Best of Both Worlds: Effective Foreign Bridge Identification in V8 Embedders for Security Analysis

Authors' version; to appear in the Proceedings of the IEEE Symposium on Security and Privacy (S&P) 2026

Georgios Alexopoulos*†‡, Thodoris Sotiropoulos*§, Zhendong Su§, and Dimitris Mitropoulos†‡
*these authors contributed equally
†University of Athens, {grgalex, dimitro}@ba.uoa.gr
‡National Infrastructures for Research and Technology, Greece
§ETH Zurich, {theodoros.sotiropoulos, zhendong.su}@inf.ethz.ch

***Abstract*—Modern JavaScript applications increasingly rely on native extensions and WebAssembly modules for performance-critical functionality. This multi-language architecture, however, introduces attack surfaces in native code that may be exploitable via JavaScript interfaces. Effective cross-language security analysis depends on accurately identifying *bridges*, i.e., connection points where JavaScript functions delegate to native or WebAssembly code. Yet, existing analysis approaches suffer from incomplete coverage and false positives due to the diversity of foreign function interfaces in JavaScript.**

We present GASKET, a novel dynamic analysis tool that effectively identifies bridges between JavaScript and low-level code. The key insight behind GASKET is that regardless of the binding framework or runtime used, all function objects are ultimately represented as uniform internal structures within the JavaScript engine (V8). By analyzing the memory layout of these structures, GASKET effortlessly identifies the native or WebAssembly functions bound to JavaScript high-level constructs, even across diverse binding frameworks and execution environments. Our evaluation demonstrates that GASKET achieves perfect recall while incurring no false positives. When integrated with existing security tools across 1,266 npm packages, GASKET enables the detection of 54 additional vulnerable flows in 23 packages that were missed by prior approaches due to incomplete bridge identification. Among these flows, 19 are confirmed to be exploitable vulnerabilities. Our systematic analysis of vulnerability reachability across ∼2K dependent packages reveals that while many packages depend on vulnerable native extensions, only a small fraction actually invoke vulnerable functions. This allows for reduced alert fatigue and actionable security insights.

1. Introduction

Modern software development increasingly relies on multi-language applications that combine high-level programming languages with native code to achieve optimal performance and code reuse [1], [2], [3], [4]. JavaScript applications exemplify this trend, frequently incorporating native extensions written in C [5], C++ [6], or Rust [7], as well as WebAssembly [8] modules. This polyglot approach enables developers to harness the expressiveness of JavaScript while accessing performance-critical functionality and established native libraries. However, this choice comes with significant security challenges [9], [10].

The security implications of multi-language architectures extend beyond individual applications, impacting software supply chain security [11], [12], [13]. In the JavaScript ecosystem, native-backed packages create complex, cross-language dependency graphs [14], [15], [16]. This complexity hinders both vulnerability detection and reachability analysis. Recent studies have shown that native extensions are prone to critical issues, such as memory- and type-unsafe operations [9], [14], while at the same time, traditional dependency scanning tools that operate at the package level often miss whether vulnerable native functions are actually reachable from JavaScript code. As a result, existing security tools suffer from high false-positive and false-negative rates, undermining developer trust in security alerts.

Cross-language program analysis presents several challenges, particularly in identifying the precise connection points—or *bridges*—between different languages. These bridges represent the critical junctures where control flow and data flow transition between language boundaries. In JavaScript applications with native or WebAssembly extensions, these bridges often take the form of JavaScript function objects that delegate execution to low-level code. Accurately identifying these bridges is important to numerous security applications, including vulnerability detection [14], [17], reachability analysis [18], attack surface analysis [19], and runtime protection mechanisms [15], [20].

Existing cross-language security analyzers, such as CHARON [14], rely on static analysis techniques to identify JavaScript-native bridges by examining binding framework APIs and native code patterns. However, these approaches suffer from a number of limitations: they support only a subset of the pathways and foreign function interfaces, while producing false positives due to over-approximation, and missing bridges due to the dynamic nature of modern JavaScript applications. Similarly, runtime protection systems such as NATISAND [15] and HODOR [16] require accurate bridge information to enforce security policies across language boundaries. The effectiveness and scope of these

tools are directly constrained by the effectiveness of bridge identification mechanisms.

Approach: We introduce a dynamic analysis technique that overcomes the limitations of existing static analysis tools by identifying cross-language bridges directly at the JavaScript engine level. The core insight behind our approach is that regardless of the binding framework, runtime, or implementation language, all JavaScript function objects are ultimately represented as internal structures in the underlying JavaScript engine (V8 in our case). Using this insight, our technique analyzes the memory layout of these internal structures and reliably recovers the native or WebAssembly functions bound to high-level JavaScript functions. To do so, it exhaustively examines loaded modules and extracts the memory addresses of functions exposed within them.

We have implemented our approach in GASKET, a runtime-agnostic tool that operates across Node.js, Deno, and Chromium environments. Our evaluation demonstrates that GASKET achieves perfect recall across twenty packages representing ten different binding frameworks and runtime environments, while incurring no false positives.

The practical impact of our work extends to real-world security applications. By integrating GASKET with existing security tools, we demonstrate significant improvements in identifying native-specific vulnerabilities in JavaScript applications. Specifically, when we replace CHARON's static bridge identification mechanism with GASKET, the enhanced system detects 2.5× more bridges. This ultimately leads to the detection of 54 vulnerable flows in 23 packages that are entirely missed by prior work. We also showcase that GASKET can enable effective cross-language reachability analysis. Our findings suggest, while many `npm` packages depend on vulnerable versions of native extensions, a small fraction actually invoke vulnerable functions, enabling more precise vulnerability prioritization and reduced alert fatigue.

Contributions.: We make the following contributions:

- We show that despite the heterogeneity of JavaScript with respect to foreign function interfaces, all bindings ultimately result in a common memory abstraction.
- Building on this observation, we introduce GASKET, the first general-purpose dynamic analysis framework for identifying bridges between JavaScript and native / Wasm code. Unlike prior work, GASKET seamlessly supports bindings implemented in various languages, created with different binding frameworks, and executed across multiple runtimes (e.g., Node.js, Deno, Chromium).
- We integrate GASKET into the CHARON vulnerability detection tool and show that it enables the discovery of 54 additional vulnerable flows across 23 new packages that are missed by CHARON. Among them, 19 vulnerabilities across 12 packages are confirmed exploitable.
- We provide the first systematic evaluation of vulnerability reachability in JavaScript packages with native extensions, analyzing 10 known vulnerabilities across 2,197 dependent packages and demonstrating significant reduction in false positive security alerts.
- We provide GASKET as an open-source tool that can be integrated with existing security analysis frameworks, enabling researchers and practitioners to benefit from improved cross-language analysis capabilities.

Remark: Our tool will be made publicly available as open source upon publication. We have followed responsible disclosure practices, notifying affected vendors within the recommended time window. Final disclosure information will be included in the published version.

2. Background and Problem Definition

2.1. Native Extensions and Security Implications

The JavaScript ecosystem exhibits *multi-execution environments*. JavaScript engines (e.g., V8), along with their embedders (e.g., Node.js, Deno, or Chromium), allow developers to extend their JavaScript programs with native code or interface with other environments, such as WebAssembly (Wasm). In this setting, many JavaScript packages deployed on `npm` combine JavaScript source files (.js) either with (1) native extensions compiled from low-level languages such as C, C++, or Rust, or (2) with Wasm modules.

Consider the example in Figure 1, which shows code fragments from a widely used `npm` package `sqlite3`. This package provides an asynchronous JavaScript API for interacting with the SQLite database engine. A client program (Figure 1a) can create a database and execute SQL statements asynchronously (e.g., lines 4–6). To run a given SQL statement, the `sqlite3` package internally calls `Statement.prototype.run` (Figure 1b; line 5), which is implemented in native code. Specifically, the implementation of `Statement.prototype.run` is given by a C++ method called `Statement::Run` (Figure 1c; line 10), which is compiled into the shared library `node_sqlite3.node` and subsequently imported into the JavaScript program just like a regular module (Figure 1b; line 1). The connection between JavaScript and native code is established using a binding framework (in this case, it is the `node-addon-api` framework [21]) which exposes native functions as JavaScript-accessible objects (Figure 1d).

Vulnerability detection: The native implementation of `Statement.prototype.run` internally calls a C++ method named `Bind` (Figure 1c; line 12). This method is used to attach user-provided parameters to an SQL statement. For example, in the client code of Figure 1a (line 5), the object `{toString: 23}` is bound to an `INSERT` statement. The `Bind` function iterates over each given parameter and invokes `BindParameter` (Figure 1c; lines 15–20), which attempts to convert the provided parameters into strings via the V8-specific method `v8::Value::ToString()` (line 17).

The problem in the implementation of `BindParameter` is that it incorrectly assumes the `toString` property of the given object is always a function. Indeed, this assumption does not hold for the program of Figure 1a, as the given object carries a property `toString` that is a number, not a

```
1  // client.js
2  const sqlite3 = require('sqlite3');
3  const db = new sqlite3.Database('test.db');
4  db.serialize(function() {
5    db.run("INSERT INTO t VALUES (?)", [{toString:
         23}]);
6  });
```

(a) A client JS program that uses the `node-sqlite3` package.

```
1  // sqlite3.js
2  const sqlite3 = require("node_sqlite3.node");
3  Database.prototype.run = function(sql, params) {
4    const stmt = new sqlite3.Statement(sql);
5    stmt.run.apply(params);
6  }
```

(b) JS code in `sqlite3` that delegates execution to C++ code.

```
1  template <class T> T* Statement::Bind(
2      Napi::CallbackInfo& info,
3      int start
4  ) {
5    Napi::Array array = info[start].As<Napi::Array>();
6    int length = array.Length();
7    ...
8    BindParameter(array.Get(i), pos)
9    }
10 Napi::Value Run(Napi::CallbackInfo &info) {
11   Statement *smt = this;
12   Baton *baton = stmt->Bind<RowBaton>(info);
13   ...
14 }
15 Field* BindParameter(Napi::Value s, int pos) {
16   if (source.IsObject()) {
17     std::string val = s.ToString().Utf8Value();
18     return new Text(pos, val.length(), val.c_str());
19   }
20 }
```

(c) C++ implementation of `Statement::Run` in `node-sqlite3`.

```
1  #include <napi.h>
2  #include <statement.h>
3  // C++ bindings
4  Napi::Object Statement::Init(Napi::Env, Napi::Object
       exports) {
5    auto t = DefineClass(env, "Statement", {
6      InstanceMethod("run", &Statement::Run),
7      InstanceMethod("get", &Statement::Get),
8      InstanceMethod("bind", &Statement::Bind),
9      InstanceMethod("all", &Statement::All),
10   ...
11   });
12 }
13
14 Napi::Object RegisterModule(Nap::Env, Napi::Object
       exports) {
15   Statement::Init(env, exports);
16   ...
17   return exports;
18 }
19 NODE_API_MODULE(node_sqlite3, RegisterModule);
```

(d) C++ bindings in `sqlite3`.

