ABSTRACT
Modern applications, written in high-level programming languages, enjoy the security benefits of memory and type safety. Unfortunately, even a single memory-unsafe library can wreak havoc on the rest of an otherwise safe application, nullifying all the security guarantees offered by the high-level language and its managed runtime. We perform a study across the Node.js ecosystem to understand the use patterns of binary add-ons. Taking the identified trends into account, we propose a new hybrid permission model aimed at protecting both a binary add-on and its language-specific wrapper. The permission model is applied all around a native add-on and is enforced through a hybrid language-binary scheme that interposes on accesses to sensitive resources from all parts of the native library. We infer the add-on’s permission set automatically on its binary side and is enforced through a set of novel program analyses. Applied to a wide variety of native add-ons, we show that our framework, BinWrap, reduces access to sensitive resources, defends against real-world exploits, and imposes an overhead that ranges between 0.71%–10.4%.

CCS CONCEPTS
• Security and privacy → Web application security; Software security engineering; • Information systems → Web interfaces.

KEYWORDS
Node.js, native add-ons, Intel MPK/PKU, seccomp-BPF

1 INTRODUCTION
Modern software development relies heavily on third-party libraries. Third-party libraries refer to software components, or modules, which are usually put together by developers that are external to a particular software system’s development team. Third-party libraries can be integrated into a software system to add functionality, improve performance, or simplify the development process. The vast majority of the libraries imported in a Node.js application are implemented in JS, and thus enjoy the memory safety guarantees provided by a high-level programming language, enforced by its managed, runtime environment—at times, augmented with language-based protection techniques [1, 17, 40, 67, 70–72].

Often, however, Node.js applications import libraries that are written in low-level languages or provided in binary-only form. These libraries, termed native add-ons, implement either functionality not available yet in the pure-JS ecosystem or components that need to be in low-level languages for performance and compatibility reasons. Native add-ons interact with the rest of the program through a thin JS layer wrapping the library enough to expose Node.js-specific naming and calling conventions.

Unfortunately, native add-ons are particularly dangerous to the rest of a JS (or, in general, a memory-safe) application—for example, over 20 CVEs are reported for a single string-interpolation add-on [65]. The complete lack of memory safety means that even a single line of memory-unsafe code may completely compromise an application’s safety and security. Native add-ons can additionally bypass the security guarantees provided by the aforementioned language-based hardening and protection techniques [41]. The exploitation risks of native add-ons compound, as these components are more likely to be targeted by malicious actors—exactly because of their vastly higher insecurity and potential impact.

In this work, we develop BinWrap: a hybrid language-binary framework for protecting against native add-ons present in modern Node.js applications. We develop a fine-grained read-write permission model, applied at the boundaries of native add-ons, offering a unified view and isolation of privilege, cutting across the barrier between the language wrapper and the corresponding binary code. Two components enforce these permissions across the language-binary barrier, during the execution of the program, protecting both sides of a native add-on: (1) language-level interposition protects against unauthorized use of the language-level bindings (BinWrap_L), and (2) binary-level indirection wraps the entire library and checks permissions to outside interfaces (BinWrap_B).
To aid developers, a pair of program analysis components infer permissions automatically, over both the binary and JS sides of a native add-on. Combined, the permission model and associated analyses aim at reducing the risk of native add-ons, while maintaining practical performance and automation characteristics to enable adoption. The evaluation of our framework demonstrates that BinWrap can effectively protect real-world applications, when vulnerabilities are exploited within the loaded native modules. BinWrap is an efficient solution, imposing a performance overhead that ranges between 0.71%–10.40%. It is also scalable and practical, since our design choices were directly influenced by the NPM ecosystem norms.

2 BACKGROUND

2.1 Node.js, V8, and NAN

Node.js is a runtime environment for executing JS code, which primarily targets server-side applications [52]. Internally, Node.js leverages the V8 JS engine [28]. V8 parses JS code and converts it to an AST (abstract syntax tree), which can later be "lowered" to V8-specific bytecode that, in turn, can be interpreted with the Ignition interpreter [25]; moreover, JS code (in AST or bytecode form) can be compiled to machine code using the (optimizing) TurboFan or (non-optimizing) SparkPlug compiler [26].

Node.js is built around V8 (both are implemented in C/C++) and provides a rich set of APIs (i.e., the Node-API [47], but also, among others, the NAN [46] and V8 [27] APIs) to JS applications. Most importantly, Node.js allows JS programs to load native add-ons (i.e., modules written in C, C++, or ASM). Typically, JS applications leverage add-ons to: (1) perform compute-intensive tasks using highly-optimized C, C++, or even handwritten-ASM code [35]; (2) have access to other (dynamically-loaded) system libraries [55]; (3) interact freely with the underlying OS kernel via the system call interface, and utilize system services for which JS abstractions are not available [48]; or even (4) perform computations on specialized hardware (e.g., GPUs [33]). (Interested readers are referred to Appendix A for more information about Node.js and V8.)

2.2 Restricting Memory Accesses

Intel MPK/PKU [32] offers user-space processes the ability to change access permissions on groups of memory pages without invoking the underlying OS. Each page group is associated with a unique key. An application can have up to 16-page groups. The access rights for each page group are mapped in a thread-local and user-accessible register, called %pkru. Since %pkru is thread-specific, MPK supports different per-thread view(s) of the process’s memory. For example, different application threads can have different access rights configured for each key in their %pkru register.

Data accesses on memory pages, associated with protection keys, are checked both against the (RW) access rights defined in the %pkru register, as well as the permissions in page tables. In contrast, instruction fetching is checked only against page table permissions.

If a memory page is executable (in the respective page tables), but configured with ‘no access’ in %pkru, the memory page is treated as execute-only [54]. This occurs since any data access will result in a mismatch between the rights defined in the page table and the %pkru register. Linux supports execute-only memory pages by leveraging MPK/PKU. A call to mprotect with only PROT_EXEC specified as permissions will result in the allocation of a new protection key, which will be associated with the corresponding memory pages; next, the %pkru register will be set to DISALLOWABLE ACCESS for the newly-allocated protection key, while the page table rights will be set to executable and readable (R-X).

For associating a memory page (or range of memory pages) with a protection key, the Linux kernel implements the pkey.mprotect system call. The access rights in the %pkru register can be modified with the wpkrux86 instruction. Since %pkru register is user-accessible, modifying the access rights does not impose significant latency, and it is much faster than invoking memory management system calls (e.g., mprotect). (Refer to Appendix B for more information about protection keys and Intel MPK/PKU.)

2.3 Restricting System Calls

seccomp is a kernel mechanism for restricting the system calls (syscalls) that an application can execute. Since v3.5, the Linux kernel supports SECure COMPuting with filters (seccomp-BPF), thereby allowing/denying syscalls based on system call numbers and arguments (pointer dereferences are not supported). The applied filter can only be supplemented by a more restrictive filter and cannot be removed. The filters applied are per-thread, and thus syscall restrictions can be applied on a specific native thread only.

3 BINWRAP OVERVIEW

We use an image processing library (§3.1) to illustrate the issues of a Node.js “module” containing vulnerabilities both on the JS and the native (i.e., add-on) part(s) of its code, and then outline how BinWrap addresses the respective problems (§3.2).

