Finding and Preventing Bugs in JavaScript Bindings

Fraser Brown* Shravan Narayan† Riad S. Wahby*
Dawson Engler* Ranjit Jhala† Deian Stefan†

*Stanford University †UC San Diego

***Abstract*—JavaScript, like many high-level languages, relies on runtime systems written in low-level C and C++. For example, the Node.js runtime system gives JavaScript code access to the underlying filesystem, networking, and I/O by implementing utility functions in C++. Since C++'s type system, memory model, and execution model differ significantly from JavaScript's, JavaScript code must call these runtime functions via intermediate *binding layer code* that translates type, state, and failure between the two languages. Unfortunately, binding code is both hard to avoid and hard to get right.**

This paper describes several types of exploitable errors that binding code creates, and develops both a suite of easily-to-build static checkers to detect such errors and a backwards-compatible, low-overhead API to prevent them. We show that binding flaws are a serious security problem by using our checkers to craft 81 proof-of-concept exploits for security flaws in the binding layers of the Node.js and Chrome, runtime systems that support hundreds of millions of users. As one practical measure of binding bug severity, we were awarded $6,000 in bounties for just two Chrome bug reports.

1 Introduction

Many web services and other attacker-facing code bases are written in high-level scripting languages like JavaScript, Python, and Ruby. By construction, these languages prevent developers from introducing entire classes of bugs that plague low-level languages—e.g., buffer overflows, use-after-frees, and memory leaks. On the other hand, high-level languages introduce new classes of severe, exploitable flaws that are often less obvious than low-level code bugs.

High-level languages push significant functionality to their *runtime systems*, which are written in low-level, unsafe languages (mainly C and C++). Runtime systems provide functionality not possible in the base scripting language (e.g., network and file system access) or expose fast versions of routines that would otherwise be too slow (e.g., sorting routines). Since the high-level, dynamically-typed scripting language and low-level language have different approaches to typing, memory management, and failure handling, the scripting code cannot call runtime routines directly. Instead, it invokes intermediary *binding code* that translates between value types, changes value representations, and propagates failure between the languages.

Binding code has the dangerous distinction of being both hard to avoid and hard to get right. This paper demonstrates the severity of the problem by demonstrating 81 proof-of-concept exploits for bugs in multiple widely-used runtimes for the JavaScript language. We picked JavaScript because of its ubiquity: it is both the most popular language on GitHub and the language with the largest growth factor [11, 110]. And though it was originally confined to web pages, JavaScript now appears in desktop applications, server-side applications, browser extensions, and IoT infrastructure. Organizations like PayPal and Walmart use JavaScript to process critical financial information, and as a result implicitly rely on runtimes and binding code for secure foundational operations [40, 68, 106]. This paper focuses on detecting and exploiting flaws in two pervasive JavaScript runtime systems—Node.js and Chrome—since binding bugs in these systems endanger hundreds of millions of people (e.g., all users of the Chrome browser).

JavaScript's variables are dynamically typed. Therefore, when JavaScript code calls a binding layer function, that C++ binding function should first determine the underlying type of each incoming parameter. Then, the function should translate each parameter's current value to its equivalent statically-typed representation in C++. The binding code should also determine if values are legal (e.g., whether an index is within the bounds of an array); if not, the binding should propagate an error back to the JavaScript layer. Finally, before the function completes, it should store any result in the memory and type representation that JavaScript expects.

In practice, writing binding code is complicated: it can fail at many points, and bindings should detect failure and correctly communicate any errors back to JavaScript. Too often, binding code simply crashes, leading to denial-of-service or covert-channel attacks (§2). If a binding function does not crash, it might still skip domain checking (e.g., checking that an array index is in bounds)—or even ignore type checking, therefore allowing attackers to use nonsensical values as legal ones. (e.g., by invoking a `number` as a `function`). One especially insidious source of errors is the fact that binding code may invoke new JavaScript routines during type and domain checking. For example, in translating to a C++ `uint32_t`, bindings may use the `Uint32Value` method, which could invoke a JavaScript "upcall" (i.e., a call *back* into the JavaScript layer). JavaScript gives users extreme flexibility in redefining fundamental language methods, which makes it hard to know all methods that an upcall can transitively invoke, and makes it easy for attackers to circumvent security and correctness checks. For example: bindings may check that a start index is within the bounds of an array before calling `Uint32Value` to get the value of an end index. The `Uint32Value` call, however, may be hijacked by a malicious client to change the value of the start index, invalidating all previous bounds checking.

These bugs are neither hypothetical nor easily avoidable. Our checkers find numerous exploitable security holes in both Node.js and Chrome, heavily-used and actively developed code bases. Furthermore, security holes in binding code may be significantly more dangerous than holes in script code. First, these bugs render attacks more generic: given an exploitable binding bug, attackers need only trigger a path to that bug, rather than craft an entire application-specific attack. Second, binding flaws do not appear in scripts themselves: a script implementor can write correct, flawless code and still introduce security errors

Violation type	Possible consequence
Crash-safety	DOS attacks, including poison-pill attacks [45]; breaking language-level security abstractions, including [41, 42, 94, 109], by introducing a new covert channel.
Type-safety	Above + type confusion attacks which, for example, can be used to carry out remote code execution attacks.
Memory-safety	Above + memory disclosure and memory corruption attacks which, for example, can be used to leak TLS keys [28] or turn off the same-origin policy [82].

Table 1—The three types of binding bugs that we describe

if their code calls flawed runtime routines. As a result, writing secure scripts requires not only understanding the language (already a high bar for many), but also knowing all of the bugs in all of the versions of all of the runtime systems on which the code might run.

To address this threat, this paper makes two contributions:

1. A series of effective checkers that find bugs in widely-used JavaScript runtime systems: Node.js, Chrome's rendering engine Blink, the Chrome extension system, and PDFium. We show how bugs lead to exploitable errors by manually writing 81 exploits, including multiple out-of-bounds memory accesses in Node.js and use-after-frees in Chrome's PDFium (two of which resulted in $6,000 in bug bounties).
2. A backwards-compatible binding-code library that wraps the V8 JavaScript engine's API, preventing bugs without imposing significant overhead. Our library does not break any of Node.js's over 1,000 tests or the test suites of 74 external Node.js-dependent modules. By design, the migration path is simple enough that we are able to automatically rewrite a portion of Node.js's bindings to use our safe API.

While we focus on (V8-based) JavaScript runtime systems, JavaScript is not special: other scripting languages have essentially identical architectures and face essentially identical challenges; it would be remarkable if these languages did not contain essentially identical flaws. Therefore, we believe that other high-level language runtimes (e.g., those for Ruby and Python) stand to benefit from lightweight checkers and more principled API design.

2 The Problems with Binding Code

In this section we introduce binding code and explain how bugs in bindings can lead to violations of JavaScript's crash-safety, type-safety, and memory-safety—and how these safety violations manifest as security holes. Crash-safety violations, the least severe, can enable JavaScript code—e.g., ads in Chrome—to carry out denial-of-service attacks. They also provide a *termination* covert channel that attackers can leverage to bypass language-level JavaScript confinement systems, such as [10, 41, 42, 94].[1] Type- and memory-safety bugs have even more severe security implications. For example, use-after-free bugs in Blink and PDFium are considered "high severity" since they may "allow an attacker to execute code in the context of,

[1] A crash or its absence can signal whether a secret is `true` or `false`.

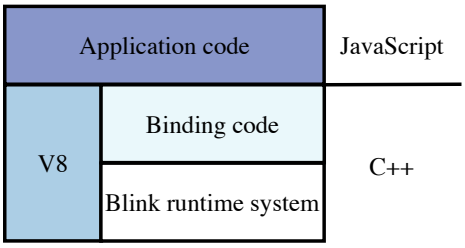


Figure 1—The Blink runtime system uses the V8 JavaScript engine to execute JavaScript application code. Blink also uses V8's APIs to extend the base JavaScript environment with new functionality and APIs, such as the DOM. This code—which bridges the JavaScript application code and Blink's C++ runtime—is binding code.

or otherwise impersonate other [website] origins" [84]. Table 1 summarizes the security consequences of these classes of bugs. In §4 we will discuss the precise security implications of safety violations with respect to the systems that we analyze.

We start with an overview of how binding code works in runtime systems and how untrusted JavaScript application code can call into the trusted C++ runtime system to exploit binding bugs. We find that these bugs often arise because JavaScript engines like V8 make it easy for developers to violate JavaScript's crash-, type-, and memory-safety; even V8's "hello world" code examples depend on hard-crashing functions [105]. We conclude with a detailed overview of V8-based binding functions.

Runtime system binding bugs. Runtime systems use JavaScript engines to execute application code written in JavaScript. For example, the Chrome rendering engine, Blink, relies on the V8 engine to interpret and run JavaScript code embedded in web pages as `<script>` elements. The JavaScript application code embedded in the `<script>` elements can use APIs like the Document Object Model (DOM), a representation of a web page, to modify the page and content layout. Binding code makes these modifications possible: Blink developers use the V8 engine API to extend the JavaScript application code's environment with such new functionality. Figure 1 illustrates the role that binding code plays in the interaction between the runtime system, the JavaScript engine, and the application.

To explain the challenges with preserving JavaScript's crash-, type-, and memory-safety in bindings, we walk through how to implement and expose a simplified version of the `Blob` interface to JavaScript [81]. This interface defines JavaScript `Blob` objects, which store binary data that can be sent over the network via other APIs (e.g., `XMLHttpRequest`). In order to efficiently pack data in memory, we implement `Blobs` in C++. We use the V8 API to expose `Blobs` to JavaScript—specifically, we use it to expose an interface for `Blob` creation and manipulation. In WebIDL [62], this interface is:

```
[Constructor(DOMString[] blobParts)]
interface Blob {
  readonly attribute unsigned long size;
  readonly attribute DOMString contentType;
  Blob slice(optional unsigned long start,
             optional unsigned long end);
};
```

Implementing this interface in C++ (as binding code) and exposing it to JavaScript allows applications to create Blobs from the array of strings (e.g., `new Blob(["foo", "bar"])`). It also allows JavaScript code to check the byte-length of a Blob (e.g., `blob.size`), get its content type (e.g., `blob.contentType`), and extract its subsets (e.g., `blob.slice(2)`).

The following binding-layer function implements the constructor for JavaScript Blobs:

```
1  void
2  blobConstr(const FunctionCallbackInfo<Value>& args)
3  {
4    // Get the current execution context
5    Local<Context> ctx
6      = args.GetIsolate()->GetCurrentContext();
7
8    // Extract first argument after type checking
9    if (args.Length() != 1 || !args[0]->IsArray())
10     // ... throw exception and return ...
11
12   Local<Array> blobParts = args[0].As<Array>();
13
14   // Create new C++ obj to back the new JS 'this' obj
15   Blob* blobImpl = new Blob(args.This());
16
17   // Add each string part to the blob
18   uint32_t n = blobParts->Length();
19   for (uint32_t i = 0; i < n; i++) {
20     // Get the ith element from array argument
21     Local<Value> part =
22       blobParts->Get(ctx, i).ToLocalChecked();
23     // Convert it to a string and add it to the blob
24     blobImpl->AddV8StringPart(part.As<String>());
25   }
26
27   // Return the receiver to the calling JS code
28   args.GetReturnValue().Set(args.This());
29 }
```

This binding code uses the V8 JavaScript engine APIs to handle JavaScript values in C++. For example, V8 represents the JavaScript arguments to the `blobConstr` function as an `args` array. `blobConstr` takes its first argument—an array of strings—and adds each string part to the underlying object. Unfortunately, `blobConstr` misuses V8 functions to introduce several errors, which we describe in the next paragraphs.

Violating JavaScript's crash-safety. `blobConstr` uses a hard-crashing function to extract elements from an array (line 20): `blobParts->Get(ctx, i).ToLocalChecked()`. This line can hard crash because `Get` returns either a wrapped value (in case of a successful get) or an empty wrapper (in case of failure)—and `ToLocalChecked` crashes when its receiver is empty. As a result, an attacker can write the following JavaScript to trigger a crash in any runtime system that exposes Blobs:

```
1 var evilarr = [];
2 Object.defineProperty(evilarr, 0, {
3   get: () => { throw 'die!'; }
4 });
5 var blob = new Blob(evilarr);
```

In this case, V8's `Get` function calls attacker-defined `get`; when `get` throws an error, `Get` returns an empty handle, and `ToLocalChecked` hard crashes. Attackers can use this kind of bug to carry out denial-of-service attacks—e.g., in Node.js, a third-party library can take down a web server while in Chrome, a third-party advertisement can essentially take down a site by crashing many users' tabs.