Figure 1: Example JavaScript client program (client.js; Figure 1a) invoking a vulnerable `sqlite3` package version with native extensions. Our approach accurately identifies the bridge from JavaScript to C++ code, enabling: (1) vulnerability detection tools to identify a type-unsafe flow from the public API of the package to a `toString` call (Figure 1c; line 17), and (2) vulnerability reachability analysis tools to determine whether a dependent package (client.js) transitively calls the vulnerable native function in its `sqlite3` dependency.

function. This causes a hard crash in Node.js when invoking `toString()` on line 17 (Figure 1c) which in turn can lead to a denial-of-service (DoS) attack.

Threat model: In the context of this work we consider attacker-controlled inputs to JavaScript applications that utilize either native extensions or WebAssembly modules. Specifically, the attacker's goal is to exploit vulnerabilities in native code [9], [14] (e.g., buffer overflows, format string attacks, integer overflows) by crafting inputs that flow from JavaScript through binding interfaces to vulnerable native functions. We assume the JavaScript runtime (V8) and underlying operating system are trusted, while third-party packages with native extensions are not trusted.

Reachability analysis: Now assume that the vulnerability described above has been discovered (e.g., by a bug-finding tool [9], [14]), patched by adding a check that ensures `toString` is a valid function [22] and assigned a CVE (CVE-2022-21227 [23]). A key follow-up task is assessing whether downstream client packages are affected this CVE. While platforms such as GitHub and `npm` provide advisories, they typically operate at package level, without checking whether the vulnerable function is actually reachable, leading to false positives. To address this, industry tools increasingly rely on fine-grained call graph analysis [24] to determine whether vulnerable dependency functions are transitively reachable from client code.

However, doing so in our example requires the construction of cross-language call graph that captures the caller-callee dependencies in both JavaScript and C++ code. In particular, the call graph needs to indicate that a call to `Database.prototype.run` in the client package (Figure 1a, line 5) transitively calls the vulnerable C++ function `BindParameter` in its dependency.

2.2. Problem Formulation

Bridges: To enable cross-language security analyses which enable vulnerability detection [9], [14], native code sandboxing [15], and attack surface reduction [16], it is essential to identify the connection points between languages. For example, to effectively detect the vulnerability in Figure 1, a taint analysis tool must track how the arguments of `Statement.prototype.run` (Figure 1b, line 5) propagate to the parameters of its native implementation `Statement::Run` (Figure 1c, line 10). Similarly, a cross-language call graph for vulnerability reachability analysis must capture that invoking `Statement.prototype.run` in JavaScript ultimately leads to a call to the native C++ method `Statement::Run`.

In this work, we refer to those connection points as *bridges*. A bridge is defined as a triple consisting of three elements: (1) the fully-qualified name (FQN) of a JavaScript function (e.g., `sqlite3.Statement.prototype.run`), (2) the FQN of the corresponding native or Wasm function it invokes (e.g., `Statement::Run`), and (3) the shared library or the Wasm module where that function is defined (e.g., `node_sqlite3.node`).

Identifying bridges: Identifying bridges between JavaScript and native or Wasm code is a challenging task. While one might argue that this can be easily achieved through static analysis by inspecting how bindings are defined (e.g., lines 5–12 in Figure 1d), prior work [14], [25], [26] has shown this static-based approach to be both imprecise and incomplete (more details in Section 6.4). This undermines the effectiveness of the cross-language security analyses that rely on accurate bridge identification. For example, failing to detect the bridge in Figure 1 leads to a vulnerability going undetected or prevents identifying that a client package is affected by the CVE.

Static approaches face several limitations: (1) language extensions span multiple languages (C, C++, Rust), each requiring distinct analysis pipelines; (2) diverse JavaScript runtimes (Node.js, Deno) employ different FFIs and over 15 binding frameworks with distinct APIs and semantics; and (3) WebAssembly modules often expose functions directly without explicit binding code.

Change in perspective: Recent work in Python, notably PyXray [26], introduced *object layout analysis*, a dynamic technique that identifies bindings between Python functions and native code. The approach imports all native extension modules to load foreign function objects into memory. Using the `id()` function, a Python built-in construct, it extracts their memory addresses and finally inspects the in-memory layout based on CPython's object model. For foreign function objects, a specific layout field contains the address of the bound native function, which PyXray resolves to establish Python-to-native bridges.

PyXray achieves 100% recall by leveraging two key observations: (1) once a native extension module is loaded, all its associated objects (including foreign function objects) reside in memory ("Observation 1"); and (2) the in-memory representation of each foreign function object contains a pointer to the native function it invokes ("Observation 2").

Focus of this work: Applying object layout analysis to JavaScript poses a number of challenges, as JavaScript supports multiple execution environments (e.g., native extensions, Wasm) with diverse FFIs across runtimes and embedders. This raises a number of fundamental questions: (1) Are bindings fully instantiated in memory at module load time, as in Python ("Observation 1")? (2) Do these execution environments share a unified memory model and object representation? (3) Do JavaScript function objects embed internal pointers to the native or Wasm functions they invoke ("Observation 2")?

To address these questions, we next examine the technical details of JavaScript function objects and their implications for object layout analysis.

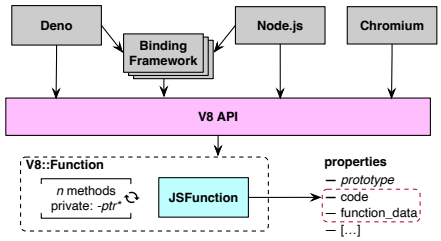


Figure 2: Using V8's high-level API, embedders create and manipulate `v8::Function` instances. Internally, each `v8::Function` points to a `JSFunction` object.

TABLE 1: Description of different function objects in V8. "Code" shows the code entry point referenced by the `code` field in the `JSFunction` layout. "Function data" indicates the metadata referenced by the `function_data` field in the `JSFunction` layout.

Category	Function Object Type	Code	function_data
JS functions	Uncompiled JS function	`v8::CompileLazy`	String with JS code
	Compiled JS function	`v8::InvokeInterpreter`	Bytecode
	Optimized JS function	Optimized assembly code	Unoptimized assembly code
Internal functions	Builtin function	Assembly code	A small integer
Foreign functions	Native function	`v8::HandleAPICall`	FunctionTemplateInfo
	WebAssembly function	`v8::JSToWasmWrapper`	WebAssembly module

3. Representation of JavaScript Functions

To understand whether object layout analysis techniques can be applied to JavaScript we need to examine how JavaScript functions are represented and implemented. This responsibility falls to execution engines (e.g., V8, JavaScriptCore), which manage the creation, layout, and runtime behavior of JavaScript functions through internal representations, such as C++ classes or structs. JavaScript embedders (e.g., Node.js, Chromium) utilize the external API of a JavaScript execution engine to define JavaScript functions that are backed by native logic. Since embedders interact with high-level APIs, the precise memory layout and behavior of JavaScript functions is determined *entirely* by the internals of the execution engine.

In this work, we focus on the V8 execution engine. We choose V8 because it holds a significant position in the JavaScript ecosystem, serving as the engine used by both of the two most popular runtimes, Node.js and Deno, as well as the most popular web browser, Chrome (based on Chromium). In V8, *all* function objects (whether native or regular ones) are allocated on the V8 heap and represented by a *single* internal C++ class called `JSFunction`. `JSFunction`'s layout is shown in Figure 2. In addition to standard properties, such as its object members and prototype, a `JSFunction` instance contains a field called `code`. This field points to a dispatcher function that determines how a call to the function object is handled, based on data stored in the `function_data` field of a `JSFunction` object.

Types of function objects: The combination of `code` and `function_data` determine the type and the behavior of

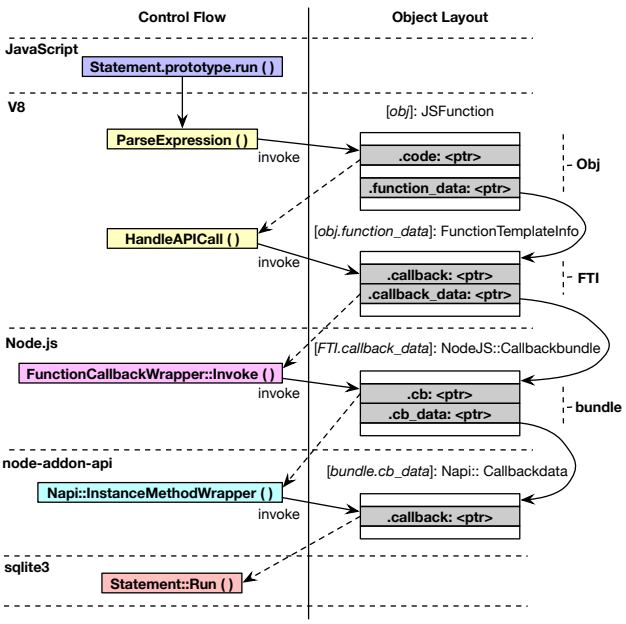


Figure 3: The runtime layout of `Statement.prototype.run` (Figure 1b, line 5). When this function is called from JavaScript, V8, along with the embedder, follow a chain of function pointers across several internal structures. This chain spans multiple layers of the object layout, ultimately leading to `Statement::Run` (Figure 1c; line 10).

function objects in V8. Table 1 classifies them into the following categories:

JavaScript functions: These are `JSFunction` instances representing functions defined at JavaScript level. Their representation varies based on compilation state. For example, if the function is optimized by TurboFan (V8's optimized JIT compiler), its `code` field points to machine code.