3.1 png-img: A Node.js Graphics Library

Consider a Node.js application that creates PNG image objects from a supplied input buffer. More specifically, the developer provides a buffer to a Node.js library (png-img), implemented (partially) in native, add-on code, which contains raw image data. In addition, assume that the size of the input data is not checked to ensure they fit into the buffer in question, and hence a memory error can occur.

Such errors are a common attack vector when code written in memory- and type-unsafe languages, like C, C++, Objective-C, and assembly (ASM) [69], is involved, and they typically manifest by exploiting missing sanitization logic, pointer arithmetic bugs, invalid type casts, etc.—i.e., bugs in code that trigger spatial [44] or temporal [45] memory safety violations, enabling attackers to corrupt or leak contents inside the (virtual) address space of victim programs. The code snippet below corresponds to the relevant application fragment of our example.

```javascript
1 const fs = require('fs');
2 const PngImg = require('png-img');
3 let buf = fs.readFileSync('./img.png');
4 let img = new PngImg(buf);
```

First, the developer loads the library png-img (ln. 2) to add image processing capabilities in their application. Consequently, they load raw (image) data into buf, using the fs module (ln. 4). Finally, the buf object is passed to the PngImg constructor for generating img, i.e., the PNG image object.
In the snippet above, we zoom into the step(s) performed by `png-img`, after the developer imports the library to the application. `png-img` uses NAN [46] to link a native function (written in C, C++, etc.) with the `PngImgImpl` object (ln. 1–2). Every time the `png-img` constructor is invoked, the buffer object, which contains the raw (image) data, is passed to the native function (ln. 4–7).

Since the add-on is written in a memory- and type-unsafe language, it may contain bugs (e.g., a buffer overflow, ln. 6) that trigger memory errors [69]. More importantly, given that the raw image data are of unknown provenance, attackers may provide special crafted inputs that exploit the underlying memory errors, potentially resulting in arbitrary memory read (disclosure, leak) and arbitrary memory write (“write-what-where”) primitives [56].

In real-world settings, attackers primarily aim for tampering-with control data (e.g., return addresses, function pointers, dynamic dispatch tables) [37], as these facilitate hijacking the control flow of the program and performing arbitrary code execution [50]—typically via means of code reuse [10]: i.e., the attacker executes benign program code, in an “out-of-context” manner, by tampering-with control data; a wide range of code-reuse attack techniques has been developed thus far [6, 12], enabling access control and policy enforcement bypasses, privilege elevation, and sensitive data leakage [66]. Considering these facts, we found a relevant vulnerability of `png-img` documented in National Vulnerability Database [42].

The vulnerability is that height and rowBytes are 32-bit integers, and thus can be overflowed. An attacker could trigger this overflow in order to cause an inadequately-sized memory allocation (ln. 3). Subsequently, image data will overwrite memory near `rowPtrs_` [] (ln. 6). As we discuss in Section 8, this arbitrary memory write can be used by an attacker to take over the control of the application.

### 3.2 Node.js Module Confinement with BinWrap

To harden Node.js applications against vulnerabilities in `png-img`, we apply BinWrap both at the JS and the native part(s) of the library. More specifically, BinWrap comes bundled with a set of tools for performing static and dynamic policy enforcement, at the level of native, binary code (BinWrap_P), as well as JS code (BinWrap_J). (The latter typically wraps the add-on code and provides a high-level API for interfacing with Node.js-based application code.) BinWrap_P consists of a set of memory isolation and code confinement techniques, tailored to the runtime environment of Node.js, which aim at restricting the execution, and side effects, of unsafe add-on code in part(s) of the virtual address space (VAS).

Specifically, the execution of native add-on code is dispatched to a special (Node.js) execution thread, which has a restricted memory view of the virtual address space, by leveraging Intel’s MPK/PKU technology [32]. The benefits of this intra-VAS isolation are twofold: first, memory errors in `png-img`’s native code cannot be used to tamper-with data of the Node.js runtime—i.e., BinWrap_P provides data confidentiality/integrity against (arbitrary) memory disclosure/corruption vulnerabilities in unsafe library code; and, second, any potential reuse of code is limited to re-using functionality that exists in `png-img` only—i.e., BinWrap_P prevents code-reuse-based, control-flow hijacking attacks (which originate from the native library) from re-using code that exists in Node.js or in other libraries in the same VAS.

In addition to the above, a seccomp-BPF filter is installed in the special execution thread to further restrict the interactions of the latter with the OS, in case the control-flow of the native (library) code is tampered-with (despite being sandboxed). BinWrap_P automatically extracts the set of syscalls required by the native code, and complements that set with syscalls that may result from the invocation of Node.js functionality via NAN (i.e., the native code invokes Node.js code via the NAN API), as well as the invocation of V8 or libc (and other system libraries) APIs.

At runtime, if JS code needs to invoke a native function that belongs to `png-img`, via NAN, BinWrap_P dispatches the execution of that function to the special thread, which executes the unsafe code under a restricted memory view that is HW-enforced by Intel MPK/PKU. The unsafe code may in turn invoke APIs that belong to Node.js, V8, libc, or any other system library. In such cases, the control flows to the target (API) entry points (and back) via special gateways, which alter the memory view(s) of the code accordingly.

The required analyses for all the above (i.e., gateway generation, syscall extraction) are performed statically during the installation of a Node.js library/module that contains native code, and need only to be repeated if the respective code is updated. BinWrap_P consists of both a static and a dynamic enforcement part. We use a state-of-the-art static analyzer, MIR [72], which hits a sweet spot between soundness and completeness. By running the MIR static analyzer on the JS wrapper code of `png-img` (i.e., `index.js`), at load-time, we get the following JSON report that summarizes the developer-intended access permissions regarding the various JS objects involved.

Armed with the above, BinWrap_P traces object accesses at runtime, and blocks any attempt to access an object that is not compatible with the extracted policy, thereby policing the interaction of `png-img` with the Node.js application that uses it.

### 4 Threat Model

We consider the exploitation of memory errors in benign native modules. An attacker can leverage memory-safety-based vulnerabilities in order to develop arbitrary memory read and write primitives [56].
The exploitation of these vulnerabilities can be used to access sensitive data or perform code-reuse attacks [66]. We do not consider malicious native modules that will actively try to evade our hardening mechanisms. Moreover, we also assume that the high-level language part of the library is confined through existing language-based mechanisms [72]. The Node.js runtime, as well as any system libraries, are considered to be trusted. Finally, we assume that side-channel attacks [38] and hardware faults [11, 43] are out of scope.

In order for BinWrap to protect the Node.js runtime environment from the above, the following OS and hardware features are required. The OS must allow seccomp-BPF [34] in order to enable syscall filtering. We also consider that the \textit{W’X} [51] policy is enforced, and that the native module does not include self-modifying code. Moreover, Node.js, system libraries, and the native module leverage Address Space Layout Randomization (ASLR) [22]. Our framework does not interfere with any other possibly deployed security mechanisms, like stack-smashing protection [15], RELRO [58], FORTIFY_SOURCE [57], etc. Rather, these mechanisms can further enhance the protection offered by BinWrap. The hardware must include MPK/PKU [32] or a mechanism that offers equivalent capabilities. While our required hardware feature(s) cannot be considered “standard”, MPK/PKU is available in modern Intel server CPUs. Moreover, MPK functionality can be emulated through memory tagging, which is available in ARM v8.5-A [4].