These security risks are not present in the base JavaScript language, since JavaScript itself is *crash-safe*: it will never hard crash. Instead, errors—even stack frame exhaustion—manifest as catchable exceptions. In contrast, in C++ code, failing gracefully requires nontrivial effort on the part of the programmer; bindings introduce the possibility of hard crashes to an otherwise crash-safe language.

Unfortunately, hard-crashing bindings can result from the design of the binding layer API itself. For example, some of V8's type-safe casting APIs are *not* crash-safe. These functions (e.g., `ToLocalChecked`, above) are supposed to convert from V8's wrapper types to unwrapped types. Developers have two options when confronted with a wrapper: they can either (1) inspect it and, if it is empty, throw an exception back to JavaScript, or (2) use the V8 function that converts wrapped to unwrapped but hard crashes when the wrapper is empty. The second choice is easier, so real bindings often follow this (unsafe) pattern.

Violating JavaScript's type-safety. An attacker could use the following JavaScript code to trigger a type-safety violation in the Blob bindings:

```
1 var evilarr = [3, 13, 37];
2 var blob = new Blob(evilarr);
```

On line two, the attacker calls the `Blob` constructor with an array of numbers. This kicks off a call to the binding layer `blobConstr` constructor, which checks that it has received a single argument of type `Array`. Then, `blobConstr` extracts the first element of the `evilarr` array and casts it to a `String` using the `As<String>` method (line 23): `blobImpl->AddV8StringPart(part.As<String>())`. Since `evilarr`'s first element is a `Number` and not a `String`, the program segfaults when `blobConstr` tries to use the incorrectly cast value. Crashes are not the only possible ramifications of type-safety errors, however. §3 describes an exploit for Node.js that leverages a function that does not type check to perform an out-of-bounds write. Type confusion bugs can also enable other kinds of attacks (e.g., remote code execution) [16–19, 38].

These attacks are a direct result of violations of (a certain notion of) JavaScript's *type-safety*. JavaScript does not have a static type system and does not satisfy the standard notion of type-safety [79]. Still, it satisfies a weaker notion of dynamic type-safety: JavaScript code cannot misuse a value by reinterpreting its underlying type representation—e.g., `numbers` cannot be reinterpreted and used as `functions`. If code tries to misuse a `number` as a `function`, for example, the JavaScript engine will raise a `TypeError`. This weak type-safety protects JavaScript from, say, accidently reading data beyond an array's bounds or calling into unexpected or unsafe parts of the runtime.

Bugs that violate type-safety appear in real binding code; they are common because neither JavaScript nor C++ nor V8 help programmers use correct types. JavaScript, by design, does not employ any static type checking. C++ is statically typed, but this applies only to C++ code—the type checking does not extend to the JavaScript code invoking the binding functions. Finally, V8 gives all values coming from JavaScript the same `Value` type, and the C++ binding-layer developer must determine, at run time, whether objects are `Objects` or `Arrays` or `Uint32`s. If the developer forgets to check a `Value`'s type before using or casting it, they can introduce type confusion vulnerabilities.

Violating JavaScript's memory-safety. There are more concerns with our `Blob` implementation. The `Blob slice` function, for example, could introduce a memory bug. In order to return a subset `Blob`, `slice` must access the receiver `Blob`'s underlying binary data—a byte array. In accessing the array, the `slice` function must be wary to read only the data that is within bounds. The starting index and length are supplied by user JavaScript, however; if the `slice` function checks bounds *before* calling JavaScript methods that might invalidate invariants, it could introduce a memory bug (e.g., an arbitrary write vulnerability). Memory-safety bugs can be used to exfiltrate sensitive information such as web server TLS keys [28].

These vulnerabilities are not present in JavaScript without bindings: JavaScript is a garbage collected, *memory-safe* language, so it can only access memory that has already been initialized by the underlying engine. Furthermore, JavaScript code may only access memory in a way that preserves abstraction—for example, it should not be able to inspect local variables as encapsulated by closures [43, 64]. C++, in contrast, is *not* memory safe—and bugs in binding code make it possible for JavaScript code to violate JavaScript's memory-safety as well.

Most binding layer memory errors arise because JavaScript values adversely affect the data- or control-flow of binding functions that perform memory operations like `memcpy` or `delete`. These bugs are *not* typically caused by developers forgetting to validate incoming JavaScript values; in fact, most binding functions check arguments in some capacity. Rather, memory bugs often arise because developers misuse V8 functions that implicitly upcall back into JavaScript. Attackers may change invariants during these upcalls; if developers unwittingly assume that invariants are still true, they may introduce vulnerabilities. For example, a JavaScript array indexing operation in the binding layer sometimes triggers an upcall into user JavaScript; attackers can use this upcall to shorten the length of the array. If a binding code developer does not re-check the length invariant and instead iterates blithely forward, they introduce an out-of-bounds memory read vulnerability.

The challenge of writing memory-safe binding code is analogous to the problem of writing memory-safe concurrent C++ code. C++ binding code upcalling into JavaScript, which can, in turn, call into C++ binding code (and so on), is a form of cooperative concurrent programming. It is no surprise that concurrent (JavaScript) code can change shared memory and cause concurrent C++ code to thereafter violate memory-safety. Unfortunately, the V8 API is "deceptive": it does not make this concurrency explicit; it does not make it clear that certain API calls may trigger upcalls into JavaScript. We list the categories of "deceptive" upcalling functions in Table 2.

Detailed overview of V8-based bindings. For completeness—and because the V8 documentation is somewhat limited—we explain the implementation of `blobConstr` in full; the uninterested reader can skip this paragraph, but may find it useful as a reference in later sections. As with all binding-layer functions, V8 calls `blobConstr` with a *callback-info* object that contains the JavaScript receiver (`args.This()`) and the list of JavaScript function arguments (`args[i]`). The receiver is an instance of `v8::Object`, the class that V8 uses to represent JavaScript objects, while the arguments are `v8::Values`; the `v8::Value` super class is used to represent arbitrary JavaScript values, which may be `v8::Objects`, `v8::Numbers`, etc. Lines 8–10 ensure that our binding-layer function is called with a JavaScript array argument. If any of these checks fail, the function raises a JavaScript exception and returns early; V8 will throw this exception upon returning control flow to JavaScript. Otherwise, the binding function creates a new C++ `Blob` instance that will be used to store the binary data (line 14). This object, `blobImpl`, serves as a backing object for the newly created JavaScript object referenced by `args.This()`. Specifically, the C++ `Blob` constructor uses the V8 API to store a pointer to `blobImpl` in one of the internal fields of the receiver object; this field is not accessible to JavaScript. The internal field ensures that whenever JavaScript calls a binding-layer `Blob` function, the bindings can retrieve the underlying C++ object (`blobImpl`) from the JavaScript object (`args.This()`). It *also* allows bindings to register a garbage collection (GC) callback with the V8 engine. When the V8 garbage collector collects the JavaScript `Blob` object, it will call the registered callback to free the corresponding C++ object. After allocating the C++ `Blob` object, `blobConstr` iterates over its array argument and adds the individual string elements to the blob (lines 17–23). Lastly, it returns the corresponding JavaScript object (line 27) that V8, in turn, hands off to the JavaScript code that called the constructor.

Summary. The bugs that our checkers target (§3) and our API aims to prevent (§5) are patterns caused by violations of three fundamental JavaScript properties: crash-, type-, and memory-safety. These violations come up repeatedly in the JavaScript systems that we analyze; in fact, our checkers automatically identified 81 real bugs. In the next section, we walk through a number of bugs, the automatic checkers that detect them, and the proof-of-concept attacks that trigger them. Afterwards, in §4, we contextualize these bugs and their security implications by describing the attacker models of the various systems that we analyze.

3 Static Checkers for Finding and Exploiting Vulnerabilities

In this section, we present static checkers for binding code and proof-of-concept exploits for the bugs that they find. The checkers, which are tailored for the systems on which they run (e.g., Node.js or PDFium), analyze a parse tree of the program source and point out potential errors. Then, as an attacker might,

Feature	Description	Example
Getters, setters	Untrusted JavaScript code can define a custom function to be called when an object property is set/get.	`object->Get(context, i)`
Prototypes	JavaScript code can poison global prototypes such as `Array.prototype` and `Object.prototype` which are then called on property access. This is especially useful when getters/setters cannot be defined.	`array->Set(context, i)`
toPrimitive, toString	JavaScript code can define a function that is called when the JavaScript engine tries to implicitly cast an object to a primitive value or `String`.	`val->Uint32Value()`
Proxy traps	JavaScript code can pass in JavaScript proxies instead of objects. This allows it to trap operations such as `set`, `get`, `delete`, `hasOwnProperty`, etc.	`obj->HasOwnProperty(context, prop)`

Table 2—Tricky JavaScript edge cases that can form the basis of exploits. All of these cases can result in upcalls to user-defined JavaScript.

we create JavaScript that triggers the binding bugs. This process demonstrates that:

1. Binding code is an exploitable weakness in JavaScript runtime systems. We write five checkers that identify 81 exploitable bugs (with 30 false positives) in binding code, including 3 use-after-free errors in PDFium. Chrome takes these errors seriously: we were awareded $6,000 in bounties for two UAF error reports [24, 25].
2. Binding code is *easy* to exploit. The static checkers in this section are at most a hundred lines long, and we typically create them in a day or two. In fact, after examining the V8 documentation for a couple days, we believe that attackers or developers could easily conceptualize and create a checker. Once our checkers identify vulnerabilities, it often takes fewer than a hundred lines to exploit them.

Checker implementation. We implement the checkers in μchex [6], a language-agnostic static checking system, because it allows us to build small and extensible checkers. Our checkers are tiny since they ignore most of the language that they check. Instead, they only parse and analyze portions of the language relevant to the checker properties themselves. The simplicity of the framework allows us to prototype quickly and to adapt checkers from one runtime system to another with little work; for example, the Node.js and Chrome invocations of one of our checkers differ by one line of code. Finally, like many static systems, our checkers are unsound: they do not guarantee the absence of bugs in any system that they check.

Checker results. Our checkers flag *binding layer functions* that unsafely use JavaScript engine APIs—V8 and shims around V8. We run the checkers on a Node.js master version from early September 2016 [74] and Chrome version 56.0.2915.0 (Developer build) [13]. We do not check test- and debugging-related files. Additionally we omit any files that have been removed (e.g., due to refactoring) from more recent versions of the runtimes—Node.js 7.7.4 and Chrome 56.0.2924.87—to simplify the bug reporting process. For each of the checker results, we manually inspected the flagged code and categorized the results. Some flags were clear *false positives* (e.g., due to our simple intra-procedural analysis). Others we confirmed by writing exploits; though most of the *exploited* bugs were in binding functions directly callable by JavaScript, in some cases we exploited helper functions that are only called by other binding code (see below) to demonstrate feasibility. Finally, we marked some results *suspicious*: we believe many of these to be exploitable, but since we do not have exploits confirming them, we count them separately. The extended version of this paper [5] will contain the updated classification of these results as we explore them in more detail.

We outline the results in Table 4 and the checkers that find them in Table 3. The checkers look for three different classes of errors. First, violations of *crash-safety*: one checker identifies hard-crashing asserts that depend on user JavaScript, and the other flags hard-crashing conversions from V8 types. Next, *type-safety*: a checker flags variables that are cast without being type-checked. Finally, *memory-safety*: one checker flags memory operations that are affected by upcalls back into JavaScript, while the other flags instances where JavaScript can force collection of a variable still used by C++. Our reports and exploits are not intended as worst case scenarios for how attackers may exploit bugs. For example, while we crash unchecked type bugs, attackers may instead leverage them to carry out remote code execution attacks. We provide links to all confirmed bugs in the extended version of this paper [5].

