

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

Cristian-Alexandru Staicu
CISPA Helmholtz Center
for Information Security
staicu@cispa.de

Sazzadur Rahaman
University of Arizona
sazz@cs.arizona.edu

Ágnes Kiss
CISPA Helmholtz Center
for Information Security
agnes.kiss@cispa.de

Michael Backes
CISPA Helmholtz Center
for Information Security
backes@cispa.de

Abstract—Scripting languages are continuously gaining popularity due to their ease of use and the flourishing software ecosystems that surround them. These languages offer crash and memory safety by design, hence developers do not need to understand and prevent most low-level security issues plaguing languages like C. However, scripting languages often allow *native extensions*, which are 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 guarantees provided by the language, e.g., crash-safety.

In this work, we first provide a comparative analysis of the security risks of the existing native extension API in three popular scripting languages. Additionally, we discuss a novel methodology for studying vulnerabilities caused by misuse of the native extension API. We then perform an in-depth study of npm, an ecosystem which appears to be the most exposed to threats introduced by native extensions. We show that vulnerabilities in extensions can be exploited in their embedding library by producing a hard crash in 30 npm packages, simply by invoking their API. Moreover, we identify five open-source web applications in which such exploits can be deployed remotely. Finally, we provide a set of recommendations for language designers, users and for the research community.

I. INTRODUCTION

Recently, modern scripting languages [49] like Python, Ruby or JavaScript are getting a lot of traction due to their versatility, ease of use, and the powerful open-source ecosystems supporting them. For example, as of March 2020, they total 7.1% of the server-side market, with a 25% increase over the previous year¹. Frameworks like Rails, Django or Express.js are an important building block for the emerging serverless computing paradigm, and various important companies rely on them for building so-called microservices, or even full-fledged web applications.

There is also a tendency to use scripting languages in several other domains, e.g., TensorFlow for machine learning in Python or Electron.js for portable desktop applications in JavaScript. This growth is supported by massive open-source ecosystems such as npm, PyPI and RubyGems. Previous work shows that there are various security risks that affect software components in these ecosystems [68], [20], [17], [57], [23],

[4] that in turn, can impact real-world websites [56]. Existing work in this domain only discusses security risks introduced by the scripting code itself and thus, ignores the important cross-language interactions in these ecosystems.

Native extensions are a convenient way to allow low-level functionality, often written in C/C++, to be directly invoked from a scripting language. Modern package management systems like npm or pip enable the smooth usage of such extensions by compiling at install time the extension's binary. At runtime, the binary is then loaded on-demand in the process of the scripting code and made available for direct invocation. In this way, the developer can expose arbitrary hardware capabilities that are beyond the reach of the scripting language, e.g., accessing various hardware ports.

One important benefit of native extensions is also the ability to reuse mature, legacy code written in low-level languages. Databases like SQLite² or cryptographic libraries like OpenSSL³ are often exposed to the scripting language using native extensions. The task of the developer in this case is to create wrappers around the native code to translate between the data representation of the two languages. There are even tools that promise to automatically generate this glue code⁴. Another important benefit of native extensions is the ability to write performance-critical code in the low-level language. For example, a non-negligible part of TensorFlow is written in C++ and exposed to the Python front-end through bindings.

One can think of native extensions as the democratization of the binding layer, which is intended for gluing the language engines with their surrounding environment, e.g., the Node.js runtime. Previous work [13], [22] discusses the security risks incurred by this layer and provides evidence that binding layer vulnerabilities are prevalent even in popular runtimes e.g., Chromium and Node.js. This type of code is usually developed by a handful of 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

¹https://w3techs.com/technologies/history_overview/programming_language

²<https://www.sqlite.org>

³<https://www.openssl.org/>

⁴<http://www.swig.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 four 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

```

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

```

Fig. 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.

typing vs. strong static typing; garbage-collected, immutable strings vs. null-terminated, self-allocated strings, etc. Moreover, a mistake in an extension may easily propagate in the ecosystem, affecting several libraries that depend on it, or even compromising production-ready applications.

Let us consider an example in Figure 1 to illustrate how bugs may arise when using native extensions and how these bugs may propagate in the ecosystem. In this example, we present a hypothetical npm package called `nativepad` that uses a native extension to pad a given string to the right with the literal "pad". The native extension depicted in Figure 1c employs four calls to the extension API: one at line 5 to retrieve the arguments, one at line 6 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 required number of characters 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 provided by the native extension. Now let us consider a client in Figure 1a that invokes the exported function with different arguments. Note that the client code 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 the `nativepad` package with a well-behaved string, e.g., "foo", the padding is performed as expected. However, when for instance, the null terminator (`\0`) is present inside the string, the native extension exposes uninitialized memory bytes. That is because this character is treated as any other character by the JavaScript runtime, i.e., counting it towards the string length, while it leads to string termination in C. Even more surprising behavior emerges, e.g., hard crash of the Node.js process if unexpected values (e.g., Booleans or certain object literals) are provided as input. Considering the no crash philosophy of JavaScript and that

uninitialized values should not leak in JavaScript space, this may surprise the users and lead to potential security incidents, e.g., compromising the availability of a web application.

While the considered example does not follow the best practices of the native extensions API, e.g., checking the argument type or the return value of the API calls, we believe that the runtime should be robust enough to protect against such *misuses*. 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 misuses are possible in each of them, though, there are important differences across languages. The Node.js API is by far the most permissive, allowing several types of misuses, 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 misuses 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 that are 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 application level.

In our evaluation, we first perform a measurement study of the native extension usage in open-source software packages. We show that packages with a native extension contain significantly more C/C++ code than JavaScript, but that a bilingual split of the code is common. We also provide justification for why developers use native extensions: e.g., reuse of legacy code and access to privileged operating system APIs. We then evaluate our methodology by studying the prevalence of misuses in 6,450 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 crash-safety violation for 30 npm packages and we identify five open-source web applications in which hard crashes can be caused remotely.

In summary, we provide the following novel contributions:

- We are the first to analyze in detail the security risks of native extensions in scripting languages. Several design decisions enable different classes of 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.

II. THREAT MODEL

We assume that the attackers do not have control over the code of the native extension, nor do they have the privilege to execute arbitrary scripting language code. That is, we consider that the developers of the extension are not malicious, but that they inadvertently introduce vulnerabilities in their code. However, we assume that attackers may control any of the arguments to a native extension, as well as their number. We consider a native extension to be 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.

Since package managers for scripting languages do not have any permission system in place, an application may transitively depend on libraries using such extensions without being aware of this fact. Attackers, thus, can exploit vulnerabilities caused by native extension in client packages. This amplification effect was previously reported for various software ecosystems [46], [25], [68].

While our threat model only considers vulnerable code, we note that native extensions may also be used to hide malicious payloads in supply chain attacks [23]. However, detecting such cases requires more sophisticated program analysis techniques than the ones employed in this work. Also noteworthy is that several extensions "vendor in" a lot of third-party code (see Section V-C for more details), hence, extending the supply chain attack surface in a stealthy way.

III. MISUSES 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. 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. [13] for JavaScript bindings, but we also add several misuses that are 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., Nan⁵ and N-API⁶, due to their prevalence in open-source projects.

In Table I, we provide an overview of our findings, along with their severity. We mark each of the misuses with a unique identifier (M_i) and will use these throughout the paper for 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 misuses. For example, none of the considered APIs prevent a crash in the native extension from compromising the availability of the application relying on it. Below, we discuss in detail each class of misuses and how they are handled by different APIs.