Our techniques aim to address three challenges: (i) prevent the native module thread from accessing memory outside of the module’s loaded address range and its heap-allocated memory; (ii) prevent the native module thread from executing code-reuse-based gadgets outside of the native module’s code area(s); and (iii) prevent the native thread from misusing syscalls. We consider the Node.js runtime environment, and our customized native module layer (NAN), as the trusted part of the application, and the native module(s) as the untrusted part(s). Our techniques ensure that any attack that targets the native module will be confined within its bounds, and will not affect the whole application.

5 LANGUAGE-SAFETY TRENDS IN NODE.JS

The NPM ecosystem contains more than 1.5 million packages, downloaded more than 151 billion times during the last month. This quantity and popularity of packages makes NPM ideal for abuse. As we focus on native libraries found in the entire NPM ecosystem, we begin with investigating the following research questions (RQs):

- **RQ1** What is the ratio of packages that use one or more native modules on NPM? (§5.1)
- **RQ2** What is the ratio of native modules that are imported from libraries on NPM? (§5.2)
- **RQ3** What are the most popular native modules used by NPM packages? (§5.3)

We downloaded the entire NPM registry on a local host to perform our analysis. We also cloned the NPM database to perform the essential queries and data extraction—our NPM registry uses CouchDB [2] for storing information (on JSON format) about Node.js packages. We used the replication mechanism of CouchDB to download the entire registry locally, and made the necessary configuration(s) to access it. Finally, we used a proxy registry, Verdaccio [73], to access our local repository of packages.

At the time of the registry replication, NPM contained 1,508,366 libraries. We used this number as the starting point of our study and analyzed all the packages that had at least one dependency with a native module. We chose the most popular native modules, used by packages, which leverage NAN or Node-API (NAPI).

5.1 Number of Native Modules (RQ1)

We found that of the 1,508,366 libraries, 63,381 of them had at least one dependency related to NAN or NAPI. A library is dependent on a native module when it includes the NAN or NAPI module directly, or indirectly via its dependency list. (In the direct case, the package imports any of the two native modules itself; in the indirect case, some of its dependencies import NAN or NAPI instead.) From the respective packages, 45,708 packages depended on NAN, 23,239 packages depended on NAPI, and 5,548 packages depended on both. From those 63,381 libraries, more than 76.7% had a single native dependency. For the remaining 23.3%, we found that 13.7% of libraries had two native dependencies, 4.2% of libraries had three native dependencies, and 2.3% of libraries had four native dependencies. The last 3.1% of libraries had five to ten native dependencies. Finally, only 99 libraries used ten native dependencies. We conclude that there is only one native package per module in most of NPM packages.

5.2 Ratio of Native Modules (RQ2)

Third-party libraries, usually include a combination of native library dependencies and ordinary JS dependencies. The packages that use either NAN or NAPI as native modules had an average of 11 total dependencies. Among those dependencies, 0.95% on average were NAN dependencies, and 0.65% were NAPI dependencies. The average ratio of NAN dependencies against the total set of dependencies was 24.22%. The average ratio of NAPI dependencies against the total set of dependencies was 11.27%.

5.3 Popularity of Native Modules (RQ3)

Packages that use native modules comprise 4.2% of the total NPM ecosystem. This percentage is a significant part of the NPM ecosystem and results in multiple daily downloads. In this section of the paper we will answer how popular this 4.2% of native packages is by measuring their total dependents. For a package to depend on another, it must be included in its dependency list.
By studying the number of dependents, we can assess the impact of a vulnerability on a native package. For the NAN-based native modules, 8,148 packages had dependents. The average number of dependents per package was 7.2. The minimum number of dependents on a package was one, and the maximum was 8,457. A total number of 58,778 packages were dependent on the NAN native module. When we conducted our analysis, the package with the most dependencies was node-sass, with over 8k dependents. The least popular package (in the top-50 list) was ledgerhq with ≈90 dependents. Meanwhile, only the top 11 packages had more than 500 dependents. Figure 1 presents the number of dependents of the top-50 (most popular) packages. We surmise that our evaluation set is an ample representation of third-party libraries in NPM.

For the NAPI native modules, 5,762 packages had dependents. The average number of dependents per package was 6.6. The minimum number of dependents on a package was one, and the maximum was 2,322. A total number of 38,383 packages were dependent on the NAPI native module. Hence, 97,161 packages are dependent on native modules, and any security incident will therefore affect a significant part of the NPM ecosystem.

6 DESIGN

The key idea behind BinWrap is to separate the runtime execution of the untrusted component from the rest of the application. Runtime separation is achieved using different execution threads for the two domains of trust, while isolating the thread responsible for executing the untrusted component involves limiting its memory visibility and syscall execution capabilities. More specifically, BinWrap limits the memory visibility of the untrusted component by creating a dedicated memory view for the untrusted thread. It also limits access to the syscall API available to the untrusted component, by wrapping and filtering syscalls.

Figure 2 presents BinWrap’s approach to hardening Node.js. First, BinWrap compiles the untrusted component and then statically analyzes the resulting ELF binary—i.e., a shared static object (DSO). We prefer analyzing binary over source code, as we can discover more easily the complete set of external symbols required (e.g., calling printf will also execute other libc functions, like write). This analysis aims at extracting (1) the full set of syscalls necessary for the execution of the native component, and (2) the set of Node.js-internal API calls used by the native component—e.g., v8::External::New(v8::Isolate*, void*), v8::Object::SetInternalField(int, v8::Local<v8::Value>), etc. Next, BinWrap creates a custom instance of a Node.js API layering library, which is loaded during the initialization of the native component.

This BinWrap-infused library sets up appropriate seccomp-BPF filters for the set of syscalls extracted in the previous step. BinWrap (re)links the native component against this library instance, effectively "injecting" the filter into the native component.

6.1 Isolation Components

Native-code execution. Native modules utilize the NAN package to wrap unmanaged code. Native code is invoked through callback objects. BinWrap dispatches these callback objects to a restricted thread, which is initialized during the first time a native function is executed. The Node.js (main) process thread that dispatched the callback object to the restricted thread will block until the native function returns, after which the main thread will be unblocked. We used a shared lock as the synchronization primitive between the two threads. This "split thread" design enables BinWrap to leverage thread-specific mechanisms (i.e., MPK/PKU and seccomp-BPF) to isolate the execution of native code. A separate thread is created for each native module loaded, since seccomp-BPF filters are per-module and can be only modified to deny more syscalls—thus native modules with different syscall sets cannot share the same thread.

Data-access filtering. Since BinWrap decouples the execution of untrusted code, by dispatching it to a restricted thread, we can prohibit arbitrary accesses to sensitive data stored in the memory (part of the VAS) of Node.js by leveraging Intel’s MPK/PKU [32]. During the initialization of the native module thread, we associate a protection key with the pages that may contain sensitive data; these include all the memory regions allocated and managed by the V8 JS engine. The native module thread will initially change the rights associated to no access, on the protection key assigned to the allocated pages of Node.js. The native module address range(s), and allocated memory, are excluded from this set. Subsequent memory allocations for executing V8 memory pools are also associated with the protection key of Node.js. Finally, through analyzing the symbol tables of the top-500 popular native modules, we found no occurrences of explicit data sharing between Node.js and native modules (e.g., via globally-scoped symbols).