3.1 Crash-Safety Violations

We write two basic checkers that flag violations of JavaScript's crash-safety in the binding layer: one checker identifies hard-crashing Node.js asserts that depend on user JavaScript, and the other flags hard-crashing conversions from V8 types. We adapt the latter hard-crashing conversion checker slightly for each of the systems that we check (Node.js and Chrome's extension system, PDFium, and Blink), a process that we describe further in the extended version of this paper. We run the checkers on the systems' source code and craft JavaScript to trigger the bugs that the checkers detect.

Hard-crashing checks on user-supplied input. This checker identifies instances in Node.js where hard-crashing checks (e.g., `CHECK`) depend on user-supplied JavaScript input. For example, the checker flags the following binding code bug [93]:

```
/* src/node_buffer.cc */
size_t Length(Local<Value> val) {
  CHECK(val->IsUint8Array());
  Local<Uint8Array> ui = val.As<Uint8Array>();
  return ui->ByteLength();
}
```

Checker Type	Problem	Example
Crash	Attacker can trigger hard crashing asserts	`CHECK(js)`
Crash	Attacker can trigger hard crashing conversions	`js->Get(..).ToLocalChecked()`
Types	Attacker can trigger bad cast	`notString.As<String>()`
Memory	Attacker can alter memory operations that depend on implicitly casting functions	`memcpy(js->ToUint32()...)`
Memory	Attacker can free object still being used by C++	`ptr*; js->ToUint32(); use(ptr)`

Table 3—The binding code bugs that our checkers identify.

Checker	System	Flagged	Exploited	Suspicious	False
Hard crash	Node.js	68	37	19	12
	PDFium	13	3	5	5
	PDFium (lib)	39	29	10	0
	Extensions	2	0	0	2
	Blink	6	1	2	3
	All	128	70	36	22
Type	Node.js	8	4	0	4
	PDFium	0	0	0	0
	Extensions	2	0	0	2
	Blink	3	0	2	1
	All	13	4	2	7
Memory	Node.js	5	4	0	1
	PDFium	9	3	6	0
	All	14	7	6	1
Total		155	81	44	30

Table 4—Bugs in JavaScript runtime systems. Our counts are conservative in several ways: (1) we do count multiple occurrence of a particular bug kind (e.g., crashing) for a single function even though in practice a crashing function, for example, can be crashed in multiple ways, and (2) we count bugs that are more difficult to trigger (e.g., because they are deeply nested) as suspicious or false positives, depending on the seeming difficulty.

The `Length` function takes a user-supplied V8 `Local<Value>`—V8's C++ base "unknown" type for a JavaScript value—as its argument; `val` is supposed to be a JavaScript byte-array whose length the function will determine. On line 246, the function `CHECK`s that `val` is actually of the correct type, hard-crashing when this is not the case. `Length` is not directly exposed to JavaScript—it is a helper function that various other Node.js binding functions use. Unfortunately, neither the other binding functions nor the JavaScript layer that calls into binding code safely enforce `val`'s type. As a result, we can sneak a malicious value argument through to trigger a crash in the `Length` function. The following code triggers the crash:

```
1 const dgram = require('dgram');
2 const util = require('util');
3 // Create object that passes instanceof Buffer check
4 function FakeBuffer() { }
5 util.inherits(FakeBuffer, Buffer);
6 const message = new FakeBuffer();
7 // Pass object to code that eventually calls Length
8 dgram.createSocket('udp4').send(message, ...);
```

The `send` function on line 8 is what eventually triggers the bug in the `Length` function above. Lines 1-6 are boilerplate to create a `message` that will fool JavaScript-layer type checks: since our message is an instance of `FakeBuffer`, which inherits from `Buffer`, it passes the JavaScript function `send`'s type checks. `send` eventually passes `message` to the binding-layer `UDP::DoSend` function, which calls `Length(message)`. This causes a hard crash: `message` is *not* a `Uint8Array`, so the `CHECK(val->IsUint8Array())` fails.

To detect hard-crashing `CHECK` bugs, the checker does a forward, intra-procedural analysis of each binding layer Node.js function. Its main computed data structure is the set of all variables that come from user JavaScript. If it detects a user JavaScript variable in a hard-crashing macro (e.g., `CHECK`, `ASSERT`, etc.), it flags an error. This simple checker works well for Node.js because, in this system, it is often clear (1) which arguments are user-supplied JavaScript and (2) how these arguments are passed in from the JavaScript layer. Furthermore, Node.js developers consistently use hard-crashing asserts in place of safe `if`-statements. In contrast, when we tried running a version of the checker on Chrome code, we drowned in a deluge of confusing reports: Chrome thoroughly performs safe checks before calling hard-crashing functions. In Blink most of these safe checks are automatically generated; in the Chrome extension system the checks are performed in JavaScript from WebIDL-like interface descriptions.

This checker flags 65 errors, 35 of which we confirmed by writing crashing exploits for Node.js. We examined 9 reports and decided that they were difficult or impossible to trigger largely because the binding functions are "monkey-patched" with safe type-checking JavaScript code before any application code can run. Of the remaining checker flags, we mark 19 as suspicious. Most of these functions are inner, helper binding functions that are more challenging to trigger than functions directly exposed to JavaScript. We could have suppressed reports for such non-public functions, but the `Length` exploit above demonstrates that it is very feasible to trigger bugs that are several layers deep in the JavaScript-C++ call stack. Moreover, `Length` is not the only deep Node.js bug we have triggered. Hence, we argue for more defensive (or less explicitly hard-crashing) bindings [90].

Hard-crashing conversions from `Maybe` types. This checker identifies instances where binding code unsafely uses hard-crashing conversions. In other words, it flags binding functions that use type conversion methods that hard-crash in the case of unexpected types. For example, the `ToChecked` function converts JavaScript values from `Maybe<T>` types—types that signal success (value of type `T`) or failure (`Nothing`)—to `T`

types, crashing when the value is Nothing. Our checker flagged the following Chrome hard crash [23]:

```
/* chrome/third_party/WebKit/Source/bindings/
core/v8/ScriptCustomElementDefinition.cpp */

template <typename T>
static void keepAlive(v8::Local<v8::Array>& array,
                      uint32_t index,
                      const v8::Local<T>& value,
                    ScopedPersistent<T>& persistent,
                      ScriptState* scriptState) {
  if (value.IsEmpty())
    return;

  array->Set(scriptState->context(), index,
    ↪  value).ToChecked();
  ...
}
```

The ToChecked call on line 94 will hard crash if its receiver is Nothing. In other words, if array->Set() returns Nothing, the method call ToChecked() on it will result in a crash. Getting array->Set(...index, value) to return Nothing is trivial. The Set function normally sets the index property of array to value (e.g., array[0] = 0). JavaScript, however, allows users to instead define custom a *setter* function to be called whenever the property is accessed. Hence, if we re-define array's index property to be an exception-throwing setter, array->Set() will return Nothing—and the tab hard crashes.

Triggering this error is a bit more subtle, though—array is not a value that comes directly from attacker-controlled JavaScript (e.g., from a web site). Instead, array is freshly created in the C++ binding code that calls keepAlive:

```
/* chrome/third_party/WebKit/Source/bindings/
core/v8/ScriptCustomElementDefinition.cpp */

v8::Local<v8::Array> array =
  ↪  v8::Array::New(scriptState->isolate(), 5);
keepAlive(array, 0, connectedCallback,
  ↪  definition->m_connectedCallback,
  ↪  scriptState);
```

On line 124, the programmer uses the New constructor to create a new array in C++. Luckily, attackers can affect the Set function even on freshly-created object. JavaScript allows developers to define properties on global prototypes (e.g., Array.prototype or Object.prototype) that are inherited by all newly created objects in the same context; attacks that take advantage of prototypes are called *prototype poisoning* attacks [2]. The following malicious JavaScript defines an exception-throwing setter function for property 0 of the Array prototype:

```
Object.defineProperty(Array.prototype, 0, {
  set: newValue => { throw "die!"; },
  enumerable: true
});
```

If we include this JavaScript in a malicious web page, all JavaScript arrays in the context will contain an exception-throwing setter as their property 0—including arrays in bindings. Therefore, when the binding code tries to access the 0 property of a freshly created array by calling array->Set(0, ...).ToLocalChecked(), the tab will crash.

The checker is implemented as a forward, intra-procedural, flow-sensitive traversal of the parse tree. Its main computed data structure is the NothingSet, which contains variables that may be Nothing; it flags an error when it sees a hard-crashing conversion call (e.g., ToLocalChecked) on a variable in the NothingSet. For each binding code function, the checker:

1. Initializes AlterSet to the empty set. The AlterSet is the set of variables whose upcalls malicious JavaScript may control; any time a user-controlled JavaScript object can override a method (e.g., js->Set()), we add that object to the AlterSet.
2. Adds user-controlled JavaScript Object or Value arguments to the AlterSet.
3. Initializes NothingSet to the empty set. NothingSet is the set of variables initialized to the result of upcalls on user-controlled JavaScript. On encountering the line x = array->Set(...), if array is in the AlterSet, the checker adds x to the NothingSet: a malicious array could override its Set function to throw an exception, leaving x as a Nothing value.
4. Removes variables from the AlterSet when they are type checked and from the NothingSet when they are compared with Nothing.
5. Flags an error any time a hard-crashing conversion (ToChecked, ToLocalChecked, and FromJust) is called on an item in the NothingSet. We can force items in the NothingSet to be Nothing, triggering a hard crash when execution hits the ToChecked.

This checker flags 27 errors, 6 of which we confirmed by writing crashing exploits—2 for Node.js, 3 for PDFium and 1 for Blink. As with our previous checker, we mark internal, hard-to-get-to functions as suspicious—in total, 7. Of the 27, 13 are false positives. Again, most false positives arise because some bugs are on impossible paths; for example, the three Blink false positives for this checker were due to series of checks performed in the functions calling the seemingly unsafe binding code. We believe that adding inter-procedural analysis to these checkers can address most of the false positives.

After looking at the initial reports for this checker, we found that it flagged hard crashes deep in PDFium's V8 wrapper library. The library wraps typical V8 functions like Uint32Value to accept PDFium JavaScript type arguments (e.g., CJS_Values) instead of V8 type arguments. We used this information to write a new 40-line, PDFium-specific twist on the original checker. The new checker identifies cases where wrapper functions are called on un-type-checked user CJS_Value arguments—usually something along the lines of "params[0].ToInt()." We identified 39 such cases, 29 of which we have triggered by embedding

JavaScript in PDFs. For example, embedding the following line of code in a single PDF crashes all open PDF tabs:

```
app.beep({ [Symbol.toPrimitive]() { throw 0; } })
```

As a final experiment, we gathered all of our Node.js crashing exploits and ran them on a different Node.js version, one that uses Microsoft's ChakraCore JavaScript engine (instead of V8) [12, 70]. Out of 37 crashing exploits, all still crash on Node.js ChakraCore. This gives us confidence that we will be able to adapt our checkers from one JavaScript engine to another relatively easily.

3.2 Type-Safety Violations

Casts without type checking. This checker flags violations of JavaScript's weaker notion of type-safety: it looks for cases where C++ code casts binding-layer JavaScript values to C++ V8 types *without* checking if values are of those types. For example, the checker detects the following Node.js binding bug, which attackers can use to carry out a type confusion attack:

```
/* node/src/node_buffer.cc */

template <typename T, enum Endianness endianness>
void WriteFloatGeneric(const
  ↪  FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);
  bool should_assert = args.Length() < 4;
  if (should_assert) {
    THROW_AND_RETURN_UNLESS_BUFFER(env, args[0]);
  }
  Local<Uint8Array> ts_obj =
  ↪  args[0].As<Uint8Array>();
  ArrayBuffer::Contents ts_obj_c =
  ↪  ts_obj->Buffer()->GetContents();
  ...
}
```

On lines 819–822, the code conditionally checks the type of the first argument (`args[0]`). Unfortunately, the condition `should_assert` depends on the user—`should_assert` is defined based on the number of arguments the user provides—so attackers can bypass the type check. On line 823, the un-type-checked `args[0]` is cast to a `Uint8Array`. Finally, from line 824 forward, `WriteFloatGeneric` calls methods on the cast object—so a well-chosen argument can amount to arbitrary code execution.