Internal functions: These are instances that wrap core ECMAScript operations (e.g., `Array.prototype.push`), using internal C++ implementations defined within the V8 codebase. The `code` field points to the compiled native code, while the `function_data` field holds a small integer (called Smi) representing the built-in function's ID.

Foreign functions: These are `JSFunction` instances that delegate execution to code *outside* the boundaries of V8. This category includes what V8 refers to as API functions, i.e., native functions defined by embedders using V8's public C++ API. The `code` field points to a call stub that dispatches into native code, while the `function_data` field references a `FunctionTemplateInfo` object containing the pointer to the actual native function. Notably, a foreign function can also wrap Wasm functions exported to JavaScript. In this case, the `code` field points to a trampoline that handles the transition from JavaScript to Wasm, and the `function_data` field holds metadata, such as the function index in the export table of the Wasm module.

Example: Figure 3 illustrates the runtime memory layout of the JavaScript function `Statement.prototype.run` from the example of Figure 1b. Its underlying `JSFunction` instance contains a chain of pointers to internal structures spanning multiple layers: V8 (`FunctionTemplateInfo`), Node.js (`CallbackBundle`), and the binding framework itself (`CallbackData`). Ultimately, this chain leads to the `CallbackData` structure from the used binding framework (`node-addon-api`). This structure contains a pointer to `Statement::Run` (Figure 1c; line 10), the C++ implementation of `Statement.prototype.run`. Figure 3 also shows the invocation path: calling `Statement.prototype.run` in JavaScript triggers the execution of `v8::HandleAPICall` (the `code` field in the `JSFunction` object), which accesses various function pointer fields to ultimately invoke `Statement::Get`.

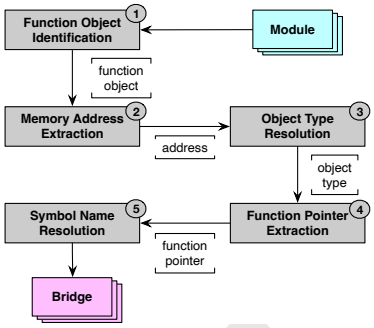


Figure 4: High-level overview of our approach for identifying foreign function bridges.

Creation of function objects: JavaScript embedders and binding frameworks can create their own function objects using *exclusively* the public V8 API. For example, to expose a native C++ function to JavaScript, embedders typically invoke `v8::FunctionTemplate::New`. This and other related API methods ultimately produce an instance of `v8::Function`, not an instance of the aforementioned `JSFunction` class. Internally, a `v8::Function` instance holds a reference to a `JSFunction` object allocated in V8's heap. However, the layout of this `JSFunction` is *opaque* to developers; they can only interact with it through specific high-level methods provided by the `v8::Function` API.

4. Bridging JavaScript and Foreign Functions

Overview: Among the different types of function objects supported in V8 (Table 1), this work focuses on *foreign function objects*, that is, those objects that do not represent pure JavaScript code but instead cross V8's execution boundary and invoke native or Wasm code. Our goal is to identify bridges that link these JavaScript functions to their underlying foreign implementations.

Figure 4 presents the high-level overview of our approach. Given an `npm` package, the approach imports its modules and recursively traverses all module members loaded into memory. When a function object is encountered (① *function object identification*), the approach extracts its memory address (② *memory address extraction*) and de-

termines the embedder-specific and/or binding framework-specific type of the object from its internal `JSFunction` representation, (③ *object type resolution*). If the provided function object is *foreign*, our approach applies specific extraction rules to recover a pointer to the native or Wasm function that will be executed when the JavaScript function is invoked (④ *function pointer extraction*). Once extracting the function pointer, the approach resolves it to a native symbol or Wasm function (⑤ *symbol name resolution*) and constructs a corresponding foreign function bridge as defined in Section 2.2.

Challenges: Each of these steps presents challenges related to the JavaScript semantics and V8 internals.

C1: Unexported objects. Step ① (*function object identification*) assumes access to all objects defined in a module. However, JavaScript supports encapsulation, meaning certain objects may be created and used internally but not exported. As a result, unexported foreign functions can be missed while traversing the contents of an imported module.

C2: Opaque JSFunction instances. In V8, the underlying `JSFunction` objects reside entirely within its internal heap and are inaccessible via JavaScript or its public API. Their fields and layout (discussed in Section 3) are hidden from embedders, which poses challenges for both memory address and function pointer extraction (steps ② and ④). Unlike Python, JavaScript provides no way to obtain the memory address of an object. And even if it did, that address does not expose the underlying `JSFunction`, which remains inaccessible across various runtimes and embedders.

C3: Binding framework-specific layouts. The native implementation of foreign function objects is stored in deeply embedded, highly diverse, and often inconsistent ways. Although this information is typically referenced via the `FunctionTemplateInfo` structure (accessible through the `function_data` field inside the `JSFunction` object; see Figure 3), its layout is not standardized. Binding frameworks and embedders are free to embed arbitrary native structures within this object and define their own calling conventions. This is exemplified by Figure 3 where recovering the native implementation of `Statement.prototype.run` in the `sqlite3` package requires traversing a deeply nested set of custom structures injected by the binding framework (`node-addon-api`). Therefore, different frameworks may embed different layouts, and the pointer to the native or Wasm function may be stored in entirely different locations. This complicates *function pointer extraction* (step ④).

C4: No explicit type information. All function objects in V8 share the same internal type, that is `JSFunction`. Even if we retrieve a memory address, there is no associated type metadata indicating which binding framework produced it to deduce what internal structure it follows (step ③). In the following, we detail how we tackle these challenges.

4.1. Function Object Identification

Basics: The input to our approach is an `npm` package (Figure 4), which may include a combination of JavaScript modules (.js files), native extension modules (.node files), shared libraries (.so files), and Wasm modules (.wasm files). All these files are grouped into two main categories:

Modules: These are files that are directly imported via `require` or `import`, such as regular JavaScript modules (.js files) or extension modules (e.g., .node files). For example, in the `sqlite3` example, native C++ code is compiled into `node_sqlite3.node`, which exports JavaScript-accessible objects (Figure 1b, line 1).

Dynamic dependencies: These are supported files (e.g., general-purpose shared libraries, Wasm modules) that are loaded at runtime programmatically For example, Deno relies on `Deno.dlopen`, which loads an arbitrary library dynamically and exposes selected native symbols to JavaScript:

Listing 1: An example module that uses Deno's `dlopen`.

```
1  const dylib = Deno.dlopen("libexample.so", {
2    add: { parameters: ["i32", "i32"], result: "i32"}
3  });
4  dylib.symbols.add(1, 3); // result: 4
```

Notably, when `Deno.dlopen` is called, Deno's runtime creates a `JSFunction` instance for every exposed symbol using V8's API (see 3). As a result, `dylib.symbols.add` (line 4) is a JavaScript function that refers to a `JSFunction` object allocated on V8's heap. This confirms our analysis in Section 3: regardless of how native code is integrated (traditional bindings or dynamic FFI), all functions are represented uniformly as `JSFunction` instances.

Identifying function objects: Our approach imports all modules (both JavaScript and native extension modules) in a given `npm` package through the `require` / `import` constructs and identifies all function objects they contain. This is done by *recursively* inspecting each imported module object and its properties using `Object.getOwnAndPrototypeEnumAndNonEnumProps`. During this step, our approach also tracks the FQN of each discovered function object to preserve its high-level JavaScript name. For example, when introspecting the `node_sqlite3.node` native extension module (Figure 1d), the JavaScript function object corresponding to the C++ `Statement::Run` function is identified as `sqlite3.Statement.prototype.run`. Similarly, in Listing 1, the function object representing the native symbol `add` is discovered as `dylib.symbols.add`.

Unexported objects: Some JavaScript objects are created within a module but never exposed via exports, This makes them invisible during introspection ("Challenge C1"). For example, in Listing 1, the module loads a shared library in an object `dylib`, but does not export this object. As a result, importing the module for analysis cannot access `dylib` or its native-backed functions.

To tackle this challenge, we complement the module traversal step with a native extension that leverages V8's built-in heap snapshot functionality, exposed via the `v8::HeapProfiler` API. This mechanism allows us to extract a complete graph of all live JavaScript objects allocated on V8's heap, regardless of whether they are accessible through module exports. In this way, we identify function

objects that reside on the heap, even if they are unexported. However, when using this fallback method, we can only infer the basename of the JavaScript function, instead of its FQN.

4.2. Memory Address Extraction

After identifying all function objects within the modules of an npm package, the next step is to recover their internal representations, that is, the memory address of JSFunction instances. However, the challenge in this context is that this internal representation is not accessible from JavaScript code or through embedder-level interfaces, such as the Node.js runtime or binding frameworks ("Challenge C2").

To overcome this challenge, our approach leverages the native extension interface of V8. Specifically, we implement a native function named jid that when given a JavaScript function object (e.g., Statement.prototype.run in Figure 1b), extracts the memory address of its underlying JSFunction representation. To do so, our approach leverages V8's calling conventions: all native functions exposed to JavaScript must adhere to the following signature:

```
void jid(v8::FunctionCallbackInfo<v8::Value>& args);
```

In the above, the parameter args holds information about the context of the call, such as the receiver, or the number and the values of arguments. If a function object is passed to jid from JavaScript, it can be accessed as args[0], which is an instance of v8::Function. While v8::Function is part of V8's public API, it abstracts away the internal details of the underlying JSFunction object. Internally, however, v8::Function wraps a pointer to this object in V8's heap. To access this internal representation, our method defines a custom C++ stub that mimics the memory layout of v8::Function. This enables us to extract the raw pointer to the underlying JSFunction object without relying on or modifying V8's public API.

4.2.1. Symbol Name and Wasm Function Resolution. Having obtained the memory address of a JSFunction object, the next step is to identify the native or Wasm function it invokes. This requires locating a pointer to the target function within the object's memory layout. The key challenge is that this information is stored in highly diverse and inconsistent ways across different binding frameworks ("Challenge C3"). Worse, given only an address α of a JSFunction, there is no explicit *type* information and metadata in the object indicating which framework created it or what internal structure it follows ("Challenge C4").