A limitation of MPK/PKU is that only 16 keys are available, and thus only 16 different memory views can be supported. As we mention in Section 5, we encountered at most 10 imported native modules, by a single library, in the entire NPM ecosystem. Moreover, this issue can be address by solutions like Libmpk [53], which virtualize protection keys. Another solution is to group sets of native add-ons under the same protection key; in this case, however, a vulnerable module can affect everything else in the same set.

Node.js and V8 export a large API in order to allow native modules to perform various tasks (object allocation and management, type conversions, etc.). Since Node.js is part of the trusted domain, when the native thread executes Node.js API functions, the access rights on Node.js data should be re-enabled. In our framework, we modified the API functions to change the rights for the corresponding protection key to allow memory operations.
We link the wrapper DSO to the native module, and resolve the functions as execute-only. Thus, arbitrary reads will fail to reveal any data, and Node.js API functions may copy data in the stack (e.g., %pkru). During every API call we spill the %pkru register on the call stack; during returns, %pkru will be modified only if the previous rights stored in the stack restrict memory accesses further (i.e., the execution “returns” to the native module). This design choice stems from the fact that API calls may be nested.

During the execution of the trusted part, the thread can access any data, and Node.js API functions may copy data in the stack (e.g., as function arguments). An attacker could access sensitive data by harvesting stale data in deallocated stack frames after API functions return. To prevent this, the native execution thread zeroes out the deallocated stack frames, before returning back to the native module, effectively deleting any residual data.

**Code-reuse prevention.** An attacker could launch a code-reuse attack, in the untrusted domain, targeting instruction snippets like `wrpkru`, which remove restrictions and allow accessing data from memory areas that are inaccessible by the native module. These instruction sequences are present in the trusted (Node.js) code since the data access restrictions are lifted during its execution. They can also be implicitly present in the untrusted part, since x86 instructions are variable-length and the architecture allows for overlapping instructions [63]. Moreover, the `xrstor` instruction can be leveraged to tamper-with `%pkru`, effectively allowing access on restricted memory. In order to prevent an attacker from using the “unlocking” code, we again utilize MPK and also rely on information hiding [39] to hide the location of the trusted code (Node.js) from the untrusted part (i.e., the native module). The key idea of this technique is that the native module can call Node.js API functions without ever knowing their address. We designed a custom linking procedure to hide Node.js, and library locations, from the untrusted part, which operates as follows.

Initially, we extract all the API and library functions, needed by the native module, from the respective DSO’s `.plt` section, along with the corresponding offset(s) in the `.got` section. Then, we create a new `wrapper` DSO, which contains a wrapper function for every Node.js API and library function required by the native module. We link the wrapper DSO to the native module, and resolve the native module’s dynamic symbols to point at the wrapper’s (at load time). Next, we utilize the capability from MPK to mark the wrapper functions as execute-only. Thus, arbitrary reads will fail to reveal API and library locations. Additionally, information hiding ensures that the addresses of Node.js, and the linked libraries, are different on separate executions.

**Figure 3** presents a high-level overview of BinWrap’s compilation and linking procedure. Finally, the wrapper library implements dynamic symbol interposition to filter syscalls that can bypass the native module sandbox, if misused [13, 61, 74]. To prevent attacks targeting implicit `xrstor` and `wrpkru` instructions, we scan the binary with the ROPgadget tool [59]. We vet any occurrence of these instructions in a similar manner as G-Free [49]. Since we can also operate on the source level of the native module, we do not strictly rely on static binary rewriting [75] to vet unsafe instructions; we rather transform the source code to prevent unsafe instructions from being emitted in the final binary. Our analysis of the top-500 native modules with ROPgadget found no implicit occurrences of `xrstor` and `wrpkru` instructions. The chances of implicit occurrence are low, since both these instructions are larger than 3-bytes. We do not need to vet unsafe instructions in Node.js or the linked libraries (e.g., libc), since we wrap all the dynamically linked symbols with execute-only wrappers in order to hide their actual location in the memory. This process can be further improved with the adoption of fine-grain, leakage-resilient code diversification [9, 16, 54].

### 6.2 System-call Set Extraction

Native modules often depend on syscalls for key functionality available by the underlying OS. There are two avenues native modules issue syscalls: (a) by directly invoking the respective syscall or `11bc` wrapper; and (b) by indirectly invoking syscalls through the use of Node.js APIs—e.g., the add-on calls `v8::External::New(v8::Isolate*, void*)`, which internally calls `brk`.

Since native components make extensive use of Node.js-internal APIs, the resulting combined set of syscalls used by the native component may be large. This set can be used as a means for an attacker to cross the protection boundary, effectively bypassing BinWrap’s enforcement mechanism [13, 74]. To extract the full set of system calls a native module requires, we use an intra-procedural, precise binary analysis [18].

**Direct syscalls.** The analysis first receives as input the native component; i.e., a `.node` (ELF) file. It proceeds to resolve dependencies to shared libraries, and then (over-)approximates the function-call graph (FCG) of the native component. This approximation is constructed over all objects in the scope of the component and its dependencies. We then use-def analysis atop the FCG to extract a tight (but safe) set of developer-intended syscalls [18].

**Inherited syscalls.** To identify the Node.js-internal API functions used by the native component, we first analyse the native module’s symbol table. We then use each function symbol as the entry point for analyzing the Node.js executable and identify the reachable syscalls. The analysis trades soundness for completeness, in that system calls performed by the native component will exist in the extracted set—however, not all extracted syscalls are expected to be used in every execution of the native component.

### 6.3 System Call Filtering

BinWrap uses the extracted system-call set to create a filter containing the complete set of syscalls that may be executed by the restricted thread. Given a set of allowed syscall numbers, BinWrap’s enforcement tool uses `seccomp-BPF` to perform syscall filtering during the execution of add-on code on the restricted thread.
To inject the filter into the add-on, BinWrap uses a set of specific Node.js API templates. BinWrap provides custom templates of Node.js API libraries that contain placeholder segments instantiated with custom filter instances. Different Node.js API wrappers—NAN, NAPI, etc.—correspond to different templates. Furthermore, syscalls requiring pointer argument filtering (see Section 7.2) are interposed in the wrapper DSO through their respective \texttt{\#include} function.

BinWrap then instantiates each template (still as source code) using (1) information extracted from the earlier static-analysis phase (§6.2), and (2) additional hard-coded policies for syscalls than can be potentially abused. It then compiles the native component, linking against the Node.js API instance, which contains the seccomp-BPF filter corresponding to this native component and the Node.js, V8, and wrapper DSO. Loading the compiled native component at runtime will result in the untrusted thread executing the appropriate seccomp-BPF filter upon initialization.

7 IMPLEMENTATION

Our framework applies across the whole stack of Node.js (§3). The JS code of the third-party library is analyzed with BinWrap to extract the permission model that will be enforced at runtime (§3.2). Our NAN modifications add \#200 LOC (C++) for initializing the restricted execution thread, the synchronization primitives, and seccomp-BPF (§6). We chose NAN (for BinWrap) due to compatibility reasons; our implementation is applicable to NAPI as well.

