BinWrap: Hybrid Protection Against Native Node.js Addons

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

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

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

Sotiris Ioannidis
TU Crete
Chania, 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, bypassing all the safety and security guarantees offered by the high-level language and associated runtime.

We perform a study across on the Node.js ecosystem to understand the use patterns of binary add-ons in Node.js applications. Taking these trends into account, we propose a new hybrid permission model aimed at protecting both a binary add-on core 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 any accesses to sensitive resources from any part of the library. A pair of program analysis components infer the add-on's permission set automatically, over both its binary and JavaScript sides. Applied to a wide variety of native addons, we show that our framework reduces access to sensitive resources, defends against real-world exploits, and imposes a modest overhead that ranges between 0.71%–10.40% (avg.:3.17%).

1 INTRODUCTION

Modern software development relies heavily on third-party libraries. The heavy use of libraries is particularly common in JavaScript applications [27, 32], and especially in those running on the Node.js platform [57], where developers have easy access to hundreds of thousands of libraries through the node package manager (npm). The vast majority of the libraries imported in a Node.js application are implemented in JavaScript and thus enjoy the memory and type safety guarantees provided by a high-level programming language and enforced by its runtime environment—at times, augmented with language-based protection techniques [1, 11, 28, 50, 52, 53].

Often, however, Node.js applications also import a few libraries written in low-level languages or provided only in binary form. These libraries, termed *native add-ons*, implement either functionality not available yet in the pure-JavaScript 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 JavaScript layer wrapping the library enough to expose Node-specific naming and calling conventions.

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Unfortunately, native add-ons are particularly dangerous to the rest of the application—for example, over 20 CVEs are reported for a single string-interpolation add-on [47]. The complete lack of memory safety means that even a single line of memory-unsafe code may compromise the application's safety and security—that is, *including* those of the (safe and secure) majority of the codebase. Native add-ons can additionally bypass the security guarantees provided by the aforementioned language-based hardening and protection techniques. The exploitation risks of native add-ons compound, as these components are more likely to be targeted by malicious adversaries—exactly because of their vastly higher insecurity and potential impact.

In this work, we develop BinWrap a hybrid language-binary framework for protecting against the few native addons present in modern Node.js applications. We develop a fine-grained read-write permission model applied at the boundaries of native addons, offering a unified view and isolation of privilege cutting across the barrier between the language wrapper and the binary core. Two enforcement components enforce these permissions across language-binary barrier during the execution of the program, protecting both sides of a native addon: (1) language-level interposition protects against unauthorized use of the language-level bindings, and (2) binary-level indirection wraps the entire library and checks permissions to outside interfaces. To aid developers, a pair of program analysis components infer permissions automatically over both the binary and JavaScript sides of a native addon. Combined, the permission model and associated analysis aim at reducing the risk of native addons while maintaining practical performance and automation characteristics to enable adoption.

The evaluation of our framework proves that BinWrap can effectively protect *real-world* applications when real-world vulnerabilities are exploited within the loaded native modules. BinWrap is an efficient solution, imposing 3.17% performance impact on average. BinWrap is a scalable and practical solution since our design choices were directly influenced by the NPM ecosystem `norms`.

2 BACKGROUND

This section presents background information on Node.js runtime, V8 JavaScript engine (§2.1), on restricting memory accesses (§2.2), on code reuse attack prevention (§2.3), and on system call restrictions (§2.4).

2.1 Node.js, V8, and NAN

Node.js is a runtime environment for executing JS code, outside the context of a web browser, which primarily targets server-side applications [38]. Internally, Node.js leverages the V8 JS engine [17]. V8 parses JS code and converts that 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 [15]; moreover, JS code (in AST or bytecode form) can be compiled to *machine code* using the (optimizing) TurboFan or (non-optimizing) SparkPlug compiler [16].

Node.js is built around V8 (both are implemented in C/C++) and provides a rich set of APIs (i.e., the Node-API [34], but also, among others, the V8 API [18]) to JS applications. Most importantly, however, Node.js allows JS programs to load native *addons* (modules written in C, C++, ASM, *etc.*), Typically, JS applications leverage addons to: (1) perform compute-intensive tasks using highly-optimized C, C++, or even handwritten-ASM code [24]; (2) have access to other (dynamically-loaded) system libraries [40]; (3) interact freely with the underlying OS kernel via the system call (syscall) interface, and utilize system services for which JS abstractions are not available [35]; or even (4) perform computations on specialized hardware, like GPUs and TPUs [23].

