that many developers use native extensions for heavy computation, e.g., cryptographic operations, it is somewhat surprising that the default behavior of all the considered APIs is to invoke the extension synchronously (M_{12}). That is, the main thread of the scripting language is blocked until the native extension computes. This may lead to serious availability issues if an attacker can control the amount of work the extension performs, especially in Node.js.

Low-level issues. Finally, we consider a handful of low-level vulnerabilities in our study to see if different APIs hinder their exploitation or not. To our dismay, in N-API, we could exploit a textbook buffer overflow (M_{13}) to overwrite local variables defined in the native extension. We also note that use-after-free (M_{14}) is allowed in most of the languages, but Ruby seems to initialize the freed memory areas with null bytes (we observe a similar behavior in case of uninitialized memory (M_9)). A double free (M_{15}) always triggers a core dump, and a format string vulnerability (M_{17}) is usually prevented by the compiler. However, in Node.js, only a warning is produced, while the compilation is aborted in other languages. Finally, none of the APIs make any effort to prevent or detect memory leaks in the extension code itself (M_{16}).

Summary. As an artifact of the presented study, we provide a set of benchmarks as supplementary material[5], exemplifying each misuse in a separate native extension for all considered scripting languages. We believe that this suite can be useful both for users trying to understand the pitfalls of each API and for language designers to inform their design decisions.

Considering the presented findings, we conclude that there is a lot of variation in implementing native extensions in various languages. Some APIs put a lot of effort into preventing users from misusing them, while others are more permissive. Node.js, in particular, seems to be very liberal in its API's design, outsourcing most of the safety checks to the developers. We filed a security issue summarizing our findings to the Node.js developers, who appreciated our report as informative. They argue that the identified issues are not security problems of the API but of the packages misusing it. As we show in Section 5.2, this relaxed design decision is the main enabler for several security issues in popular npm packages. The Node.js maintainers promised, however, to fix some of the identified issues, e.g., the behavior responsible for M_6. While the misuses presented in this section aim to emulate realistic user interactions, the reader may wonder whether such cases appear in practice and, if so, if they affect real-world applications. We now proceed to designing a methodology for studying this aspect.

4 Methodology

While native extensions can be directly integrated into (web) applications, we believe it is more common for these extensions to be first encapsulated in a package. Hence, we propose two levels of static analysis to detect native API misuse vulnerabilities. We depict our analysis pipeline for Node.js and npm in Figure 2, but we believe it can be easily adapted for other scripting languages and their ecosystems. First, we run a package-level analysis to detect vulnerable npm packages due to insecure native extensions under the strong attacker model. We propose running both simple, intra-procedural analyses, but also cross-language ones. Specifically, we create a common representation for both C/C++ and JavaScript code present at the language boundary to detect problematic native extensions within a package. After finding a vulnerable package, we use inter-procedural backward data-flow analysis to study its impact on applications that use the package under the weak attacker model assumption.

4.1 Package analysis

Since most native extensions we encountered are relatively small, and many misuses can be formulated as flow problems, we propose specifying the misuse detection as a graph

[5] https://www.staicu.org/native-extension-risks

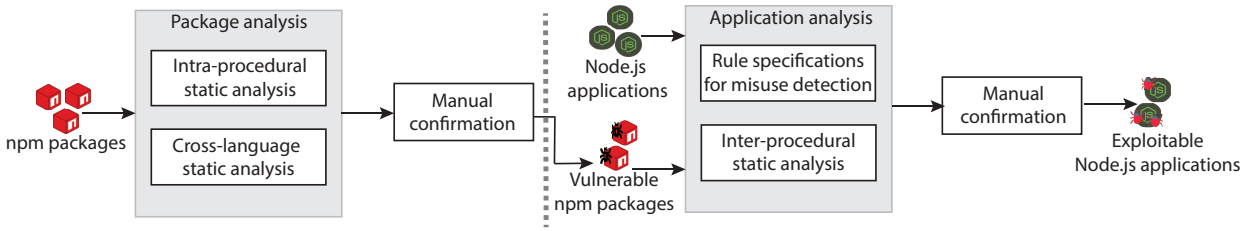

Figure 2: Overview of our methodology for identifying native extension vulnerabilities and for studying their impact.

traversal problem on the data-flow graph. However, as we show in Section 5, this may lead to a significant number of false positives because the analysis does not have information about how data is handled in the upper layer, i.e., in the scripting language. Hence we also propose unifying the data-flow graphs of the two languages.

Intra-procedural analysis. The first step of our analysis is to create a data-flow graph of the target functions. Our definition for such a graph is very permissive: nodes N represent lines of the program, and edges E depict explicit information flows. For instance, the green part of Figure 3 shows the data-flow graph for the example in Figure 1c. The nodes represent statements, and the edges represent data flows between them. We argue that the exact representations may vary as long as the semantics of the edges are preserved.

We then associate special meaning to particular nodes in the graph. n_0 is the root node of the graph where the traversal starts from, corresponding to the method definition statement in the source code. Thus, it has outer edges towards all the nodes in which parameters are referenced. S is the list of sink nodes that the analysis is interested in, while $\overline{S}$ is the list of sanitizers that invalidate a given flow to the sink.

Our analysis reports a security vulnerability if and only if:

- $\exists s \in S$ such that $n_0 \rightsquigarrow s$,
- $\nexists \overline{s} \in \overline{S}$ such that $n_0 \rightsquigarrow \overline{s}$,

where $a \rightsquigarrow b$ represent a path from a to b on the graph. We note that the presented analysis is not argument-sensitive, if any data flow to the sanitizer is detected, the flow to the sink is considered safe. This pragmatic design decision can lead to many false negatives in practice. Nevertheless, in this work, we do not aim for a complete solution to the described problem but to show the feasibility of an automated detection technique in this domain.

Cross-language analysis. We observe that many relevant API calls, e.g., sanitizers, happen in the two functions that are closest to the language boundary: one in JavaScript and one in C/C++. For identifying such pairs, we search for calls to the native extension API that map low-level functions to their high-level names. All the considered APIs in Section 3 require such calls during the initialization of a native extension. Let us assume we want to expose the `Foo` function from C/C++ to the scripting language, with the name `"foo"`. The syntax used by the considered APIs for binding the two entities is:

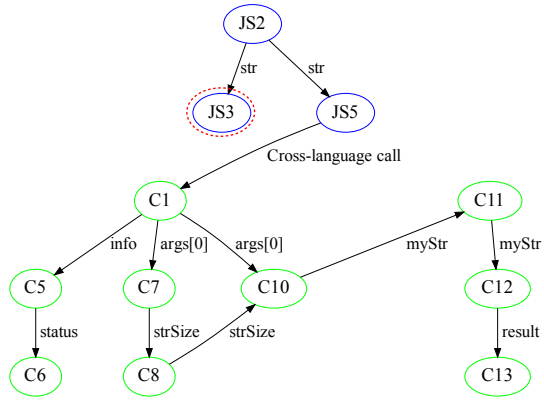

Figure 3: A cross-language data-flow graph for our example shown in Figure 1. We depict the JavaScript nodes with blue and the C/C++ ones with green. Numbers denote line numbers in Figure 1. With red we mark a potential location for sanitization in the JavaScript front-end.

```
1 // Node.js-Nan
2 Set(module, New<v8::String>("foo"),
3       New<v8::FunctionTemplate>(Foo));
4 // Node.js-N-API
5 napi_define_properties(...,{"foo",...,Foo,...});
6 // Ruby
7 rb_define_method(module,"foo",Foo,1);
8 // Python
9 PyModule_Create({...,{"foo",(PyCFunction)Foo
    },...});
```

Once we identify this mapping, we merge the data-flow graphs of the two functions by adding an edge from the node in the JavaScript graph corresponding to the native extension call to the definition node of the invoked C/C++ function. Finally, we perform the same analysis described above on the obtained cross-language graph.