7.1 Memory Isolation

Node.js and V8 API modifications. We modified any V8 and Node.js API function reachable through the NAN API. We identified the full set of these functions by analyzing the test suite of the NAN package. Our analysis discovered 122 dynamic symbols that point to V8 and Node.js code. We additionally analysed the native modules that consist our evaluation set and found that they link less than half of these functions—i.e., \approx 50 symbols. Our modifications remove memory restrictions upon entry (to Node.js/V8 code) and reimpose them before the API function returns to the native module (§6.1).

Wrapper code. Wrapper libraries are generated using shell scripts and range between 240 – 600 LOC ( Bash), depending on how many V8 and Node.js symbols are dynamically linked to the node add-on. The wrapper functions are pure (i.e., no stack frame) and consist of two instructions implementing a computed branch.

7.2 Memory-sandbox Hardening

Several studies have shown that certain syscalls can be leveraged to bypass MPK/PKU-based restrictions [61, 74].

Memory management. Syscalls that are used for memory management, like \texttt{mmap}, \texttt{mprotect}, \texttt{mremap}, etc., can be used to allocate executable memory, execute instructions that mangle protection keys, and access data in normally-inaccessible memory locations. Protection keys can also be wiped by de-allocating and re-allocating the target memory region(s). In BinWrap, we hook these system calls and disallow them to target the protected domains as well as allocating executable memory. We also disallow remapping executable pages, since it is possible to form unsafe instructions on page boundaries. Using \texttt{set_thread_area} with \texttt{READ,IMPLIES,EXEC} an attacker can render any subsequent allocated pages executable. None of the native modules required this system call, and thus we safely deny its execution. We finally disallow userfaultfd, since it can enable arbitrary writes on MPK-protected pages [61].

Process/thread control. Another family of dangerous system calls is related to process creation ( \texttt{fork, execve, clone} ). In clone, the MPK configuration is inherited to the new thread, and is thus safe; \texttt{fork}, however, can be combined with \texttt{kill} to force the child process produce a core dump, which can then be read by the untrusted domain. We found that \texttt{fork} and \texttt{execve} are not really required by native add-ons and deny their execution. We also disable core dumps by configuring the application process with \texttt{PR_SET_DUMPABLE} and \texttt{SU_ID.DUMP_DISABLE}. This also prohibits access to \texttt{procs} and thus prevents the misuse of file-related syscalls. We also deny \texttt{ptrct} and \texttt{set_thread_area}, which can remap thread-local storage.

Signal handling. During signal delivery, the kernel stores the register state (including \texttt{\%rkr}) on the call stack. When the signal handler finishes its execution, the register state is restored through \texttt{rt\_sigreturn}. An attacker can craft a register state where the \texttt{\%rkr} register allows full memory access and execute a \texttt{sigreturn} gadget in order to obtain universal access [8]. There is no wrapper for \texttt{rt\_sigreturn} in Node.js, since it is not supposed to be called by applications. However, an attacker can call \texttt{rt\_sigreturn} through manipulating \texttt{syscall/sysenter} instructions. In BinWrap, we treat such instructions as unsafe, and we prohibit them in the executable section(s) of the add-on. Thus, an attacker cannot invoke \texttt{rt\_sigreturn}.

8 EVALUATION

To assess BinWrap, we use a set of real-world, native NPM packages, investigating the following evaluation questions (EQs):

- **EQ1** How effective is BinWrap at defending against attacks that exploit real-world vulnerabilities? (§8.1)
- **EQ2** How much BinWrap reduces the set of syscall in the context of native modules, what is the set breakdown? (§8.2)
- **EQ3** How efficient and scalable are each of BinWrap components (BinWrap, BinWrap)? (§8.3)

Add-ons and workloads. We evaluated each of BinWrap components, assessing their security guarantees and the runtime overhead imposed. To address EQ1, we evaluate BinWrap against exploits targeting vulnerabilities (pulled from the Snyk [64] database) in popular third-party libraries. We found that BinWrap successfully prevents the exploitation of the selected vulnerable packages.
To address EQ2 and EQ3, we evaluated the overhead of BinWrap using real-world applications that stress individual components and provide insights about the micro- and macro-aspects of the performance impact. Our results indicate that BinWrap can offer strong security guarantees to JS applications that utilize third-party native add-ons, while imposing an overhead between 0.71%–10.40%.

To benchmark each package and application, we tried to execute its corresponding test suite. If the application developer(s) did not provide any test suite, we used the example code provided in the respective repository. We used two sets of benchmarks: the first, and major set, consists of benchmarks that implement the behavior of an actual application, which uses third-party libraries—i.e., executing mostly JS code and offloading heavy computations on the native add-on(s); the second set consists of benchmarks that execute native functions in tight loops, thereby stressing the cross-domain transitions of our framework.

**Testbed.** We used a host armed with an Intel Core i9-10900 CPU and 32GB of RAM, running Linux v5.4. We implemented our modifications on Node.js v8.9.4. BinWrap does not require any kernel modifications to run and only needs support for MPK/PKU and seccomp-BPF. We run our benchmarks in the latest Ubuntu distribution, and so we had to recompile each library that Node.js loads dynamically to remove (Intel) CET/IBT instrumentation [14], which is added by default in most packages. (Intel IBT uses a customized .plt section that is not supported by sysfilter [18].) Note that removing IBT instrumentation does not affect the analyses, or code transformations, of BinWrap. Finally, we disabled C-states and Turbo Boost, and locked the clock frequency at 2.8GHz.

**Evaluation set.** To find a representative set of libraries to evaluate BinWrap, we analysed the entire NPM ecosystem. The goal of this analysis was to find applications covering the following criteria: (i) large number of dependencies; (ii) compatibility with our tool; and (iii) security exigency. In addition, the respective NPM packages should run successfully in the unmodified baseline. To analyze the applications on NPM, we used the local registry from our study (§5).

We took multiple steps to get from the 1,508,366 libraries to the 20 in the evaluation set (Table 1). All these steps are displayed on Figure 4. First, we found all the libraries that have NAN as a dependency (5,073 packages). We only consider packages that can be installed without manual effort (4,201 packages), while we also removed duplicates (3,508 packages). Next, we selected packages shipped with test cases that could run out of the box (400 packages). Finally, we reduced our set to 20 based on the criteria outlined in Figure 4. The evaluation set includes packages that do not have large number of dependents.

8.1 Security Evaluation (EQ1)

To assess the security of BinWrap we implemented exploits for four distinct CVEs, by analyzing vulnerabilities reported in Snyk [64] for NPM packages. These vulnerabilities occur due to memory errors in the DSO(s) that ship(s) with various NPM packages. To evaluate the security of BinWrap, we exploit these vulnerabilities, and bypass the boundaries of the untrusted part, by mimicking similar, publicly-available exploits [19–21].

**CVE-2018-11499** is an *information disclosure* vulnerability that manifests via a use-after-free. The vulnerability is present in the node-sass package until v3.5.5, and occurs due to lack of proper exception handling. Our exploit manages to leak pointers to heap addresses, which can be used to construct arbitrary memory read primitives. With BinWrap, any attempt to read beyond the memory allocated to libsass fails due to the restrictions (memory sandboxing) imposed through MPK/PKU (§6.1).