Error containment. As mentioned earlier, scripting languages follow a no crash philosophy in their operation. For example, in 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 can not 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
                        runtime
} catch(e) {
  // never invoked
}
```

This code snippet produces a hard crash that can not be handled in the corresponding try-catch. Such an outcome may surprise users that consider a catch clause as a universal safety net for their application. 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 (for C++ support, Ruby requires third-party libraries). Python allows C++ exceptions by default, while Node.js requires a special flag to be set.

Arguments 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 sets of as-

⁵<https://www.npmjs.com/package/nan>

⁶<https://nodejs.org/api/n-api.html>

TABLE I: 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.

Type	Id	Misuse	Node.js-N-API	Node.js-Nan	Python	Ruby	Severity
Errors	M_1	Unhandled C++ exception	●	●	●	N/a	Low
	M_2	Runtime error in C/C++	●	●	●	●	Medium
Arguments	M_3	Wrong argument type	●	●	○	○	High
	M_4	Wrong number of arguments	●	◐	○	○	High
	M_5	Different semantics for \0	●	●	○	○	High
	M_6	Overflow numeric types	●	●	○	○	High
Ret.	M_7	Missing return statement	●	○	◐	●	Low
	M_8	Void return type or null return value	○	○	◐	○	Low
Mem.	M_9	Leak uninitialized memory content	●	○	◐	○	Medium
	M_{10}	Memory leak due to cross-language pointers	○	○	●	○	Low
High-level	M_{11}	Read/write private variables	○	○	○	●	High
	M_{12}	Default blocking/sync call	●	●	●	●	Medium
Low-level	M_{13}	Buffer overflow	●	○	○	○	High
	M_{14}	Use-after-free	●	●	●	◐	High
	M_{15}	Double free	◐	◐	◐	◐	High
	M_{16}	Memory leaks	●	●	●	●	Low
	M_{17}	Format string	◐	◐	○	○	High

sumptions. In Ruby, one needs to specify the number of arguments at extension declaration time, while in Python, the API for retrieving the arguments mandate 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 [13] for an extensive discussion about the implications of breaking type safety in V8-based runtimes.

We further stress that there are 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 and throw an exception. N-API signals this by returning a non-empty status code, while Nan does not detect the mismatch.

Even when the types are correctly aligned, there are still problems caused by the different ways in which 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/C++ it marks the end of a string. Hence, if such characters are allowed to freely cross the language boundary (M_5), as it 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. 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 problems. 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, one can expose both uninitialized string values and buffers, while in Python only buffers are allowed. Ruby and Nan proactively initialize such memory areas with null bytes. Memory issues may also appear due to the garbage collector not freeing pointers to interface objects, exchanged across the language boundary (M_{10}). While all considered APIs prevent this by default, Python makes it easy to overwrite this behavior by claiming ownership of certain pointers. While this is not a problem per se, carelessly using this feature may compromise the availability of the entire application.

High-level issues. Most of the considered APIs expose only opaque pointers to the C/C++ world. That is, the native extensions can not 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 not only the modification 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 (see Section V-D for more details), it is somewhat surprising that the default behavior of all the considered APIs is to invoke the extension in a synchronous manner (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.

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 remind the reader that we observe 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 in other languages the compilation is aborted. 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 in the supplementary material of this paper⁷, exemplifying each misuse in a separate native extension, for every considered scripting language. We believe that this suite can be useful not only for users trying to understand the pitfalls of each API, but also for language designers to inform their design decisions.

Considering the presented findings, we conclude that there is a lot of variation in the implementation of native extensions in various languages. Some of the APIs put a lot of effort in 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. While they appreciated our report as informative, they argued that the identified issues are not security problems of the API, but of the packages misusing it. They promised, however, to fix some of the identified issues, e.g., the behavior responsible for M_6 . While the presented misuses 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.

IV. METHODOLOGY

While native extensions can be directly integrated in (web) applications, we believe that it is more common for these extensions to be first encapsulated in a package and then included in the application. 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 that it can be easily adapted for other scripting languages and their ecosystem. First, we run a

package-level analysis to detect vulnerable npm packages due to insecure native extensions. To that end, we advocate 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 find its impact on applications that use the package.

A. Package analysis

Since most of the 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 traversal problem on the data-flow graph. However, as we show in Section V, 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 graph is very permissive: nodes N represent program entities 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. A slightly different representation is the green part of Figure 12 in Appendix A, where some of the nodes are only partial statements. 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. It corresponds to the method definition statement in the source code, and 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, e.g., the `Buffer::Data()` call in Figure 12 for a wrong argument type vulnerability (M_3). \bar{S} is the list of sanitizers that invalidate a given flow to the sink.

Our analysis reports a security vulnerability iff:

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

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 is a pragmatic design decision that 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 for showing 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 III require such calls during the initialization of a native extension.

⁷<https://native-extension-risks.herokuapp.com/>

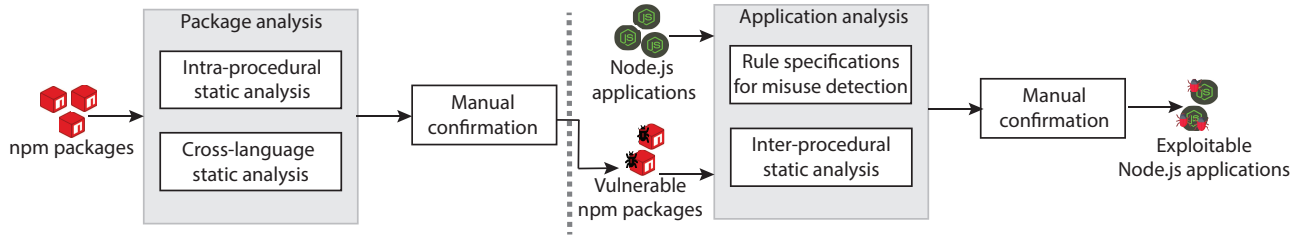


Fig. 2: Overview of our methodology for identifying native extension vulnerabilities and for studying their impact.

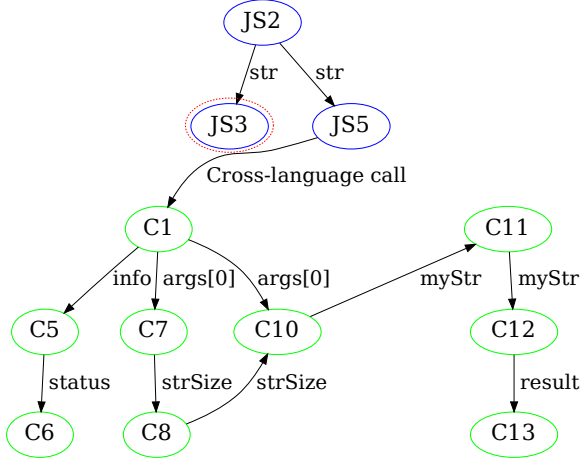


Fig. 3: A cross-language data-flow graph for our example shown in Figure 1. We depict the JavaScript nodes with blue and the C/C++ ones with green. Numbers denote line numbers in Figure 1. With red we mark a potential location for sanitization in the JavaScript front-end.

Let us assume we want to expose the `Foo` function from C/C++ to the scripting language, with the name `"foo"`. We provide the syntax used by the considered APIs for binding the two entities below:

```

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 identified 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 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 `JS3`, and would not produce a warning.

Implementation details. For extracting the data-flow graphs, we use Joern [66] for C/C++ files and Google Closure Compiler [1] for JavaScript. We instruct Joern to output the code property graph as a dot⁸ 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.

For finding the two functions at the language boundary, we first perform a simple AST-based 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 identified 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. We now present the details of performing this analysis step.