Key idea: To address these challenges, we leverage two observations. First, while binding frameworks may embed custom native structures within JSFunction objects, these structures are consistent across all bindings created by the *same* framework. For example, native extensions built with node-addon-api store the native function pointer in the Napi::details::CallbackData structure, which is referenced via a CallbackBundle embedded in the function_data field of the function object (Figure 3). This consistency allows us to define framework-specific rules for extracting the native function pointer from the object layout.

However, to apply these rules effectively, we must first determine which binding framework created the function object. This leads us to our second observation: although we cannot directly infer the binding framework from a raw JSFunction address, we observe that each framework produces a characteristic call chain when a function object is invoked. These call chains follow a *common* V8 prefix but diverge at the point where the control flow crosses into framework-specific code. For example, according to Figure 3, a call to a JavaScript function created with node-addon-api includes a call to FunctionCallbackWrapper::Invoke before reaching the target native function. Notably, we do *not* rely on executing the function to observe this call chain. Instead, we reconstruct the *call-chain suffix* statically by inspecting the memory layout of the JSFunction object and its embedded function pointers. This suffix serves as a unique signature for the used binding framework, allowing us to apply the appropriate rule to locate and resolve the native/Wasm function.

Extraction and resolution rules: Given the address of a JSFunction object obtained from the previous step, we define our pointer extraction and resolution rules as follows:

$$\sigma(H, L, W, \alpha) = \langle s, l \rangle$$

Given a heap H, a set of shared libraries L loaded into the address space, a set of Wasm modules W, and an address α pointing to a JSFunction object in the heap, the function $\sigma(H, L, W, \alpha)$ returns a pair consisting of (1) s, the name of the native or Wasm function that is executing when calling the function object, and (2) l, the path to the shared library or Wasm module where s is defined.

Figure 5 presents a subset of inference rules for resolving native and Wasm functions across different binding frameworks and embedders (the remaining rules can be found in Appendix A). In these rules, the notation $H(\alpha, p)$ refers to the value found by following the access path p (e.g., function_data.callback) starting from the base address α. The function $L(\alpha)$ returns the path of the shared library in which the symbol pointed by α resides. Similarly, $W(E)$ returns the Wasm module associated with the export table E. Our inference rules also rely on two resolution functions: $R_{so}(\alpha)$ and $R_{wasm}(E, i)$. The former resolves the symbol located at address α in native code, whereas the latter returns the name of the Wasm function at index i within the export table E. Finally, the $\cdot$ notation indicates path concatenation, while p_{cb} and p_d are notational conventions for access paths function_data.callback and function_data.callback_data respectively.

Example: Consider the example in Figure 3. To resolve the C++ function associated with Statement.prototype.run, our approach applies the NODE-ADDON-API rule. This rule matches a call chain suffix containing Node::FunctionCallbackWrapper::Invoke followed by Napi::InstanceWrap::Wrapper, indicating that the binding was created using the node-addon-api framework.

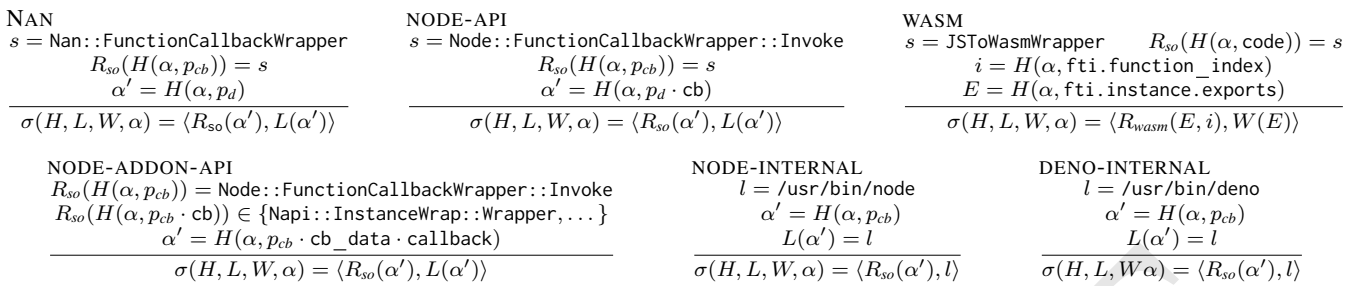


Figure 5: Inference rules for resolving the name of the native or Wasm function and the corresponding shared library or Wasm module across different binding frameworks and embedders. The remaining rules can be found in Appendix A.

When this is the case, the pointer α' to the native function is located at the access path $p_{cb} \cdot \texttt{cb\_data} \cdot \texttt{callback}$, relative to the base address α of the object. This resolves to the symbol $R_{so}(\alpha') = \texttt{Statement::Run}$, which is defined at the shared library $L(\alpha') = \texttt{node\_sqlite3.node}$.

Handling WebAssembly functions: Our approach identifies JavaScript functions that wrap Wasm functions by checking whether their code field points to `JSToWasmWrapper` (see Table 1). This is in contrast to native-backed functions, which are handled via `HandleAPICall`. The `JSToWasmWrapper` function essentially the Wasm call by converting JavaScript arguments into Wasm-compatible values.

Following the WASM rule in Figure 5, we extract two fields from the `JSFunction` object: (1) `function_data.function_index`, which is the index of the function within the Wasm module, and (2) `function_data.instance.exports`, which contains the export table of the Wasm module. We then resolve the Wasm function name by applying $R_{wasm}(E, i)$, where E is the export table and i is the function index.

Resolving Wasm module names: Given that the `JSFunction` object does not store the module name or its origin, we employ a sort of fingerprinting to identify the Wasm module (.wasm file) that defines the wrapped Wasm function. This fingerprinting process works as follows: we hash the export table of every Wasm binary included in the installed `npm` package. If the export table E found in the `JSFunction` layout matches one of these, we retrieve the name of the corresponding Wasm module using $W(E)$. If no match is found, we issue a warning. This typically means the Wasm module is fetched dynamically (e.g., from a remote source), and not bundled within the `npm` package.

5. Implementation

We implement our approach as a JavaScript command-line tool called GASKET, containing roughly 1.8k lines of JavaScript and C++ code. GASKET consists of two components: (1) a frontend written in JavaScript (700 LoC) that implements the *function object identification* step (Figure 4.1) and (2) a backend (1.1k LoC) which corresponds to a Node.js native extension written in C++ that implements the remaining steps. Notably, although we implement GASKET's backend as a Node-based native extension, it is still compatible with other runtimes, such as Deno. We refer the reader to Appendix B for further details on GASKET's frontend and backend design, as well as additional optimizations.

Limitations: As a dynamic analysis tool, GASKET inherits several limitations. First, it currently targets Linux ELF binaries and relies on GDB for symbol resolution; extending it to support other platforms is straightforward by integrating equivalent tooling. Second, GASKET focuses exclusively on forward flows, (i.e., from JavaScript to native or Wasm functions) and does not analyze reverse flows (e.g., callbacks from native code into JavaScript). Detecting such flows would require a different approach, potentially based on static analysis. To mitigate this issue, a simple strategy could assume that if a callback (a common programming idiom in JavaScript) is passed as an argument to a native function identified by GASKET, then the native code may invoke it internally. While conservative, this assumption can help approximate reverse flows without dynamic tracking. Third, GASKET may miss bridges created dynamically at runtime. However, our evaluation indicates that such cases are highly rare (Section 6) as most bindings are initialized during *module loading*.

Additionally, GASKET requires that the native extension modules contain symbol information, that is, the corresponding binaries are not stripped. By default, the native extensions in Node.js and Deno are compiled with the `-g` compiler option enabled. Furthermore, because GASKET installs and loads `npm` packages to inspect their memory state, analyzing untrusted packages may require sandboxed environments to mitigate the risk of arbitrary code execution. Finally, our current implementation relies on V8's memory abstractions, and is therefore limited only to V8 embedders. In principle, it can be extended to other JavaScript engines.

6. Evaluation

We seek answers for the following research questions:

RQ1 What is the precision and recall of GASKET in identifying connections between JavaScript functions and their corresponding native or Wasm targets?

RQ2 What is the performance of GASKET?

RQ3 How applicable is GASKET across different JavaScript runtimes and binding frameworks?

RQ4 Can GASKET enhance existing cross-language security analysis tools in terms of detecting native call targets and uncovering vulnerabilities?

RQ5 Can GASKET help developers determine whether their code reaches native-backed vulnerabilities in third-party dependencies?

6.1. RQ1: Effectiveness of GASKET

Binding frameworks and runtimes: We evaluate GASKET's effectiveness by applying it to Node.js and Deno packages that utilize native or Wasm extensions. For Node.js, we examine the six most popular binding frameworks: `NaN`, `Node-API`, and `node-addon-api` for C/C++ bindings, and `napi-rs`, `neon`, and `node-bindgen` for Rust bindings. We also analyze internal modules such as `fs`, which use ad hoc mechanisms to interact directly with V8, as well as packages that utilize Wasm. For Deno, we examine internal bindings coming from its standard library as well as packages that use the `Deno.dlopen` API.

Benchmarks: Packages that use `NaN`, `Node-API`, or `node-addon-api` often rely on helper libraries such as `napi-macros` and similar tools. To identify `npm` packages that use these binding frameworks, we query the `npmjs` registry for packages that depend on the relevant helper libraries and select the top two packages with the highest monthly downloads for each framework. For internal bindings, we focus on the `fs` and `os` modules, as these are widely used in both Node.js and Deno applications, and studied by previous research [15], [16].

To identify benchmarks utilizing the remaining binding frameworks, we utilize GitHub's code search functionality with framework-specific query strings. For example, to find packages using `neon`, we search for `cx.export_function`, a common function call in Neon-based projects, We then examine the first two pages of results and select the two most popular packages based on GitHub stars. Our final benchmark comprises twenty packages, two per binding framework (see Table 2).

Ground truth: To establish ground truth, we manually inspect the source code of each package to identify files (e.g., C++, Rust) that implement binary extensions, ensuring that no additional code is generated dynamically at build time. We then examine all native functions in these files and map them to their corresponding JavaScript function objects, based on how they are bound via the respective binding frameworks. For the Wasm and `Deno.dlopen` categories, we additionally analyze all JavaScript files to locate loading points for Wasm modules and shared libraries via calls to `Deno.dlopen` or `Wasm.Instantiate`. For Wasm, we then inspect the export tables of the .wasm binaries; for `Deno.dlopen`, we track the exported symbols from shared libraries at the time of loading. The complete ground truth for each package is reported in the "Native funcs (Ground truth)" column of Table 2. While manual inspection comes with a risk of missing a bridge, we mitigate this by ensuring that any native function not marked as a binding is indeed not used by the binding framework in any way.