Let us consider Figure 3 that shows the cross-language data-flow graph corresponding to the native extension in Figure 1. We assume that we are interested in detecting unchecked type conversions. By applying our analysis starting from $JS2$, we can detect a path to $C7$ that corresponds to a call to `napi_get_value_string_utf8()`, i.e., the sink. Since there is no statement either in JavaScript nor in C/C++ that checks that the type of the argument is string (sanitizer), the analysis produces a warning for this case. Let us assume that

in line 3 of Figure 1b there is a type check instead of a null check. Our cross-language analysis is path-insensitive, i.e., if we detect a flow from the source to a sanitizer, we consider the usage safe. Therefore, the analysis would detect a flow to the sanitizer marked with a dashed red circle, i.e., in *JS* 3, and would not produce a warning.

Implementation details. For extracting the data-flow graphs, we use Joern [75] for C/C++ files and Google Closure Compiler[6] for JavaScript. We instruct Joern to output the code property graph as a dot[7] file and further pre-process it by only preserving the data-flow edges. We also add edges from the function definition node, i.e., the first node, to the nodes accessing the `info[*]` and `args[*]` objects, which are the arguments coming from JavaScript. Joern fails to detect these edges because the arguments do not appear verbatim in the function declaration. For the Google Closure Compiler, on the contrary, we build our custom compiler pass to extract def-use pairs from its internal representation and output them in a dot file. We run both Joern and the Closure-based analysis with a budget of 15 minutes per analyzed package.

To find the two functions at the language boundary, we first perform a simple static analysis of the JavaScript code to detect which of the exposed C/C++ functions are called directly and in which JavaScript function. We then proceed by resolving these calls by analyzing the C/C++ code and identifying API calls to functions such as `napi_define_properties` described above. Once we identify the two functions, we retrieve their corresponding dot representations and merge them as described earlier. We then analyze the obtained graph and output security violations. Thereafter, we manually verify each security violation by attempting to exploit the misuse through the package's API. In case of success, we proceed to study the vulnerability's impact on real-world web applications. Additionally, whenever we identify an exploitable violation, we proceed to manually look for other misuses in that package, under the assumption that misuses tend to occur together.

Security Modelling. Our current prototype is targeted towards studying an important subset of the misuses identified in Table 1: missing type checks (M_3, M_4). For M_3, we specify the list of sinks based on the APIs we studied in Section 3 and the list of sanitizers based on idiomatic type checks in the two languages, together with the APIs provided by N-API and Nan for type checking. We provide the complete list of sinks and sanitizers in Appendix A. For M_4, we additionally consider checks on number of arguments as sanitizers. The supplementary manual analysis step described above allows us to identify misuses that go beyond (M_3, M_4) once an initial missing type check is identified in a given package.

[6] https://developers.google.com/closure/compiler

[7] https://en.wikipedia.org/wiki/DOT_(graph_description_language)

4.2 Application analysis

The existence of native extension vulnerabilities in npm packages motivated us to investigate their impact on Node.js web applications. Specifically, after manually confirming the vulnerabilities found in Section 4.1, we study their exploitability in the web application context. We formulate the detection of web applications using vulnerable packages as a flow problem, which can be automatically detected. To make our analysis scalable, we chose to use static analysis over dynamic analysis. This is because dynamic analysis requires running an application to monitor its runtime behavior [11, 29, 62]. Manually setting up and running a diverse set of Node.js web applications with their heterogeneous software and library dependencies are infeasible.

Therefore, we build a new demand-driven, def-use-based, static data-flow analysis framework for JavaScript, named FlowJS, for our needs. Finally, we use FlowJS on Node.js applications to detect exploitable uses of insecure native extensions. It is worth noting that demand-driven data-flow analysis has already been proven to effectively detect various kinds of API misuse in other languages [17, 35, 61, 64]. In this section, we discuss different components of our FlowJS framework. Note that FlowJS guarantees neither soundness (i.e., absence of bugs) nor completeness (i.e., absence of false alarms). However, like other practical static analysis tools, it favors completeness and efficiency over soundness. This design choice is acceptable for our use case since our goal is to run FlowJS scalably on a large number of web applications.

Rule specification. Our analysis takes rule specifications as input, which are manually created for a given vulnerable native extension API. A rule specification contains the API of interest and a callback function to check its misuses. The API definition consists of the function name and the parameter of interest. For example, to detect unsanitized inputs from the network to the function `run(query, data)` of the `sqlite3` package, one might specify the rule as follows.

$$IsMisuse\ \frac{p_o : run(\_, \textbf{\textit{data}}),\ \ P : \{p_i\},\ \text{s.t. } p_i \rightsquigarrow p_o, \forall i \in [1, |P|]}{\text{if } \texttt{req} \textbf{ or } \texttt{req.body} \in P \text{ then } \textbf{true} \text{ else } \textbf{false}}$$

Here, *IsMisuse* is the callback function that takes P as input and outputs *true* if misuse is found and *false* otherwise. p_o is the API definition, and P is the set of all unsanitized influences on p_o. `req` is the object containing the request data to the server. $\rightsquigarrow$ represents the *direct influence* on p_o. Our demand-driven analysis starts from the API invocation to find all program entities that influence it.

Intra-procedural backward data-flow analysis. An analysis to find the data flows to a given program point (invocations of the defined APIs) is known as *backward data-flow analysis*. We build our intra-procedural backward data-flow analysis on top of Google Closure Compiler's internal data-flow analysis framework. Closure's data-flow analysis framework provides an implementation of the worklist algorithm [58, 61]. To calculate data flows at a given program point, we implemented a

flow-insensitive def-use analysis by using the Abstract Syntax Tree (AST) representation of the code provided by the Closure Compiler. *Definition* (in short, *def*) of a variable x is an instruction that writes to x. *Use* of a variable y is an instruction that reads y. An analysis that utilizes the def-use relationship of variables is known as *def-use analysis*. Our def-use analysis collects direct influences and avoids any orthogonal function invocations. This is because we use FLOWJS to find *raw inputs* from the network or file system (Section 5.3), where processing is typically performed with orthogonal function calls.

Call-graph generation. We implement our call-graph generator on top of Google Closure's AST traversal algorithm. We traverse the AST to find function *definition* and *invocation* nodes and collect all the caller-callee relationships within a JS file. We represent anonymous function definitions with their line number and the starting position. For this prototype implementation, we do not handle function aliasing.