Security Modelling. Our current prototype is targeted towards studying an important subset of the misuses identified in Table I: missing type checks (M_3 - M_6). For this, we specify the list of sinks based on the APIs we misused in Section III, 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 C. As we discuss in

⁸[https://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language))

Section VI, we believe that our prototype can be extended to cover other misuses by performing additional modelling for the remaining ones. However, this requires significant engineering effort without providing additional insights about the feasibility of our methodology, which we study in this work.

B. Application analysis

The existence of naive extension vulnerabilities in npm packages motivated us to find their impact on Node.js web applications. We encountered two of the following challenges to run a large-scale analysis of this kind. First, we need to find exploitable scenarios that can be automatically detected. Second, we need to build a scalable method to detect them. To overcome the first challenge, we manually confirm that the vulnerabilities detected in npm packages are exploitable at package-level. Then, we formulate the detection of web applications using vulnerable packages as a flow problem, which can be automatically detected. To overcome the second challenge, we decided to use static analysis over dynamic analysis. This is because dynamic analysis requires running an application to monitor its runtime behavior [54], [8], [22]. Manually setting up and running a diverse set of Node.js web applications with their heterogeneous software and library dependencies are infeasible. Therefore, we built a new demand-driven, def-use based, static data-flow analysis framework for JavaScript, named FlowJS for our need. Finally, we used 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 misuses in other languages [52], [12], [55], [29]. In the rest of this section, we discuss different components of our FlowJS framework. Note that FlowJS neither guarantees soundness (i.e., absence of bugs), nor completeness (i.e., it has a potential to generate false alarm). 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.

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

Here, *IsMisuse* is the callback function that takes *P* as input and outputs *true* if a 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. To calculate data-flows at a given program point, we implemented a def-use analysis by using the 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 which utilizes the def-use relationship of variables is known as *def-use analysis*. Our def-use analysis only 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 V-G), 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 relationship 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. Then, it runs intra-procedural backward data-flow analysis on all the caller functions by 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 the inter-procedural data-flow analysis results to find misuses.

V. EVALUATION

We first discuss our research questions in Section V-A and the setup for our npm ecosystem study in Section V-B. Then present the collected empirical results in Sections V-C-V-G.

A. Research questions

We aim to answer the following research questions that help us evaluate the benefits of the proposed methodology. We investigate the presence of native extension misuses in open-source packages, and their impact on real-world web applications:

- *RQ₁*: How do npm packages use native extensions?
- *RQ₂*: Why do developers use native extensions?
- *RQ₃*: Can we automatically detect native extension misuses?
- *RQ₄*: Can attackers exploit misuses in npm packages?
- *RQ₅*: What is the impact of misuses on web applications?

While these research questions do not allow us to exhaustively study all the misuses identified in Section III, they allow us to draw important conclusions about the nature and impacts in real-world applications. *RQ₁* and *RQ₂* show the relevance of insecure native extensions in npm ecosystem and inform the community about the techniques required for automated analysis. For example, if only a handful of packages used native extensions, the relevance of the described problem would be very limited. Similarly, if packages with native extensions mostly consisted of C/C++ code, then a single-language analysis would suffice. *RQ₃*, *RQ₄* and *RQ₅* inform

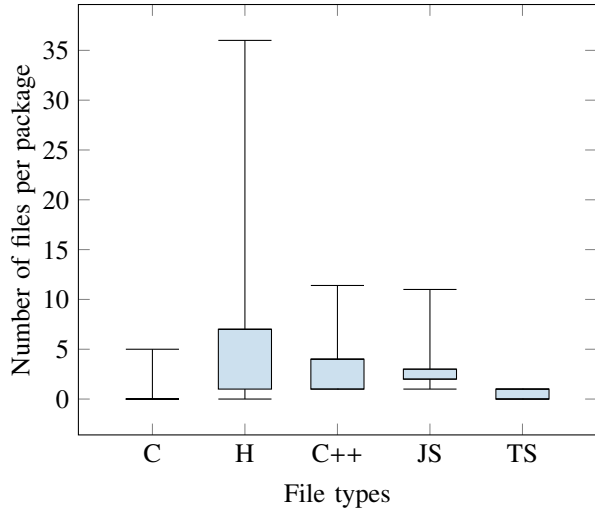


Fig. 5: The number of C/C++ and JavaScript files per package based on file types. C refers to .c files, H indicates header files with .h/.hpp extension, C++ refers to .cpp/.cc files, while JS and TS refer to .js and .ts files, respectively. The boxes indicate the lower quartile (25%) and the upper quartile (75%) and the whiskers mark the 10th and the 90th percentiles. The median should be marked with a horizontal line but it collides with the top (C, H, C++) and/or bottom (C, JS, TS) of the box in all cases.

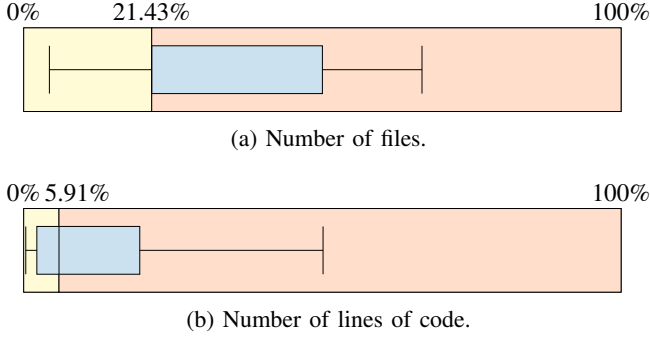


Fig. 6: The distribution of JavaScript versus C/C++ code in percentage in the studied 6,450 packages. The boxes indicate the lower quartile (25%) and the upper quartile (75%) and the whiskers mark the 10th and the 90th percentiles. The median is marked with a horizontal line and we color the background with yellow (left side of median) for JavaScript and with orange (right side of median) for C/C++ code.

D. RQ2: Reasons for usage

In this section, we study different reasons why developers may use native extensions. Understanding this helps us see how likely it is that native extensions are used in real-world web applications. We identify three main categories while looking at example npm packages that use native extensions: (1) packages that wrap operating system or low-level functionalities, (2) packages that wrap existing C/C++ libraries, and (3) packages that are implemented for enhancing performance of the JavaScript code.

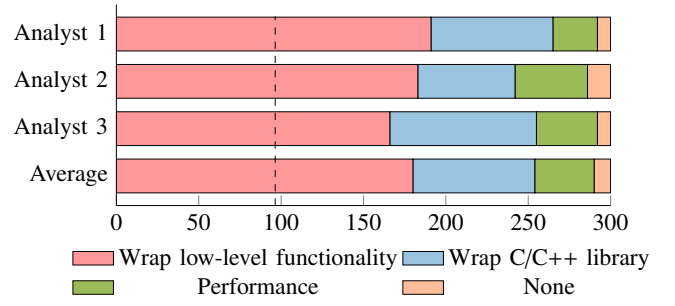


Fig. 7: Number of packages for each usage category from 300 randomly selected packages. With a dashed line we note the 104 automatically generated packages by NodeRT.

In order to provide insights on the distribution of these categories, we randomly select 300 packages out of the total number of 6,450 packages. Three authors of this paper independently labeled the selected packages, reaching moderate agreement (Cohen Kappa’s coefficient score is 0.57). We depict our findings in Figure 7. Of the 300 packages, we note that 104 were chosen from the automatically generated packages by NodeRT⁹ that provide wrappers for Windows functionalities (the package names start with `nodert-win`). This high number is due to the high prevalence of these packages in our study: there are altogether 1,883 packages generated by NodeRT in our set of 6,450 packages, i.e., 29.2%. Therefore, in Figure 7, 104 of the first category belong to these packages that wrap Windows OS functionalities (we mark these with a dashed line in the figure). Anecdotaly, the most common application domains that we see in these randomly selected packages are IoT and cryptographic libraries. Our findings and Figure 7 show that in many cases, it is not trivial to tell which category a package belongs to as it may, e.g., include an external library to enhance performance. We conclude that a user study with the package developers would have provided us with more precise results (and more refined categories), however, performing such a study is beyond the scope of this paper.