Results: Table 2 presents the analysis results for GASKET. In all cases, GASKET achieves 100% recall, meaning it successfully identifies every bridge included in the ground truth without any false negatives. Notably, we do not explicitly compute the precision of GASKET, since its dynamic nature ensures no false positives. Our results validate the core insight behind GASKET: *bindings are mostly created when loading a binary extension module or library into memory, not during execution*. Although creating bindings during the execution of JavaScript functions is possible, it appears uncommon in practice, as we do not observe such cases in the studied benchmarks.

6.2. RQ2: Performance of GASKET

Measuring performance: For each package in Table 2, we track the overall running time of GASKET and the total number of function objects examined in a single run. This performance measurement (single-threaded) is done on a Linux machine equipped with an AMD EPYC 7413 processor and 24GB of RAM.

Results: Our results indicate that GASKET is efficient, analyzing most packages in under 100 seconds. For all categories except Deno FFI, execution time is ∼14 seconds, primarily due to the overhead of attaching and detaching GDB to the process for symbol resolution.

We observe that execution time scales well with package size. For example, analyzing `@u4/opencv4nodejs` requires examining 43 times more objects than `re2`; yet it leads to only a 2-fold increase in execution time. GASKET's running times are acceptable given that the analysis only needs to be performed once per package-version combination and the results can then be reused.

6.3. RQ3: Applicability of GASKET

Our findings show that, regardless of the binding framework or programming language used, the JavaScript objects created during module instantiation are *ultimately* internal V8 objects. GASKET *effortlessly* captures the bridges across different embedders and binding frameworks by incorporating an average of 70 lines of code for locating and extracting the function pointer from the internal `JSFunction` object (Section 5). Notably, GASKET is the *first* approach to extend bridge identification to Rust-based frameworks, as well as Wasm and the `Deno.dlopen` API.

Applicability of static analysis: Regarding existing static analysis techniques, the final columns of Table 2 show the categories that each technique supports. We use checkmarks to indicate that a technique *attempts* to capture JavaScript-Native bridges for the associated binding framework, not that it does so comprehensively (more details in Section 6.4).

We consider two security tools that statically compute JavaScript-native bridges. CHARON [14] builds on the Joern

TABLE 2: Descriptive characteristics of the selected benchmarks and GASKET's execution results.

Package	Effectiveness			Performance			Applicability				
	Native funcs (Gr. truth)	Found	Recall	Time (s)	Objects	Function Objects	Languages	Runtime	Binding Framework	CHARON	HODOR
re2	10	10	100	14	604	402	C++	Node.js	NaN	✓	✗
@u4/opencv4nodejs	685	685	100	34	26,293	12,549	C++		Node-API	✓	✗
leveldown	23	23	100	14	1,476	991	C++				
fs-native-extensions	14	14	100	14	1,187	847	C		node-addon-api	✓	✗
sharp	13	13	100	13	1,146	831	C++				
sqlite3	27	27	100	14	1,426	943	C++		ad-hoc*	✗	✓
fs (internal)	67	67	100	15	2,837	1,907	C++				
os (internal)	13	13	100	14	552	451	C++		neon	✗	✗
skia-canvas	174	174	100	15	6,459	3,407	Rust				
@cubejs-backend/native	16	16	100	15	1,245	879	Rust		napi-rs	✗	✗
lz4-napi	12	12	100	14	1,113	8,155	Rust				
@napi-rs/canvas	101	101	100	16	4,163	2,239	Rust, C++		node-bindgen	✗	✗
@fluvio/client	35	35	100	15	1,872	1,183	Rust				
venbind	3	3	100	14	816	671	Rust		WASM	✗	✗
tiny-secp256k1	22	22	100	97	6,711	5,239	-				
@cwasm/lodepng	5	5	100	14	846	686	-				
fs (internal)	61	61	100	18	22,585	13,602	Rust	Deno	ad-hoc*	✗	✗
os (internal)	16	16	100	18	22,585	13,602	Rust				
@db/sqlite	73	73	100	1	5,346	4,422	-		Deno FFI	✗	✗
@b-fuze/deno-dom	5	5	100	4	40,338	33,366	-				

TABLE 3: Bridge-finding results of GASKET vs. CHARON.

Technique	Total	False Pos.	Statistical Measures (bridges/pkg)			
			5%	Mean	Median	95%
GASKET	20,427	0	1	16	3	48
CHARON	8,147	1341 (16%)	0	6	1	21

static analyzer [27] to generate code property graphs (CPGs) for both JavaScript and C++ code. It identifies vulnerable data flows from JavaScript function parameters to unsafe C++ functions (e.g., `memcpy`). To connect the JavaScript and C++ sides, CHARON uses regular expressions and specific code patterns to detect binding-related C++ calls, such as `Nan::SetMethod`, and extract their arguments statically.

Contrary to GASKET, CHARON only supports three Node.js-based binding frameworks, namely `Nan`, `Node-API`, and `node-addon-api`. While in theory CHARON could be extended to support additional frameworks and runtimes, doing so is non-trivial. Its static analysis engine, Joern, lacks support for Rust and Wasm, making it difficult to analyze popular bindings of Rust- and Wasm-specific code, without substantial backend changes. Even with such extensions, CHARON remains prone to false positives and negatives, as we show in Section 6.4.

For completeness, we note that HODOR [16] targets internal bindings in Node.js. In contrast, GASKET is runtime-agnostic and can enhance HODOR's runtime protection mechanism by broadening its scope. For example, GASKET can enable HODOR to apply runtime protection mechanisms in arbitrary popular `npm` packages.

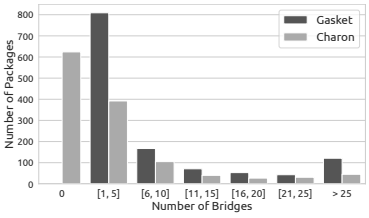
6.4. RQ4: Extending Security Tools with GASKET

Experimental setup: In this research question, we evaluate how GASKET can actively enhance existing security tools that perform cross-language analysis. Specifically, we focus on CHARON [14], a vulnerability detection tool that relies on cross-language taint analysis (see Section 6.3).

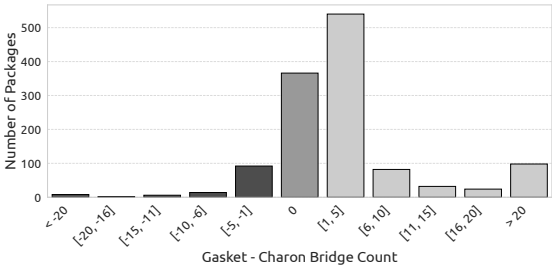
We evaluate the benefits that GASKET brings to CHARON along two dimensions: (1) by comparing the set of JavaScript-to-native function connections (bridges) identified by GASKET versus those detected by CHARON's static analysis, and (2) by *replacing* CHARON's static bridge detection mechanism with GASKET's dynamic one, and assessing how this impacts CHARON's overall effectiveness in finding vulnerabilities in native bindings of JavaScript applications.

Benchmarks: To assess how GASKET enhances CHARON, we begin with the original CHARON dataset of 9,083 `npm` packages with native extensions. We manually exclude 2,314 packages that are platform-specific (e.g., Windows-only or macOS-only), We also manually add 1,446 extra packages, identified by querying `npm` for dependents of helper packages commonly used in native extensions (e.g., `node-pre-gyp`). This yields a total benchmark of 8,214 `npm` packages, *all* of which rely on binding frameworks supported by CHARON (Table 2). We successfully install 1,266 of these packages and perform our comparison only on this subset, generating bridge information using both GASKET and CHARON. The remaining packages are excluded from the comparison either because of installation failures of broken and deprecated packages or because CHARON encounters unexpected analysis errors.

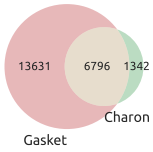
6.4.1. Results: Bridge-Finding Capability. Table 3 summarizes the bridge-finding results of GASKET and CHARON across the 1,266 `npm` packages. Overall, GASKET detects approximately 2.5× more bridges than CHARON: 20,427 bridges compared to 8,147. On average, GASKET reports 16 bridges per package (median: 3), whereas CHARON reports only 3 bridges per package on average (median: 1). In summary, GASKET uncovers 13,631 not covered by CHARON, while CHARON detects a total of 1,342 bridges not reported by GASKET (Figure 6c). Importantly, all bridges found by GASKET are guaranteed to be true positives: by design, GASKET does *not* produce false bridges. Later, we discuss the bridges exclusively found by CHARON (see "False bridges in CHARON").



(a) Distribution of the number of bridges detected by GASKET and CHARON across all analyzed packages.



(b) Differences in reported number of bridges between GASKET and CHARON; positive values indicate packages where GASKET detect more bridges.



(c) Venn diagram with the overall bridges found by GASKET and CHARON. 1,340 bridges reported exclusively by CHARON are false bridges.

Figure 6: Comparing bridge-finding capability of GASKET and CHARON across 1,266 npm packages.

Figure 6a shows that CHARON fails to detect any bridge in nearly half of the packages (624/1,266), while GASKET detects at least one in all. Most GASKET detections fall in the [1–5] range, but it identifies over 25 bridges in more than 100 packages. Figure 6b shows that GASKET finds more bridges in 61% of the cases (776/1,266), CHARON more in 10% (122), and both agree in 29% (349) instances.

False bridges in CHARON: We manually study each of the bridges exclusively detected by CHARON (1,342 bridges in total). We find that *all* but one case are false bridges, accounting for 16% of CHARON's overall results.

CHARON's false bridges stem from three main reasons. First, it over-approximates `*.Set()` calls, which are used to expose both functions and primitive values (714 cases). Second, CHARON may misinterpret C++ namespace labels as native function names (470 cases). Third, CHARON analyzes source files regardless of whether they are compiled (157 cases). For example, in `@div_arora/libpg-query`, it reports a bridge from a file that was removed from the build process six years ago. Since CHARON does not distinguish between compiled and unused files, it erroneously includes this outdated code in its analysis.