2.2 Restricting memory accesses

We restrict the memory view of the native module code by leveraging Intel's Memory Protection Keys (MPK) [22]. BinWrap offloads the execution of native functions to a different thread. By leveraging Intel MPK we are able to restrict the memory access rights of the thread responsible for executing native functions. Thus, the exploitation of a memory vulnerability in the native module will not affect the rest of the application.

2.3 Code reuse attack prevention

Native modules need to execute Node.js and V8 API functions for benign reasons (e.g., JavaScript object allocation). During API invocations, we need to re-enable memory accesses in the API's function prologue and disable them upon return. Since we can operate on source code level, we ensure that there no occurrences of instructions that can reinstate memory accesses in the native module. However, an attacker could launch a Code Reuse Attack (CRA) that first re-enables memory access and then copies data to an accessible area. To prevent these attacks, we again utilize MPK and also rely on Address Space Randomization Layout (ASLR) to `hide` the location of the trusted code (Node.js) from the untrusted part (native module). The key idea of this technique is that the native module can call Node API functions without ever knowing their real address. In BinWrap$_B$ we link the native module with a wrapper library that interposes each API function required by the native module. The interposition functions call the actual API functions; however, the memory region they reside is configured as execute-only and thus, the actual address of the API functions is not visible from the native module thread.

2.4 System call restrictions

Native modules also execute several system calls for benign reasons. An attacker can misuse these system calls to bypass BinWrap$_B$ sandbox. In BinWrap$_B$, we deploy two techniques to restrict system call execution. The first technique relies on finding the actual set of system calls required for the execution of the native module. To extract this set, we deploy Sysfilter [12]. Sysfilter is a static binary analysis framework that extracts an application's set of system calls. The enforcement submodule of Sysfilter produces a BPF filter that can be used with seccomp-bpf [**?**] to deny the execution of system calls not present in the set.

Seccomp is a kernel mechanism that can restrict the system calls an application can execute. Since version 3.5 Linux kernel supports SECure COMPuting with Berkeley Packet Filter (seccomp-BPF). The filter rules, allow or deny system calls based on system call numbers and arguments. The applied filter can only be replaced by a more restrictive filter and cannot be removed. The filters applied are per-thread, and thus we can restrict system calls only to the native module thread. However, seccomb-bpf cannot dereference pointer arguments.

Our second technique aims to restrict system calls that attackers can misuse and are present in the system call set required by the native module. For these cases we remove any implicit and explicit `syscall` and `sysenter` instructions from the native module code and we only allow system calls through `libc` library. Finally, we interpose `libc` functions that wrap systen calls required by native modules in order to filter their arguments.

3 BINWRAP OVERVIEW

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

3.1 `png-img`: A Node.js Portable Network 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, addon 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 (e.g., a "buffer overflow").

Such errors are a common attack vector when code written in memory- and type-unsafe languages, like C, C++, Objective-C, and assembly (ASM) [54], 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* [30] or *temporal* [31] 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 to link a native function (written in C, C++, *etc.*) with the PngImgImpl object (ln. 1–2). Every time the png-img constructor in invoked, the buffer object, which contains the raw (image) data, is passed to the native function (ln. 4–7).

Since the addon is written in a memory- and type-unsafe language, it may contain bugs (e.g., a buffer overflow, ln. 6) that trigger memory errors [54]. More importantly, given that the raw image data are of *unknown provenance*, attackers may provide specially-crafted inputs that *exploit* the underlying memory errors, potentially resulting in *arbitrary memory read* (disclosure, leak) and *arbitrary memory write* ("write-what-where") primitives [41].

In real-world settings, attackers primarily aim for tampering-with *control data* (e.g., return addresses, function pointers, dynamic dispatch tables) [26], as these facilitate *hijacking the control flow* of the program and performing *arbitrary code execution* [37]—typically via means of *code reuse* [8]: 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 proposed and developed thus far [6, 9], enabling access control and policy enforcement bypasses, privilege elevation, and sensitive data leakage [48].

Considering these facts, we found a relevant vulnerability of png-img documented in National Vulnerability Database [29].

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

The vulnerability here is that height and rowbytes variables are 32-bit integers and thus can be overflowed. An attacker could trigger this overflow in order to cause an inadequately sized memory allocation. Subsequently, image data will overwrite adjacent memory regions. As we discuss in 8, this arbitrary memory write can be levelaged by an attacker in order to take over the control of the application.

