BinWrap: Hybrid Protection against Native Node.js Add-ons

George Christou
FORTH-ICS
Heraklion, Crete, Greece
gchri@ics.forth.gr

Grigoris Ntousakis*
Brown University
Providence, RI, USA
gntousakis@brown.edu

Eric Lahtinen
Aarno Labs
Cambridge, MA, USA
elahtinen@aarno-labs.com

Sotiris Ioannidis†
TU Crete
Chania, Crete, Greece
sotiris@ece.tuc.gr

Vasileios P. Kemerlis
Brown University
Providence, RI, USA
vpk@cs.brown.edu

Nikos Vasilakis
Brown University
Providence, RI, USA
nikos@vasilak.is

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 over both its binary and JavaScript sides, via 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`

ACM Reference Format:
George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P. Kemerlis, and Nikos Vasilakis. 2023. BinWrap: Hybrid Protection against Native Node.js Add-ons. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579856.3590330

*Also with TU Crete.
†Also with FORTH-ICS.

ASIA CCS '23, July 10–14, 2023, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-0098-9/23/07.
https://doi.org/10.1145/3579856.3590330

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 userspace 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 `DISABLE_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 `wrpkru` x86 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.

```
1  const fs = require('fs');
2  const PngImg = require('png-img');
3
4  let buf = fs.readFileSync('./img.png');
5  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.

```
1  const PngImgImpl =
2      require('./build/Release/png_img').PngImg;
3
4  module.exports = class PngImg {
5      constructor(rawImg) {
6          this.img_ = new PngImgImpl(rawImg);
7      }
8  }
```

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].

```
1  void PngImg::InitStorage_() {
2      rowPtrs_.resize(info.height, nullptr);
3      data_ = new png_byte[info.height * info.rowbytes];
4
5      for(size_t i = 0; i < .height; ++i) {
6          rowPtrs_[i] = data_ + i * info.rowbytes;
7      }
8  }
```

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$_B$), as well on JS code (BinWrap$_L$). (The latter typically wraps the add-on code and provides a high-level API for interfacing with Node.js-based application code.)

BinWrap$_B$ consists of a set of memory isolation and code confinement techniques, tailored to the runtime environment of Node.js, which aim at restringing 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$_B$ 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$_B$ 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$_B$ 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$_B$ 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$_L$ 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.

```
1  "/node_modules/png_img/index.js": {
2   "module": "r",
3   "module.exports": "w",
4   "require": "rx",
5   "require('./build/Release/png_img')": "ir",
6   "require('./build/Release/png_img').PngImg": "rx"
7  }
```

Armed with the above, BinWrap$_L$ traces object accesses at runtime, and blocks any attempt to access an object in a way 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 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—the 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.

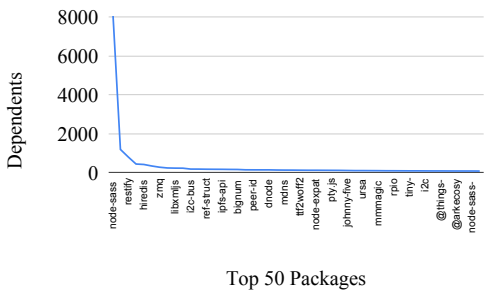

Figure 1: Number of dependents in the top 50 NPM packages.

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 incidence 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 `.so` dynamic shared 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

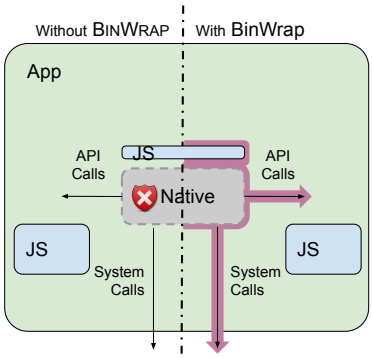

Figure 2: Hardening Node.js with BinWrap.

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 synchronisation 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 expanding V8 memory pool(s) 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.

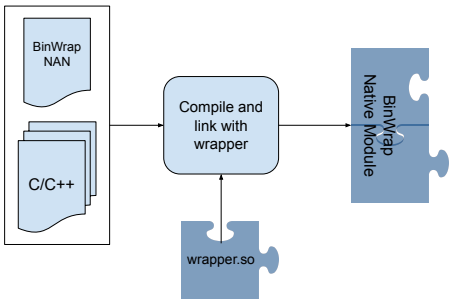

Figure 3: Compilation order of native modules in BinWrap.

(We reimpose restrictions before an API function returns back to the native module.) 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 `libc` 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 `libc` 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$_L$ 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$_L$) 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., ≈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 native add-on. The wrapper functions are *pure* (i.e., no stack frame) and consist of two instructions implementing a computed branch.