E. RQ3: Automatic misuse detection

As discussed in Section IV-A, we study the feasibility of automatic misuse detection by focusing on an important class of misuses: missing type checks (M_3 - M_6). This allows us to answer our research question, without investing tremendous engineering effort into modelling the APIs corresponding to all the misuses in Table I.

In order 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 every one of these APIs react in the same way under misuse conditions, but they all proceed with an unsafe value, i.e., they do not throw an exception.

In Figure 8, we depict the total number of packages that explicitly convert values to a given type. Casting to object, number or string types are the most prevalent conversions.

⁹<https://github.com/NodeRT/NodeRT>

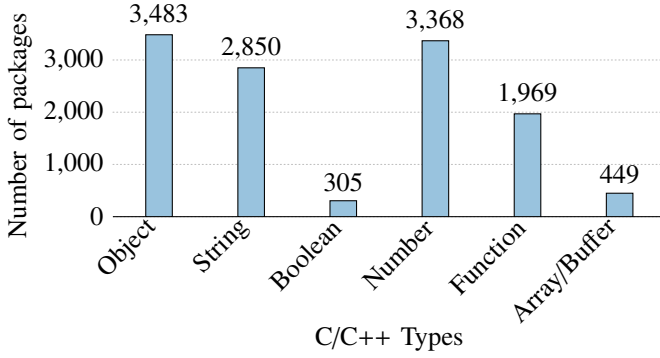


Fig. 8: Number of packages explicitly converting values to a given type.

One can observe that the majority of packages perform these conversions. The relatively low number of functions shows that at most one in three packages perform non-blocking, asynchronous operations (M_{12}). That is because these operations require a function object to be invoked upon completion.

Detecting the presence of a type conversion is not enough for estimating the number of misuses in the ecosystem. Hence, we 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 2,873 packages with type conversions, of which 1,600 have a flow to the conversion API. Of these, 962 were type checked in the native extension code, and 638 were not. The difference between the number of type conversions and the results reported in Figure 8 are due to limitations of our analysis pipeline, e.g., timeout of Joern.

Next, we concentrate on three APIs that are known to produce hard crashes on misuse, since we want to manually validate that we can exploit the misuse in practice. 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 of the work in the fuzzing domain uses a similar testing oracle. In Figure 9, 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 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 9). 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 were missed by our intra-procedural analysis (“false positives”). We then try to install 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

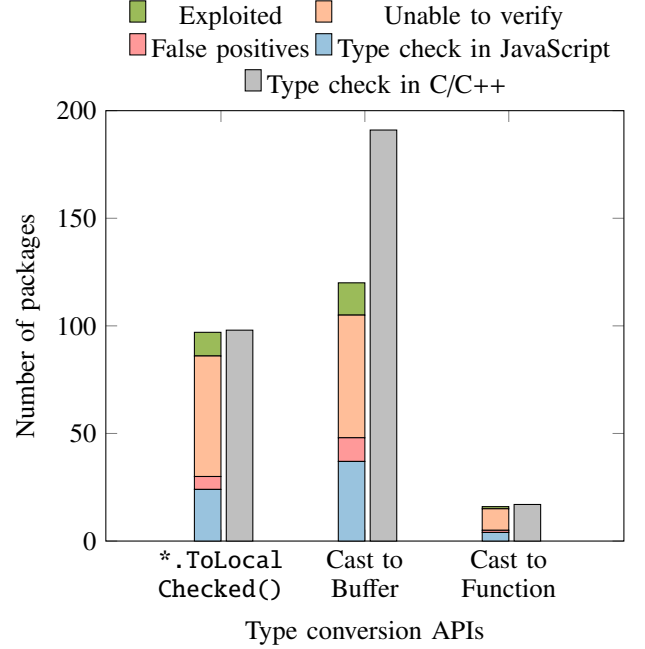


Fig. 9: Number of packages with data-flows to the type conversion APIs. The first bar represents unsanitized flows while the second bar depicts sanitized flows. For the unsanitized flows we further split our results in different categories based on our manual inspection.

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. We defer the discussion about the produced exploits to the next section, and now we continue by presenting the results of our cross-language analysis.

The main benefit of performing cross-language analysis, as described in Section IV-A, 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 9 (depicted in blue) to the grey bar. Another more subtle benefit is the improvement in user experience for the analyst. We found ourselves often 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.

We analyzed 2,372 cross-language flows, and detected 170 flows to the sink. Out of these, 81 sanitize in C/C++, 29 sanitize in JavaScript, 13 sanitize in both, and 60 do not perform any sanitization. We provide the obtained cross-language data-flow graphs in the supplementary material of this paper¹⁰ in order to increase confidence in our analysis method. Furthermore, we provide three examples in Appendix A: one for an unsanitized flow (Figure 10), one for a flow sanitized in JavaScript (Figure 11) and one for a flow sanitized both in C/C++ and JavaScript (Figure 12).

¹⁰<https://native-extension-risks.herokuapp.com/>

TABLE II: 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 ¹¹	Endpoint
64	0.0.1	4	encode
@alien.sh/signals	1.0.0	1	Register
@discordjs/opus	0.4.0	1	encode
@arunwij/read-bytes	1.0.0	1	#main#
bigint-hash	0.1.0	3	update
bignum	0.13.1	168	powm
binary-diff	1.0.0	1	#main#
ced	0.1.0	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	indexOfSkip
gs-node-lmdb	0.7.8	1	Cursor
int64-napi	1.0.1	3	divide
jitterbuffer	0.1.0	3	put
libasar_enc	1.0.0	1	#main#
libxmljs	0.19.7	237	parseXml
multi-hashing	1.0.0	10	scriptjane
node-crc	1.3.0	4	crc64iso
node-lzma	0.0.1	1	compress
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	1.0.0	1	getNativeFunction
sendto	1.0.3	1	#main#
sqlite3	5.0.1	1905	run
termios-fixedv12	0.1.8	2	getattr
time	0.12.0	56	setTimezone
zopfli-node	2.0.3	1	deflateSync

F. RQ4: Break safety of the language

In Table II we show the list of npm packages for which we could cause a hard crash, their reach in the ecosystem, and the API we used for the exploit. We reported security issues in `sqlite3` and `libxml`, the most high-profile packages. While we did not hear back from the maintainers of `libxml`, the `sqlite3` team decided to fix the problem and to make the exploit public in a GitHub issue. We are in the process of reporting the other vulnerabilities and will elaborate on the results in the final version of the paper.

Some of these packages do not compile 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.

One may argue that the majority of the identified vulnerabilities are merely “crashes in obscure, legacy packages”. Though we agree that the importance of some of these packages is limited, we believe that they show a symptom of the ecosystem, and the fact that some popular packages are in this list makes the matter even worse. With more powerful analysis tools, e.g., inter-procedural cross-language analysis, one may identify even more serious vulnerabilities.

TABLE III: Vulnerable applications per package. TP refers to true positives.

Package	Criteria	#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
time	setTimezone(tz)	298	0	0
Total	–	1,170	6/5	6/5

G. RQ5: 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.

Application selection and pre-processing. For this experiment, we selected vulnerable extensions with reach more than 50 (`sqlite3`, `libxml`, `bignum`, `time`) from Table II. Next, our goal is to find 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 1,170 Node.js applications (Table III). Note that, a dependent repository does not imply a web application. However, sorting web applications from other repositories 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. We use Google Closure Compiler to merge all the JavaScript files from a repository before running our inter-procedural data-flow analysis with FlowJS. However, during our experiment, Closure failed to merge files for 704 out of 1,170 packages. If Closure fails to merge files for a repository, we analyze each of the files separately. This has a potential to miss flows across JavaScript files.