3.2 Node.js Module Confinement with BinWrap

To harden the respective Node.js application 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 analyses, and policy enforcement, at the level of native, binary code (BinWrap$_B$), as well on JS code (BinWrap$_L$). (The latter typically wraps the addon code and provides a high-level API for interfacing with Node.js-based application code.) We envision BinWrap$_B$ as the set of binary-focused methods and techniques that is to be applied during library installation time, whereas BinWrap$_L$ is the {load, run}-time counterpart, targeting the JS wrapper 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 the side effects, of unsafe addon code in *part(s)* of the corresponding virtual address space (VAS). More specifically, the execution of native addon 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 (Memory Protection Keys) technology [22]. The benefits of this *intra-VAS isolation* are twofold: first, memory errors in png-img's native code *cannot* be used to tamper-with the 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 functionality that exists in the code of Node.js itself or that of any other library 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 system calls (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 APIs, 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. Note that 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. Lastly, our techniques are *complete* and have minimal requirements (i.e., access to symbols) for increased precision.

BinWrap$_L$ consists of both a static and a dynamic part. By running the static analyzer on the JS wrapper code of png-img (i.e., index.js), at load-time, we get the following (RWX-like) 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 access map, 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 further *locking* the interaction of `png-img` with the Node.js application that uses it.

4 THREAT MODEL

Our threat model assumes that the adversarial capabilities allow the exploitation of memory and type unsafety of *benign* native modules. An attacker therefore, can leverage memory corruption vulnerabilities in order to develop arbitrary memory read and write primitives. The exploitation of these vulnerabilities can be used in order to access sensitive data and even perform Code Reuse Attacks. 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 restricted through existing languages based mechanisms e.g., MIR. Node.js runtime, as well as system libraries are considered trusted and free of vulnerabilities. Finally, we consider side-channel attacks and hardware faults as out of scope.

In order for BinWrap to protect Node.js runtime environment from these cases, the following OS and hardware features are required. The OS must include Seccomp BPF in order to enable system call filtering. We also consider that $W\hat{X}$ 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). Our framework does not interfere with any other possibly deployed security mechanisms e.g., AppArmor, RELRO, *etc.*. Rather, thse mechanisms can further enhance the protection offered by BinWrap. The hardware must include Memory Protection Keys or a mechanism that offers equivalent capabilities. While our required hardware feature cannot be considered as standard as our OS prerequisites, it is part of latest Intel's server CPU series. Moreover, MPK functionallity can be emulated through memory tagging which is widely available in ARMs latest processor series [4].

Our techniques aim to address three challenges: (i) prevent the native module thread from accessing memory outside of the native module's loaded address range and its heap-allocated memory, (ii) prevent the native module thread from executing CRA gadgets outside of the native module's `.text` area, and (iii) prevent the native thread from misusing system calls.

We consider the Node.js JavaScript runtime environment and our customized native module layer (NAN) as the trusted part of the application. We consider the source code of the native module as the untrusted part of the application. We deploy many techniques to ensure that any exploitation attempts targeting the native module code will be confined within its bounds and will not affect the whole application.

5 LANGUAGE SAFETY TRENDS IN NODE

As of today, the NPM ecosystem contains more than 1.5 million packages, downloaded more than 151 billion times just in the last month. This quantity and popularity of packages makes the NPM ecosystem ideal for attackers to target and exploit. Since on this paper we focus on native libraries found in the entire NPM ecosystem, we investigate the following questions:

- **Q1** What is the ratio of packages that use one or more native modules on NPM? (§5.1)
- **Q2** What is the ratio of native modules that libraries on NPM import? (§5.2)
- **Q3** What are the most popular native modules used by NPM packages? (§5.3)

This section presents (i) an overview of the experimental setup used to perform the analysis and (ii) a detailed analysis of native modules on the NPM ecosystem.

Implementation The entire NPM registry was cloned on a local server to execute the analysis. We cloned the database to perform the essential queries and data extraction. The NPM registry uses a Couch-DB [2] database under the hood, storing the information on JSON fields. We used the replication mechanism of Couch-DB to download the entire registry locally. After we downloaded our local copy of the registry, we made the necessary configurations to access it. Lastly, we use a proxy registry called Verdaccio [55] to access the private registry and query packages from there.

At the moment of the registry replication, the NPM registry contained 1.508.366 libraries. We used this number as the starting point of the analysis 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 to add native abstractions to their programs, namely NAN.

5.1 Number of native modules (Q1)

We found that of the 1,508,366 libraries, 63,381 of them had at least one dependency related to NAN or Node-API. A library is dependent on a native module when it includes the NAN or Node-API module directly or indirectly on 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 Node-API. From the sum of the packages, 45,708 packages depended on NAN, 23,239 packages depended on Node-API, 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 use ten native dependencies.

5.2 Ratio of native modules (Q2)

To improve the development process, developers include on their applications third-party libraries. The third-party libraries, include on some cases a combination of native library dependencies and ordinary JavaScript dependencies. In this section we answer the native modules' ratio against the rest of the dependencies.