**CVE-2018-18577** is a *heap-based buffer overflow* vulnerability enabling the construction of arbitrary write primitives. The vulnerability is present in libtiff, which the picha package loads for processing image files. The bug stems from libtiff ignoring the size of a destination buffer when decompressing JBIG-compressed images. With BinWrap, this exploit fails in corrupting data that do not belong to the native module’s benign memory pages (§6.1).

**CVE-2019-3822** is a *stack-based buffer overflow* that can lead to the execution of a ROP gadget chain. The node-libcurl NPM package links with libcurl, which contains the stack-based buffer overflow vulnerability from v7.36 to v7.64. The vulnerability is due to the fact that during an NTLM negotiation, libcurl sends a message to the server containing the server’s original response. If that response is large enough, it overflows a buffer in the call stack.

We used Ropper [60] to create a ROP chain that loads the command we want to pass to the system function of libc, and executes it. This exploit is unsuccessful when BinWrap is deployed, since the system function ends-up executing the execve syscall, which is not part of libcurl’s benign syscall set (§6.3).

**CVE-2020-28248** leads to an under-allocated buffer due to an *integer overflow* in a memory initialization function. The exploit is present in the png-1ng NPM package in all versions up to v3.1. This condition introduces a heap-based buffer overflow that can be exploited using a specially-crafted PNG file. Libpng registers a callback function in a struct for reporting errors; the inadequately-sized buffer is in the lower addressed region of the struct containing the error callback function.
Table 1: Third-party libraries in our evaluation set.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Dependents</th>
<th>C LOC</th>
<th>sycalls</th>
<th>Test Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>node-sass</td>
<td>Style sheet preprocessor</td>
<td>8457</td>
<td>37365</td>
<td>93</td>
<td>Macro</td>
</tr>
<tr>
<td>bip32</td>
<td>Bitcoin wallet client</td>
<td>632</td>
<td>5559</td>
<td>82</td>
<td>Macro</td>
</tr>
<tr>
<td>xml.js</td>
<td>XML and SAX parser</td>
<td>344</td>
<td>170K</td>
<td>93</td>
<td>Macro</td>
</tr>
<tr>
<td>iconv</td>
<td>Text recoding</td>
<td>329</td>
<td>96967</td>
<td>91</td>
<td>Micro</td>
</tr>
<tr>
<td>zeroMQ</td>
<td>Networking library</td>
<td>323</td>
<td>8131</td>
<td>114</td>
<td>Macro/Micro</td>
</tr>
<tr>
<td>node-ref</td>
<td>Memory buffer utilities</td>
<td>289</td>
<td>6900</td>
<td>91</td>
<td>Macro</td>
</tr>
<tr>
<td>tiny-secp256k1</td>
<td>Optimised library for ECDA</td>
<td>187</td>
<td>24511</td>
<td>82</td>
<td>Macro</td>
</tr>
<tr>
<td>heap-dump</td>
<td>V8 heap dump</td>
<td>173</td>
<td>6395</td>
<td>91</td>
<td>Macro</td>
</tr>
<tr>
<td>ttf2woff2</td>
<td>TTF to WOFF2 converter</td>
<td>155</td>
<td>28185</td>
<td>91</td>
<td>Macro</td>
</tr>
<tr>
<td>pty.js</td>
<td>Pseudo terminal for Node.js</td>
<td>128</td>
<td>6999</td>
<td>93</td>
<td>Macro</td>
</tr>
<tr>
<td>blake2</td>
<td>Hash function library</td>
<td>19</td>
<td>26207</td>
<td>91</td>
<td>Macro</td>
</tr>
<tr>
<td>pngImg</td>
<td>PNG image processing library</td>
<td>6</td>
<td>66244</td>
<td>93</td>
<td>Macro</td>
</tr>
<tr>
<td>picha</td>
<td>JPEG encoder/decoder</td>
<td>4</td>
<td>7522</td>
<td>92</td>
<td>Macro</td>
</tr>
<tr>
<td>statvfs</td>
<td>File-system information</td>
<td>3</td>
<td>6463</td>
<td>93</td>
<td>Micro</td>
</tr>
<tr>
<td>mtrace</td>
<td>Native memory tracing and logging</td>
<td>3</td>
<td>6263</td>
<td>91</td>
<td>Micro</td>
</tr>
<tr>
<td>node-uri parser</td>
<td>Native library for URI parsing</td>
<td>3</td>
<td>8111</td>
<td>90</td>
<td>Macro/Micro</td>
</tr>
<tr>
<td>node-hll-native</td>
<td>Hyper log log algorithm</td>
<td>2</td>
<td>6663</td>
<td>91</td>
<td>Macro/Micro</td>
</tr>
<tr>
<td>syncrunner</td>
<td>Return output from binary execution</td>
<td>2</td>
<td>10363</td>
<td>91</td>
<td>Micro</td>
</tr>
<tr>
<td>node-fs-ext</td>
<td>File system utilities</td>
<td>0</td>
<td>6863</td>
<td>95</td>
<td>Micro</td>
</tr>
<tr>
<td>node-delta</td>
<td>Delta compression algorithm</td>
<td>0</td>
<td>6680</td>
<td>82</td>
<td>Macro</td>
</tr>
</tbody>
</table>

| Video Thumb Grid| Video thumb grid generation (uses picha) | na | 7522 | 92 | Application|
| Manta Minnow    | Storage utilization agent for Manta project (uses statvfs) | na | 6463 | 91 | Application|

Figure 5: Syscall-set size for the 20 native add-ons (combined with syscalls inherited due to Node.js APIs).

8.2 System-call Set Analysis (EQ2)

We analysed the 20 native modules with `sysfilter` to extract the set of syscalls: (i) required by the native module itself; and (ii) required through Node.js API functions. We found that Node.js API functions require the same 62 system calls in all the 20 (native) add-ons. We also found that the native module DSOs require roughly the same syscalls as the Node.js API functions. The number of system calls inherited from Node.js range from 2 (`zeromq`) to 35 (`node-delta`).

This vulnerability can be used to overwrite the callback (function pointer), with an address of our choosing; we chose to overwrite the error callback with the address of `system` in `libc`. Similarly to CVE-2019-3822, this exploit is unsuccessful when BinWrap is deployed, due to syscall filtering (§6.3).

As shown in Figure 5, we can safely block more than \( \approx \frac{2}{3} \) of the available syscalls in most cases, except in `zeromq`, which requires 114 system calls due to utilizing network sockets.

8.3 Performance Evaluation (EQ3)

To assess the performance impact of BinWrap, when enabled on third-party libraries, we compare its runtime performance against the vanilla Node.js. We break BinWrap down to three different parts (to measure their contributing overhead): (i) the native function sandbox; (ii) the dynamic analysis privilege checks; and (iii) the combined impact of (i) and (ii) on runtime performance. Figure 6 summarizes the results of our evaluation.
Macro benchmarks. We evaluated BinWrap by running the test suite provided by each NPM package. We run the npm test command 100 times and measured the average execution time for the unmodified NPM package and with BinWrap enabled.