```
1  __attribute__ ((aligned(4096), pure))
2  void
3  wrap_node_api_func(void)
4  { asm ("movq 0xdeadcafe, %rax; jmpq *%rax"); }
```

Syscalls with pointer arguments that can be potentially misused are intercepted by *preloading* their `libc` wrapper. The wrapper includes a *constructor* method that will be the first function executed when the native add-on is loaded. Each pure wrapper is patched by the constructor in order to store the wrapped symbol's address (e.g., `0xdeadcafe`) in the auxiliary register (`%rax`), which will be dereferenced during the computed branch. The native module's `.got` is configured to point at the wrapper functions. Finally, the constructor maps the wrappers as execute-only. The net effect of the above is the effective "hiding" (in a leakage-resilient manner) of the wrapped symbols. (Refer to Appendix C and D for more information about symbol wrapping and Node.js and V8 API modifications.)

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 `mmap`, `mprotect`, `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 `personality()` with `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 (`fork`, `execve`, `clone`). In `clone`, the MPK configuration is inherited to the new thread, and is thus safe; `fork`, however, can be combined with `kill` to force the child process produce a core dump, which can then be read by the untrusted domain. We found that `fork` and `execve` are not really required by native add-ons and deny their execution. We also disable core dumps by configuring the application process with `PR_SET_DUMPABLE` and `SUID_DUMP_DISABLE`. This also prohibits access to `procfs` and thus prevents the misuse of file-related syscalls. We also deny `prctl` and `set_thread_area`, which can remap thread-local storage.

Signal handling. During signal delivery, the kernel stores the register state (including `%pkru`) on the call stack. When the signal handler finishes its execution, the register state is restored through `rt_sigreturn`. An attacker can craft a register state where the `%pkru` register allows full memory access and execute a `sigreturn` gadget in order to obtain universal access [8]. There is no wrapper for `rt_sigreturn` in `libc`, since it is not supposed to be called by applications. However, an attacker can call `rt_sigreturn` through reusing `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 `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$_L$, BinWrap$_B$)? (§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.

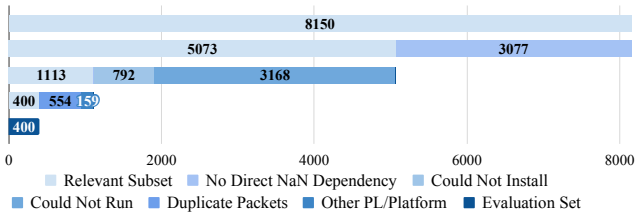

Figure 4: Overview of the analysis for our evaluation set.

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 behaviour 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 dependents; (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.

For example `uriparser` and `node-hll-native` have under 5 dependents. However, these libraries are shipped with tests that are suitable for benchmarking the micro-aspects of BinWrap. On the other hand, `statvfs` and `picha` are required by `Manta Minnow` and `Video Thump Grid` respectively, which are complete Node.js applications and offer insights about performance in real-world settings. `node-fs-ext`, `syncrunner`, `node-delta` and `mtrace` where chosen to diversify the types of libraries in our set. Finally, `png-img` is also used in our security evaluation, since it contained known vulnerabilities. The packages in the selected evaluation set contain one native library, which is the most common scenario (§5.2).

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-img` 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.

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

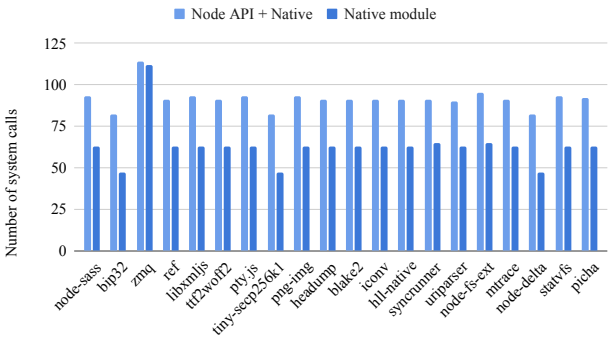

Figure 5: Syscall-set size for the 20 native add-ons (combined with syscalls inherited due to Node.js APIs).

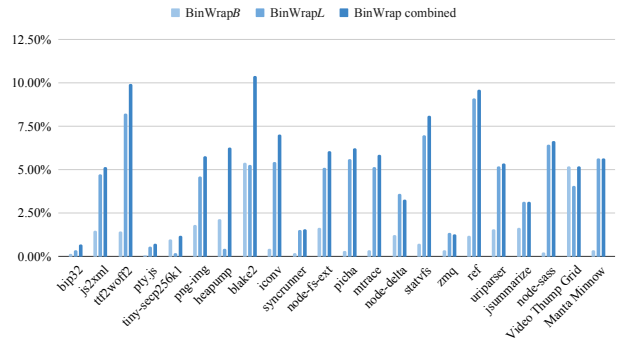

Figure 6: Runtime overhead when deploying BinWrap on the 20 native add-ons in our evaluation set.

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).

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`).

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 (§3). The overhead originating from the modifications in the native module's DSO is related to the synchronisation between Node.js and the restricted thread. Since `pkey_set` instructions are executed in userland, changing the rights during domain switches imposes negligible overhead. Interposing syscalls system and Node.js API functions is also lightweight since the number of extra instructions executed due to interposition is small (§7).

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 ≈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.

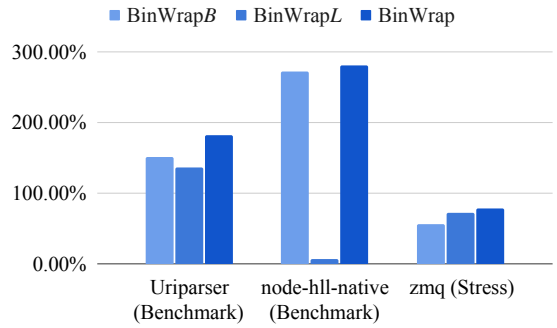

Figure 7: Micro-benchmarking results.

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 Voulimeneas 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.