The packages that use either NAN or NAPI as native modules have an average of 11 total dependencies. Among those dependencies, 0,95% on average are NAN dependencies, and 0,65% are NAPI dependencies. The average ratio of NAN dependencies against the total dependencies is 24,22%. The average ratio of NAPI dependencies against the total dependencies is 11,27%. We conclude that there is only one native package per module in most of NPM packages. Due to the small number of native modules, the application's attack surface that we need to defend is small. Also since, we need

to defend against a small attack surface, we expect the overhead imposed by our tools to be minimal.

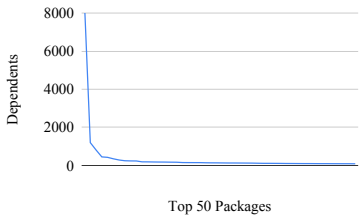
5.3 Popularity of native modules (Q3)

Packages that use native modules comprise 4.2% of the total NPM ecosystem. This 4.2% of packages 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 dependants. For a package to depend on another, it must be included in the dependency list. By evaluating the number of dependants, we can assess a vulnerability's impact when it appears on a native package.

For the NAN 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 is one, and the maximum number of dependents on a package is 8,457. A total number of 58,778 packages are dependent on the NAN native module.

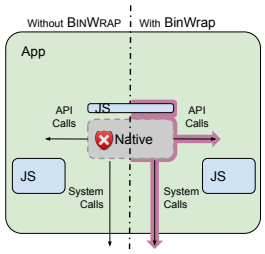
On the day we conducted the analysis, the package with the most dependencies was `node-sass` with over 8k dependents. The least popular package in the top 50 list was `ledgerhq` with 90 dependents. Meanwhile, only the top 11 packages had more than 500 dependents. Thus, we can safely conclude that our evaluation set is an ample representation of third party libraries in the NPM ecosystem.

For the NAPI native modules, 5,762 packages had dependents. The average number of dependents per packet was 6.6%. The minimum number of dependents on a package is one, and the maximum number of dependents on a package is 2,322. A total number of 38,383 packages are dependent on the NAPI native module. As we can see, 97,161 packages are dependent on native modules, thus any security incidence will affect a significant part of the NPM ecosystem.



6 DESIGN

The key idea behind the design of BinWrap is to separate the runtime execution of the of the untrusted component from the rest of the application. Runtime separation is achieved using different execution threads for the two trust domains while isolating the thread responsible for executing the untrusted component—effectively, limiting its memory visibility and system-call execution capabilities. BinWrap limits the memory visibility of the untrusted component by creating a dedicated memory view for the untrusted thread. Also it limits access to the system calls available to the untrusted component, by wrapping and filtering system calls in the untrusted thread.



This section describes BinWrap's mechanism for isolating the execution of untruste natived components. Fig. 6 presents BinWrap's wrapping. BinWrap first compiles the source code of the untrusted component and then statically analyzes the resulting binary —a `.so` shared object.We prefer analyzing over the source code since we can find Analyzing the shared object instead of the source code the complete set of external symbols required, for instance calling `printf` will also execute other `libc` functions e.g.,write, *etc.* This analysis aims at extracting (1) the full set of system calls 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.*.

BinWrap then creates a custom instance of a Node.js API *layering* library—loaded during the initialization of the native component. This BinWrap-infused library sets up appropriate seccomp filters for the set of system calls extracted in the previous step. BinWrap then recompiles the native component, linking against the library instance, thus injecting the filter into the native component.

6.1 Isolation techniques

Native function execution Native modules utilize the Native Abstractions for Node (NAN) package to wrap native functions. Native functions are invoked through callback info objects. In BinWrap we handle these callback info objects to the restricted thread. This thread is initialized during the first time a native function is executed. The Node.js process thread that dispatched the callback info object to the native function thread will block until the native function returns. After the native function returns, the main thread will be unblocked. We used a shared variable as the synchronisation primitive between the two threads. The separate threads design enables BinWrap to leverage thread specific mechanisms (i.e., MPK and seccomp) in order to isolate the execution of native libraries.

Data access filtering BinWrap restricts the memory accesses of third party libraries in Node.js. Since we decouple the execution of untrusted code by creating a new thread, we can prohibit arbitrary accesses o sensitive data stored in Node's memory by leveraging Intel's Memory Protection Keys (MPK) technology [22]. During the initialization of the native module thread, we associate a protection key with the pages that may contain sensitive data. These data include all the memory allocated and managed by the V8 JavaScript engine. The native module thread will initially change the rights associated to `no access` on the protection key associated with Node's allocated pages. The native module address ranges and allocated memory is excluded by this set. Subsequent memory allocations for expanding V8 memory pool are also associated with Node's protection key. Finally, we ensured that there is no explicit data sharing between Node and native modules (e.g., globals) by analyzing the symbol table of the top 500 popular native module.