We trigger this bug using the public `buffer` API, which attempts to apply JavaScript-layer checks before calling into the buggy binding function:

```
/* node/lib/buffer.js */

Buffer.prototype.writeFloatLE = function
  ↪  writeFloatLE(val, offset, noAssert) {
  val = +val;
  offset = offset >>> 0;
  if (!noAssert)
    binding.writeFloatLE(this, val, offset);
  else
    binding.writeFloatLE(this, val, offset, true);
  return offset + 4;
};
```

This JavaScript-layer code converts `val` and `offset` to number values in lines 1245 and 1246, but does nothing to type check the receiver `this`, which should be a `Buffer`. Then, depending on the user-supplied `noAssert`, it calls the binding layer `writeFloatLE` (which calls the buggy binding function `WriteFloatGeneric`) with either three or four arguments. In the latter case, the binding layer's `should_assert` argument is `false`, disabling type checking and triggering the incorrect cast. The following exploit triggers this bug:

```
Buffer.prototype.writeFloatLE.call(0xdeadbeef, 0,
  ↪  0, true);
```

This code snippet triggers a call to `WriteFloatGeneric` with `0xdeadbeef` as `args[0]`, `0` as `args[1]`, etc. The exploit will cause a type confusion attack: it almost always hard crashes, but a well-crafted argument (in place of `0xdeadbeef`) can cause the `Buffer` method call on `ts_obj` to execute meaningful code. Attackers could embed this seemingly benign code deep in the dependency tree of publicly available, anonymous, and unsigned Node.js packages and go unnoticed [83, 91].

The type-casting checker is implemented as another intra-procedural forward code traversal. Its main computed data structure is the set of un-type-checked user arguments; whenever it sees a cast of an un-type-checked argument, it flags an error. For each binding layer function, the checker:

1. Initializes the set of `UncheckedTypes` to the empty set.
2. Adds any user-controlled JavaScript arguments to the set of `UncheckedTypes`.
3. Removes any argument that is type checked from the `UncheckedTypes` set.
4. Flags an error when a variable in `UncheckedTypes` is cast using V8's `As<T>()` function.

The checker flags 13 bugs; we confirm 4 by crafting exploits for them. Most false positives—especially in the Chrome systems—occur because of impossible paths into our flagged reports; inter-procedural checking and checking between JavaScript-layer and C++-layer functions would make our reports far cleaner.

3.3 Memory-Safety Violations

The checkers in this section identify memory-safety violations. They look for instances where user JavaScript can alter values used in memory operations and instances where user JavaScript can force the deallocation of objects still used by C++ code. Attackers could leverage these sorts of bugs to, for example, read the TLS keys of a Node.js web application.

Memory operations dependent on implicit casts. V8 provides built-in functions that return C++ representations of JavaScript values. For example, the statement

"`uint32_t y = x->Uint32Value()`" assigns y to the C++ unsigned integer value of x. Programmers occasionally depend on the results of these functions for sensitive operations such as memory allocations (e.g., `malloc(y)`). If the JavaScript receiver is a primitive type (e.g., x is a `Number`), this is fine; if the receiver is a non-primitive type, calls like `x->Uint32Value()` can be dangerous. In particular, when x is an `Object`, the JavaScript engine upcalls the `x[Symbol.toPrimitive]` function (if defined) within the `Uint32Value` function. Attackers can leverage this function in order to, say, evade bounds checks. We will call functions like `Uint32Value`—functions that implicitly cast a value by calling `Symbol.toPrimitive`—*implicitly casting*.

This checker flags instances where the binding layer does *not* perform type checking before depending on the result of an implicitly casting function for a memory operation. It identifies an out-of-bounds write error in Node.js's buffer `fill` function, which fills in a user-provided buffer `buf` with a single value starting at a `start` index and going to an `end` index [92]. `fill` must ensure that both `start` and `end` are within the bounds of `buf`. Bounds checking, though, is not as straightforward as it seems: `fill` tries to implement some checking in the JavaScript layer and some in the C++ binding layer. We give the JavaScript checks below:

```
   /* node/lib/buffer.js */

662 function fill(val, start, end, encoding) {
663   ...
664   // bounds checks
665   if (start < 0 || end > this.length)
666     throw new RangeError('Out of range index');
667   if (end <= start)
668     return this;
669
670   // calls binding code
671   binding.fill(this, val, start, end, encoding);
672 }
```

The checks that start on line 664 are supposed to ensure that the `start` and `end` values are within the bounds of the buffer. After these checks, on line 671, the JavaScript code calls the C++ binding layer implementation of `binding.fill` [92]:

```
   /* node/src/node_buffer.cc */

604 void Fill(const FunctionCallbackInfo<Value>& args) {
605   size_t start = args[2]->Uint32Value();
606   size_t end = args[3]->Uint32Value();
607   size_t fill_length = end - start;
608   ...
609   CHECK(fill_length + start <= ts_obj_length);
610
611   if (Buffer::HasInstance(args[1])) {
612     SPREAD_ARG(args[1], fill_obj);
613     str_length = fill_obj_length;
614     memcpy(ts_obj_data + start, fill_obj_data,
       ↪  MIN(str_length, fill_length));
615     ...
616   }
617 }
```

Lines 605 and 606 get the unsigned integer values of arguments two and three, the start and end index of the fill operation. If the start and end indices are unsigned 32-bit integers like 0 and 5, everything is fine; if an attacker passes in an object, though, they can take advantage of implicit casting to call their malicious `Symbol.toPrimitive` function. In doing so, they can return values that evade the single bounds check on line 609, a check that tries to ensure that the length of the write is less than the length of the buffer object.

In the next paragraphs, we will explain *how* a malicious `Symbol.toPrimitive` function returns a negative value; in this one, we will explain what happens when `Symbol.toPrimitive` returns such a value for `start` (though `end` can be abused the same way). Since `start` (line 605) is an unsigned `size_t`, a negative value will cause it to overflow. When `start` is very large, the addition in the bounds check (line 609) wraps around: `fill_length + start` becomes less than `ts_obj_length`. Since the bounds check passes, the `memcpy` starting at location `ts_obj_data + start` executes; the negative value passed in for `start` clearly controls the location of the write.

The following exploit code carries out this attack:

```
1  var buff = Buffer.alloc(1);
2  var ctr = 0
3  var start = {
4    [Symbol.toPrimitive](hint) {
5      if (ctr == 0) {
6        // evade the check in lib/buffer.js
7        ctr = ctr + 1;
8        return 0;
9      } else {
10       // in the C++ implementation of fill:
11       return -1;
12     }
13   }
14 };
15 buff.fill(victim, start, 1);
```

Line 3 defines an object `start` to be passed in as the start value of the write (line 15). Since there is no type checking in either the JavaScript or C++ `fill` functions, our `start` is a legal argument value. On line 4, we define the malicious `Symbol.toPrimitive` function. Now, whenever someone tries to get the number value of `start`, our function will be called. This function uses the counter `ctr`, defined on line 2, to evade bounds checking in the JavaScript code. It returns a benign value of 0 the first time it is called. The next time `start[Symbol.toPrimitive]` is called, however—in the C++ binding code—the function returns a negative value.

To identify such errors, our checker does a forward traversal of each function. Its main computed data structure is the set of `DangerousValues`, values that are the results of upcalls into user JavaScript. We flag a bug if a memory operation depends on a dangerous value. The checker:

1. Initializes the `UncheckedTypes`, the set of variables whose types have not been checked, to the empty set.
2. Adds any user-controlled JavaScript arguments to the `UncheckedTypes` set.

3. Removes any argument that is actually type checked from the UncheckedTypes set. For example, the following line of code would cause the checker to remove argument arg from the set of UncheckedTypes: `if (!arg->IsUint32()) return`.
4. Adds the results of any implicitly casting calls on UncheckedTypes to the DangerousValues set.
5. Adds any values that are assigned using values in DangerousValues to DangerousValues: if x is in DangerousValues, the line `y = x + 5` will cause y to be added to DangerousValues.
6. Flags an error if any value in DangerousValues appears in an expression that is used as an argument to a memory operation (e.g., malloc or memcpy).

The checker flags 5 errors, of which 4 are true and 1 is false. All of these reports are in Node.js. Two of our true bugs appear in template code—WriteFloatGeneric and ReadFloatGeneric—that is actually used by four exposed binding layer functions: WriteFloatLE, WriteFloatBE, ReadFloatLE, and ReadFloatBE. We write exploits that resemble the Fill exploit in this section for all 4 errors. The false positive, in Node.js's crypto bindings, arises because these bindings do careful invariant re-checking that accounts for wraparound.

PDFium use-after-frees. This checker flags potential use-after-free errors, instances in PDFium bindings where malicious user JavaScript can force an object to be freed while C++ maintains a live reference to that object. Consider the following bug [24]:

src/third_party/pdfium/fpdfsdk/javascript/Annot.cpp

```
72 bool Annot::name(IJS_Context* cc, CJS_PropValue& vp,
   ↪  CFX_WideString& sError) {
73   CPDFSDK_BAAnnot* baAnnot =
     ↪  ToBAAnnot(m_pAnnot.Get());
74   if (!baAnnot) return false;
75   ...
76   CFX_WideString annotName;
77
78   vp >> annotName;
79   baAnnot->SetAnnotName(annotName);
80 }
```

This bug appears in the binding layer of PDFium's JavaScript API, an API that allows JavaScript embedded in PDFs to make changes to the underlying PDF representation. The name function above, for example, is supposed to set the name of a PDF annotation. name's CJS_PropValue& argument, vp, is a user-supplied JavaScript value; we can craft a JavaScript vp argument that causes pointer baAnnot to be used (line 79) after it is freed (line 78).

The function initializes baAnnot and checks that it is non-null. The next two lines are supposed to assign annotName, a special type of PDFium String, to the value of baAnnot, the annotation name. This assignment uses the overloaded "`>>`" operator; when annotName is a CFX_WideString, the operator calls the function ToCFXWideString with vp as the receiver. ToCFXWideString is part of PDFium's layer which wraps the V8 API: internally, this function just calls V8's ToString on the vp object. Naturally, an attacker can provide their own definition of ToString function to delete baAnnot and trigger the UAF.

For example, the following exploit is embedded as JavaScript code into a PDF with radio-button widgets:

```
1 const annots = this.getAnnots();
2 annots[0].name = {
3   toString: () => {
4     this.removeField("myRadio");
5     gc();
6     return false;
7   }
8 }
```

In this snippet, `annots[0]` corresponds to vp in the binding layer. We override name's toString function to remove the "myRadio" field, which corresponds to baAnnot in the binding layer. Now, there are no more JavaScript references to "myRadio"; when we call gc and force garbage collection on line 5, the GC frees the memory associated with "myRadio". This memory, however, is *also* associated with baAnnot in the binding layer. Unfortunately, when control returns to the bindings, baAnnot is used without any checks (`baAnnot->SetAnnotName(annotName)`)—even though the JavaScript call already caused it to be freed.

Our UAF checker does a forward traversal of each PDFium function parse tree. Its main computed data set is FreeablePointers, pointers that may have been freed in user JavaScript; it flags a bug whenever a freeable pointer is used. For each function, it:

1. Initializes the set of all pointers that have been initialized, InitPointers, to empty.
2. Initializes FreeablePointers, the set of pointers that may be altered by user JavaScript, to empty.
3. Adds newly initialized pointers to InitPointers (e.g., after the line `BAAnnot* x = foo()`, x is in InitPointers).
4. Adds all InitPointers to FreeablePointers when it encounters a PDFium function that can upcall into user JavaScript. For example, it adds x to FreeablePointers after the line `jsval.ToInt()`.
5. Flags an error when FreeablePointers are used (e.g., at the line `*x`).

This checker moves x in InitPointers to FreeablePointers when x's initialization is followed by an upcall into JavaScript. The checker does so because the JavaScript upcall may remove the JavaScript object associated with x and then force garbage collection, thereby making any subsequent uses of x in C++ a use-after-free violation. In our checker implementation, we consider any potential upcall (e.g., `x.ToInt()`) to be a feasible upcall since PDFium does not perform any binding layer type checking. Therefore, we know we can almost always pass an Object with maliciously overridden methods to the function. We do not add obvious unique pointers to the InitPointers or FreeablePointers sets, since we cannot trigger a UAF attack on a unique pointer.

This checker flags 9 errors. We wrote exploits for 3 of them and mark the remaining 6 suspicious. All the suspicious bugs are easy to reach, but we are not sure which PDF fields can be removed from user JavaScript. We are in contact with PDFium developers about how to remove certain elements (such as annotations, above) from PDFs.