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 a web application or its dependencies pass unsanitized request data 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 III, we provide the APIs corresponding to each rule specification.

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 III presents the summary of our experimental findings. FlowJS reported four vulnerabilities in four applications (out of 283) that are using the `sqlite3` package and two vulnerabilities in two applications that are using the `libxml` package. FlowJS did not find any vulnerabilities in any other categories. Our manual investigation shows that five out of six alerts in five applications are true

¹¹As defined by Zimmermann et al. [68].

positives (Table III). 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 Appendix B. We also ran `FlowJS` to find how often `libxml` is used on content read from local files. Our analysis found that 27 applications directly read such files, which may be security relevant for some web applications.

Below we provide an example misuse of `sqlite3`'s `run` API, detected by our approach. Note that this code is protected against SQL injection by the use of prepared statements. However, the vulnerability 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,category,description,link} = req.
    body
3   const query = 'INSERT INTO ideas (image, title,
    category, description, link) VALUES
    (?, ?, ?, ?, ?)'
4   const values = [img,title,category,description,
    link]
5   db.run(query, values, function(err) {
6     if(err) {
7       console.log(err)
8       return res.send("Erro no banco de dados")
9     }
10    //redireccionando pra pagina de ideias
11    return res.redirect("/ideias")
12  })
13  ...
14 })

```

Similarly, we show a misuse of the `parseXml` API of the `libxml` package, which was detected with `FlowJS`. Here, passing data of invalid types in the request body can trigger a hard crash.