Inter-procedural backward data-flow analysis. Our analysis starts by finding all the call sites of the provided API invocations. It then runs intra-procedural backward data-flow analysis on all the caller functions using the given call sites. We run the process recursively by following the caller-callee chain upward. Then, we stitch all the intra-procedural data-flow summaries to form inter-procedural data-flow results. Finally, FLOWJS invokes the rule-specific callback functions on these results to find misuses.

5 Evaluation

To study the security impact of native extension misuse and the feasibility of its automatic detection, we first present an empirical study of missing type checks (M_3, M_4) in npm packages (Section 5.1). We then identify multiple zero-day vulnerabilities in real-world packages and report them to their maintainers (Section 5.2). Finally, we show that vulnerabilities in libraries can be exploited remotely in web applications (Section 5.3).

5.1 Missing type checks in npm

As discussed in Section 4.1, we study the feasibility of automatic misuse detection by focusing on an important class of misuses: missing type checks (M_3, M_4). This allows us to draw relevant conclusions about our hypothesis without investing tremendous engineering effort into modeling the APIs corresponding to all the misuses in Table 1.

Setup. To identify packages likely to contain native extensions, we analyze the entire npm graph available on 9th of February 2021. We consider all the packages that directly depend on five popular helper packages that are widely used for developing native extensions: `bindings`, `node-gyp`, `prebuild-install`, `node-addon-api` and `nan`. While this approach may have some false negatives, it is a cost-effective way to identify packages of interest without the need to download and analyze all the 1.5 million packages on npm and instead, allocate more resources for the in-depth study. In total, we download 7,605 npm packages that comply with the aforementioned requirement. Of these, we identify 1,173 false positives that do not contain any C/C++ code. After excluding these packages, we are left with 6,432 packages that we further use in our study. We believe that this is a large enough sample for drawing conclusions about how native extension APIs are used in Node.js. For each package, we download and analyze its latest version at the time of our study.

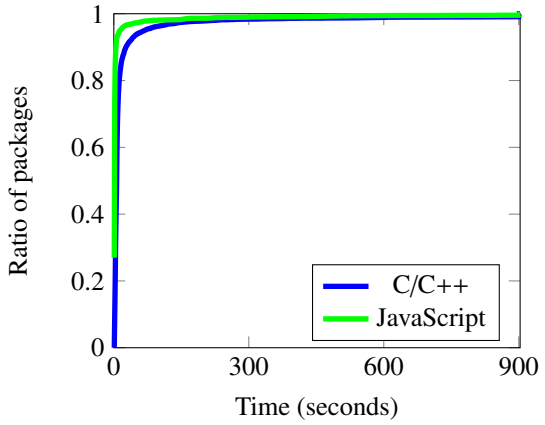

Figure 4: Cumulative distribution function (CDF) for the performance of data flow graph extraction, for the two languages. For any number of seconds between 0 and 900, we depict the ratio of the packages that have finished (or timed out) by then.

In Figure 4, we show the time used by our prototype to extract the JavaScript and the C/C++ graphs. We run all our experiments on a server with 64 AMD EPYC 7H12 cores, 2TB of memory, and running Debian 11. For 75% and 93% of the packages, the analysis finishes in less than 10 seconds for native and JavaScript parts, respectively. This shows that the proposed approach scales well and packages with native extensions tend to have more complex C/C++ code than JavaScript. For 69 packages, the graph extraction times out while analyzing the C/C++ part, while for 34 packages it does so for the JavaScript part. Since this represents less than 0.5% of the considered packages, we deem it is a minor shortcoming of our current prototype, and not a fundamental limitation of the proposed methodology.

To investigate the effect of missing type checks (M_3), we first search in all the C++ files for calls to type conversion APIs. We note that there are multiple ways in which arguments coming from JavaScript can be converted to a given type, e.g., `*.As<Type>`, `*.To<Type>`, and not all of these APIs react in the same way to misuse, but they all proceed with an unsafe value. In Figure 5, we depict the total number

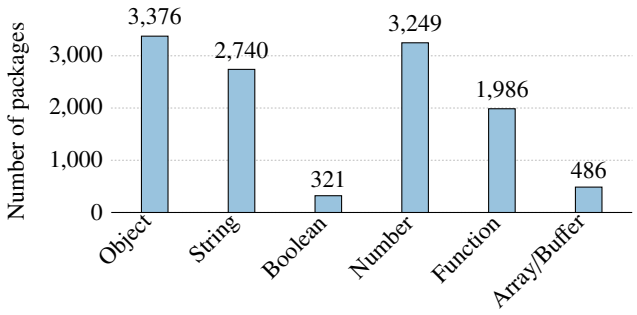

Figure 5: Number of packages explicitly converting values to various C/C++ types.

of packages that explicitly convert values to a given type. Casting to object, number, or string types are the most prevalent conversions, performed by 77% of the packages. The relatively low number of functions (16.3%) shows that at most one in three packages perform non-blocking, asynchronous operations (M_{12}), as these operations require a function object as an argument to be invoked upon completion.

We then perform intra-procedural analysis on the C/C++ native extensions, namely on the output `.dot` graphs of Joern, for detecting missing type checks. In total, we identify 2,802 packages with type conversions, of which 1,669 have a flow to the conversion API. Of these, 939 were type-checked in the native extension code, and 730 were not.

To evaluate our tool's effectiveness at finding flows to relevant APIs, we perform a controlled experiment with single C++ functions, all containing a flow to the target APIs. We collect a set of 25 functions from real-world packages, aiming for a diverse set of code constructs, e.g., macros, different APIs, intermediary variables, or chained calls. We then run our tool on these benchmarks and detect a flow in 21 of them (84% recall). One failure was due to Joern's imprecise data flow graph that failed to capture an important data flow step, one due to a missing sink, and two due to our analysis' inability to handle type checks or sinks present in macros or functions. We provide the set of microbenchmarks in the supplementary material to assist replication. We also verify all the produced flows and find a single false positive caused by a lack of path sensitivity (95% precision). We conclude that our prototype can handle a wide range of code constructs present in real-world code, producing a manageable amount of both false positives and negatives. We next proceed to identify zero-day vulnerabilities.

To this end, we concentrate on three APIs that produce hard crashes on misuse. Since we want to validate each finding manually, it is easier to judge the presence of a crash than the success of other types of payloads, e.g., the effect of integer overflow. We note that most work in the fuzzing domain uses similar testing oracles. In Figure 6, we show the total number of npm packages that contain detected data flows to (i) `*.ToLocalChecked()`, which is a method on the V8's `Maybe` type that concretizes a given value, (ii) APIs for casting

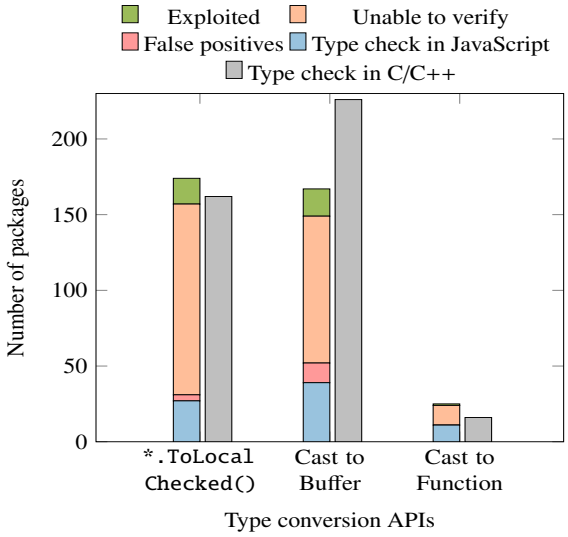

Figure 6: Packages with flows to type conversion APIs. The first bar represents unsanitized flows while the second one depicts sanitized flows. We further split unsanitized flows in different categories based on our manual inspection.

to Buffer, and (iii) APIs for casting to function. We depict both sanitized (grey bar on the right) and unsanitized flows (colorful bar on the left) and further categorize unsanitized flows based on their exploitability after manual verification.