4 Runtime System Design and Attacker Models

In the previous sections, we outlined several classes of binding layer bugs; in this section, we contextualize the real-world impact of our results in the systems that we analyze—Blink, the Chrome extension system, Node.js, and PDFium. We also outline the attacker models that the systems assume and efforts they make to mitigate the effects of binding layer bugs. In some cases, we propose changes to their efforts and attacker models.

4.1 Blink

Chrome's rendering engine, Blink, relies on V8 to expose APIs (e.g., the DOM) to JavaScript web applications. Blink assumes that JavaScript application code may be malicious [14]—that it may, for example, try to leak or corrupt data of different origins by exploiting a bug in the binding layer. As a result, Blink treats type- and memory-safety violations as security concerns. Blink does not consider crashing bugs and denial-of-service attacks to be security errors because malicious JavaScript can always hang the event loop and deny service. Nevertheless, Blink tries to mitigate the risk and likelihood of all three categories of bug: they use automatically generated bindings, a C++ garbage collector, and out-of-process iframes (Figure 2a).

Blink addresses most type- and crash-safety binding bugs by automatically generating most of its bindings from WebIDL specifications of web platform APIs (e.g., the DOM, `XMLHttpRequest`, etc.). Once the generating templates are correct, generated code can perform type checking in a consistent, crash safe way, avoiding type confusion and hard-crashing bugs. Templates and WebIDL compilers may still be buggy [15], but they are more reliable than manual type checking.

Blink avoids memory leaks and use-after-free vulnerabilities with a garbage collector, called Oilpan, for C++ binding objects. Oilpan prevents memory errors that arise when binding layer functions call back into JavaScript, altering or removing pointers on which C++ code still relies [39].

Blink is also protected by Chrome's new out-of-process iframes (OOPIFs). OOPIFs isolate iframes with different origins in separate processes [22], reducing the severity of some attacks (e.g., by making it more difficult to leak cross-origin data)—even attacks that exploit binding bugs.

Unfortunately, neither OOPIFs nor the combination of code generation and garbage collection protect all binding layer Blink code. Some Blink bindings are still handwritten (since they need to manipulate or allocate JavaScript objects directly); these bindings are still vulnerable to programmer error. For example, we identified a crashing bug in the Custom Elements DOM APIs that we can trigger with crafted JavaScript.

OOPIFs are not a comprehensive defense either: Chrome only deploys them for high-profile websites [22], leaving the rest of the web unprotected. Moreover, OOPIFs only defend at coarse granularity, and many client-side, language-level mechanisms rely on JavaScript memory- and type-safety for fine-grained security [50, 60, 63, 94, 109]. As a result, a JavaScript attacker who can exploit binding bugs to break safety assumptions may violate these systems' language-level guarantees—even though the attacker cannot break Chrome's isolation guarantees.

4.2 Chrome Extension System

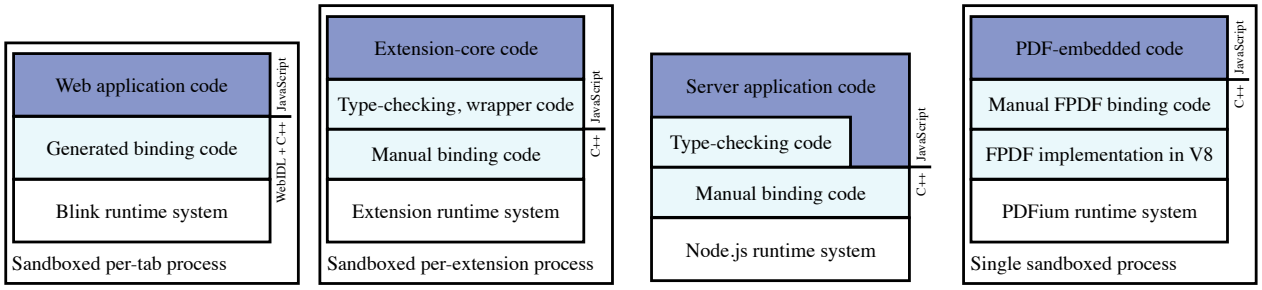
The Chrome extension system uses V8 to expose privileged APIs to JavaScript extension code (e.g., to allow extensions to create new tabs or read page contents on certain origins). Chrome assumes that extensions are "benign-but-buggy" [4]—that they may contain errors but are not intentionally malicious. The pages that extensions interact with, however, *may* be malicious; they may even try to exploit vulnerabilities in extension code. To address attacks from malicious pages, the runtime isolates the *core* part of the extension—the code that has access to privileged APIs—from the *content scripts* that directly interact with the page: Chrome runs the core extension in an isolated process. Though, for performance reasons, multiple extensions are placed in the same process [32].

Even with the extension system's isolation and privilege separation mechanisms in place, attackers have exploited extension system vulnerabilities and managed to abuse privileged APIs [20, 21, 60]. Unfortunately, binding-layer bugs can further amplify these exploit strategies. Type- and memory-safety vulnerabilities are particularly serious, since these classes of binding bugs may allow JavaScript code to use the privileged APIs of co-located extensions, otherwise not requested by the vulnerable extension nor approved by the user (for this extension). Crash-safety bugs, on the other hand, do not have security implications—they can only be used to crash the isolated extension process.[2]

The Chrome extension system uses binding layer defenses to reduce the risk of crash-, type-, and memory-safety bugs: it relies on a trusted JavaScript layer to do crash-safe type checking before calling into binding code (Figure 2b). Attackers may bypass the trusted JavaScript layer, though [20, 21]; a bug in the trusted JavaScript layer and a bug in the bindings combine to form a security vulnerability. Moreover, Chrome extension system does not use C++ garbage collection or code generation to eliminate binding bugs by construction.

We believe that Chrome extension system should assume a stronger attacker and treat extensions as potentially malicious code. Numerous extensions—used by millions of people—have turned out to be malicious [46, 48, 94, 103], while other popular extensions such as AdBlock Plus [51] have been sold to untrustworthy parties. Chrome currently does not assume malicious extensions in their threat model and, to make matters worse, automatically downloads extension updates as long as those updates do not request new privileges. Thus, if an attacker maintains a least-privileged extension, they can update that ex-

[2] Since Chrome notifies the user when an extension crashes, however, a malicious page may exploit hard-crashing bugs to annoy the user into disabling or uninstalling a targeted extension such as HTTPS Everywhere [33].



(a) Chrome's Blink relies on process isolation and automatic binding code generation to address binding-layer vulnerabilities.

(b) The Chrome extension system relies on a small, isolated, and trusted JavaScript layer to type check arguments before calling into hand-written binding code.

(c) Node.js implements most of its core libraries in JavaScript, atop a small hand-written binding layer. However, the binding layer is accessible to user JavaScript and the JavaScript layer is not isolated from application code.

(d) PDFium wraps the V8 API with a small, but less safe, C++ API that it then uses to expose APIs to JavaScript.

Figure 2—The binding layers and their defenses across JavaScript runtime systems. The trustworthiness of code decreases with color—white is the most trustworthy, while dark blue is the often untrusted JavaScript application code.

tension with code that leverages a binding layer bug to, perhaps, escalate the malicious extension's privileges. Chrome will automatically download this malicious update, and the extension will operate with unauthorized access to user information.

4.3 Node.js

Node.js is a JavaScript runtime system for building servers and desktop applications. The runtime uses V8 to expose APIs for filesystem, networking, and crypto utilities. Node.js (Figure 2c) exposes low-level binding APIs (e.g., `process.binding('fs')`) which JavaScript code, in turn, uses to implement the core standard libraries (e.g., `fs`). By implementing most code in a high-level, memory- and type-safe language instead of C++, Node.js makes it easier for developers to safely create new features.

Despite only implementing minimal machinery in C++, Node.js still struggles with binding layer bugs (§3). Node.js does *not* consider binding bugs to be security risks; to the best of our understanding, the Node.js attacker model assumes that JavaScript application code is benign.[3] However, discordantly, Node.js recently added support for zero-filling buffers. Zero-filled buffers make it more difficult for remote attackers to exploit benign but buggy application code that relies on the `buffer` library to disclose memory (in the style of Heartbleed) [1]. Binding layer vulnerabilities reintroduce the problem that zero-filling buffers are designed to fix; attackers can use binding bugs (e.g., in `buffer`) to read and write arbitrary parts of Node.js processes (§3).

[3]Personal communication with the Node.js security list, unfortunately, did not lead to a clear explanation of Node.js's attacker model. For example, our arbitrary memory write exploit was not considered a security bug, while our less severe out-of-bounds write was flagged as a security issue. In this paper, we conservatively assume a relatively weak attacker. We, however, remark that since our original reports, the Node.js team has established a security working group to, among other things, address some of concerns raised by this work. We are actively working within the scope of this group to refine Node.js's attacker model [73].

Furthermore, we believe that the buggy but benign model is not generally appropriate: the node package manager (NPM) and Node.js workflow make it easy to download and execute untrusted code [76, 83]. Members of the Node.js team and NPM recommend that developers "not execute any software . . . [that they] do not trust [75]." Binding bugs make it hard to follow this advice. Most binding bugs are reachable from core Node.js libraries, so developers cannot easily audit and therefore trust NPM packages. Even if a program does not `require` any module—a first indication that it may be trying to do something sensitive—that code can nevertheless leverage a binding bug to be extremely damaging. For example, a malicious NPM package could exploit one of the out-of-bounds vulnerabilities that we found in the core `buffer` library, which is always loaded, to read and write arbitrary parts of the Node.js process (e.g., users' secret keys). Even our crashing bugs may be useful: since Node.js is popular for implementing web-servers, attackers could use hard crashing binding errors to take down a server that otherwise handles crashes gracefully.

Finally, attackers may use binding bugs against language-level security mechanisms for Node.js, including [8, 27, 50]. The security systems defend against language-level attacks but assume JavaScript's memory- and type-safety. Bugs in the binding layer can violate these assumptions (as we show in §bugs), therefore violating the security guarantees of the language-level mechanisms. Neither the language-level systems nor more general JavaScript mechanisms (e.g., [26]) can safely expose subsets of the Node.js API without giving up on their guarantees.

4.4 PDFium

PDFium, Chrome's PDF rendering engine, parses and renders PDF documents. PDFs may contains JavaScript that customizes the document at runtime (e.g., by drawing new widgets or filling in a form); embedded JavaScript may even submit forms to remote servers. PDFium exposes an API for customizing PDFs as such using V8 bindings [96].

Chrome assumes that PDF documents may be malicious, and treats type- and memory-safety violations as security concerns. Chrome is especially concerned about binding layer attacks, since, for example, "a PDFium UAF will usually lead to remote code execution, particularly when it is triggered from [JavaScript] where the adversary has substantial control over what happens between the free and the subsequent re-use" [85].

Despite their attacker model, PDFium does not use any serious binding layer defenses: their bindings are hand-written using a crash-unsafe library that minimally wraps V8's APIs (Figure 2d). Chrome still runs the PDFium renderer in an isolated, sandboxed process, though, which limits the damage of binding errors. This is because Chrome's OS-sandbox restricts PDFium to communicating with other Chrome processes by using message passing. Unfortunately, PDF documents of different origins are rendered in the same process. As a result, binding layer memory-read exploits may, say, violate the same-origin policy by reading the contents of a different-origin PDF. The Chrome team is working on a more robust architecture that will isolate origins, making cross-origin attacks extremely difficult [85].

5 Preventing Errors By Construction

This section presents a new V8-based binding-layer API, one that makes it easier for developers to preserve JavaScript's crash-, type-, and memory-safety. We describe the API's design, implementation, and evaluation: it is backwards compatible and imposes little overhead and little porting burden. The API helps developers avoid bindings bugs by automatically type-checking JavaScript values and by forcing developers to more gracefully handle errors.

5.1 Safe API Design

A safe binding-layer API should:

1. Force developers to handle failures (e.g., exception-throwing upcalls) in a crash-safe way, by propagating errors back to JavaScript instead of hard crashing.
2. Disallow developers from using JavaScript values before checking their types.
3. Make the concurrent programming model explicit by making clear which C++ functions can trigger JavaScript upcalls that may change invariants (§3).