Missed bridges in GASKET: False negatives in GASKET arise when function objects are created dynamically at runtime rather than at module load time. We observe one such case in the `v8-sandbox` package, where its native function `Initialize` creates an additional JavaScript function only when it is invoked. Since GASKET analyzes the memory immediately after module loading (i.e., without executing any further code), these dynamically created objects are not visible to GASKET's analysis. Note that our manual inspection of all bridges reported by CHARON but not reported by GASKET reveal that the majority of them (99.9%) are not missed bridges of GASKET; rather, they stem from false positives in CHARON.

6.4.2. Results: Vulnerability-Finding Capability. We integrate GASKET into CHARON by replacing its JavaScript-to-native bridge identification mechanism, requiring only ten lines of additional code. This enhanced version, denoted as CHARON$_+$ (CHARON with GASKET), leverages GASKET's results to perform cross-language taint analysis and detect native-level vulnerabilities, such as memory leaks, buffer overflows, or integer overflows. This is achieved by computing data flows from user-controlled inputs in the JavaScript code (sources) to unsafe function calls in native code (e.g., `strcpy`), without any intervening sanitization. We refer the reader to the original CHARON's paper [14] for more details about this taint analysis. We evaluate the impact of this integration by comparing the bug-finding results of CHARON$_+$ against the original CHARON.

TABLE 4: Comparison of vulnerability detection results between CHARON and CHARON$_+$.

Tool	Packages	Vln. flows
CHARON	68	404
CHARON$_+$	91	458

TABLE 5: Summary of vulnerability types in the flows exclusively detected by CHARON$_+$.

Vulnerability type	Packages	Vln. Flows	Exploitable	Time Bomb
Integer Overflow	7	14	2	0
Division By Zero	10	12	3	0
Memory Leak	4	8	8	0
Buffer overflow	4	9	3	1
Null Ptr Dereference	2	2	2	0
malloc corruption	8	9	1	5
Total	-	**54**	**19**	**6**

Table 4 summarizes the bug-finding results. Across the 1,266 analyzed packages, CHARON detects 404 vulnerable flows in 68 unique packages. In contrast, CHARON$_+$ uncovers 458 vulnerable flows across 91 packages. Importantly, 23 of these packages (34% more than CHARON) contain vulnerabilities that CHARON misses. This demonstrates that GASKET enables CHARON to discover new vulnerable flows that were previously undetectable. Notably, CHARON may report multiple flows per vulnerability due to common native suffixes (e.g., many flows lead to the same sink). As such, the raw number of flows can be skewed by packages with multiple vulnerable flows ($>$ 60 flows).

Exploitable vulnerable flows: We manually analyze the 54 vulnerable flows *exclusively* reported by CHARON$_+$ to assess their real-world exploitability. For each flow, we examine

the source JavaScript function and craft test cases with carefully chosen arguments aimed at triggering the reported vulnerability type (e.g., buffer overflow or division by zero). Where necessary to confirm our findings, we recompile each package with sanitizers [28] (e.g., ASan and UBSan) to verify that our test cases trigger alarms.

Table 5 presents the results of this analysis. We successfully create proof-of-concept exploits for 19 out of the 54 flows, residing in 12 packages. These include, among others, 8 memory leaks (e.g., repeatedly invoking native functions that allocate memory without releasing it), 3 memory corruption bugs (e.g., buffer overflows, `malloc` corruption), 2 arithmetic overflows, and 3 division-by-zero errors. We use the term `malloc` corruption to refer to cases where a signed integer is used as an argument to `malloc`, where the implicit cast to `size_t` can potentially underflow.

The security impact of these vulnerabilities ranges from DoS attacks to potential arbitrary code execution. This demonstrates the practical value of GASKET in helping CHARON uncover previously undetected issues.

Time bombs: We are unable to exploit 35 of the reported vulnerable flows. However, this does not imply that these flows are unexploitable. In fact, we identify six flows that we consider "time bombs". These include vulnerabilities that are currently unexploitable only due to limitations in the JavaScript runtime but could become exploitable if these constraints evolve in future versions. A recurring pattern involves storing the length of a `v8::String` in a signed `int`, adding a constant offset (e.g., 32), and then calling `malloc`. Currently, V8 limits string lengths to 2^{29} characters. This cap has fluctuated over time (from 2^{28} to 2^{30}) in response to user feedback [29]. If raised further, adding even a small offset could cause an integer overflow and pass a negative size to `malloc` (assuming wrap-around semantics).

Discussion: The results demonstrate the significant advantage GASKET offers to security tools, such as CHARON. Specifically, the 13,631 bridges identified exclusively by GASKET (Figure 6c) present opportunities for uncovering additional vulnerabilities. This is because many of these bridges remain underutilized by CHARON primarily due to limitations in its taint analysis. For example, CHARON does not reason about asynchronous execution paths, such as those implemented using `libuv` [30], which are common in native JavaScript bindings.

A concrete case is the `sqlite3` package, where GASKET successfully detects all relevant bindings. One such binding asynchronously invokes a native callback via `libuv`, and in this callback, user-controlled input is passed to an unsafe `memcpy(ptr, 0, 0)` call. This call exhibits undefined behavior, which can lead to vulnerabilities in the presence of optimizing compilers [31], [32]. We manually confirm this flow and issue by constructing a test-case that results in a UBSan alarm. The developers have addressed our report. Both CHARON and CHARON$_+$ fail to detect vulnerable flow because they do not analyze asynchronous paths in native code. This demonstrates that our work could be further leveraged by more advanced security analysis tools.

TABLE 6: Reachability analysis results for 10 vulnerabilities affecting Node.js packages with native extensions. "vuln. ver." indicates analyzed dependents using vulnerable versions. "Call" shows the subset where the vulnerable symbol(s) is reachable in the cross-language call graph.

Vulnerability ID	Package	Vuln. symbol(s)	Dependents	
			vuln. ver.	Call
CVE-2022-25324 [33]	bignum	`Upowm, Bpowm`	169	11
CVE-2022-21144 [34]	libxmljs	`XmlDocument::FromHtml`	223	150
CVE-2022-21227 [23]	sqlite3	`Statement::BindParameter`	203	4
CVE-2022-25852 [35]	pg-native	`SendQuery, SendQueryParams`	97	4
CVE-2022-25345 [36]	@discordjs/opus	`OpusEncoder::Encode`	105	3
CVE-2024-21526 [37]	speaker	`speaker_open`	155	126
CVE-2022-21164 [38]	node-lmdb	`TxnWrap::putString`	22	4
CVE-2024-34394 [39]	libxmljs2	`get_local_namespaces`	40	1
CVE-2025-3194 [40]	bigint-buffer	`toBigInt`	198	56
GHSA-9v62-24cr [41]	node-sass	`get_importer_entry`	208	27

6.5. RQ5: Reachability Analysis with GASKET

We demonstrate how GASKET enhances vulnerability prioritization through cross-language reachability analysis. Current tools such as `npm-audit` check dependencies against the GitHub Advisory Database [42] but generate numerous false positives, overwhelming developers with alerts for unreachable vulnerabilities. Existing reachability analysis techniques are limited to pure JavaScript code and cannot handle vulnerabilities in native extensions, where many critical issues reside. We address this gap by leveraging GASKET to construct cross-language call graphs, which enable reachability analysis across JavaScript and native code boundaries.

Experimental setup: We utilize our dataset from RQ4. First, we sort the packages by popularity and then examine if they include vulnerabilities in their previous versions by querying the GitHub advisory database [42]. If they do so, we check if the vulnerabilities reside in their native extensions by examining fixing commits. In this way we identify 10 packages and their associated vulnerable version ranges, six of which are also studied by Staicu et al. [9].

For each vulnerable package, we collect dependent packages using `npm`'s API, which limits results to a maximum of 396 dependents per package. For packages with more dependents, we analyze this maximum returned subset. Our final dataset comprises 2,197 dependent packages across all vulnerable packages.

Cross-language call graph construction: We employ a multi-step approach to construct call graphs spanning JavaScript and native code. For JavaScript analysis, we utilize Jelly [43], [44], [45], a well-established static call graph generator (see Section 7). We also use Ghidra [46] to generate call graphs from compiled binaries.

Our analysis pipeline works as follows. First, we resolve version constraints in `package.json` to determine usage of vulnerable versions. Then, we generate JavaScript call graphs using Jelly and construct binary call graphs for native extensions using Ghidra. Finally, we identify JavaScript-native bridges using GASKET and integrate each component to determine the reachability of vulnerable native functions from JavaScript entry points.

Results and impact: Table 6 presents our findings, which demonstrate substantial reduction in false positives: for example, while 169 packages depend on vulnerable versions of `bignum`, only 11 actually invoke the vulnerable functions `Upowm` and `Bpowm`. Similarly, among 203 packages using vulnerable `sqlite3` versions, only 4 reach the vulnerable `Statement::BindParameter` function.

Our findings indicate that cross-language reachability analysis can significantly reduce alert fatigue by filtering out non-exploitable vulnerability warnings. We responsibly disclosed our findings to maintainers of 15 transitively affected packages, recommending dependency version updates where appropriate.

Threats to validity: One threat to the validity of our results is the potential inaccuracy of the cross-language call graph, which may include false positives or false negatives due to limitations in the tools used (i.e., Jelly and Ghidra). However, our approach is agnostic to the choice of call graph generators and can be integrated with other tools as well.

7. Related Work

Multi-language program analysis: Multi-language program analysis has gained significant attention due to the prevalence of polyglot applications. Staicu et al. [9] analyze the misuse of native extension APIs in scripting languages, highlighting potential security threats introduced by these extensions. CHARON is another related tool, which we refer the reader to Section 6.4 for a detailed comparison.

For JavaScript-specific environments, several approaches have been developed to handle language interactions and framework-specific extensions. Bae et al. [47] propose a type system that supports interactions between JavaScript and Java to detect bugs in Android applications. Lee et al. [48] employ static analysis to detect these interactions and identify potential information leaks. Similarly, Bai et al. [49] focus on Android applications by providing a taint-tracking approach to examine interoperations between JavaScript and native code. ReactAppScan [50] constructs component graphs to track data flows across JavaScript-XML components for React vulnerability detection.