During our manual analysis, we first verify if there is a JavaScript check that protects the reported vulnerable endpoint ("type check in JavaScript" in Figure 6). We then verify if, indeed, there is an unsanitized flow in the C/C++ part, i.e., that there are no method calls that perform sanitization that was missed by our intra-procedural analysis ("false positives"). We then proceed with the installation of the given package on our machine, which turned out to be a very challenging task. We were unable to install the majority of the reported packages ("unable to verify") due to several reasons: legacy code not running in our considered Node.js runtimes, missing hardware, different operating system, missing installed libraries. To maximize the number of packages that we can install, we attempt installing with five different Node.js versions: 15.4.0, 14.15.0, 12.22.1, 8.17.0, 0.12.18. For the packages that we could install, we attempt to write an exploit that produces a hard crash ("exploited"). If we fail to do so, we reanalyze the code and assign it to one of the other categories mentioned earlier.

Figure 6 shows first of all that 63% of the packages with a flow to type conversion APIs, also type check the arguments. Out of these, 82% do so in C/C++, showing that checking often occurs in the proximity of the API usage. These findings are good news since they show that most developers are aware of this best practice of the API. We successfully exploit 38 flagged packages, showing the feasibility of our automated approach. We identify a total of 22 clear false positives, all of them caused by type checks placed in differ-

Table 2: The most important npm packages for which we identified a zero-day vulnerability. Vulnerabilities M_2 and M_9 were found through manual inspection of flagged packages.

Package Name	#Downloads[9]	Misuse	Remote Exploitability	Status
bignum	5,091	M_3	Yes	CVE-2022-25324
ced	1,765	M_3	No	CVE-2021-39131
libxmljs	28,629	M_3	Yes	CVE-2022-21144
sqlite3	452,737	M_3,M_9	Yes	CVE-2022-21227
pg-native	92,436	M_3	Yes	CVE-2022-25852
utf-8-validate	917,251	M_3, M_4	No	Reported
@discordjs/opus	63,007	M_2, M_3,M_9	Yes	CVE-2022-25345
fast-string-search	25	M_3	Yes	CVE-2022-22138
time	1,701	M_3	Yes	Reported
bigint-buffer	159,067	M_3	No	Reported

ent functions / macros. This implies a false positive rate of 6%, in line with our controlled experiment. It also suggests that our lightweight analysis design (path-insensitive, intra-procedural) is adequate for the problem at hand. However, these results should be taken with a grain of salt since, for the majority of the reports, we could not verify their exploitability empirically, and thus, we rely on manual code analysis to judge whether the report is a false positive or not.

The main benefit of performing cross-language analysis, as described in Section 4.1, is to automate the first part of our manual process: the analysis should assign all the packages assigned to "type check in JavaScript" in Figure 6 (depicted in blue) to the grey bar. Another more subtle benefit is the improvement in user experience for the analyst. We often found ourselves switching between the C/C++ file and the JavaScript part of a package during manual analysis. A cross-language visual representation of the code would significantly ease this process. Using our prototype, we analyzed 6,401 cross-language flows and detected 300 flows to the sink. Out of these, 144 sanitize in C/C++, 45 sanitize in JavaScript, 22 sanitize in both, and 111 do not perform any sanitization. We provide the obtained cross-language data-flow graphs in the supplementary material of this paper[8] to increase confidence in our analysis method. Furthermore, we show graphs for three examples in the full version of our paper [67, Appendix A]: an unsanitized flow, a flow sanitized in JavaScript, and one sanitized in both languages.

5.2 Zero-day vulnerabilities

We identify a total of 33 libraries for which we can exploit a misuse through their public API, i.e., under the strong attacker model. In Table 3, we show the list of npm packages for which we could trigger a hard crash, their reach in the ecosystem, and the API we used for the exploit. We remind the reader that we use the hard crashes as a testing oracle, indicating a potential security problem. Some of these packages do not compile

[8] https://www.staicu.org/native-extension-risks

[9] Weekly downloads at the time of writing

[10] As defined by Zimmermann et al. [76].

Table 3: Npm packages in which we identified a previously unknown hard crash. The endpoint represents the package's method that we use for the proof of concept. With #main# we depict the default method exposed by the package.

Package name	Version	Reach[10]	Endpoint
@alien.sh/signals	1.0.0	1	Register
bigint-hash	0.2.2	3	update
bigint-buffer	1.1.5	97	toBigIntLE
bignum	0.13.1	168	powm
binary-diff	1.0.0	1	#main#
bkjs-utils	0.2.8	1	getUser
ced	0.0.1	3	#main#
csac-ed25519	0.0.3	1	Verify
csocket	1.0.3	1	send
cuckaroo29b-hashing	1.0.0	1	cuckaroo29b
fast-string-search	1.4.1	1	indexOf
gs-node-lmdb	0.9.0	1	Cursor
int64-napi	1.0.1	3	divide
jitterbuffer	0.1.14	3	put
libasar_enc	1.0.0	1	#main#
libxmljs	0.19.7	237	parseXml
multi-hashing	1.0.0	10	scryptjane
node-crc	1.3.0	4	crc64iso
node-lzma	0.1.0	1	compress
pg-native	3.0.0	92	query
libpq	1.8.9	4	$execParams
node-mbus	1.2.1	3	#main#
phin-ecdh	1.0.0	1	encrypt
pixel-change	1.0.0	2	#main#
rapid-crc	1.0.10	2	crc32c
roaring	1.0.6	6	_initTypes
sbffi	1.0.4	1	getNativeFunction
sendto	1.0.3	1	#main#
sqlite3	5.0.1	1905	run
termios-fixedv12	0.1.9	2	getattr
time	0.12.0	56	setTimezone
utf-8-validate	5.0.8	551	#main#
zopfli-node	2.0.3	1	deflateSync

with the latest Node.js version and require a legacy version of the runtime instead. Others require a specific library on the operating system before installation. On our setup we could, however, meet such strict constraints by acting on the compilation error we observed on unsuccessful installation attempts. We reported all these issues to the package maintainers.