Our API achieves these goals by forcing functions that interact with JavaScript to use a special type, `JS<T>`, that encapsulates either a JavaScript value of type `T` (e.g., `v8::String`) or a JavaScript exception of type `v8::Error`. Our API satisfies the first goal because a `JS<T>` forces the developer to handle a `v8::Error` explicitly instead of triggering a hard crash; it satisfies the second goal by only providing functions that automatically type-check values before casting them; and it satisfies the third goal by forcing each potentially upcalling function to return a `JS<T>`, explicitly signalling that these functions may throw errors or have other side effects. Table 5 outlines the interface that our API exposes to C++ binding code. The API includes three kinds of functions that interact with JavaScript: `JS<T>` accessor methods, `Value`-marshaling functions, and `Object` methods.[4]

`JS<T>` accessor methods

```
onVal: JS<T0 x ... x Tn> -> ((T0, ..., Tn) -> JS<T>) -> JS<T>
onFail: JS<T> -> (Error -> JS<Error>|void) -> JS<Error>|void
```

`Value`-marshaling functions

```
marshal: Value v0 -> ... -> Value vN -> JS<T>
implicitCast: Value v0 -> ... -> Value vN -> JS<T>
toString: Value val -> JS<String>
```

`Object` methods

```
getProp: Object obj -> Value key -> JS<T>
getOwnPropDesc: Object obj -> String key -> JS<Value>

setProp: Object obj -> Value key -> Value newVal -> JS<bool>
defineOwnProp: Object obj -> Name key -> Value v -> JS<Value>

delProp: Object obj -> Value key -> JS<bool>

hasProp: Object obj -> Value key -> JS<bool>
hasOwnProp: Object obj -> Value key -> JS<bool>

getPropNames: Object obj -> JS<Array>
getOwnPropNames: Object obj -> JS<Array>
```

Table 5—The interface that our JavaScript engine API exposes. We use ML-style types to describe the function types: `T0 x ... x Tn` denotes a product type; `T0|T1` denotes a sum type; `T0 -> T1` denotes a function type. Like V8, all calls take an `Isolate*` as a first argument, and `Value`s and `Object`s are wrapped in `Local<>` handles; we have omitted these for brevity.

`JS<T>` accessor methods. `JS<T>` is the only type that describes JavaScript values in our API. We force programmers to properly handle JavaScript values by only allowing them to access `JS<T>`s using two safe accessor methods, `onVal` and `onFail` (see Table 5). The programmer interacts with `JS<T>`s by registering callbacks using these methods; the API invokes the callbacks after type checking and casting. `onFail` handles a `JS<T>` encapsulating a `v8::Error` by registering an error handler that the API calls when type checking fails or when JavaScript upcalls throw an error. The programmer must register an error handler: failing to do so causes a compile-time warning. This means that all functions that return a `JS<T>` are guaranteed to have associated error-handling code.

`onVal` registers a callback that accepts one or more values; the callback's type signature indicates which values the programmer expects. For convenience, the programmer can implement overloading by chaining multiple `onVal` calls. In that case, the API invokes the first callback with a matching type signature, or the error handler if no signature matches.

`Value`-marshaling functions. The API provides three func-

[4] Our API is inspired by Haskell's monads and JavaScript's promises. It differs from V8's usage of `Maybe<T>` types (§3.1) in two ways: (1) `JS<T>` keeps track of exceptions raised by JavaScript code and (2) the methods on `JS<T>` are crash- and type-safe.

tions for converting JavaScript Values to JS<T>s. Value is V8's base "unknown" type for all JavaScript values. Value-marshaling functions convert Values either to a specific type (e.g., v8::String) or to v8::Error in case of failure. To enforce type- and crash-safety, these functions always check the type of their Value val arguments before casting. Internally, this amounts to calling val->IsString(), say, to check that val is truly of type v8::String. If so, the marshaling function returns a JS<v8::String>; if not, it returns a v8::Error. Because the marshaling functions return a JS<T>, they require the programmer to explicitly handle errors; recall that, to access JS<T> values, the programmer must register callbacks with onVal and onFail.

Object methods. The API also provides methods for safely manipulating Objects. These methods are similar to V8's object methods—e.g., Get, which gets the value of a property—but they make side effects explicit. As an example, V8's Get may silently upcall into a user-defined JavaScript getter, leading to an unexpected exception. Like Get, our API's getProp method gets the value of a property—but it returns a JS<T> instead of a Value.[5] This return value makes it clear that an upcall is possible and forces the programmer to handle the potential side effects of that upcall by registering an onFail error handler.

Example: blobConstr. We re-implement blobConstr from §2 using our safe API:

```
1  void
2  blobConstr(const FunctionCallbackInfo<Value>& args)
3  {
4    // marshal arg[0] v8::Value from JavaScript
5    marshal(args.GetIsolate(), args[0])
6    // if marshaling to Array succeeded:
7    .onVal([&](Local<Array> blobParts) {
8      // Add each string part of the array to the blob
9      uint32_t n = blobParts -> Length();
10     for (uint32_t i = 0; i < n; i++) {
11       // Get the ith element from array argument
12       getProp(context, blobParts, i)
13       // Getting succeeded and returned a string:
14       .onVal([&](Local<String> part) {
15       // Add already-casted string part to the blob
16         blobImpl->AddV8StringPart(part);
17       })
18       // if above failed or element is not a string:
19       .onFail([&](Local<Error> err) {
20       // handle unexpected field
21       })
22     }
23   })
24   // if arg[0] is not an Array or onVal failed:
25   .onFail([&](Local<Error> err) {
26     // handle error
27   });
28 }
```

This function creates a new JavaScript Blob object out of an array of JavaScript strings. First, it uses the marshal function to safely convert the JavaScript value args[0] to a C++ value (line 4).[6] In order to use the result of the marshal call, the programmer must register callbacks via onVal and onFail. onVal and onFail each take one argument, a C++ lambda, that the API invokes after executing marshal. The formal argument to the onVal lambda (blobParts on line 6) specifies the type that the programmer expects marshal to return (Array). At runtime, the API checks the type of args[0] before casting it and executing the callback. If args[0] is an Array, the onVal callback executes; if not, the onFail one runs instead, allowing the programmer to pass an exception back to JavaScript code. In this way, casting and type checking are always coupled, eliminating a range of type-safety bugs.

blobConstr's top-level onVal callback uses the safe API to extract the String values at each index in the blobParts JavaScript array (line 11). Since blobConstr uses getProp to access these values, the API type checks and casts the values before invoking the correct callback, preserving JavaScript's crash- and type-safety. Since failing to register onVal and onFail callbacks results in a compile-time warning, the programmer is forced to account for both success and failure.

5.2 Implementation

We implement our API as a C++ library on top of the existing public V8 API. The API implementation comprises 1100 lines of C++. The library-based approach introduces little performance overhead (§5.3) and, more importantly, allows binding code developers to incrementally migrate their existing systems from V8 proper. Furthermore, this approach lets security-critical modules (e.g., the Node.js password-hashing library bcrypt [86]) use our safe API *without* waiting for the Node.js runtime or V8 engine to incorporate our changes.

The C++ class JS<T> is a templated class that implements onVal and onFail. Programmers use onVal and onFail to register success and failure callbacks. For example, any time a programmer wants to use a specific V8 type, they must marshal a Value to that type, registering their callbacks along the way.

Our API makes JS<T> values easier and safer to use by attaching the warn_unused_result compiler attribute [36] to the return value of the onVal method. This strongly encourages binding code developers to register onFail handlers: if onVal's result is unused (i.e., the call to onVal is not chained to a call to onFail), the compiler emits a warning.

Our API also uses restrictions on method arguments to enforce type-safety. Developers must declare the concrete expected type (e.g., v8::Array or v8::String) of every JavaScript argument in an onVal lambda. The API uses recent C++ features like **decltype** and std::declval to introspect the type of the lambda; it uses this information to generate specialized versions of each function accepting JS<T> arguments (e.g., marshal). These specialized versions perform runtime type checking and casting according to the types specified in the programmer's registered lambda. Other than onVal and onFail, our API does not provide any way to directly manipu-

[5] Or a Maybe<T> values that can be converted to a Value with hard-crashing conversion functions.

[6] marshal takes an Isolate* as a first argument; we discuss these further in §2, "Detailed overview of V8-based bindings."

late (e.g., check or cast) `v8::Values`.

5.3 Evaluation

We evaluate our API design and implementation by answering three questions:

1. Is the API backwards compatible?
2. Is the API's performance overhead acceptable?
3. How hard is it to port existing code to the API?

To answer these questions, we rewrote the Node.js binding-layer libraries for `buffer` and `http`. We chose these libraries both because they are representative of Node.js bindings and because they are widely used: `http` and `buffer` are essential for building web applications, Node.js's most prominent use case. In particular, we rewrote the buffer binding library `node_buffer.cc`, HTTP parsing binding library `node_http_parser.cc`, and several smaller support libraries' bindings (e.g., `uv` and `util`) in Node.js version 7.0.0. In the rest of this section we answer the three evaluation questions by comparing vanilla Node.js against SaferNode.js, our safer version of Node.js.

Compatibility

Porting binding-layer functions to our safe API should preserve their semantics—except their crashing semantics. We specifically *want* to eliminate hard crashes.[7] To measure SaferNode.js's backward compatibility, we used Node.js's existing compatibility-checking test suite. We ran Node.js's built-in test suite [66], which consists of 1,265 tests; SaferNode.js passed all of them. We also used the Canary in the Gold Mine (CITGM) tool [67] to run the test suites of 74 popular Node.js packages with SaferNode.js [72]; this is the same setup that Node.js developers use to find regression bugs in candidate releases [71]. Once again, we found no difference between vanilla Node.js and SaferNode.js in terms of compatibility.

Performance

We ran Node.js's performance benchmarks, two micro-benchmarks, and a macro-benchmark to measure SaferNode.js's overhead. In the worst case, SaferNode.js is 11% slower than Node.js when the latter uses V8 APIs in an unsafe (hard-crashing) way. On the other hand, when Node.js uses V8 APIs safely, SaferNode.js imposes no significant additional overhead. Finally, for real-world applications, SaferNode.js imposes less than 1% overhead. We describe these results and benchmarks in more detail below.

All measurements were conducted on a single machine with an Intel i7-6700K (4 GHz) with 64 GiB of RAM, running Ubuntu 16.10. We disabled dynamic frequency scaling and hyper-threading, and pinned each benchmark to a single core.

Node.js benchmarks. To measure the performance difference between Node.js and SaferNode.js, we used Node.js's benchmarking suite [69]. This suite is designed to find performance regressions. It works by benchmarking a set of Node.js modules on two different versions of the Node.js runtime and comparing their performance; our tests compare Node.js and SaferNode.js. We ran the `buffer` and `http` benchmark suites 50 and 10 times, respectively. (We chose these numbers in order to complete the benchmarking in reasonable time, roughly 10 hours.)

Each benchmark runs hundreds of tests on both Node.js and SaferNode.js, where each test invokes an operation a fixed number of times, depending on how long the operation takes to run. For example, the `buffer` benchmark for `indexOf`, a relatively slow operation, measures the time to execute 100,000 calls to `buff.indexOf`. It times these calls for different combinations of search strings, encodings, and buffer types, reporting total execution time in operations per second for each combination.

Figures 3a and 3b plot the speed of SaferNode.js normalized to Node.js for each test in the `buffer` and `http` benchmark suites, respectively. Each dot represents one test from the suite; results are sorted from slowest to fastest. The average overhead for `buffer` is 1%, with a maximum of 11%. `http` shows essentially no overhead on average; in the worst case, it is 5% slower. Below we use micro-benchmarks to show that SaferNode.js's overhead is the result of the API's added type checking; Node.js is faster because it does not perform these checks.

A few tests in both benchmark suites show modest speed-ups; these are spurious. To confirm this, we built Node.js and SaferNode.js using two different compilers, GCC 6.2 and Clang 3.8.1, and ran both test suites (Figure 3 shows results for GCC). We found that compiler-to-compiler performance variation on individual tests was on the order of 1–2%, comparable to the measured speed-ups. Moreover, tests that showed speed-ups for GCC often showed slow-downs for Clang, and vice-versa. A few tests show >2% average speed-ups. In these tests, however, individual runs showed widely varying results, with both speed-ups and slow-downs. We expect further benchmarking would show that SaferNode.js and Node.js have essentially the same performance on these tests.