Table 2: Comparison of MPK sandboxes. [1]Only addresses system call issues and is based on Donky. [2]Is an API for MPK-based sand boxes. [3]PKRU-safe does not address MPK-based sandbox issues (i.e., system calls, stray unsafe instructions). [4]Cerberus does not prevent exploitation through `sigreturn`.

	Erim	Hodor	Donky	Jenny	Cerberus	PKRU-Safe	BinWrap
In-Process Isolation	✓	✓	✓	✓[1]	✓[2]	✓	✓
No Kernel Modifications	✗	✗	✗	✗	✗	✓[3]	✓
System Call Restrictions	Partial	Partial	Partial	Complete	Partial	No	Complete
Unsafe Instruction vetting	Partial	Partial	NA	Partial	Partial	No	Complete
PKU Pitfalls Protection [13]	✗	✗	✗	✓	✓[4]	✗	✓
New PKU Pitfalls Protection [74]	✗	✗	✗	✓	✓	✗	✓
Performance Overhead	Low	Moderate	Low	Moderate	Low	Low	Low

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.

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

[1] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In *Annual Computer Security Applications Conference (ACSAC)*. 1–10.
[2] Apache. 2022. CouchDB. https://docs.couchdb.org/en/stable/.
[3] ARM. 2018. Domains. https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/Virtual-Memory-System-Architecture--VMSA-/Memory-access-control/Domains.
[4] Steve Bannister. 2018. Memory Tagging Extension: Enhancing memory safety through architecture. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety.
[5] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen, and Pierre Wilke. 2019. Compiling Sandboxes: Formally Verified Software Fault Isolation. In *European Symposium on Programming (ESOP)*. 499–524.
[6] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *ACM Asia Symposium on Information, Computer and Communications Security (ASIACCS)*. 30–40.
[7] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proc. of USENIX Summer*. 87–98.
[8] Erik Bosman and Herbert Bos. 2014. Framing Signals—A Return to Portable Shellcode. In *IEEE Symposium on Security and Privacy (S&P)*. 243–258.
[9] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *Network and Distributed System Security Symposium (NDSS)*.
[10] Bugtraq. [n. d.]. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63.
[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium (SEC)*. 249–266.
[12] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming without Returns. In *ACM Conference on Computer and Communications Security (CCS)*. 559–572.
[13] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *USENIX Security Symposium (SEC)*. 1409–1426.
[14] Intel Corporation. 2019. *Control-flow Enforcement Technology Specification*.
[15] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium (SEC)*, Vol. 98. 63–78.
[16] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE Symposium on Security and Privacy (S&P)*. 763–780.
[17] Willem De Groef, Fabio Massacci, and Frank Piessens. 2014. NodeSentry: Least-privilege Library Integration for Server-Side JavaScript. In *Annual Computer Security Applications Conference (ACSAC)*. 446–455.
[18] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 459–474.
[19] ExploitDB. 2000. cURL 6.1 < 7.4 – Remote Buffer Overflow. https://www.exploit-db.com/exploits/20293.
[20] ExploitDB. 2004. LibPNG Graphics Library – Remote Buffer Overflow. https://www.exploit-db.com/exploits/389.
[21] ExploitDB. 2010. LibTIFF Buffer Overflow (Metasploit). https://www.exploit-db.com/exploits/16869.
[22] Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. Building Diverse Computer Systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*. 67–72.
[23] Google. 2017. Orinoco: young generation garbage collection. https://v8.dev/blog/orinoco-parallel-scavenger.
[24] Google. 2018. V8 Garbage Collector. https://github.com/thlorenz/v8-perf/blob/master/gc.md.
[25] Google. 2022. Ignition. https://v8.dev/docs/ignition.
[26] Google. 2022. Sparkplug – a non-optimizing JavaScript compiler. https://v8.dev/blog/sparkplug.
[27] Google. 2022. V8's public API. https://v8.dev/docs/api.
[28] Google. 2022. What is V8? https://v8.dev.

[29] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX Annual Technical Conference (ATC)*. 489–504.
[30] IBM. 2022. Kernel Storage-Protection Keys. https://www.ibm.com/docs/en/aix/7.1?topic=concepts-kernel-storage-protection-keys.
[31] Intel. 2000. Intel IA-64 Architecture Software Developer's Manual. http://refspecs.linux-foundation.org/IA64-softdevman-vol2.pdf.
[32] Intel. 2022. Memory Protection Keys. https://www.kernel.org/doc/html/latest/core-api/protection-keys.html.
[33] kashif. 2022. `node-cuda` provides NVIDIA CUDA bindings for Node.js. https://github.com/kashif/node-cuda.
[34] The Linux Kernel. 2023. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.
[35] keyhash. 2022. Cryptonight hashing functions for Node.js. https://github.com/keyhash/node-cryptonight-old-hardware.
[36] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking Down the Heap Between Safe and Unsafe Languages. In *European Conference on Computer Systems (EuroSys)*. 132–148.