The results indicate that BinWrap imposes moderate overhead(s), ranging between 0.71%–10.40%. The typical workload is the execution of JS code with sporadic invocation of native functions. The overhead of the dynamic enforcer of BinWrap is bound to the size of the access rights extracted during the analysis phase ($\S$3).

Micro benchmarks. During domain transitions, the restricted thread is unlocked and executes the native function. The main thread, in turn, waits for the native module thread to finish the callback execution. We evaluated several synchronisation algorithms to measure the performance in this scenario. Our baseline micro benchmark is a function that increments a global variable, called 100M times. Next, we implemented the same scenario but this time the process spawns a thread that will be responsible to increment the variable. The new thread increments the variable once and then locks until the main thread unlocks it. In a similar manner, the main thread will lock until the thread responsible to increment the variable unlocks the synchronization variable. When utilising futex the overhead compared to the benchmark without threads is 240x. When using inline ASM memory operations to spin on the synchronization variable the overhead was reduced to 80x.

We also evaluated BinWrap with test cases included in NPM packages that stress various security mechanisms; we present our results in Figure 7. In the case of uriparser, the benchmark code consists of only two loops that parse a URL 2M times. The first loop uses the JS implementation of the parser, while the latter uses the natively-implemented parser. The majority of the code executed triggers the synchronization mechanism between Node.js and the restricted thread. A similar scenario appears in node-hll-native. The benchmark implements a tight loop that executes a native hyperloglog function 50M times, which only executes $\approx$300 instructions. Finally, zeromq consists of two instances (sender and receiver) communicating with small (1KB) TCP packets. The receiver expects 1M packets from the sender. In this case, both the synchronization and the system call filtering components are stressed.

9 RELATED WORK

OSes rely on process isolation (e.g., via means of virtual memory) to prevent processes from arbitrarily interfering with each other. Intra-process isolation, is required in applications that need to isolate components within the same VAS. For example, web browsers isolate the execution of different pages in order to prevent malicious JS code from accessing sensitive data. A notable family of intra-process isolation techniques is SFI (Software Fault Isolation); SFI instruments memory operations in order restrict memory access beyond a designated area.

Other instrumentation approaches ensure that out-of-bound pointers are transformed to in-bound. Research efforts focus on in-process techniques, offering isolation guarantees with minimum cost [5]. Beyond software-only solutions for intra-process isolation, there are different mechanisms in widely used architectures that can be leveraged for that purpose.

BinWrap utilizes Intel MPK/PKU [32] to differentiate access rights on memory when accessed from the trusted and untrusted parts of Node.js applications. A comparison with various related systems is displayed on Table 2. Several other research efforts, also leverage MPK for intra-process isolation: ERIM [68] and Hodor [29] introduce security domains in applications, and protect sensitive data from being accessed by untrusted components. The access rights are modified through call gates (ERIM) and trampolines (Hodor). Moreover, binary inspection is used in order to vet occurrences of MPK-mangling instructions. Regarding system calls, ERIM only intercepts memory management system calls, while Hodor denies any system call originating from untrusted domains by modifying the underlying OS. Recent studies [13, 74], however, have demonstrated bypasses on both ERIM and Hodor; extended system call filtering with ptrace in ERIM solves some issues, but incurs substantial overhead [61].

Donky [62] modifies a RISC-V processor, enabling protection keys and user-level interrupts. Domain transitions and memory management syscalls are managed by a per-process monitor; only the monitor has access to the protection key registers. Jenny [61] resolves several limitations of Donky, effectively implementing more complete system call filtering. However, Jenny and Donky cannot be directly applied in x86 and require custom hardware.

PKRU-Safe [36] shepherds inter-domain data flows in MPK-based sandboxes, but does not address the security issues presented by Connor et al. and Voulimeanis et al. [13, 74] (i.e., syscall misuse, stray MPK instructions). PKRU-Safe is orthogonal to BinWrap and could be deployed in order to enhance our memory restriction policies (i.e., what can be shared between Node.js and native add-ons). Cerberus [74] aims to address the issues of MPK-based sandboxes presented by Connor et al. [13], and also presents novel attacks. Cerberus is an API, offering primitives for protecting other MPK sandboxes, like Hodor and ERIM. System calls are handled through a kernel-side monitor, but as the monitor is implemented in the OS, it is not able to thwart sigreturn-based attacks.
We presented BinWrap, a framework that applies across the whole stack of Node.js applications in order to isolate the execution of potentially-vulnerable, third-party native add-ons. We studied the Node.js ecosystem and presented insights about how native add-ons are used. By identifying certain trends, we implemented a hybrid protection scheme that wraps both the native and high-level language components of a native library. We evaluated the security of our framework against exploits in real-world applications, and we assessed the runtime performance overhead of BinWrap, which ranges between 0.71%–10.4%. We believe that BinWrap is a practical framework that can protect Node.js applications in the presence of native add-ons that are vulnerable to memory errors.

Availability
The prototype implementation of BinWrap is available at: https://github.com/atlas-brown/binwrap

ACKNOWLEDGMENTS
We thank the anonymous reviewers for their valuable feedback. This work was supported by European Union’s Horizon 2020 research and innovation programme, under grant agreements 883540 and 958478, as well as in part by the National Science Foundation (NSF), through award CNS-2238467, and the Defense Advanced Research Projects Agency (DARPA), through contracts HR001120C0191 and HR001120C0155. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the EU, the US government, NSF, or DARPA.

REFERENCES

Table 2: Comparison of MPK sandboxes. 1Only addresses system call issues and is based on Donky. 2Is an API for MPK-based sand boxes. PKRU-safe does not address MPK-based sandbox issues (i.e., system calls, stray unsafe instructions). 4Cerberus does not prevent exploitation through sigreturn.

<table>
<thead>
<tr>
<th></th>
<th>Erim</th>
<th>Hodor</th>
<th>Donky</th>
<th>Jenny</th>
<th>PKRU-Safe</th>
<th>BinWrap</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-Process Isolation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>No Kernel Modifications</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>System Call Restrictions</td>
<td>Partial</td>
<td>Partial</td>
<td>Partial</td>
<td>Complete</td>
<td>Partial</td>
<td>No</td>
</tr>
<tr>
<td>Unsafe Instruction vetting</td>
<td>Partial</td>
<td>Partial</td>
<td>NA</td>
<td>Partial</td>
<td>Partial</td>
<td>No</td>
</tr>
<tr>
<td>PKU Pitfalls Protection [13]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>New PKU Pitfalls Protection [74]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Performance Overhead</td>
<td>Low</td>
<td>Moderate</td>
<td>Low</td>
<td>Moderate</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

10 CONCLUSION
We presented BinWrap, a framework that applies across the whole stack of Node.js applications in order to isolate the execution of potentially-vulnerable, third-party native add-ons. We studied the Node.js ecosystem and presented insights about how native add-ons are used. By identifying certain trends, we implemented a hybrid protection scheme that wraps both the native and high-level language components of a native library. We evaluated the security of our framework against exploits in real-world applications, and we assessed the runtime performance overhead of BinWrap, which ranges between 0.71%–10.4%. We believe that BinWrap is a practical framework that can protect Node.js applications in the presence of native add-ons that are vulnerable to memory errors.