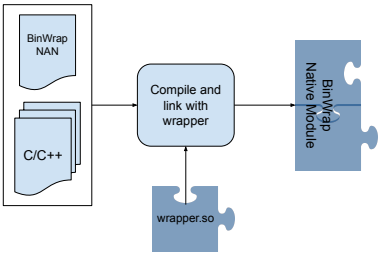
An obvious limitation of protection keys is that only 16 are available and thus only 16 different *views* on memory can be supported. This issue has been addressed by the literature through virtualizing

memory protection keys. In libmpk [39] the authors implement a software abstraction for MPK that virtualises the hardware protection keys. Another solution is grouping sets of native modules under the same protection key. In this approach however, a vulnerable module could also affect the other modules in the set. Finally, as we presented in section 5, we encountered at most 10 imported native modules by a library in the entire NPM ecosystem.

Node.js and V8 export a large API set of functions in order to allow native modules to perform various tasks (e.g., object allocation and management, type conversions, *etc.*). Since Node.js is part of the trusted domain when the native thread executes Node API functions, the access rights on Node's data should be re-enabled. In our framework, we modified the API functions to change the rights for the protection key to allow memory operations. We reimpose restrictions before an API function returns back to the native module. During every API call we store the previous contents of `pkru` register in API's function stack variable. During returns, the `pkru` will be modified only if the previous `rights` stored in the stack restrict memory accesses (execution returns to native module). This design choice stems from the fact that API calls may be nested.

An attacker could access sensitive data by harvesting stale data in deallocated stack frames after API functions execution. During the execution of the trusted part, the thread can access any data, and Node API functions can copy data in the stack (as function arguments, *etc.*). After the function returns, the residual data can be accessed without restrictions. To prevent this, the native execution thread zeroes out the deallocated stack frames before returning back to the native module, effectively deleting residual data.

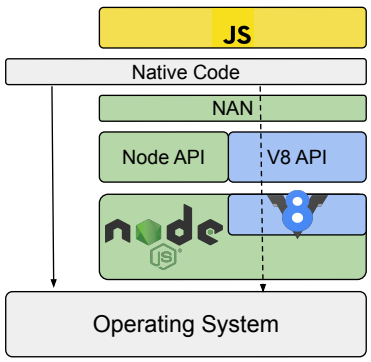
Preventing Code Reuse An attacker could launch a code reuse attack targeting instruction snippets that remove the restrictions i.e., `wrpkru` and fetch data to memory areas accessible by the native module. These instruction sequences are present in the trusted code since the data access restrictions must be lifted during the execution of the trusted part. These instructions can also be implicitly present in the untrusted part since x86 instructions have variable length. Moreover, `xrstor` instruction can be leveraged to restore a crafted register state with modified `pkru` in order to allow access on restricted memory. In order to prevent an attacker from using the unlocking functions, we again utilize MPK and also rely on Address Space Layout Randomization (ASLR) to `hide` the location of the trusted code (Node.js) from the untrusted part (native module). The key idea of this technique is that the native module can call Node.js API functions without ever knowing the address of these functions.



We designed a custom linking procedure to hide Node API and library locations from the untrusted part. Initially, we extract all the API and library functions needed by the native module from the module's shared object `.plt` section, along with the offset in the `.got` section. Then, we create a new *wrapper* shared object which contains a wrapper function for every Node.js API and library function required by the native module. We link the wrapper library to the native module and manually resolve the native modules dynamic symbols to point at the wrappers (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 finally, ASLR also ensures that the address of Node.js and linked libraries is different on separate executions. Fig. 6.1 presents a high-level overview of BinWrap modules compilation procedure. Finally, the wrapper library implements dynamic symbol interposition to filter system calls that can bypass the native module sandbox if misused.

To prevent attacks targeting `implicit xrstor` and `wrpkru` instructions, we scan the binary with ROPgadget tool [42]. We vet any occurence of these instructions in a similar manner as G-free [36]. Since we can also operate on the source level of the native module, we do not only rely on Static Binary Instrumentation (SBI) to vet unsafe instructions, rather we can transform the source code to prevent unsafe instructions from being emitted in the final binary. Our analysis on the top 500 native modules with ROPgadget tool found no implicit occurrences of `xrstor` and `wrpkru` instructions. The chances of implicit occurrence are low since both of these instructions are more 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.