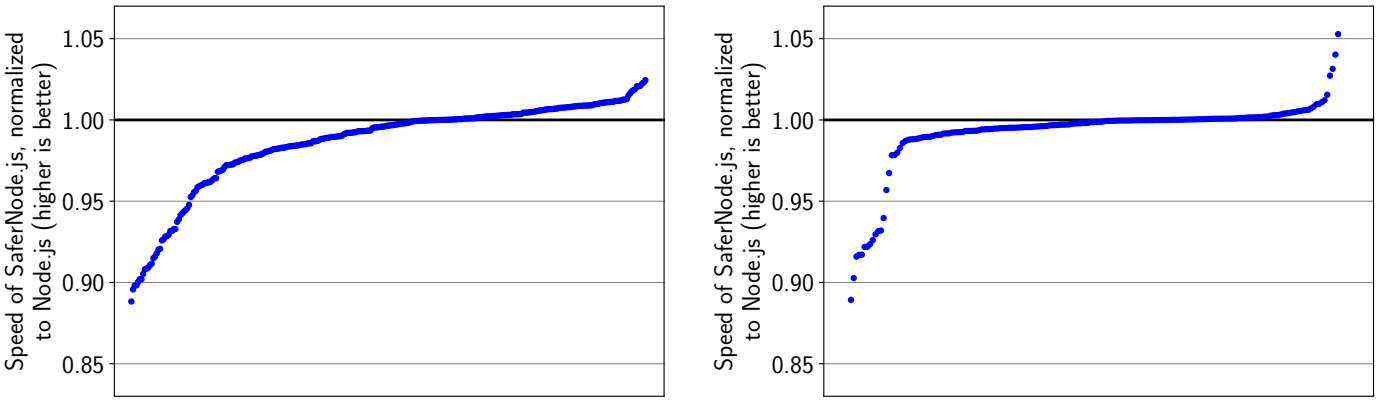
Micro-benchmarks. To test our hypothesis that SaferNode.js's overhead is due to extra checking in the safe API, we created three micro-benchmarks, each of which marshals a `Number` from JavaScript to C++ and back. We compare our safe API's version of this function with two normal V8 API versions, one that does type checking and one that does not. `echo_nocheck` uses the normal V8 API and performs no type checking:

```
void echo_nocheck(const FunctionCallbackInfo<Value>&
  ↪  args) {
  Local<Number> ret = args[0].As<Number>();
  args.GetReturnValue().Set(ret);
}
```

`echo_check` uses the same V8 API calls as above, but adds explicit type checking:

```
void echo_check(const FunctionCallbackInfo<Value>&
  ↪  args) {
  if (args[0]->IsNumber()) {
    Local<Number> ret = args[0].As<Number>();
    args.GetReturnValue().Set(ret);
  } else { // handle error
  }
}
```

[7] There is one exception to this: we do not rewrite code that hard-crashes for legitimate reasons, e.g., because it can no longer allocate memory.



(a) `buffer` benchmark suite: 304 tests, 50 runs each

(b) `http` benchmark suite: 182 tests, 10 runs each

Figure 3—Speed of SaferNode.js normalized to Node.js on a subset of the Node.js benchmarking suite [69] (§5.3). Each dot represents one benchmark from the suite; results are sorted slowest to fastest. SaferNode.js's speed ranges from ≈89% to ≈105% of Node.js's.

Finally, `echo_safeAPI` uses the safe API:

```
void echo_safeAPI(const FunctionCallbackInfo<Value>&
  ↪  args) {
  return safeV8::With(args->GetIsolate(), args[0])
  .onVal([&](Local<Number> ret) {
    args.GetReturnValue().Set(ret);
  })
  .onFail([&](Local<Error> exception) {
    // handle error
  });
}
```

We benchmark these functions by calling each one in a 10^7-iteration loop and measuring execution time. We call all three functions with an argument of the correct type; still, note that `echo_nocheck` would crash if given a non-numeric argument.

`echo_safeAPI` executes 12% more slowly than `echo_nocheck`, close to SaferNode.js's worst-case overhead on the benchmarks in the previous section. On the other hand, since both `echo_safeAPI` and `echo_check` check the type of their argument, they show no significant performance difference (less than 1%). As a result, we conclude that most of the overhead in the SaferNode.js benchmarks comes from the safe API's extra checking.

Macro-benchmark. Finally, to measure the performance of our safe API in a real-world setting, we measured the performance of the popular `express.js` web framework [95] by comparing the performance of Node.js and SaferNode.js using `express.js`'s speed benchmark [31]. This benchmark uses the `wrk` web server stress testing tool, running 8 concurrent threads and 50 open connections to measure the throughput of the web server. Node.js and SaferNode.js performance was within 1%, each serving about 15,000 requests/second.

Porting burden

Porting Node.js's `buffer` module to our safe API required adding about 1000 lines of code to `node_buffer.cc`, originally a 1300-line file. For `http`, we added about 150 lines of code to `node_http_parser.cc`, which was originally about 800 lines.

While porting, we realized that we were repeatedly rewriting similar code, so we built a prototype tool that assists the programmer by flagging binding functions and automatically rewriting common unsafe patterns. Specifically, the tool identifies top-level binding functions that are exposed to JavaScript (i.e., Node.js functions that accept one argument of type `const v8::FunctionCallbackInfo<v8::Value>&`).

For these functions, the tool rewrites (1) hard-crashing `CHECK` calls, (2) casts using `As<Type>` with no preceding `IsType` check, and (3) calls to `Get`, `Set`, and `ToString`. Our tool is conservative in that it only rewrites code that is easy to reason about. For example, it does not rewrite functions that include `goto`s. The tool comprises about 7,500 lines of Java.

Despite being a prototype, this tool was useful in porting Node.js to our API. In Node.js 7.0.0, we manually counted 378 functions with the required type signature; our tool flagged 371 of them. Of these, 201 did not need to be rewritten. Another 35 used `goto`s or similar patterns that the tool cannot handle. The tool rewrote the remaining 135 functions, but required manual intervention in two cases, about 30 lines of code total. As a sanity check, we ran CITGM (the regression suite from above) and the full Node.js benchmarks on the rewritten code. We found that it was fully functional and paid a performance overhead roughly commensurate with the results in Figure 3.

From the above, we surmise that porting to our API is reasonable, even in complicated code bases. We regard further automation of this porting effort as future work.

6 Related Work

We discuss our contributions with respect to related work on securing binding code in multi-language systems. Specifically, we consider the literature on *finding* bugs in binding code, *avoiding* bugs by construction, and *tolerating* bugs using isolation.

6.1 Finding Binding Bugs

The first line of work looks at finding errors that arise at the boundaries of multi-language systems, either via dynamic checking, property-independent translation to a common IR, or property-specific static checking.

Dynamic checking. Jinn [55] generates dynamic bug check-

ers for arbitrary languages from state machine descriptions of foreign function interface (FFI) rules. In doing so, Jinn finds bugs in both the Java Native Interface (JNI) and Python/C code. While running similar checkers in production is probably prohibitively expensive, this kind of system would complement existing browser debug-runtime checks.

Translation to common IR. Several groups have looked into translating multi-language programs into a common intermediate language, and then applying off-the-shelf analysis tools to their translation [9, 54, 58, 100]. For large code bases like Chrome, this approach is as feasible as the analysis approach is scalable; consequently, an alternative approach is to develop custom checks for particular classes of bugs.

Crash-safety bugs. Kondoh, Tan, and Li present different static analysis techniques for finding bugs caused by mishandled exceptions in Java JNI code [52, 56, 99]. Safer binding-layer APIs would address some of the issues these works tackle. For example, both our API the recent V8 API addresses similar memory management and exception-catching concerns by construction [30, 37]. Still, their approach can address a concern that neither our API nor existing JavaScript engine APIs handle: finding bugs in binding code where exceptions are raised by native code.

Type-safety bugs. Tan and Croft find type safety bugs that result when developers expose C pointers to Java as integers [99]. Their work also illustrates that static checkers can find type safety bugs in practice, as we suggest in §3. Exposed pointer errors, however, are unlikely in JavaScript binding code, since JavaScript engines provide hidden fields for C code to save raw pointers across contexts. Still, since these errors would be catastrophic, it may be worthwhile to implement Tan and Croft's technique for browser binding code. More broadly, Furr and Foster [34, 35] present a multi-language type inference systems for the OCaml and Java FFIs. Since JavaScript is dynamically typed, using type-inference approaches like these is difficult, but a similar analysis could help ensure that binding layer dynamic type checks are correct. Moreover, they would be applicable to runtime systems like Node.js where type checks are not generated but implemented manually.

Memory-safety bugs. Li and Tan present a static analysis tool that detects reference counting bugs in Python/C interface code [57]. Their results also show that static checkers can help secure cross-language code. A tool like this one would assist binding code in which the C++ side performs reference counting (e.g., Blink before Oilpan).

6.2 Avoiding Binding Bugs by Construction

Another line of work looks at avoiding binding bugs by construction, either with formal models of inter-language interaction to verify safety, or with new languages that restrict interactions to ensure safety.

Formal models. Several projects develop formal models for JavaScript, multi-language systems, and FFIs [53, 59, 61, 98, 104]. These works not only help developers understand multi-language interaction but also allow developers to formally reason about tools for multi-language systems (tools like static checkers or new APIs). We envision developing a similar formalization for JavaScript, perhaps based on [59, 80], or by extending and combining the formal models for C and JavaScript developed in the K-framework [29, 77].

Language design. Janet [7] and Jeannie [44] are language designs that allow users to combine Java and C code in a single file, therefore building multi-language systems more safely. SafeJNI [101] provides a safe Java/C interface by using CCured [65] to retrofit C code to a subset that abides by Java's memory and type safety. While these language designs are well-suited for isolated browser features, it is unclear how these approaches would apply to existing large JavaScript runtime system. By refactoring existing FFI code to use domain-specific languages such as our safe API, our approach provides a way to gradually increase security across different components. Specifically, it strongly encourages the programmer to put in suitable checks and handle all possible errors, one module at a time, thereby providing greater safety while remaining in the original host language (e.g., C++).

Safe linking. A recent line of work by Ahmed et al. [3, 78] aims to address the problem of (safely) composing multi-language programs by separately compiling the components into a *gradually* typed [88, 89] target language and then linking the results. The gradually typed target language would have support for more, less, or completely untyped sub-components, and would automatically insert run-time checks to ensure that typing invariants are preserved as values move across the different parts [65]. In the context of binding code, this approach has the benefit of shifting the burden of placing suitable checks from programmer to the compiler and could provide formal safety guarantees. On the other hand, gradual typing implementations still have non-trivial run-time overheads [97] and it remains to be seen whether the above approach can be made practical for complex systems like Node.js and Chrome.

6.3 Tolerating Binding Bugs

One last approach, orthogonal to finding and preventing binding bugs, is to design systems to tolerate such bugs by isolating components at the language or browser level.

Language-level isolation. Running different languages' runtimes in isolated environments addresses many security bugs in FFI code. Klinkoff et al. [49] present an isolation approach for the .NET framework. They run native, unmanaged code in a separate sandboxed process mediated according to the high-level .NET security policy. Robusta [87] takes a similar approach for Java, but, to improve performance, uses software fault isolation (SFI) [108].

Browser-level isolation. Redesigning the browser to run iframes in separate processes would address many of the vulnerabilities that lead to same-origin policy bypasses. Unfortunately, as illustrated by Chrome's ongoing efforts [22] and several research browsers (e.g., Gazelle [107], IBOS [102], and Quark [47]) this is not an easy task. Redesigns often break compatibility and have huge performance costs. Moreover, applying such techniques beyond the browser to runtime systems such as Node.js and PDFium is not easily achievable—in these systems

we do not have the security policies that allow browsers to more easily decide where to draw isolation boundaries.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Nikhil Swamy, for many insightful comments and for pointing out a bug in an early version of this paper. Úlfar Erlingsson and Bryan Parno for their accomodations. Thomas Sepez for helping us understand the PDFium attacker model and confirming and fixing some of our bugs. Bryan Eglish, Colin Ihrig, Devon Rifkin, Sam Roberts, Rod Vagg, and Brian White for useful discussions of Node.js's attacker model and for incorporating our feedback on how to improve the runtime's safety and security. Colin Ihrig and Timothy Gu for promptly fixing many of our Node.js bugs. Adrienne Porter Felt, Joel Weinberger, Lei Zhang, Nasko Oskov, and Devlin Cronin for helping to explain the security model for Chrome extensions. Hovav Shacham, David Kohlbrenner, and Joe Politz for fruitful discussions. Sergio Benitez and Andres Nötzli for help, comments, and formatting magic. Mary Jane Swenson for making everything easier. This work was supported by NSF Grant CNS-1514435 and an NSF Fellowship.