```

1 router.post('/', function (req, res, next) {
2   var options = {
3     noent: true,
4     dtdload: true
5   }
6   var xmlDoc = libxml.parseXml(req.body, options);
7   ...
8 }

```

VI. DISCUSSION

The presented results show that misuses of native extension API can be detected automatically and even exploited remotely in web applications. However, the developed prototype is by no means a complete solution to the problem of identifying security problems caused by native extensions in scripting languages. There are several components that can be improved by future work. First, we rely mostly on intra-procedural static analysis for finding misuses in JavaScript libraries. While this 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. Second, while we provide initial evidence that cross-language analysis can aid analysts in the vulnerability detection task, we argue that

the costs of this intellectually attractive idea may be too high for practitioners. Instead, a taint summarization approach [9], [58] may scale better.

To generalize our prototype to 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 for the scripting language code, (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 modelling to identify security-relevant sinks and sanitizers. While this can be done with sufficient engineering effort for the three scripting languages studied in Section III, we believe that our results for the npm ecosystem suffice for drawing conclusions about the feasibility of the proposed methodology.

In the current evaluation, we consider all crash-safety violations to be equally harmful, while one may argue that some of them are more important than others. For example, a crash caused by an exploitable buffer overflow vulnerability is a more serious problem than a crash produced by division by zero. However, we argue that hard crashes alone are a serious enough issue, in particular in the context of web applications, where restarting a server instance can take several seconds or even minutes. Nevertheless, future work should propose techniques for identifying complex attack vectors that go beyond the crashes presented in this work.

We also believe that there are several takeaways uncovered by our work that are relevant to practitioners. First, developers should handle native extensions with utmost care and only use them when absolutely necessary, e.g., for enabling low-level hardware features. We believe that the community should discourage the use of native extensions for performance reasons and encourage safer alternatives such as `WebAssembly`. Second, API creators should opt for safe design decisions whenever possible. For instance, instead of signaling errors by returning a flag that developers may ignore, the API should avoid proceeding with erroneous data. Experience tells us that *if an API can be misused in obvious ways, it will eventually be*. If for various reasons (e.g., use of legacy APIs) safe API design is not possible, we encourage practitioners to provide compile-time support to detect API misuses. We also recommend practitioners to experiment with more radical designs, such as using fault isolation techniques against native extensions, or outsourcing the extension's computation to a different process and invoke it asynchronously.

VII. RELATED WORK

Binding layer and engine issues. Vulnerabilities in JavaScript engines and in binding layer code seriously undermine the security guarantees of the language [13], [51]. Analyzing this code got a lot of traction recently [13], [22], [15], [31], [64], [32], [50], [44], [51]. The work in this domain can be categorized in two groups: fuzzing-based [32], [2], [31], [44], [50], [22] and static analysis-based [13], [15] approaches.

Holler et al. proposed `LangFuzz` [32], 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 [2] generates test cases from context-free grammars. The main limitation of these solutions is the lack of semantic-awareness. Han et al. [31] fixes this problem by proposing a code combining mechanism that uses a def-use analysis to find snippets with important semantic dependencies. Favocado [22] 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 [15] 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 [13] 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 μ chex [14]. 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 [13] are also prevalent in this new setting.

Unsafe APIs uses. Almanee et al. [7] show that developers have an inertia to update vulnerable native libraries in Android apps, which consequently makes these apps vulnerable. Zimmermann et al. [68] show that the problem is prevalent in the Node.js ecosystem as well. Mastrangelo et al. [46] 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. [25] show that the use of unsafe Rust features is widespread as well. To minimize the impact of unsafe Rust, Liu et al. propose XRust [25], 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 [52], [24], [5], [40], SSL/TLS APIs [26], Fingerprint APIS [12], as well as non-system APIs [69].

Node.js security. Analyzing the security of the Node.js ecosystem has been a very active research field recently. Related work studies several types of threats in this ecosystem: regular expression denial-of-service [56], [17], [19], [18], [16], code injections [57], [28], [34], path traversals [30], trivial packages [4], [41], hidden property abuse [65], vulnerable dependencies [21], APIs [59] and supply chain attacks [68], [23]. While not strictly Node.js-specific, the adoption of WebAssembly may also pose additional risks for the runtime [45]. Existing solutions for reducing the attack surface of web applications using third-party code include package vetting [57], [23], compartmentalization [63], and debloating [37]. To the best of our knowledge, we are the first to study the risk of native extensions in this context.

Comparative analysis of scripting languages. Related work studies various security issues across multiple languages. Decan et al. [20] and Kikas et al. [36] 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. [23] proposes a technique

for detecting supply chain attacks for the same set of scripting languages we consider in our work. As discussed in Section VI, by integrating the same data flow analysis tools used by Duan et al. [23] 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 [61], [60], [42], [10], [11], [43]. Nguyen et al. built a cross-language program slicing framework to analyze PHP, HTML and JavaScript code in the same context [48]. There has been attempts to build cross-language dynamic taint analysis platforms as well [38], [39]. Alimadadi et al. propose an approach to model temporal and behavioral information in full-stack JavaScript applications, which can be leveraged to detect cross-stack bugs [6]. We are the first to perform cross-language analysis in the context of native extensions.

JavaScript/C static analysis. There are several open-source tools that support easily extensible static analysis for JavaScript, e.g., JSAI [35], TAJIS [33], Closure [1], and for C/C++, e.g., Joern [66], and PhASAR [53]. Our approach relies on frameworks like these to retrieve the data-flow graph for the package analysis step.

Fault isolation. There are several proposals to minimize the impact of insecure low-level code by isolating faulty components [3], [47], [62], [27], [67]. Generally speaking, fault isolation techniques add an additional layer of protection against low-level problems, which we believe can be deployed for native extensions too.

VIII. CONCLUSIONS

In this work, we first systematically analyzed 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 exploitable buffer overflow vulnerabilities or modifications of encapsulated values of the scripting language.

We then construct a methodology that shows how to automatically detect misuses of native extension API and how the security problems propagate in the dependency chain, first to the enclosing library and then to the web application relying on it. We study in detail missing type checks in native extensions on npm, an ecosystem that is particularly exposed to such problems. We show that many libraries fail to type check arguments coming from the scripting language and that attackers can cause hard crashes by providing well-crafted inputs to the library API. In total, we create a proof-of-concept crash in 30 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] “Closure compiler,” <https://developers.google.com/closure/compiler>, accessed April 19, 2021.
- [2] “Funfuzz,” <https://github.com/MozillaSecurity/funfuzz>, accessed April 19, 2021.
- [3] C. Abate, A. A. de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hritcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach, “When good components go bad: Formally secure compilation despite dynamic compromise,” in *Conference on Computer and Communications Security (CCS)*, 2018.
- [4] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, “Why do developers use trivial packages? an empirical case study on npm,” in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.
- [5] Y. Acar, M. Backes, S. Fahl, S. L. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, “Comparing the usability of cryptographic apis,” in *Symposium on Security and Privacy (S&P)*, 2017.
- [6] S. Alimadadi, A. Mesbah, and K. Pattabiraman, “Understanding asynchronous interactions in full-stack JavaScript,” in *International Conference on Software Engineering (ICSE)*, 2016.
- [7] S. Almanee, A. Unal, and M. Payer, “Too quiet in the library: An empirical study of security updates in android apps’ native code,” 2021.
- [8] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C. Staicu, “A survey of dynamic analysis and test generation for javascript,” *ACM Comput. Surv.*, vol. 50, no. 5, pp. 66:1–66:36, 2017.
- [9] S. Arzt and E. Bodden, “Stubdroid: automatic inference of precise data-flow summaries for the android framework,” in *International Conference on Software Engineering (ICSE)*, 2016.
- [10] S. Arzt, T. Kussmaul, and E. Bodden, “Towards cross-platform cross-language analysis with soot,” in *International Workshop on State Of the Art in Program Analysis (SOAP@PLDI)*, 2016.
- [11] S. Bae, S. Lee, and S. Ryu, “Towards understanding and reasoning about android interoperations,” in *International Conference on Software Engineering (ICSE)*, 2019.
- [12] A. Bianchi, Y. Fratantonio, A. Machiry, C. Kruegel, G. Vigna, S. P. H. Chung, and W. Lee, “Broken fingers: On the usage of the fingerprint API in android,” in *Network and Distributed System Security Symposium (NDSS)*, 2018.
- [13] F. Brown, S. Narayan, R. S. Wahby, D. R. Engler, R. Jhala, and D. Stefan, “Finding and preventing bugs in javascript bindings,” in *Symposium on Security and Privacy (S&P)*, 2017.
- [14] F. Brown, A. Nötzli, and D. 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.
- [15] F. Brown, D. Stefan, and D. R. Engler, “Sys: A static/symbolic tool for finding good bugs in good (browser) code,” in *USENIX Security Symposium*, 2020.
- [16] J. C. Davis, “Rethinking regex engines to address redos,” in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2019.