6.2 System call extraction



Native modules often depend on system calls for key functionality available by the operating system. There are two avenues native modules issue system calls. First, system calls are by default available directly to the native component—e.g., component calls `mmap` for mapping memory. Second, they are available indirectly, through Node.js-internal APIs that the component uses—e.g., component calls `v8::External::New(v8::Isolate*, void*)` from Node core, which internally calls `brk` system call. Since native components make extensive use of Node.js-internal APIs, the resulting combined set of system calls used by the native component may be large. Both of these classes can be used as a means for an attacker to cross the protection boundary, effectively bypassing BinWrap's enforcement mechanism [10, 56].

To extract the full set of system calls a native module requires, we use an intra-procedural, flow-sensitive binary analysis. We analyse each native module in order to extract both the directly included system calls, as well as the system calls inherited through V8 and Node.js API functions (dashed arrow) Fig 6.2.

Direct System Calls The analysis first receives as input the shared native component, i.e., a `.node` file. It proceeds to resolve dependencies to shared libraries, and then (over-)approximates function-call

graph (FCG) of the native component. This approximation is constructed over all objects in the scope of the component and its dependencies. The analysis then performs a set of analyses atop the FCG to extract a tight (but safe) set of developer-intended syscalls.

Inherited System Calls To identify the Node.js-internal API functions used by the native component, we first analyse the native module's shared object symbol table. We then uses each function symbol as entry point for analyzing the Node.js executable and identify the reachable system calls. The analysis trades soundness for completeness, in that system calls performed by the native component will exist in the extracted set—but not all extracted system calls are expected to be used in every (or, indeed, any) execution of the native component. The set of extracted calls includes system calls directly used by the native component, calls used by Node.js-internal APIs during that execution, system calls in `libc`, and any other shared libraries loaded dynamically.

6.3 System call restriction

BinWrap uses the extracted system-call set to create a filter containing the complete set of system calls that may be executed by the native thread. Given a set of allowed system-call numbers, BinWrap's enforcement tool first converts them to a BPF program and then uses `seccomp-BPF` to execute, and thus enforce, this filter during the execution of the unsafe thread. The core of the BPF filter's logic is centered around conditionals that check the system call number and non-pointer system call arguments. To inject the filter into the binary program, BinWrap leverages BinWrap-specific Node.js API templating. BinWrap provides custom templates of Node.js API libraries that contain placeholder segments instantiated with custom filter instances. Different Node.js API custom wrappers—e.g., NAN, N-API *etc.*—correspond to different templates. On the other hand system calls requiring pointer argument filtering (described in 7.1) are interposed in the library wrapper object through their respective `libc` function.

BinWrap then instantiates each template (still as source code) using (1) information extracted from the earlier static-analysis phase, and (2) additional hardcoded policies for system calls than can be potentially misused. It then compiles the native component, linking against the Node.js API instance, that contains the `seccomp-BPF` filter corresponding to this native component and the Node.js, V8 and dynamic library interposition wrapper object. 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 a Node.js application. The JavaScript code of the third party library is analyzed with BinWrap$_L$ to extract the permission model that will be enforced at runtime. Our NAN modifications add 200 lines of code for initializing the native execution thread, the synchronization (i.e., the execution of native functions) and the seccomp-bpf configuration. We chose NAN due to BinWrap$_L$ compatibility, however our implementation is directly applicable to N-API as well as Node.js.

Our wrapper library is generated using bash scripts and ranges between 240 and 600 lines of code depending on how many V8 and Node.js symbols are dynamically linked to the native library.

Node and V8 API modifications We modified any V8 and Node API function reachable through NAN API. We identified the full set of these functions by analysing the test binaries shipped with NAN package. Our analysis discovered 122 dynamic symbols that point to V8 and Node.js. We additionally analysed the native modules that consist our evaluation set and found that they link less than half of these functions i.e., around 50 symbols. The modifications are functions that remove memory restrictions upon entry and reimpose then before the API function returns to the native module.

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

Wrapper library Wrapper libraries are generated using bash scripts and ranges between 240 and 600 lines of code depending on how many V8 and Node.js symbols are dynamically linked to the native library. The wrapper functions are pure (i.e. they do not create a stack frame) and consist of two instructions implementing an indirect jump. `mv` and `jmp`. System calls with pointer aarguments 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 module is loaded. Each pure wrapper is patched by the constructor in order to store the wrapped symbol's address in the auxiliary register, which will be dereferenced during the indirect jump instruction. The native modules `.got` is configured to point at the wrapper functions. Finally, the constructor maps the wrappers as execute only. This will cause the allocation of a new memory protection key associated with the pages containing the wrapper functions.