Python-native bridge analysis has also received considerable attention. Our work is directly inspired by PyXray [26], which introduces the concept of object layout analysis. However, CPython differs fundamentally from JavaScript engines. In Section 4, we outline the unique challenges in adapting this approach to the JavaScript ecosystem. Frog [25] focuses specifically on Python-native bridges by statically analyzing Python and C source files to generate unified call graphs. PolyCruise [51] enables dynamic information flow analysis through symbolic dependencies in Python-C programs, facilitating vulnerability discovery. Python-C interactions have been further analyzed using abstract interpretation [52] and declarative analysis [53]. Cross-language analysis has also been extensively explored for Android applications [54], [55], [56], [57], Java Native Interface [2], [58], [59], [60], [61], and Go-C/C++ interactions via differential fuzzing [5].

Securing script execution environments: Runtime protection systems have emerged as a critical defense mechanism for script execution environments. HODOR [16] implements a runtime protection system for Node.js applications that enforces system call restrictions based on call graphs spanning JavaScript and C/C++ layers of Node's internal library. Wyss et al. [11] develop a similar system call filtering mechanism with a specific focus on install scripts. Abbadini et al. [15], [20] propose runtime sandboxing solutions employing eBPF to control native code access to system resources for both Node.js and Deno environments.

Browser execution environments have also received attention in security research. Zhao [1] introduces an object-oriented permission system to confine potentially harmful JavaScript operations. Vasilakis et al. [13], [62] propose the use of both process-level isolation and language-centric methods to achieve compartmentalization in scripting language code. Narayan et al. [63] focus on the Firefox ecosystem and C++, combining static information flow analysis with lightweight runtime checks to confine faulty components of the runtime system. Wasm, despite its sandboxed execution environment, still faces security challenges from low-level flaws [8]. Michael et al. [64] address this by proposing memory safety principles that enable compiler-based enforcement against runtime attacks.

Ecosystem-level security analysis: Large-scale ecosystem analysis has become increasingly important for understanding security risks in modern software supply chains. JSGo [19] addresses vulnerability validation in Node.js by leveraging test suites for automatic input generation, successfully reproducing known vulnerabilities. Ecosystem mapping approaches [65] utilize call graphs to analyze calling relationships across software systems, providing insights into dependency structures.

Studies have revealed important patterns in vulnerability propagation. Mir et al. [66] demonstrate that while one-third of Maven packages contain vulnerable dependencies, only 1% actually invoke vulnerable methods. Zahan et al. [12] analyze 1.6 million `npm` packages to detect supply chain risk indicators, revealing concerning trends in dependency management. Dependency bloat analysis has been explored for Python [24], indicating that vulnerabilities often reside in bloated areas of utilized packages. GASKET enables cross-language reachability analysis as shown in Section 6.5.

8. Conclusion

We have presented GASKET, a dynamic analysis tool that leverages V8's internal object representation to detect foreign JavaScript bridges across diverse binding frameworks and runtime environments. By analyzing function object layouts at the JavaScript engine level, GASKET achieves perfect recall without false positives, successfully handling different binding frameworks.

Our evaluation across numerous `npm` packages demonstrates that GASKET enables existing security tools to discover additional vulnerable flows in previously unexplored

packages. Cross-language reachability analysis using GASKET further highlights that a small fraction of applications actually invoke vulnerable functions in their dependencies, enabling better vulnerability prioritization and actionable security alerts.

References

[1] R. Zhao, "Beast in the cage: A fine-grained and object-oriented permission system to confine javascript operations on the web," in *Proceedings of the ACM on Web Conference 2025*, ser. WWW '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 3171–3182.

[2] G. Fourtounis, L. Triantafyllou, and Y. Smaragdakis, "Identifying Java calls in native code via binary scanning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 388–400. [Online]. Available: https://doi.org/10.1145/3395363.3397368

[3] T. Roth, J. Näumann, D. Helm, S. Keidel, and M. Mezini, "AXA: Cross-language analysis through integration of single-language analyses," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1195–1205. [Online]. Available: https://doi.org/10.1145/3691620.3696193

[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 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 385–395.

[5] A. Sorniotti, M. Weissbacher, and A. Kurmus, "Go or no go: Differential fuzzing of native and c libraries," in *2023 IEEE Security and Privacy Workshops (SPW)*, 2023, pp. 349–363.

[6] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in JavaScript bindings," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 559–578.

[7] A. N. Evans, B. Campbell, and M. L. Soffa, "Is Rust used safely by software developers?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 246–257.

[8] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: binary security of webassembly," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20. USA: USENIX Association, 2020.

[9] C.-A. Staicu, S. Rahaman, Á. Kiss, and M. Backes, "Bilingual problems: Studying the security risks incurred by native extensions in scripting languages," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6133–6150. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/staicu

[10] J. Rack and C.-A. Staicu, "Jack-in-the-box: An empirical study of javascript bundling on the web and its security implications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 3198–3212.

[11] E. Wyss, A. Wittman, D. Davidson, and L. De Carli, "Wolf at the door: Preventing install-time attacks in npm with latch," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1139–1153.

[12] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 331–340.

[13] N. Vasilakis, C.-A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel, "Preventing dynamic library compromise on node.js via rwx-based privilege reduction," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1821–1838.

[14] R. Scholtes, S. Khodayari, C.-A. Staicu, and G. Pellegrino, "CHARON: Polyglot code analysis for detecting vulnerabilities in scripting languages native extensions."

[15] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "NatiSand: Native code sandboxing for JavaScript runtimes," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 639–653. [Online]. Available: https://doi.org/10.1145/3607199.3607233

[16] W. Wang, X. Lin, J. Wang, W. Gao, D. Gu, W. Lv, and J. Wang, "HODOR: Shrinking attack surface on Node.js via system call limitation," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2800–2814. [Online]. Available: https://doi.org/10.1145/3576915.3616609

[17] L. Zhao, S. Chen, Z. Xu, C. Liu, L. Zhang, J. Wu, J. Sun, and Y. Liu, "Software composition analysis for vulnerability detection: An empirical study on java projects," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 960–972.

[18] S. Lee and M. Böhme, "Statistical reachability analysis," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 326–337.

[19] C. Luo, P. Li, W. Meng, and C. Zhang, "Test suites guided vulnerability validation for node.js applications," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 570–584. [Online]. Available: https://doi.org/10.1145/3658644.3690332

[20] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "Cage4deno: A fine-grained sandbox for deno subprocesses," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 149–162.

[21] "node-addon-api module," https://www.npmjs.com/package/node-addon-api, 2025, online accessed; 04-06-2025.

[22] "bug: fix segfault of invalid toString() object," https://github.com/TryGhost/node-sqlite3/pull/1450, 2022, online accessed: 2025-06-04.

[23] NIST, "CVE-2022-21227," https://nvd.nist.gov/vuln/detail/CVE-2022-21227, National Institute of Standards and Technology, 2022, accessed: 2025-06-04.

[24] G.-P. Drosos, T. Sotiropoulos, D. Spinellis, and D. Mitropoulos, "Bloat beneath Python's scales: A fine-grained inter-project dependency analysis," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660821

[25] M. Hu, Q. Zhao, Y. Zhang, and Y. Xiong, "Cross-language call graph construction supporting different host languages," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 155–166.

[26] G. Alexopoulos, T. Sotiropoulos, G. Gousios, Z. Su, and D. Mitropoulos, "Building cross-language call graphs in python via dynamic memory analysis," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15607105

[27] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: a fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 28.

[29] "String length limit is small-ish," https://issues.chromium.org/issues/42209439, 2017, online accessed: 2025-06-04.

[30] "Cross-platform asyncronous I/O," https://libuv.org/, 2025, online accessed: 2025-06-04.

[31] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 260–275. [Online]. Available: https://doi.org/10.1145/2517349.2522728

[32] S. Li and Z. Su, "Finding unstable code via compiler-driven differential testing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 238–251. [Online]. Available: https://doi.org/10.1145/3582016.3582053

[33] NIST, "CVE-2022-25324," https://nvd.nist.gov/vuln/detail/CVE-2022-25324, National Institute of Standards and Technology, 2022, accessed: 2025-06-04.

[34] ——, "CVE-2022-21144," https://nvd.nist.gov/vuln/detail/CVE-2022-21144, National Institute of Standards and Technology, 2022, accessed: 2025-06-04.

[35] ——, "CVE-2022-25852," https://nvd.nist.gov/vuln/detail/CVE-2022-25852, National Institute of Standards and Technology, 2022, accessed: 2025-06-04.

[36] ——, "CVE-2022-25345," https://nvd.nist.gov/vuln/detail/CVE-2022-25345, National Institute of Standards and Technology, 2022, accessed: 2025-06-04.

[37] ——, "CVE-2024-21526," https://nvd.nist.gov/vuln/detail/CVE-2024-21526, National Institute of Standards and Technology, 2024, accessed: 2025-06-04.

[38] ——, "CVE-2022-21164," https://nvd.nist.gov/vuln/detail/CVE-2022-21164, National Institute of Standards and Technology, 2022, accessed: 2025-06-04.

[39] ——, "CVE-2024-34394," https://nvd.nist.gov/vuln/detail/CVE-2024-34394, National Institute of Standards and Technology, 2024, accessed: 2025-06-04.

[40] ——, "CVE-2025-3194," https://nvd.nist.gov/vuln/detail/CVE-2025-3194, National Institute of Standards and Technology, 2025, accessed: 2025-06-04.

[41] GitHub, "GHSA-9v62-24cr-58cx," https://github.com/advisories/GHSA-9v62-24cr-58cx, GitHub Advisory Database, 2022, accessed: 2025-06-04.

[42] ——, "GitHub Advisory Database," https://github.com/advisories, 2019, online accessed: 2025-06-04.

[43] "Jelly: A static call graph generator for javascript," https://github.com/cs-au-dk/jelly, 2022, online accessed: 2025-06-04.

[44] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of node.js applications," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 29–41.

[45] A. Møller, B. B. Nielsen, and M. T. Torp, "Detecting locations in javascript programs affected by breaking library changes," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[46] National Security Agency, "Ghidra," https://ghidra-sre.org/, 2025, online accessed; 10-03-2025.

[47] S. Bae, S. Lee, and S. Ryu, "Towards understanding and reasoning about android interoperations," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 223–233. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00038

[48] S. Lee, J. Dolby, and S. Ryu, "Hybridroid: static analysis framework for android hybrid applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 250–261.