- [17] J. C. Davis, C. A. Coghlan, F. Servant, and D. 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.
- [18] J. C. Davis, F. Servant, and D. Lee, “Using selective memoization to defeat regular expression denial of service (redos),” 2021.
- [19] J. C. Davis, E. R. Williamson, and D. 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.
- [20] A. Decan, T. Mens, and M. Claes, “An empirical comparison of dependency issues in OSS packaging ecosystems,” in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.
- [21] A. Decan, T. Mens, and E. Constantinou, “On the impact of security vulnerabilities in the npm package dependency network,” in *International Conference on Mining Software Repositories (MSR)*, 2018.
- [22] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. 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.
- [23] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, “Towards measuring supply chain attacks on package managers for interpreted languages,” in *Network and Distributed System Security Symposium (NDSS)*, 2021.
- [24] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, “An empirical study of cryptographic misuse in android applications,” in *Conference on Computer and Communications Security (CCS)*, 2013.
- [25] A. N. Evans, B. Campbell, and M. L. Soffa, “Is rust used safely by software developers?” in *International Conference on Software Engineering (ICSE)*, 2020.
- [26] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben, “Why Eve and Mallory love Android: an analysis of Android SSL (in)Security,” in *Conference on Computer and Communications Security (CCS)*, 2012.
- [27] T. Frassetto, P. Jauernig, C. Liebchen, and A. Sadeghi, “IMIX: in-process memory isolation extension,” in *USENIX Security Symposium*, 2018.
- [28] F. Gauthier, B. Hassanshahi, and A. Jordan, “AFFOGATO: runtime detection of injection attacks for node.js,” in *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.
- [29] D. Gens, S. Schmitt, L. Davi, and A. Sadeghi, “K-miner: Uncovering memory corruption in linux,” in *Network and Distributed System Security Symposium (NDSS)*, 2018.
- [30] L. Gong, “Dynamic analysis for javascript code,” Ph.D. dissertation, University of California, Berkeley, 2018.
- [31] H. Han, D. Oh, and S. K. Cha, “Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines,” in *Network and Distributed System Security Symposium (NDSS)*, 2019.
- [32] C. Holler, K. Herzig, and A. Zeller, “Fuzzing with code fragments,” in *USENIX Security Symposium*, 2012.
- [33] S. H. Jensen, A. Møller, and P. Thiemann, “Type analysis for javascript,” in *International Symposium on Static Analysis (SAS)*, 2009.
- [34] R. Karim, F. Tip, A. Sochurkova, and K. Sen, “Platform-independent dynamic taint analysis for JavaScript,” *IEEE Transactions on Software Engineering*, 2018.
- [35] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Saracino, B. Wiedermann, and B. Hardekopf, “JSAI: a static analysis platform for javascript,” in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2014.
- [36] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, “Structure and evolution of package dependency networks,” in *International Conference on Mining Software Repositories (MSR)*, 2017.
- [37] I. Koishybayev and A. Kapravelos, “Mininode: Reducing the attack surface of node.js applications,” in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.
- [38] J. Kreindl, D. Bonetta, and H. Mössenböck, “Towards efficient, multi-language dynamic taint analysis,” in *International Conference on Managed Programming Languages and Runtimes (MPLR)*, 2019.
- [39] J. Kreindl, D. Bonetta, L. Stadler, D. Leopoldseider, and H. Mössenböck, “Multi-language dynamic taint analysis in a polyglot virtual machine,” in *International Conference on Managed Programming Languages and Runtimes (MPLR)*, 2020.
- [40] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, “Crysl: An extensible approach to validating the correct usage of cryptographic apis,” in *European Conference on Object-Oriented Programming (ECOOP)*, 2018.
- [41] R. G. Kula, A. Ouni, D. M. Germán, and K. Inoue, “On the impact of micro-packages: An empirical study of the npm javascript ecosystem,” *CoRR*, vol. abs/1709.04638, 2017.
- [42] S. Lee, J. Dolby, and S. Ryu, “Hybridroid: static analysis framework for android hybrid applications,” in *International Conference on Automated Software Engineering (ASE)*, 2016.
- [43] S. Lee, H. Lee, and S. 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.

- [44] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided javascript engine fuzzer," in *USENIX Security Symposium*, 2020.
- [45] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *USENIX Security Symposium*, 2020.
- [46] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at your own risk: the Java unsafe API in the wild," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.
- [47] S. Narayan, C. Disselkoben, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the firefox renderer," in *USENIX Security Symposium*, 2020.
- [48] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Cross-language program slicing for dynamic web applications," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015.
- [49] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *IEEE Computer*, 1998.
- [50] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *Symposium on Security and Privacy (S&P)*, 2020.
- [51] T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz, "Nojitsu: Locking down javascript engines," in *Network and Distributed System Security Symposium (NDSS)*, 2020.
- [52] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects," in *Conference on Computer and Communications Security (CCS)*, 2019.
- [53] P. D. Schubert, B. Hermann, and E. Bodden, "PhASAR: An interprocedural static analysis framework for C/C++," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2019.
- [54] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs, "Jalangi: a selective record-replay and dynamic analysis framework for javascript," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 488–498.
- [55] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java," in *European Conference on Object-Oriented Programming (ECOOP)*, 2016.
- [56] C. Staicu and M. Pradel, "Freezing the web: A study of redos vulnerabilities in javascript-based web servers," in *USENIX Security Symposium*, 2018.
- [57] C. Staicu, M. Pradel, and B. Livshits, "SYNODE: understanding and automatically preventing injection attacks on NODE.JS," in *Network and Distributed System Security Symposium (NDSS)*, 2018.
- [58] C. Staicu, M. T. Torp, M. Schäfer, A. Möller, and M. Pradel, "Extracting taint specifications for javascript libraries," in *International Conference on Software Engineering (ICSE)*, 2020.
- [59] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated analysis of security-critical javascript apis," in *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011, pp. 363–378.
- [60] G. Tan and J. Croft, "An empirical security study of the native code in the JDK," in *USENIX Security Symposium*, 2008.
- [61] G. Tan and G. Morrisett, "Ilea: inter-language analysis across java and c," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.
- [62] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: secure, efficient in-process isolation with protection keys (MPK)," in *USENIX Security Symposium*, 2019.
- [63] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization," in *Network and Distributed System Security Symposium (NDSS)*, 2018.
- [64] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: grammar-aware greybox fuzzing," in *International Conference on Software Engineering (ICSE)*, 2019.
- [65] F. Xiao, J. Huang, Y. Xiong, G. Yang, H. Hu, G. Gu, and W. Lee, "Abusing hidden properties to attack the node.js ecosystem," in *USENIX Security Symposium*, 2021.
- [66] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Symposium on Security and Privacy (S&P)*, 2014.
- [67] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Symposium on Security and Privacy (S&P)*, 2009.
- [68] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *USENIX Security Symposium*, 2019.
- [69] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Symposium on Security and Privacy (S&P)*, 2019.

APPENDIX

A. Example cross-language data-flow graphs

In this section, we provide examples for cross-language data-flow graphs for different scenarios. Additionally, we provide more complex examples in the supplementary material of this paper¹².

In Figure 10, we provide a small example unsanitized flow from the buffer parameter in JavaScript to the type conversion API `Buffer::Data(*)` in C++ in package `zopfli-node`, version 2.0.3. In this figure, it can be observed that the buffer parameter flows into `Buffer::Data(*)` in C++ with no sanitization.

In Figure 11, we provide an example sanitized flow from the input parameter to the type conversion API `Buffer::Data(*)` in C++ in package `iltorb`, version 1.0.0. We mark the node where sanitization takes place with red and note that in this example, the sanitization is carried out in the JavaScript front-end.

In Figure 12, we provide a somewhat complex sanitized flow from the rounds parameter to the type conversion API `Int32Value()` in C++ in package `nan-bcrypt`, version 0.7.7. We mark the nodes where sanitization takes place with red and note that in this example, sanitization is carried out both in the JavaScript front-end and in the C++ native extension code.

B. False positive produced by FlowJS

In this section, we provide additional details about the false positive detected by FlowJS. In Listing 1 we show the relevant source code for the flagged application. The application processes seven attributes of the post request's body by saving them in the `sqlite3` database. To deploy our payload identified in Section V-F, we need to pass a specially-crafted **object** as an entry in the array passed as the second argument to `sqlite3.run()`. The type conversions in line 5, 6, 7, 8, and the type checks in line 27, 34, 41 ensure that only numbers and strings are allowed, respectively. The presented false positive is caused by limitations in our current prototype, and not by fundamental shortcomings of the presented approach.

¹²<https://native-extension-risks.herokuapp.com>

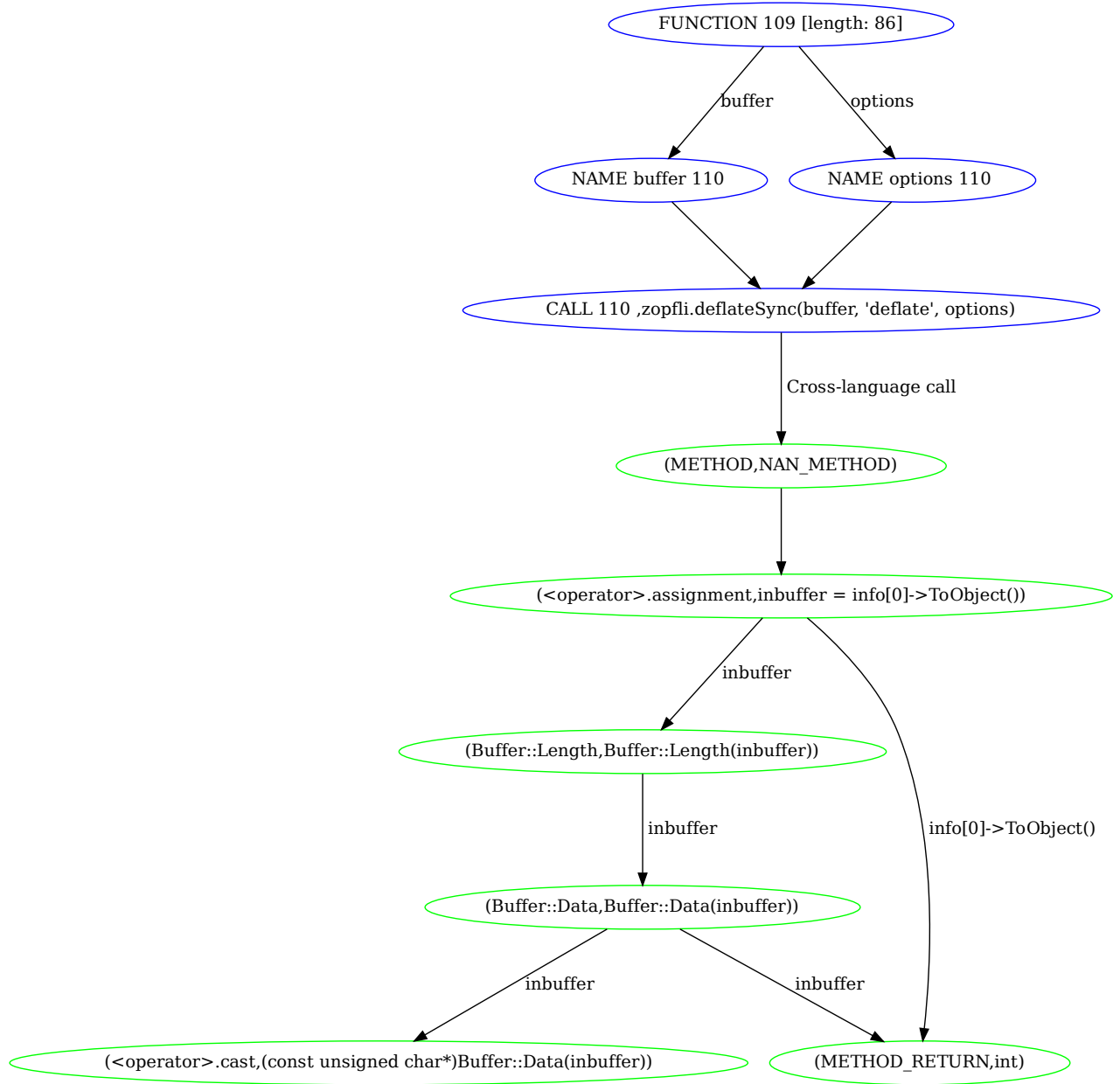


Fig. 10: A cross-language data-flow graph for the package `zopfli-node`, version 2.0.3. The graph corresponds to the method `CompressBinding::Sync` in C++ and `lib/zopfli.js` in JavaScript. With blue we show the JavaScript nodes and with green the C++ ones. The graph shows an unsanitized data-flow from the `buffer` parameter in JavaScript to the type conversion API `Buffer::Data(*)` in C++.

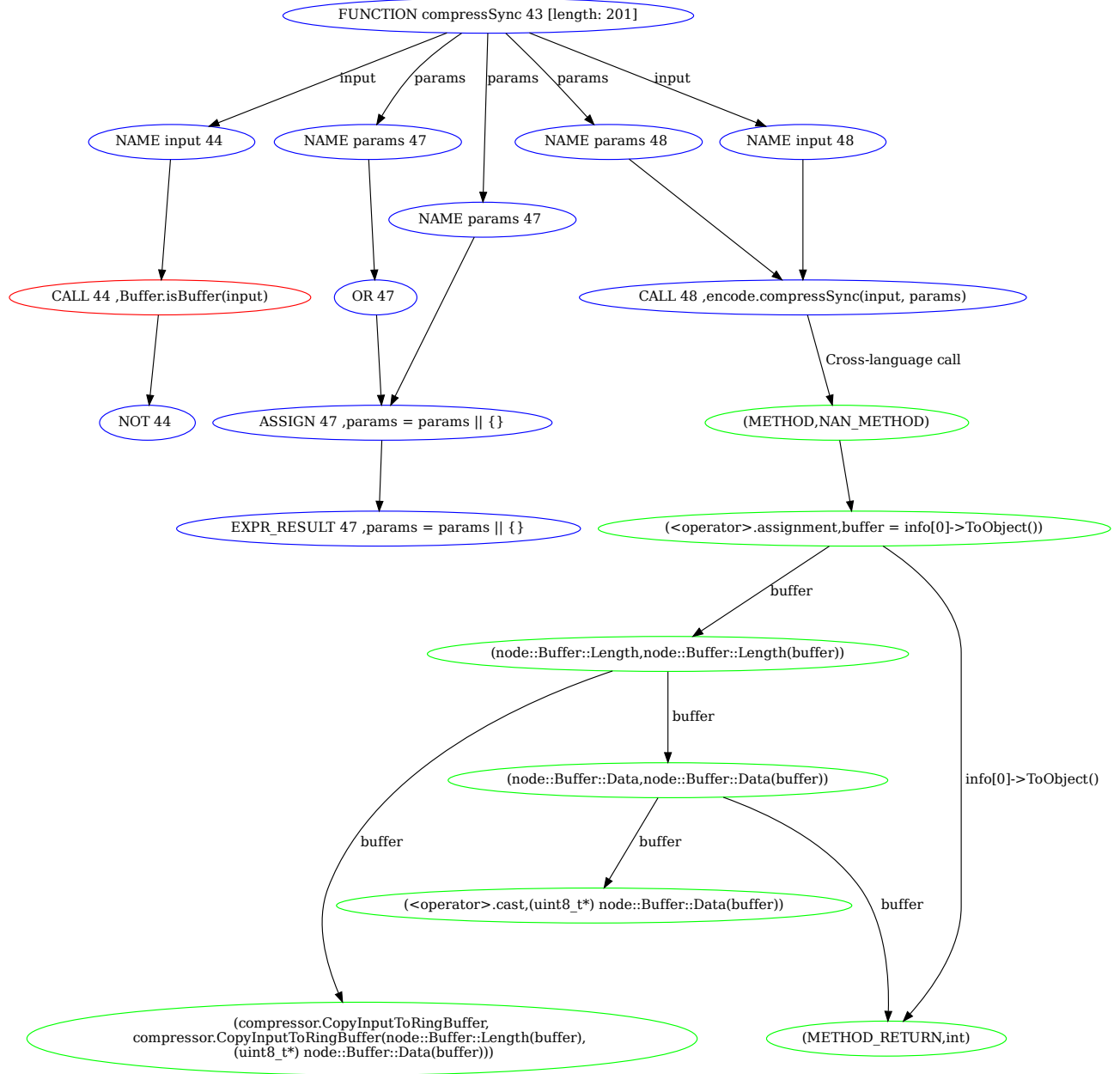


Fig. 11: A cross-language data-flow graph for the package iltorb, version 1.0.0. The graph corresponds to the method `compressSync` in C++ and `index.js` in JavaScript. With blue we show the JavaScript nodes and with green the C++ ones. The graph shows a sanitized data-flow from the input parameter in JavaScript to the type conversion API `Buffer::Data(*)` in C++. The sanitizer is marked with red and it can be found in the JavaScript part of the code.