7.1 System Call policies

Several system calls can be leveraged to bypass MPK based restrictions [44, 56] ***Memory management*** System calls that are used for memory management, like `mmap`, `mprotect`, `munmap`, `brk`, or `mremap`, could be used to allocate executable memory, execute `pkey_set` instructions or move data to memory locations that can be accessed. Protection keys can also be wiped by de-allocating and re-allocating the target memory region. 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 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 [44].

Process and thread control Another family of dangerous system calls is related to process creation (`fork`, `exec`, and `clone`). These system calls create a new process that is either new (exec system calls) or executed in the same address space. In `clone` the MPK configuration is inherited to the new thread, and is thus safe. `Fork` system call however, can be combined with `kill` system call in order to force the child process to produce a core dump which can be read by the untrusted domain. We found that `fork` and `exec` are not required by native modules and deny their execution. We also disable core dumps by configuring the application process with `(PR_SET_DUMPABLE, SUID_DUMP_DISABLE)`. This kernel utility also prohibits access to `procfs` and thus prevents the

misuse of file-related system calls. We also safely 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 stack. When the signal handler finishes its execution, the register state is restored through `rt_sigreturn` system call. An attacker can craft a register state where `pkru` register allows memory access and execute a sigreturn gadget in order to obtain universal access. 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 the `.text` area. In BinWrap, we treat such instructions as unsafe, and we vet them. Thus, there is no way for an attacker to call `rt_sigreturn`.

8 EVALUATION

To evaluate BinWrap, we use a set of real-world native npm packages, investigating the following questions:

- **Q1** How effective is BinWrap at defending against attacks that exploit real-world vulnerabilities? (§8.1)
- **Q2** How much BinWrap reduces the set of system calls in the context of native modules, what is the set breakdown? (§8.2)
- **Q3** How efficient and scalable are each of BinWrap components? (§8.3)

Libraries and workloads We evaluated each of BinWrap components, testing the security guarantees and the runtime overhead imposed. We investigate each one of the topics, answering a set of relevant questions. To address Q1, we evaluate BinWrap against exploits targeting vulnerabilities in popular third-party libraries. The vulnerabilities were pulled from the Snyk [46] vulnerability database, and exploited by us. We found that BinWrap successfully prevents the exploitation of the selected vulnerable packages. To address Q2 and Q3, 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 JavaScript applications that utilise third-party native libraries, while imposing an average of 3.17% runtime overhead.

To benchmark each package and application, we tried to execute the test suite that was provided by the developers. If the application developer did not provide any test suite, we used the example code provided in the repository for our evaluation. All of the third-party libraries used in the evaluation ship with npm based test suites. We used two sets of benchmarks during our evaluation. The first and major set consists of benchmarks that implement the behaviour of an actual application that uses the third party library (npm packages and applications), i.e., executing mostly JavaScript code and offloading heavy computations on the native module. The second set consist of benchmarks that execute native functions in tight loops, thus stressing the cross-domain transition of our framework. The first set is suitable for presenting the performance impact of BinWrap in real-world applications, while the second is suitable for measuring the micro aspects of the runtime overhead.

Setup Our system is configured with an Intel Core i9-10900 CPU with 32GB RAM and runs Linux version 5.4.0-84. We implemented our modifications on Node version 8.9.4. BinWrap does not require any kernel modifications to run and only needs MPK and seccomp-bpf to be available on the system. Since we run our benchmarks in the latest Ubuntu distribution, we had to recompile each library that Node.js loads dynamically to remove Intel CET instrumentation, which is added by default in most packages. Intel CET uses a customized `.plt` section that is not supported by sysfilter. Disabling CET does not affect the security of the system, since hardware support is not available in our system's processor and CET instructions are treated as NOPs. Finally, we disabled cstates and Intel turbo boost and locked the clock frequency at 2.8GHz.

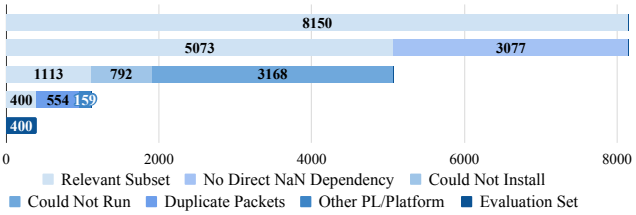


Figure 1: Overview of the analysis for the evaluation set.

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. The applications have (i) large number of dependents, (ii) compatibility with our tool, and (iii) security exigency. It is also of critical importance that the npm package is not corrupted and can run successfully in the base-line case (i.e., unmodified, without BinWrap). To analyze the applications on NPM, we used the local NPM registry from our study (§5). Our study in section (§5) highlights how critical native applications are to the developers. The local NPM registry that we used as a base contained 1,508,366 libraries.