We emphasize our most important findings in Table 2, highlighting the identified misuse types. In all the 33 libraries, we first detect an issue caused by careless type conversions (M_3, M_4), and for three of them, we identify additional types of misuses (M_2, M_9), using the supplementary manual analysis described in Section 4.1. To confirm that all the identified vulnerabilities caused by misuse are indeed security-relevant and worth fixing, we approached the maintainers of the library to report our findings. We describe below the disclosure process and the outcome of these interactions.

Vulnerability disclosure. We reported the discovered vulnerabilities to the maintainers using their email address provided in the package description. We have received a few responses and a CVE with this strategy, and after the grace period of six months has passed, we worked together with Snyk[11] to report the vulnerabilities affecting the most high-profile packages depicted in Table 2. This disclosure strategy has resulted in an additional six CVEs. All CVE-assigned high-profile packages have also been fixed, thanks to our disclosure efforts. The maintainer of `utf-8-validate` dismissed our report saying that it is unlikely that an attacker can trigger this bug remotely, while we are still pending responses for the other two we reported. Six CVEs were assigned a high severity label, and one of them medium. These interactions show that developers acknowledge the risks posed by native extensions and are willing to mitigate it when provided with actionable reports. Below we discuss three examples illustrating security-relevant consequences that can be obtained by exploiting the identified native extension misuses.

Uninitialized memory. Let us consider a type confusion vulnerability with surprising security implications. `fast-string-search` is a package that promises to be "10 times faster than the `indexOf` function of a Node.js string". Using our automated approach, we detect missing type checks corresponding to the two arguments of the `indexOf` method. The sequence of API calls for retrieving the string from JavaScript is very similar to the one in Figure 1. Thus, if an attacker passes numbers instead of strings, a very large string length is retrieved in the first native extension call, which will then crash the process when trying to allocate a C string that large. After investigating further, we notice that the compiler reuses memory locations between API calls and hence, strings can be leaked between calls:

```
const fss = require('fast-string-search');
fss.indexOf("My password is Foo123#", "is");
let res = fss.indexOf(1, "Foo123");
console.log(res); // prints 15
```

Let us assume the call in line 2 is performed using the user's arguments, while the one in line 3 uses arguments under the attacker's control. The first call correctly returns 12, the position of the substring "is" in the larger string containing the password. The second call, though, passes an invalid string as a first argument and a password guess as second argument. In this case, the libraries reuse the argument provided in the first call, thus the result of the second call is 15 - the position of "Foo123" (passed in the second call) in the string "My password is Foo123#" (passed in the first call). Therefore, the attacker can guess parts of the user's password in this way. In general, when presented with invalid arguments, this npm package reuses uninitialized memory as input. Exploiting such bugs in real systems can have a detrimental impact on user's privacy, similar to that of vulnerabilities caused by the deprecated `Buffer` constructor[12]. We believe this is an extremely dangerous behavior for a third-party package in a scripting language. Most users of such packages are not familiar with such subtle memory-related bugs and would not expect to see them surface in their scripting language. We again stress that developers can import native extensions carelessly with `npm install fast-string-search`, most of them probably not knowing that they use native code under the hood. We also note that this bug is not possible in any of the other scripting languages we analyzed in Section 3, and it is a consequence of the permissive API design of Node.js.

Hard crash. Let us consider the following proof of concept using the modular exponentiation in the `bignum` package:

```
1 let bignum = require('bignum');
2 try {
3     bignum(10).powm(1, {});
4 } catch(e) {
5     console.log(e); // never executed
6 }
```

When this API is provided with an object literal instead of a number, it instantly crashes the Node.js process, without the possibility of recovering, e.g., error handling. Prior work reports that Node.js' parallelism relies on a single point of failure, i.e., the event loop [26, 65]. In the context of web applications, the impact of a hard crash on the server side is an instant halt of the server. This implies dropping all pending requests and hence loss of precious data. Restarting the server may take several seconds in which user requests cannot be served, seriously impacting the availability of the server. If, however, the server runs as part of a cloud platform with autoscaling, a new server instance needs to be spawned immediately and the current one has to be restarted. By sending multiple such requests, an attacker can mount a Yo-Yo attack [18], causing significant economic damage.

Memory leak. Let us consider the following example, showing a memory leak in one of the analyzed packages:

```
1 const { OpusEncoder } = require('@discordjs/opus')
2 while (1) new OpusEncoder(48000,3);
```

When monitoring the memory consumption for this simple code snippet, we observe a linear growth, indicating the lack of garbage collection. After a few seconds, the machine becomes extremely slow, and then the process crashes. While this example is artificial, if this package is used as part of a web application and such objects are created in response to user requests, the attacker can mount a memory-based DoS attack. This would, in turn, slow down or even stop the event loop like in the case above, causing a hard-to-diagnose performance problem that would impair Node.js' parallelism.

[11] snyk.io

[12] https://snyk.io/blog/exploiting-buffer/

The root cause of the problem is the ObjectWrap API[13] in Node.js (N-API), in particular the `Napi::Persistent` method, which impairs the garbage collector. Therefore, fixing the indicating problem requires migrating away from that API or from all the packages using it. We found a similar problem in the `sqlite3` package.

5.3 Impact on web applications

Vulnerabilities in native Node.js extensions motivated us to measure the problem's impact on web applications' security. In this section, first, we present our application selection criteria and pre-processing steps. Then, we discuss our findings. Note that, given a web application using a vulnerable extension, FLOWJS can be used to detect if the vulnerability can be exploited remotely. However, since it requires prior knowledge about the vulnerability, we only run experiments on the manually confirmed ones.

Application selection and pre-processing. For this experiment, we select seven most used (> 1,000 downloads) vulnerable extensions from Table 2 that can be exploited remotely: `sqlite3`, `libxml`, `bignum`, `time`, `pg-native`, `discord/opus` and `bigint-buffer`. Next, our goal is to measure their impact on web applications that are using them. For each extension, we retrieve their dependent applications from GitHub. We download at most 300 repositories per vulnerable package, consisting of in total 1,993 Node.js applications (Table 4). Note that a dependent repository does not imply a web application. However, sorting web applications from others would require manual effort, which we conservatively employ – if and only if FLOWJS reports an alert.

Our current prototype, FLOWJS, can only analyze one JavaScript file at a time. To find vulnerabilities across multiple files, FLOWJS requires the files to be merged. We use Google Closure Compiler to merge all the JavaScript files from a repository before running FLOWJS. However, setting up a repository properly for Closure, would require properly setting up the custom module dependencies, proper handling of duplicate variable declarations, etc., which requires non-trivial manual efforts. *Without such efforts, Closure successfully merged 1,141 out of 1,993 repositories.* If Closure fails to merge files for a repository, we analyze each of the files separately. Note that this is a fundamental limitation of single-pass compiler platforms like Closure that we used to implement FLOWJS. They work on the boundary of one translation unit (TU)—at the file level, and miss cross-TU flows. Solving this issue would require designing a multi-pass analysis of our own, which comes with its own challenges (similar to merging files by Closure).

Exploitable misuses as program flows. A common property of all the selected native extension APIs is that the vulnerability can be triggered if an attacker can control the input to them. In web applications, an attacker can control the request data. If unsanitized request data is passed to those APIs, then an attacker can turn the vulnerabilities into exploits. Based on this insight, we use FLOWJS to find unsanitized flows from request data to the vulnerable APIs. We report a misuse if an element of the network request directly influences the API parameter of interest. In Table 4, we provide the APIs corresponding to each rule specification.