[37] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea nd R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–163.
[38] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–37.
[39] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *ACM Conference on Computer and Communications Security (CCS)*. 280–291.
[40] Jonas Magazinius, Daniel Hedin, and Andrei Sabelfeld. 2014. Architectures for Inlining Security Monitors in Web applications. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*. 141–160.
[41] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-language Attacks. In *Network and Distributed System Security Symposium (NDSS)*.
[42] MITRE. 2020. CVE-2020-28248. https://nvd.nist.gov/vuln/detail/CVE-2020-28248.
[43] Onur Mutlu and Jeremie S Kim. 2019. Rowhammer: A Retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39, 8 (2019), 1555–1571.
[44] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 245–258.
[45] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *International Symposium on Memory Management (ISMM)*. 31–40.
[46] Node.js. 2022. Native Abstractions for Node.js. https://github.com/nodejs/nan.
[47] Node.js. 2022. What is Node-API? https://nodejs.github.io/node-addon-examples/about/what/.
[48] ohmu. 2022. The missing POSIX system calls for Node. https://github.com/ohmu/node-posix.
[49] Onarlioglu, Kaan and Bilge, Leyla and Lanzi, Andrea and Balzarotti, Davide and Kirda, Engin. 2010. G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries. In *Annual Computer Security Applications Conference (ACSAC)*. 49–58.
[50] Aleph One. 1996. Smashing The Stack For Fun And Profit. *Phrack Magazine* 7, 49 (1996).
[51] OpenBSD. 2003. i386 W^X. https://marc.info/?l=openbsd-misc&m=105056000801065.
[52] openJS Foundation. 2009. Node.js. https://nodejs.org/en/.
[53] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX Annual Technical Conference (ATC)*. 241–254.
[54] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *European Conference on Computer Systems (EuroSys)*. 420–436.
[55] Prior99. 2022. Unofficial bindings for node to `libpng`. https://github.com/Prior99/node-libpng.
[56] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*. 563–577.
[57] Red Hat Blog – Huzaifa Sidhpurwala. 2018. Security Technologies: FORTIFY_SOURCE. https://www.redhat.com/en/blog/security-technologies-fortifysource.
[58] Red Hat Blog – Huzaifa Sidhpurwala. 2019. Security Technologies: RELRO. https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro.
[59] Jonathan Salwan. 2015. ROPgadget Tool. https://github.com/JonathanSalwan/ROPgadget.
[60] Sascha Schirra. 2022. Ropper. https://github.com/sashs/Ropper.
[61] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium (SEC)*. 936–952.
[62] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *USENIX Security Symposium (SEC)*. 1677–1694.
[63] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*. 552–561.
[64] Snyk. 2021. Vulnerability Database. https://snyk.io/vuln?type=npm.
[65] Snyk. 2022. `node-sass` vulnerabilities. https://security.snyk.io/package/npm/node-sass.
[66] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*. 48–62.
[67] Mike Ter Louw, Phu H Phung, Rohini Krishnamurti, and Venkat N Venkatakrishnan. 2013. SafeScript: JavaScript Transformation for Policy Enforcement. In *Nordic Conference on Secure IT Systems (NordSec)*. 67–83.
[68] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium (SEC)*. 1221–1238.
[69] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 86–106.
[70] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization.. In *NDSS*.
[71] Nikos Vasilakis, Grigoris Ntousakis, Veit Heller, and Martin C. Rinard. 2021. Efficient Module-Level Dynamic Analysis for Dynamic Languages with Module Recontextualization. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1202–1213.
[72] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In *ACM Conference on Computer and Communications Security (CCS)*. 1821–1838.
[73] Verdaccio. 2022. Lightweight private npm proxy registry built in Node.js. https://verdaccio.org/docs/what-is-verdaccio.
[74] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-based Sandboxing. In *European Conference on Computer Systems (EuroSys)*. 266–282.
[75] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-agnostic Binary Recompilation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 133–147.

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 latter is designed with aggressive GC in mind, and, to this end, is partitioned to multiple *(semi-)spaces*, which are object allocation arenas that host short- and long-lived objects, in a way that makes GC performant and effective [23]. The heap region also includes special *SLAB-like* [7] 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 (`ld.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 [47]. 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` x86 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