[49] J. Bai, W. Wang, Y. Qin, S. Zhang, J. Wang, and Y. Pan, "Bridgetaint: A bi-directional dynamic taint tracking method for javascript bridges in android hybrid applications," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 677–692, 2019.

[50] Z. Guo, M. Kang, V. Venkatakrishnan, R. Gjomemo, and Y. Cao, "Reactappscan: Mining react application vulnerabilities via component graph," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 585–599.

[51] W. Li, J. Ming, X. Luo, and H. Cai, "PolyCruise: A Cross-Language dynamic information flow analysis," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2513–2530. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/li-wen

[52] R. Monat, A. Ouadjaout, and A. Miné, "A multilanguage static analysis of Python programs with native C extensions," in *Static Analysis*, C. Drăgoi, S. Mukherjee, and K. Namjoshi, Eds. Cham: Springer International Publishing, 2021, pp. 323–345.

[53] D. Youn, S. Lee, and S. Ryu, "Declarative static analysis for multilingual programs using CodeQL," *Software: Practice and Experience*, vol. 53, no. 7, pp. 1472–1495, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3199

[54] S. Bae, S. Lee, and S. Ryu, "Towards understanding and reasoning about Android interoperations," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 223–233. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00038

[55] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: static analysis framework for Android hybrid applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 250–261. [Online]. Available: https://doi.org/10.1145/2970276.2970368

[56] A. D. Brucker and M. Herzberg, "On the static analysis of hybrid mobile apps," in *Engineering Secure Software and Systems*, J. Caballero, E. Bodden, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2016, pp. 72–88.

[57] S. Almanee, A. Ünal, M. Payer, and J. Garcia, "Too quiet in the library: An empirical study of security updates in android apps' native code," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, p. 1347–1359.

[58] S. Lee, H. Lee, and S. Ryu, "Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 127–137.

[59] J. Park, S. Lee, J. Hong, and S. Ryu, "Static analysis of JNI programs via binary decompilation," *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3089–3105, 2023.

[60] G. Tan and G. Morrisett, "Ilea: inter-language analysis across java and c," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 39–56.

[61] G. Tan and J. Croft, "An empirical security study of the native code in the jdk," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08. USA: USENIX Association, 2008, p. 365–377.

[62] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization," in *Networked and Distributed Systems Security*, ser. NDSS'18, 2018.

[63] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the firefox renderer," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20. USA: USENIX Association, 2020.

[64] A. E. Michael, A. Gollamudi, J. Bosamiya, E. Johnson, A. Denlinger, C. Disselkoen, C. Watt, B. Parno, M. Patrignani, M. Vassena, and D. Stefan, "Mswasm: Soundly enforcing memory-safe execution of unsafe code," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, Jan. 2023.

[65] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 101–104.

[66] A. M. Mir, M. Keshani, and S. Proksch, "On the effect of transitivity and granularity on vulnerability propagation in the Maven ecosystem," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 201–211.

[67] I. Stepanyan, "Unsafe fast JS calls," https://v8.dev/blog/v8-release-87#unsafe-fast-js-calls, 2020, online accessed: 2025-06-04.

[68] The Chromium Project, "Omnibox API implementation," https://github.com/chromium/chromium/blob/main/chrome/browser/extensions/api/omnibox/omnibox_api.cc, 2024, accessed: 2025-06-04.

[69] Chrome Developers, "Chrome Extensions: API Reference," https://developer.chrome.com/docs/extensions/reference/api, 2025, online accessed; 12-03-2025.

Appendix A. Extraction and Resolution Rules

Figure 7 summarizes the extraction and resolution rules for all remaining binding frameworks and V8 embedders supported by our approach. Notably, our method seamlessly handles Chromium's internal bindings, Rust-based frameworks in Node.js, the `Deno.dlopen` API, and V8's fast API mechanism [67], which enables optimized native calls that bypass traditional JavaScript bindings.

This enables existing security tools to handle broader scenarios. For example, HODOR [16], originally designed to restrict system calls in Node.js internal bindings, can now apply the same protections to `npm` packages using Rust-based extensions. Likewise, CHARON [14], which analyzes native extensions in Node.js packages, can now be extended with our approach to identify issues in the code of Chrome browser add-ons, which although written in JavaScript, interact with the natively-backed [68] Chrome Extension API [69].

Appendix B. Implementation Details of GASKET

```
1  const gasket = require("gasket_native.node");
2
3  function introspect(modpath) {
4    // import module under analysis
5    const mod = require(modpath);
6    // Init an object layout analysis
7    const ola = new gasket.OLAnalysis();
8    const worklist = [[mod, modpath]];
9    while (mod.length > 0) {
10     obj, objname = worklist.pop();
11     if (typeof obj === "function") {
12       // Visit internal JSFunction object and
13       // extract their function pointers (if present)
14       ola.visit_obj(obj, objname);
15     }
16     for (const child_attr of
           getOwnAndPrototypeEnumAndNonEnumProps(obj)) {
17       const child_obj = obj[child_attr];
18       worklist.push([child_obj, objname + "." +
             child_attr]);
19     }
20   }
21   // Resolve all registered function pointers and
         return bridges.
22   return ola.resolve();
23 }
```

Figure 8: A sketch implementation of the `introspect` function in GASKET's frontend.

Figure 8 presents a high-level overview of GASKET's frontend. Its core function, `introspect`, takes a path to a JavaScript or native module, dynamically imports it (line 5), and recursively traverses the structure of the imported module object using `getOwnAndPrototypeEnumAndNonEnumProps` (lines 9–20). The introspection logic leverages GASKET's backend via the `OLAnalysis` class (lines 1 and 7), which exposes two native instance methods:

- `visit_object(obj, objname)` (lines 11–14): This method accepts a JavaScript function object and its FQN. It locates the corresponding `JSFunction` object in memory (Section 4.1) and extracts its wrapped function pointer using the rules outlined in Figure 5. The extracted pointers are stored in an internal map called `pointer_set`, which maps each object's FQN to its native function pointer.
- `resolve()` (line 22): This method resolves the symbolic names of all stored function pointers in `pointer_set`, using debugging tools, such as GDB (see below). It returns a set of triples, where each triple corresponds to a detected bridge consisting of: the FQN of the JavaScript function object, the name of the native or Wasm function it invokes, and the file (.so or .wasm) where that function is defined.

Extracting function pointers: The rules in Figure 5 require accessing fields of `JSFunction` objects, whose internal layout is *opaque* to embedders and their native extensions. While one could attempt to mimic this

CHROMIUM

$$\frac{l = \texttt{/usr/bin/chromium} \quad \alpha' = H(\alpha, p_{cb}) \quad L(\alpha') = l}{\sigma(H, L, W, \alpha) = \langle R_{so}(\alpha'), l \rangle}$$

NEON

$$\frac{s = \texttt{neon::sys::fun::call\_boxed} \quad R_{so}(H(\alpha, p_d \cdot \texttt{cb})) = s \quad \alpha' = H(\alpha, \texttt{name})}{\sigma(H, L, W, \alpha) = \langle R_{so}(\alpha'), L(\alpha') \rangle}$$

NAPI-RS/NODE-BINGEN

$$\frac{s = \texttt{Node::FunctionCallbackWrapper::Invoke} \quad R_{so}(H(\alpha, p_{cb})) = s \quad \alpha' = H(\alpha, p_d \cdot \texttt{cb})}{\sigma(H, L, W, \alpha) = \langle R_{so}(\alpha'), L(\alpha') \rangle}$$

DENO-DLOPEN

$$\frac{s = \texttt{deno::v8::CFnFrom<F>::mapping::c\_fn} \quad R_{so}(H(\alpha, p_{cb})) = s \quad \alpha' = H(\alpha, \texttt{name})}{\sigma(H, L, W, \alpha) = \langle R_{so}(\alpha'), L(\alpha') \rangle}$$

FAST-API

$$\frac{\alpha' = H(\alpha, \texttt{function\_data} \cdot \texttt{cfunction\_overloads})}{\sigma(H, L, W, \alpha) = \langle R_{so}(\alpha'), L(\alpha') \rangle}$$

Figure 7: Inference rules for resolving the name of the native function in binding created by the Chromium browser, Rust-based binding frameworks in Node.js, `Deno.dlopen` API, and V8's fast API.

layout with stub structs, doing so is brittle and error-prone. Instead, we rely on V8's internal debugging utility `_v8_internal_Print_Object`, which, given a raw pointer, prints the object's type and key fields. For example, calling this function on the address of a `JSFunction` object (0x2c6edfe37001) yields:

```
0x2c6edfe37001: [Function]
 - map: 0x12a374c022f1 <Map[64]>
 - prototype: 0x12a374c0ed89 <JSFunction>
 - code: 0x315b00cd3d29 <Code BUILTIN HandleApiCall>
 - function_data: 0x2c6edfe36bc <FunctionTemplateInfo>
```

We parse this output and recursively apply `_v8_internal_Print_Object` to follow relevant access paths (e.g., `function_data`) until we extract the function pointer as specified by the rules.

Resolving function pointers: Once a function pointer is extracted from the object layout, we resolve it using GDB by running the `info symbol <ptr>` command. This resolves both the symbol name and the shared library where it is defined. For Wasm functions, resolution is based on the export table of the corresponding Wasm module instance, which is accessible from the object layout (Figure 5, rule WASM).

Optimizations: To improve performance, GASKET defers symbol resolution. When `visit_object` is called (Figure 8; line 14), it extracts and stores function pointers in the internal state of the receiver object without resolving them immediately. Only after introspection is complete, GASKET invokes `resolve` (line 22) to resolve all stored pointers at once. This avoids repeatedly attaching GDB into the current process, which is an expensive operation.

Furthermore, during the initialization of the object layout analysis (line 7), GASKET performs another optimization to avoid repeated symbol resolution via GDB. It uses `dlsym`[1] to retrieve the memory addresses of key framework-specific symbols (e.g., `FunctionCallbackWrapper::Invoke`) and caches them. Later, during layout analysis (line 14), GASKET identifies which binding framework created a given object by comparing function pointers directly against these pre-resolved addresses (see Figure 5).

1. https://man7.org/linux/man-pages/man3/dlsym.3.html