REFERENCES
A NODE.JS AND V8

Memory organization. V8 follows a hierarchical (virtual) memory organization scheme that is primarily geared towards garbage collection (GC). Irrespective of how JS code is executed atop V8 (interpreted vs. compiled), JS programs are represented by a so-called resident set, which is the collection of memory pages that V8 allocates to facilitate the execution of the respective program. The resident set is further divided to memory (sub)regions that correspond to the runtime (execution) stack of the JS program, as well as the heap. The heap is designed with aggressive GC in mind, and to this end, it is partitioned to multiple (semi-)spaces, which are object allocation areas that host short- and long-lived objects, in a way that makes GC performant and effective. The heap region also includes special SLAB-like areas (or spaces), to support the fast allocation of special, typed objects, “large” ( mmap-ed) objects, as well as jitted code [24]. Lastly, V8 uses pointer tagging to differentiate between plain data and pointer values, while (dynamically-allocated)

read-only-relo.


A NODE.JS AND V8

Memory organization. V8 follows a hierarchical (virtual) memory organization scheme that is primarily geared towards garbage collection (GC). Irrespective of how JS code is executed atop V8 (interpreted vs. compiled), JS programs are represented by a so-called resident set, which is the collection of memory pages that V8 allocates to facilitate the execution of the respective program. The resident set is further divided to memory (sub)regions that correspond to the runtime (execution) stack of the JS program, as well as the heap. The heap is designed with aggressive GC in mind, and to this end, it is partitioned to multiple (semi-)spaces, which are object allocation areas that host short- and long-lived objects, in a way that makes GC performant and effective. The heap region also includes special SLAB-like areas (or spaces), to support the fast allocation of special, typed objects, “large” ( mmap-ed) objects, as well as jitted code [24]. Lastly, V8 uses pointer tagging to differentiate between plain data and pointer values, while (dynamically-allocated)
JS objects are represented by opaque handles and accessed/modified via specific accessor functions.

Native modules. JS applications can load native modules by using the require function—i.e., the same mechanism that is used to load JS-only libraries. In Linux, such add-ons are implemented as dynamic shared objects (DSOs), using the .node file extension, which are essentially ELF DSOs mapped in the (virtual) address space of the Node.js process via means of dlopen—the dynamic linker/loader (1d.so) will first relocate the ELF object accordingly, then recursively load all its .so dependencies (represented by DT_NEEDED entries in the .dynamic section), perform symbol resolution (fill the respective entries in the GOT sections), and, finally, invoke the module constructors (functions annotated with __attribute__((constructor)), .init, etc.). Add-ons may export functions, and objects, to JS code, directly invoke JS functions passed as callbacks, and even wrap C++ objects/classes in a way that enables their instantiation directly from JS code (e.g., using the new operator).

The interoperability between JS and native, C/C++ code is directly facilitated by V8’s native code bindings. More specifically, by leveraging the Node API (NAPI), or even the more esoteric V8 API [27], add-ons can (un)marshal function arguments and return values to/from JS code, invoke JS code (mostly asynchronously), raise (and handle) exceptions, access/pass objects in JS scope, perform JS-to-C/C++ type conversion, and more [37]. The NAN (Native Abstractions for Node.js) API [46] is yet another Node.js API, designed as a portable, stable API for add-on development, given that both the NAPI and the V8 API are version- and platform-dependent (and therefore hinder the portability on add-on code).

B PROTECTION KEYS

Protection keys is a relatively common architectural feature, first introduced in IBM System/360. (Today, IBM storage protection keys [30] are part of the Z architecture.) A protection key is assigned to each virtual page and represents the access authority required for each context. An authority mask register is used for specifying the access rights of each context. IA-64 protection keys [31] are designed to restrict permissions on memory by tagging each virtual page with a unique domain identifier. IBM extended the protection keys architecture with 16 Protection Key Registers used as a cache for the access rights on the protection domains required by a process. During memory accesses, if a key is found during memory translation, it is looked-up in the available protection key registers to check the access rights. ARM memory domains [3] offer multiple sandboxes to a process. There can be 16 memory domains in each process, and a domain access control register (DACR) defines the access rights on each domain. DACR is a privileged register, and thus, domain switches are handled by the supervisor level.

Intel MPK/PKU offers the ability to userspace processes to change access permissions on groups of pages. Each page group is associated with a unique key. An application can have up to 16-page groups. The access rights for each page group are mapped in a thread-local and user-accessible register called protection keys rights register (for user) %pkru. Since the %pkru register is thread-specific, MPK supports per-thread views of the process’s memory. For example, different application threads have different access rights [56] configured for each key in their %pkru register.

The key benefits of MPK over page table permissions are performance and the ability to configure different memory permissions for each thread in a process. The access rights supported by the page groups are read/write, read-only, and no access. Data accesses on memory pages associated with protection keys are checked both against the access rights defined in the %pkru register, as well as the permissions in the page table. Instruction fetching is checked through the permissions in the page table only.

If a memory page is executable in the page table but configured with no access in %pkru, the memory page is treated as execute-only. This occurs since any data access will result in a mismatch between the rights defined in the page table and the %pkru register. Linux supports execute-only memory pages by leveraging MPK. A call to mprotect with only PROT. EXEC specified as permissions will result in the allocation of a protection key which will be associated with the memory pages passed to mprotect. Next, the %pkru register will be set to DISABLE_ACCESS for the newly allocated protection key, while the page table rights will be set to executable and readable. Any access to execute only pages except for instruction fetching will result in a memory violation exception.

For associating a memory page (or range of memory pages) with a protection key, the Linux kernel implements the pkey_mprotect system call. Similarly to the traditional mprotect system call, it will also set the access rights passed as an argument in the %pkru register. The access rights in the %pkru register can be modified with the wrpkru x866 instruction. Since %pkru is user-accessible, modifying the access rights does not impose significant latency, and it is much faster than invoking memory management system calls (e.g., mprotect). Finally, rdpkru instructions returns the contents of the executing thread’s %pkru register.

C WRAPPER LIBRARY TEMPLATE

```c
__attribute__((constructor)) void init_method(void)
{
    ...
    mprotect((void*) wrap_node_api_func, 4096, PROT_WRITE);
    rewrite_loc = wrap_node_api_func;
    rewrite_loc = &node_api_function << 16 | 0x8080;
    ...
    mprotect((void*) wrap_node_api_func, 4096, PROT_EXEC);
    write.got = native.module.address +
                node_api_func.got_entry;
    ...
}
```

The constructor method first finds the location of the native module’s shared object in the process memory map. Then, the addresses of each symbol are collected using dlsym. The wrapper functions load an address in the auxiliary register %rax and then indirectly jump to that address with jmpq +%rax. The constructor then marks wrapper functions as writable and patches the new instructions in order to store the actual symbol’s address. Then the native module’s GOT is patched to point at the wrapper functions. Finally, the wrapper functions are configured as execute-only.
MODIFICATIONS IN NODE.JS AND V8 API

Each of the 122 entry points of Node.js and V8 where modified in order to remove memory restrictions upon entry and reinstate them before returning to the untrusted native module. Since API calls may be nested, we keep a copy of the previous rights in the stack frame in order to know when the execution transfers to the native module.