[13] https://nodejs.org/api/n-api.html#object-wrap

Table 4: Vulnerable applications per package. TP refers to true positives.

Package	Criteria (Sinks)	#Repos	#Misuses (Total/TP)	#Exploitable (Total/TP)
`sqlite3`	`run(_, data)`	283	4/3	4/3
`libxml`	`parseXml(xml)`	296	2/2	2/2
`bignum`	`powm(_, pow)`	293	0/0	0/0
`time`	`setTimezone(tz)`	298	0/0	0/0
`pg-native`	`query(_, values, _)`	270	1/1	1/1
`discordjs/opus`	`encode(data)`	272	0/0	0/0
`bigint-buffer`	`toBigIntLE(buff)`	282	0/0	0/0
Total	–	1,993	7/6	7/6

Our findings. To find misuses corresponding to each of the selected APIs, we create the corresponding rule specification. Then, we run FLOWJS with the rule specifications on the selected GitHub repositories. Table 4 presents the summary of our experimental findings. FLOWJS reported four exploitable vulnerabilities in four applications (out of 283) using `sqlite3`, two vulnerabilities in two applications using `libxml`, and one vulnerability in one application using `pg-native`. FLOWJS did not find any vulnerabilities in any other categories. Our manual investigation shows that six out of seven alerts in six applications are true positives (Table 4). In the false positive case also, data from the request attributes are directly passed to the `sqlite3` API, however, before doing so, a type check is performed. Since our current implementation of FLOWJS is not path-sensitive, it cannot detect such type-checking constraints. Therefore, it raised an alert. We show the source code of the false positive and additional details in the full version of our paper [67, Appendix C]. We also ran FLOWJS to find how often `libxml` is used on content read from local files. Our analysis found 27 such cases, which may be security-relevant for some web applications.

Below we provide an example misuse of `sqlite3`'s `run` API, detected by our approach. This code is protected against SQL injection by the use of prepared statements. However, the vulnerability we identified in `sqlite3` allows attackers to trigger hard crashes remotely, e.g., by providing the value `{toString: 23}` for the `img` attribute of the request's body.

```
1  server.post("/", (req, res) => {
2    const {img,title,cat,desc,link} = req.body
3    const query = 'INSERT INTO ideas (image, title,
         cat, desc, link) VALUES (?,?,?,?,?)'
4    const values = [img,title,cat,desc,link]
5    db.run(query, values, function(err) {
6     if(err)
7       return res.send("Erro no banco de dados")
8      return res.redirect("/ideias")
9    })
10 })
```

Our results show that after identifying an exploitable misuse in an npm package, an adversary can further detect vulnerable endpoints of open-source web applications that rely on this package. To prevent such exploitation, it is crucial for the community to detect and patch vulnerable npm packages.

6 Discussion

Impact of security findings. Our success with identifying high-severity vulnerabilities in native extensions of popular libraries implies that even well-maintained libraries struggle with using native extensions securely. The empirical evidence clearly shows that most bugs are caused by the permissive nature of Node.js. Hence, we recommend that maintainers of this runtime reassess whether this design is in the best interest of their users.

Feasibility of the weak attacker model. Prior work [19, 29, 38, 42, 60] assumes a strong adversary that can run arbitrary high-level code to exploit bugs in the native layer. To the best of our knowledge, we are the first to propose a methodology for finding low-level vulnerabilities that can be exploited remotely by weak, web attackers. By showing the feasibility of this scenario, our findings raise concerns about a hidden attack surface of web applications written in scripting languages. Code analysis tools should challenge the assumption that low-level code is trustworthy. Additionally, the community should continuously vet packages with native extensions for both vulnerable and malicious code.

Extensibility of the prototype. The developed prototype is by no means a complete solution for identifying security problems caused by native extensions in scripting languages. There are several components that can be improved by future work. We opted in our design for a path-insensitive, intra-procedural analysis. While very scalable, this strategy results in a non-negligible number of false positives, which can be reduced by employing more sophisticated analysis techniques. Also, while this simple analysis may suffice for identifying missing type checks, it is not enough for detecting more complex problems, such as use-after-free or buffer overflow. That is because type conversions often appear at the language boundary, while unsafe buffer operations may appear anywhere in the program. While we provide initial evidence that cross-language analysis can aid analysts in the vulnerability detection task, we recognize that this method might be costly for practitioners in its current form. A taint summarization approach [12, 68] might scale better. Similar limitations exist in FLOWJS too. This is because to run large-scale analysis we traded precision and soundness for scalability. For example, FLOWJS is flow- and path-insensitive, which hurts its precision. Additionally, incorporating alias analysis in FLOWJS would result in better soundness.

Applying our methodology to other languages. We believe that our high-level methodology that emphasizes the analysis of libraries can be applied to any other scripting language. To add support to our prototype for other scripting languages, one can reuse the C/C++ extraction and the post-processing of the dot graphs, i.e., the graph traversals. However, for each scripting language, one would additionally need: (i) a data flow extraction tool that can produce graphs in the dot format, (ii) a way to identify the program locations in which the native extensions are invoked so that the cross-language graphs can be generated, and (iii) additional modeling to identify security-relevant sinks and sanitizers. While this can be done with sufficient engineering effort for all the languages studied in Section 3, we believe that our results for npm suffice for drawing conclusions about the feasibility of the methodology.

7 Related Work

Binding layer and engine issues. Vulnerabilities in JavaScript engines and in binding layer code seriously undermine the security guarantees of the language [19, 60]. Analyzing this code got a lot of traction recently [19, 21, 29, 37, 38, 50, 59, 60, 73]. The work in this domain can be categorized in two groups: fuzzing-based [1, 29, 37, 38, 50, 59] and static analysis-based [19, 21] approaches. Holler et al. proposed LangFuzz [38], which found 105 severe vulnerabilities in Mozilla's JavaScript interpreter. Given a set of seed programs, LangFuzz generates test cases by combining fragments of the seed programs. Instead of seed programs, Mozilla Security's FunFuzz [1] generates test cases from context-free grammars. The main limitation of these solutions is the lack of semantic-awareness. Han et al. [37] fixes this problem by proposing a code combining mechanism that uses a def-use analysis to find snippets with important semantic dependencies. Favocado [29] is the first fuzzing-based tool to detect binding layer bugs. Favocado extracts semantic information from the API references and uses this to generate semantic-aware test cases. Sys [21] is an analysis framework that combines static analysis and selective symbolic execution to identify low-level vulnerabilities in browser code. The most closely related work to ours is Brown et al.'s [19] approach to find binding layer issues in JavaScript runtimes. Specifically, they describe various bugs that undermine crash-, type- and memory-safety of the scripting language and propose using lightweight static checkers written in μchex [20]. By using these checkers, they detect high profile vulnerabilities in the analyzed runtimes, showing the severity of the problem. In this work we study native extensions, which democratize the access to low-level code to non-expert users. Our results confirm that many of the issues introduced by Brown et al.'s [19] are also prevalent in this new setting. However, Brown et al. [19] use a much stronger attacker model that assumes that JavaScript code is untrusted, hence there is no need for cross-language analysis. On the contrary, we assume that the JavaScript part is benign, but vulnerable. We also consider languages beyond JavaScript to understand how API design decisions can enable misuses.