```

1  module.exports = (db) => {
2      app.get('/health', (req, res) => res.send('Healthy'));
3
4      app.post('/rides', jsonParser, (req, res) => {
5          const startLatitude = Number(req.body.start_lat);
6          const startLongitude = Number(req.body.start_long);
7          const endLatitude = Number(req.body.end_lat);
8          const endLongitude = Number(req.body.end_long);
9          const riderName = req.body.rider_name;
10         const driverName = req.body.driver_name;
11         const driverVehicle = req.body.driver_vehicle;
12
13         if (startLatitude < -90 || startLatitude > 90 || startLongitude < -180 || startLongitude > 180) {
14             return res.send({
15                 error_code: 'VALIDATION_ERROR',
16                 message: 'Start latitude and longitude must be between -90 - 90 and -180 to 180 degrees
17                     respectively'
18             });
19         }
20
21         if (endLatitude < -90 || endLatitude > 90 || endLongitude < -180 || endLongitude > 180) {
22             return res.send({
23                 error_code: 'VALIDATION_ERROR',
24                 message: 'End latitude and longitude must be between -90 - 90 and -180 to 180 degrees
25                     respectively'
26             });
27         }
28
29         if (typeof riderName !== 'string' || riderName.length < 1) {
30             return res.send({
31                 error_code: 'VALIDATION_ERROR',
32                 message: 'Rider name must be a non empty string'
33             });
34         }
35
36         if (typeof driverName !== 'string' || driverName.length < 1) {
37             return res.send({
38                 error_code: 'VALIDATION_ERROR',
39                 message: 'Rider name must be a non empty string'
40             });
41         }
42
43         if (typeof driverVehicle !== 'string' || driverVehicle.length < 1) {
44             return res.send({
45                 error_code: 'VALIDATION_ERROR',
46                 message: 'Rider name must be a non empty string'
47             });
48         }
49
50         var values = [req.body.start_lat, req.body.start_long, req.body.end_lat, req.body.end_long, req.body.
51             rider_name, req.body.driver_name, req.body.driver_vehicle];
52
53         const result = db.run('INSERT INTO Rides(startLat, startLong, endLat, endLong, riderName, driverName,
54             driverVehicle) VALUES (?, ?, ?, ?, ?, ?, ?)', values, function (err) {
55             if (err) {
56                 return res.send({
57                     error_code: 'SERVER_ERROR',
58                     message: 'Unknown error'
59                 });
60             }
61         });
62     }
63 }

```

Listing 1: False positive produced by FlowJS. The type conversions in line 5, 6, 7, 8, and the type checks in line 27, 34, 41 prevent the deployment of remote type confusion payloads.

C. 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 mentioning both `napi_get_value_int32()` and `napi_get_value_string_utf8()`, we use the notation `napi_get_value_#type#()` to refer to all 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()`