```
1  __attribute__ ((aligned(4096), pure))
2  void
3  wrap_node_api_func(void)
4  { asm ("movq 0xdeadcafe, %rax; jmpq *%rax"); }
5  ...
6  static __attribute__((constructor)) void
7  init_method(void)
8  {
9      ...
10     mprotect((void*) wrap_node_api_func, 4096, PROT_WRITE);
11     rewrite_loc = wrap_node_api_func;
12     *rewrite_loc = &node_api_function << 16 | 0xb848;
13     ...
14     mprotect((void*) wrap_node_api_func, 4096, PROT_EXEC);
15     write_got = native_module_address +
16                 node_api_func_got_entry;
17     *write_got = wrap_node_api_func;
18     ...
19 }
```

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 `mov` 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.

D MODIFICATIONS IN NODE.JS AND V8 API

```
unsigned
enable_access(void)
{
    unsigned previous_rights = pkey_get(node_memory_pkey);
    if (previous_rights == PKEY_DISABLE_ACCESS)
        pkey_set(node_memory_pkey, PKEY_ALLOW_ACCESS);
    return previous_rights;
}

void
restrict_access(unsigned previous_rights)
{
    if (previous_rights == PKEY_DISABLE_ACCESS)
        pkey_set(node_memory_pkey, PKEY_DISABLE_ACCESS);
}
```

Each of the 122 entry points of Node.js and V8 where modified in order to remove memory restrictions uppon 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.