Unsafe APIs uses. Almanee et al. [10] show that developers have an inertia to update vulnerable native libraries in Android apps, which consequently makes these apps vulnerable. Zimmermann et al. [76] show that the problem is prevalent in the Node.js ecosystem as well. Mastrangelo et al. [54] show that third-party library developers use unsafe Java virtual machine APIs for the sake of performance, which seriously undermines the security guarantees provided by the language. Evans et al. [32] show that the use of unsafe Rust features is widespread as well. To minimize the impact of unsafe Rust, Liu et al. propose XRust [32], which ensures data integrity by logically dividing the safe and unsafe memory allocations into two mutually exclusive regions. Studies showed that there is a widespread tendency to misuse non-native APIs as well. For example, Java developers often misuse common platform-provided library APIs, e.g., Crypto APIs [7, 31, 46, 61], SSL/TLS APIs [33], Fingerprint APIS [17], as well as non-system APIs [77].

Node.js security. Analyzing the security of the Node.js ecosystem has been a very active research field recently. Related work studies several threats in this ecosystem: regular expression denial-of-service [23–26, 65], code injections [34, 40, 66], path traversals [36], trivial packages [6, 47], prototype pollution [52, 53, 63], hidden property abuse [74], sandbox escape [8], vulnerable dependencies [28], APIs [69] and supply chain attacks [30, 76]. While not strictly Node.js-specific, the adoption of WebAssembly may also pose additional risks for the runtime [51]. Existing solutions for reducing the attack surface of web applications using third-party code include package vetting [30, 66], compartmentalization [72], and debloating [43]. Recently, Bhuiyan et al. [16] propose SecBench.js, a suite of known vulnerabilities with exploits for Node.js, but this suite does not include low-level vulnerabilities. We are the first to study in depth the native extensions' risks in Node.js' context.

Comparative analysis of scripting languages. Related work studies various security issues across multiple languages. Decan et al. [27] and Kikas et al. [41] were the first to analyze the structure of third-party dependencies in various programming languages, and draw conclusions about interesting trends. More recently, Duan et al. [30] proposes a technique for detecting supply chain attacks for the same set of scripting languages we consider in our work. As discussed in Section 6, by integrating the same data flow analysis tools used by Duan et al. [30] and by modelling additional sinks and sources, our prototype can be extended to support Python and Ruby as well. Nonetheless, we are the first to perform an in-depth security study of an equivalent API in different scripting languages.

Cross-language program analysis. Researchers study various static analysis approaches to augment the insights of non-Java code to detect cross-lingual vulnerabilities in Java and Android applications [13, 14, 22, 48, 49, 70, 71]. Nguyen *et al.* built a cross-language program slicing framework to analyze PHP, HTML and JavaScript code in the same context [56]. Brucker *et al.* propose static cross-language call graphs for hybrid JavaScript/Java mobile apps. There has been attempts to build cross-language dynamic taint analysis platforms as well [15, 44, 45]. Alimadadi *et al.* propose an approach to model temporal and behavioral information in full-stack JavaScript applications to detect cross-stack bugs [9]. Recently, Hu *et al.* [39] study the usage of native extension API in Python, while Monat *et al.* [55] propose a sophisticated static analysis for detecting cross-language bugs in Python. We are the first to perform cross-language analysis in the context of native extensions security.

8 Conclusions

In this work, we first systematically analyze the pitfalls of using native extensions in three scripting languages. We show how a failure to adhere to best practices can cause serious problems such as an exploitable buffer overflows or modifications of encapsulated values of the scripting language. We further propose a methodology to systematically detect misuses of a native extension API. By leveraging that, we show how the security problems propagate in the dependency chain, first to the enclosing library, and then to the web application relying on it. We show that many libraries fail to type check arguments coming from the scripting language and that attackers can cause uninitialized memory reads, hard crashes or memory leaks by providing well-crafted inputs to the library API. In total, we create proof-of-concept exploits in 33 real-world npm packages and show that some of the vulnerabilities could be exploited remotely in open-source web applications. This paper is first of all a warning for developers: native extensions in third-party code may violate all your assumptions about the safety of the scripting language you use. To put it more poetically, *tell me what you include, so I can tell your language guarantees.*

References

[1] FunFuzz. https://github.com/MozillaSecurity/funfuzz. Accessed April 19, 2021.

[2] Protestware on the rise: Why developers are sabotaging their own code. https://techcrunch.com/2022/07/27/protestware-code-sabotage/. Accessed: 2022-10-07.

[3] Resolving dependencies with native extensions with gem. https://guides.rubygems.org/gems-with-extensions/. Accessed: 2022-10-07.

[4] Resolving dependencies with native extensions with npm. https://docs.npmjs.com/cli/v8/commands/npm#dependencies. Accessed: 2022-10-07.

[5] Resolving dependencies with native extensions with pip. https://packaging.python.org/en/latest/tutorials/installing-packages/#source-distributions-vs-wheels. Accessed: 2022-10-07.

[6] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? An empirical case study on npm. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.

[7] Yasemin Acar, Michael Backes, Sascha Fahl, Simson L. Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the usability of cryptographic APIs. In *Symposium on Security and Privacy (S&P)*, 2017.

[8] Abdullah Alhamdan and Cristian-Alexandru Staicu. SandDriller: A fully-automated approach for testing language-based JavaScript sandboxes. In *USENIX Security Symposium*, 2023.

[9] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *International Conference on Software Engineering (ICSE)*, 2016.

[10] Sumaya Almanee, Arda Unal, and Mathias Payer. Too quiet in the library: An empirical study of security updates in Android apps' native code. 2021.

[11] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for JavaScript. *ACM Comput. Surv.*, 2017.

[12] Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the Android framework. In *International Conference on Software Engineering (ICSE)*, 2016.

[13] Steven Arzt, Tobias Kussmaul, and Eric Bodden. Towards cross-platform cross-language analysis with soot. In *International Workshop on State Of the Art in Program Analysis, (SOAP@PLDI)*, 2016.

[14] Sora Bae, Sungho Lee, and Sukyoung Ryu. Towards understanding and reasoning about Android interoperations. In *International Conference on Software Engineering (ICSE)*, 2019.

[15] Junyang Bai, Weiping Wang, Yan Qin, Shigeng Zhang, Jianxin Wang, and Yi Pan. Bridgetaint: A bi-directional dynamic taint tracking method for JavaScript bridges in Android hybrid applications. *IEEE Trans. Inf. Forensics Secur.*, 2019.

[16] Masudul Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. SecBench.js: An executable security benchmark suite for server-side JavaScript. In *International Conference on Software Engineering (ICSE)*, 2023.