References

[1] F. Aboukhadijeh. Buffer(number) is unsafe. https://github.com/nodejs/node/issues/4660.

[2] B. Adida, A. Barth, and C. Jackson. Rootkits for JavaScript environments. In *WOOT*, Aug. 2009.

[3] A. Ahmed. Verified compilers for a multi-language world. In *Summit on Advances in Programming Languages, SNAPL 2015*, May 2015.

[4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*, Feb. 2010.

[5] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and preventing bugs in JavaScript bindings: Extended version. https://bindings.programming.systems.

[6] F. Brown, A. Nötzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *ASPLOS*, Apr. 2016.

[7] M. Bubak, D. Kurzyniec, and P. Luszczek. Creating Java to native code interfaces with Janet extension. In *Worldwide SGI UsersâĂŹ Conference*, Oct. 2000.

[8] E. Budianto, R. Chow, J. Ding, and M. McCool. Language-based hypervisors. In *CANS*, Nov. 2016.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, Dec. 2008.

[10] caja. Caja. https://developers.google.com/caja/.

[11] P. Carbonnelle. PopularitY of Programming Language. http://pypl.github.io/PYPL.html.

[12] chakraCore. Microsoft chakracore. https://github.com/Microsoft/ChakraCore.

[13] checkerChromeVersion. Chromium version 56.0.2915.0. https://chromium.googlesource.com/chromium/src.git/+/56.0.2915.0.

[14] Chromium. Security faq. https://www.chromium.org/Home/chromium-security/security-faq.

[15] Chromium. Side by side diff for issue 196343011. https://codereview.chromium.org/196343011/diff/20001/Source/bindings/templates/attributes.cpp, 2016.

[16] Chromium. Issue 395411 and CVE-2014-3199. https://bugs.chromium.org/p/chromium/issues/detail?id=395411, 2016.

[17] Chromium. Issue 456192 and CVE-2015-1217. https://bugs.chromium.org/p/chromium/issues/detail?id=456192, 2016.

[18] Chromium. Issue 449610 and CVE-2015-1230. https://bugs.chromium.org/p/chromium/issues/detail?id=449610, 2016.

[19] Chromium. Issue 497632 and CVE-2016-1612. https://bugs.chromium.org/p/chromium/issues/detail?id=497632, 2016.

[20] Chromium. Issue 603748. https://bugs.chromium.org/p/chromium/issues/detail?id=603748, 2016.

[21] Chromium. Issue 603725. https://bugs.chromium.org/p/chromium/issues/detail?id=603725, 2016.

[22] Chromium. Out-of-process iframes. https://www.chromium.org/developers/design-documents/oop-iframes, 2016.

[23] Chromium. Issue 671488: Hard crash in webkit customelement bindings. https://bugs.chromium.org/p/chromium/issues/detail?id=671488, 2016.

[24] Chromium. Issue 679643: Security: Use after free in pdfium's annot::name. https://bugs.chromium.org/p/chromium/issues/detail?id=679643, 2017.

[25] Chromium. Issue 679642: Security: Use after free in pdfium's field::page. https://bugs.chromium.org/p/chromium/issues/detail?id=679642, 2017.

[26] D. Crockford. ADsafe: Making JavaScript safe for advertising. http://www.adsafe.org, 2008.

[27] W. De Groef, F. Massacci, and F. Piessens. Nodesentry: least-privilege library integration for server-side javascript. In *ACSAC*, Dec. 2014.

[28] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, et al. The matter of Heartbleed. In *IMC*, Nov. 2014.

[29] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, Jan. 2012.

[30] B. English. <=v4: process.hrtime() segfaults on arrays with error-throwing accessors. https://github.com/nodejs/node/issues/7902.

[31] Express. Benchmarks run. https://github.com/expressjs/express/blob/master/benchmarks/run.

[32] A. P. Felt, J. Weinberger, L. Zhang, N. Oskov, and D. Cronin. Private communication, March 2017.

[33] E. F. Foundation. HTTPS everywhere. https://www.

eff.org/https-everywhere, 2017.

[34] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *PLDI*, June 2005.

[35] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *ESOP*, Mar. 2006.

[36] gcc. Declaring attributes of functions. https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html.

[37] M. Hablich. API changes upcoming to make writing exception safe code more easy. https://groups.google.com/forum/#!topic/v8-users/gQVpp1HmbqM.

[38] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical type confusion detection. In *ACM CCS*, Oct. 2016.

[39] K. Hara. Oilpan: GC for Blink. https://docs.google.com/presentation/d/1YtfurcyKFS0hxPOnC3U6JJroM8aRP49Yf0QWznZ9jrk, 2016.

[40] J. Harrell. Node.js at PayPal. https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/, November 22 2013.

[41] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *ACM SAC*, Apr. 2014.

[42] S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *POST*, Apr. 2015.

[43] M. Hicks. What is memory safety? http://www.pl-enthusiast.net/2014/07/21/memory-safety/, 2014.

[44] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *ACM SIGPLAN Notices*, volume 42:10, 2007.

[45] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *IEEE S&P*, May 2013.

[46] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *USENIX Security*, Aug. 2015.

[47] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, Aug. 2012.

[48] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security*, Aug. 2014.

[49] P. Klinkoff, E. Kirda, C. Kruegel, and G. Vigna. Extending .NET security to unmanaged code. *Journal of Information Security*, 6(6):417–428, 2007.

[50] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE EuroS&P*, Apr. 2017.

[51] J. Koetsier. Ad Block Plus is now...an ad network. https://www.forbes.com/sites/johnkoetsier/2016/09/13/adblock-plus-is-now-an-ad-network/#697cbff41bca.

[52] G. Kondoh and T. Onodera. Finding bugs in Java native interface programs. In *Symposium on Software Testing and Analysis*, Apr. 2008.

[53] A. Larmuseau and D. Clarke. Formalizing a secure foreign function interface. In *SEFM*, Sept. 2015.

[54] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, Mar. 2004.

[55] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *ACM SIGPLAN Notices*, volume 45:6, 2016.

[56] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *ACM CCS*, Nov. 2009.

[57] S. Li and G. Tan. Finding reference-counting errors in Python/C programs with affine analysis. In *ECOOP*, July 2014.

[58] P. Linos, W. Lucas, S. Myers, and E. Maier. A metrics tool for multi-language software. In *SEA*, Nov. 2007.

[59] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for javascript. In *APLAS*, Dec. 2008.

[60] P. Marchenko, Ú. Erlingsson, and B. Karp. Keeping sensitive data in browsers safe with ScriptPolice. Technical report, UCL, 2013.

[61] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *TOPLAS*, 31(3):1–44, 2009.

[62] C. McCormack. Web IDL. *World Wide Web Consortium*, 2012.

[63] L. A. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE S&P*, May 2010.

[64] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *ACM SIGPLAN Notices*, volume 44:6, 2009.

[65] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37:1, 2002.

[66] nodeBenchmarks. Node.js core benchmarks. https://github.com/nodejs/node/tree/master/benchmark.

[67] Node.js. Canary in the Gold Mine. https://developers.google.com/v8/embed, .

[68] Node.js. Node.js helps NASA keep astronauts safe and data accessible. https://nodejs.org/static/documents/casestudies/Node_CaseStudy_Nasa_FNL.pdf, .

[69] Node.js. Node.js benchmarking branch. https://github.com/nodejs/node/tree/master/benchmark, .

[70] Node.js. Node.js on ChakraCore. https://github.com/nodejs/node-chakracore.

[71] Node.js. Node.js CITGM lookup list. https://github.com/nodejs/citgm/blob/master/lib/lookup.json, .

[72] Node.js. Canary in the Gold Mine – node.js 7.0.0. https://github.com/nodejs/citgm/blob/2434cceb09f2e7966cfdf70b523e0bea57be9598/lib/lookup.json, .

[73] Node.js security working group. What is/is not a "vulnerability"/"security issue"? https://github.com/nodejs/security-wg/issues/18.

[74] B. Noordhuis. src: remove unneeded environment error methods. https://github.com/nodejs/node/commit/0e6c3360317ea7c5c7cc242dfb5c61c359493f34.

[75] NPM. Package install scripts vulnerability. https://blog.npmjs.org/post/141702881055/package-install-scripts-vulnerability.

[76] T. npm Blog. kik, left-pad, and npm. https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm.

[77] D. Park, A. Stefanescu, and G. Rosu. KJS: a complete formal semantics of JavaScript. In *PLDI*, June 2015.

[78] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP*, Apr. 2014.

[79] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[80] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in javascript. In *ACM SIGPLAN Notices*, volume 48:2, 2013.

[81] A. Ranganathan, J. Sicking, and M. Kruisselbrink. File API. *World Wide Web Consortium*, 2015.

[82] R. Rogowski, M. Morton, F. Li, K. Z. Snow, F. Monrose, and M. Polychronakis. Revisiting browser security in the modern era: New data-only attacks and defenses. In *IEEE EuroS&P*, Apr. 2017.

[83] S. Saccone. npm hydra worm disclosure. https://www.kb.cert.org/CERT_WEB/services/vul-notes.nsf/6eacfaeab94596f5852569290066a50b/018dbb99def6980185257f820013f175/$FILE/npmwormdisclosure.pdf.

[84] G. A. Security. Severity guidelines for security issues. https://sites.google.com/a/chromium.org/dev/developers/severity-guidelines.

[85] T. Sepez. Private communication, March 2017.

[86] R. Shtylman. bcrypt. https://www.npmjs.com/package/bcrypt.

[87] J. Siefers, G. Tan, and G. Morrisett. Robusta: Taming the native beast of the JVM. In *ACM CCS*, Oct. 2010.

[88] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Sept. 2006.

[89] J. G. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, July 2007.

[90] D. Stefan. spawnSync's SyncProcessRunner::CopyJsStringArray segfaults with bad getter. https://github.com/nodejs/node/issues/9821, .

[91] D. Stefan. npm shrinkwrap allows remote code execution. https://hackernoon.com/npm-shrinkwrap-allows-remote-code-execution-63e6e0a566a7#.e7an55fo2, .

[92] D. Stefan. Buffer.fill has an out of bounds (arbitrary) memory write. https://github.com/nodejs/node/issues/9149, 2016.

[93] D. Stefan. Buffer::Length hard crashes. https://github.com/nodejs/node/issues/11954, 2017.

[94] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazieres. Protecting users by confining JavaScript with COWL. In *OSDI*, Oct. 2014.

[95] StrongLoop/IBM. Express—Node.js web application framework. https://expressjs.com.

[96] A. Systems. JavaScript for Acrobat api reference. http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf.

[97] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *POPL*, Jan. 2016.

[98] G. Tan. JNI Light: An operational model for the core JNI. In *APLAS*, Nov. 2010.

[99] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *USENIX Security*, July 2008.

[100] G. Tan and G. Morrisett. ILEA: Inter-language analysis across Java and C. In *ACM SIGPLAN Notices*, volume 42:10, 2007.

[101] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java native interface. In *Secure Software Engineering*, volume 97, 2006.

[102] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In *OSDI*, Oct. 2010.

[103] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, et al. Ad injection at scale: Assessing deceptive advertisement modifications. In *IEEE S&P*, May 2015.

[104] V. Trifonov and Z. Shao. Safe and principled language interoperation. In *ESOP*, Mar. 1999.

[105] V8. Getting started with embedding. https://github.com/v8/v8/wiki/Getting%20Started%20with%20Embedding.

[106] walmart. Walmart. https://www.walmart.com, 2016.

[107] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle Web Browser. In *USENIX Security*, Aug. 2009.

[108] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, May 2009.

[109] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys*. ACM, Apr. 2009.

[110] C. Zapponi. Programming languages and GitHub. http://githut.info/.