[17] Antonio Bianchi, Yanick Fratantonio, Aravind Machiry, Christopher Kruegel, Giovanni Vigna, Simon Pak Ho Chung, and Wenke Lee. Broken fingers: On the usage of the fingerprint API in Android. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[18] Anat Bremler-Barr, Eli Brosh, and Mor Sides. DDoS attack on cloud auto-scaling mechanisms. In *Conference on Computer Communications (INFOCOM)*, 2017.

[19] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in JavaScript bindings. In *Symposium on Security and Privacy (S&P)*, 2017.

[20] Fraser Brown, Andres Nötzli, and Dawson R. Engler. How to build static checking systems using orders of magnitude less code. In *International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS)*, 2016.

[21] Fraser Brown, Deian Stefan, and Dawson R. Engler. Sys: A static/symbolic tool for finding good bugs in good (browser) code. In *USENIX Security Symposium*, 2020.

[22] Achim D. Brucker and Michael Herzberg. On the static analysis of hybrid mobile apps - A report on the state of Apache Cordova Nation. In *Engineering Secure Software and Systems (ESSoS)*, 2016.

[23] James C. Davis. Rethinking Regex engines to address ReDoS. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2019.

[24] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2018.

[25] James C Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service (ReDoS). 2021.

[26] James C. Davis, Eric R. Williamson, and Dongyoon Lee. A sense of time for JavaScript and Node.js: First-class timeouts as a cure for event handler poisoning. In *USENIX Security Symposium*, 2018.

[27] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *International Conference*

on Software Analysis, Evolution and Reengineering (SANER), 2017.

[28] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories (MSR)*, 2018.

[29] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[30] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[31] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Conference on Computer and Communications Security (CCS)*, 2013.

[32] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is Rust used safely by software developers? In *International Conference on Software Engineering (ICSE)*, 2020.

[33] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why Eve and Mallory love Android: an analysis of Android SSL (in)Security. In *Conference on Computer and Communications Security (CCS)*, 2012.

[34] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. AFFOGATO: Runtime detection of injection attacks for Node.js. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[35] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-Miner: Uncovering memory corruption in Linux. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[36] Liang Gong. *Dynamic Analysis for JavaScript Code*. PhD thesis, University of California, Berkeley, 2018.

[37] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[38] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, 2012.

[39] Mingzhe Hu and Yu Zhang. The Python/C API: evolution, usage statistics, and bug patterns. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020.

[40] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. Platform-independent dynamic taint analysis for JavaScript. *IEEE Transactions on Software Engineering*, 2018.

[41] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *International Conference on Mining Software Repositories (MSR)*, 2017.

[42] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Symposium on Security and Privacy (S&P)*, 2019.

[43] Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of Node.js applications. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[44] Jacob Kreindl, Daniele Bonetta, and Hanspeter Mössenböck. Towards efficient, multi-language dynamic taint analysis. In *International Conference on Managed Programming Languages and Runtimes (MPLR)*, 2019.

[45] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. Multi-language dynamic taint analysis in a polyglot virtual machine. In *International Conference on Managed Programming Languages and Runtimes (MPLR)*, 2020.

[46] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.

[47] Raula Gaikovina Kula, Ali Ouni, Daniel M. Germán, and Katsuro Inoue. On the impact of micro-packages: An empirical study of the npm JavaScript ecosystem. *CoRR*, abs/1709.04638, 2017.

[48] Sungho Lee, Julian Dolby, and Sukyoung Ryu. Hybridroid: Static analysis framework for Android hybrid applications. In *International Conference on Automated Software Engineering (ASE)*, 2016.

[49] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis. In *International Conference on Automated Software Engineering, (ASE)*, 2020.

[50] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided JavaScript engine fuzzer. In *USENIX Security Symposium*, 2020.

[51] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of WebAssembly. In *USENIX Security Symposium*, 2020.

[52] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Detecting Node.js prototype pollution vulnerabilities via object lookup analysis. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[53] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining Node.js vulnerabilities via object dependence graph and query. In *USENIX Security Symposium*, 2022.

[54] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The Java unsafe API in the wild. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[55] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A multilanguage static analysis of Python programs with native C extensions. In *International Symposium on Static Analysis (SAS)*, 2021.

[56] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Cross-language program slicing for dynamic web applications. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015.

[57] John K Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 1998.

[58] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in Soot using value contexts. In *International Workshop on State Of the Art in Java Program analysis (SOAP)*, 2013.

[59] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript engines with aspect-preserving mutation. In *Symposium on Security and Privacy (S&P)*, 2020.

[60] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. NoJITsu: Locking down JavaScript engines. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[61] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In *Conference on Computer and Communications Security (CCS)*, 2019.

[62] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013.

[63] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent spring: Prototype pollution leads to remote code execution in Node.js. In *USENIX Security Symposium*, 2023.

[64] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In *European Conference on Object-Oriented Programming (ECOOP)*, 2016.

[65] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *USENIX Security Symposium*, 2018.

[66] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. SYNODE: understanding and automatically preventing injection attacks on Node.js. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[67] Cristian-Alexandru Staicu, Sazzadur Rahaman, Ágnes Kiss, and Michael Backes. Bilingual problems: Studying the security risks incurred by native extensions in scripting languages. *arXiv preprint arXiv:2111.11169*, 2021.

[68] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. Extracting taint specifications for JavaScript libraries. In *International Conference on Software Engineering (ICSE)*, 2020.

[69] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated analysis of security-critical JavaScript APIs. In *Symposium on Security and Privacy (S&P)*, 2011.

[70] Gang Tan and Jason Croft. An empirical security study of the native code in the JDK. In *USENIX Security Symposium*, 2008.

[71] Gang Tan and Greg Morrisett. Ilea: Inter-language analysis across Java and C. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[72] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. BreakApp: Automated, flexible application compartmentalization. In *Network and Distributed System Security Symposium, (NDSS)*, 2018.

[73] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: grammar-aware greybox fuzzing. In *International Conference on Software Engineering (ICSE)*, 2019.

[74] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the Node.js ecosystem. In *USENIX Security Symposium*, 2021.

[75] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Symposium on Security and Privacy (S&P)*, 2014.

[76] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium*, 2019.

[77] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? Uncovering the data leakage in cloud from mobile apps. In *Symposium on Security and Privacy (S&P)*, 2019.

A Sinks and sanitizers used by our prototype

Below, we enumerate the sinks and sanitizers used by our prototype for detecting missing type checks. For conciseness, we use the metavariable #type# for specifying several sinks or sanitizers from the same family, e.g., instead of listing `napi_get_value_int32()` and `napi_get_value_string_utf8()`, we use the notation `napi_get_value_#type#()` to refer to APIs of that form.

Set of sinks (all in C/C++):

- `napi_get_buffer_info()`
- `Buffer::Data()`
- `Buffer::Length()`
- `*.As<#type#>`
- `*.To<#type#>`
- `*.To#type#()`
- `*.ToLocalChecked()`
- `*::Cast()`
- `napi_get_value_#type#()`

Set of sanitizers:

- (C/C++) `napi_is_#type#()`
- (C/C++) `napi_typeof()`
- (C/C++) `Nan::Check()`
- (C/C++) `*.HasInstance()`
- (C/C++) `*.Is#type#()`
- (JavaScript) `typeof`
- (JavaScript) `Buffer.isBuffer()`