We took multiple steps to get from the 1,508,366 libraries to the 20 in the evaluation set. First, we found all the libraries that have NAN as a dependency, i.e., 5.073 packages. We only consider packages that can be installed without manual effort (4.201 packages) while we removed remove 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 our criteria.

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 (i.e., tightloops executing third-party library functions). 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 with regards to performance in real-world scenarios. With regards to `node-fs-ext, syncrunner, node-delta and mtrace` where chosed to diversify the different types of libraries in our set (i.e., we could choose more popular parser packages, but this would result in a set of libraries with similar behaviour). Finally, `png-img` is also used in our security evaluation, since it contained known vulnerabilities.

Table 1: Third party libraries available in npm and applications using these libraries. * C lines of code. ** Number of System Calls

Name	Description	Dependents	CLoC*	NoSC**	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	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 using picha	na	7522	92	Application
Manta Minnow	Storage utilization agent for Manta project, uses statvfs	na	6463	91	Application

8.1 Security evaluation (Q1)

To assess the security of BinWrap we implemented exploits for four distinct CVEs. We analyzed vulnerabilities reported in Snyk.io [46] database for npm packages. These vulnerabilities occur from memory bugs in the shared object that ships with npm packages. To evaluate the security of BinWrap, we exploited these vulnerabilities and bypass the boundaries of the untrusted part.

```
1  std::string exp_src = exp->to_string (ctx.c_options);
2  Selector_List_Obj sel = Parser::parse_selector
3                          (exp_src.c_str(), ctx, traces);
4  parsedSelectors.push_back(sel);
```

CVE-2018-11499 Is an information disclosure attack that exploits a use-after-free vulnerability. The vulnerability was present in node-sass package until version 3.5.5. The use-after-free occurs due to lack of exception safety in a loop. In the above code example, exp_src is allocated in the stack, while parse_selector stores a pointer in a memory buffer. If the parse_selector throws an exception, the stack is unwinded and the exp_src buffer is deallocated. During the construction of the exception message, the exception handler de-references the freed memory region. Our exploit manages to leak pointers to heap addresses, which can be used to perform arbitrary reads. With BinWrap, any attempt to read beyond the memory allocated for libsass will fail due to the restrictions imposed using memory protection keys.

CVE-2018-18577 Is a heap buffer overflow vulnerability enabling arbitrary writes. The exploit is present in libtiff that picha npm package, loads for processing tagged image file format files. The bug stems from the fact that, libtiff ignores the size of the destination buffer when decompressing JBIG compressed images. Thus, an attacker can write arbitrary amounts of decoded data in the destination buffer. With BinWrap, we are able to prevent this exploit from overwriting data that do not belong to the native module's memory address ranges. In this scenario, an attacker could replace sensitive data used by the JavaScript part of the application. Since these areas are not accessible by the thread executing libtiff's code, when a store instruction targets a memory address beyond its visibility, a memory violation exception will be raised.

CVE-2019-3822 Is a stack buffer overflow that can lead to the execution of a ROP gadget chain. The node-libcurl npm package links to libcurl library, which had a stack buffer overflow vulnerability from version 7.36.0 to 7.64.0. 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 leads to a stack buffer overflow. In our version, the response buffer was statically allocated with 1024 bytes. The response buffer is base64 decoded and also memcpy is used instead of strcpy (making it easier to exploit, since copies will not stop when zero is encountered). We used Ropper [43] to create a ROP chain that loads the command we want to pass to system libc function and execute it. This exploit does not work when BinWrap is deployed, since system function will end-up executing execl system call. In every native module we analysed with sysfilter, execl system call was not present in the system call set that the native module might need to execute (for benign reasons).

CVE-2020-28248 This vulnerability leads to an under-allocated buffer due to an integer overflow in a memory initialization function. The exploit is present in png-img npm package in all versions up to 3.1.0. This condition introduces a heap-based buffer overflow that can be exploited using a specially crafted png file.

```
void PngImg::InitStorage_() {
    rowPtrs_.resize(.height, nullptr);
    data_ = new png_byte[.height * .rowbytes];
    for(size_t i = 0; i < .height; ++i) {
        rowPtrs_[i] = data_ + i * .rowbytes;
    }
}
```

The `InitStorage` function height and `rowbytes` are 32-bit integers, thus this calculation can easily overflow and allocate an inadequately sized region. Data from the decoded image may then be written past the end of the buffer. Library `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. This vulnerability can be used to overwrite the callback In our exploit we overwrite the error callback with the address of `system` function. Similar to **CVE-2019-3822** this exploit will not work in BinWrap, due to system call filtering.

Take away: BinWrap can effectively defend Node.js applications from memory vulnerabilities present in native third party libraries.

8.2 System call set analysis (Q2)

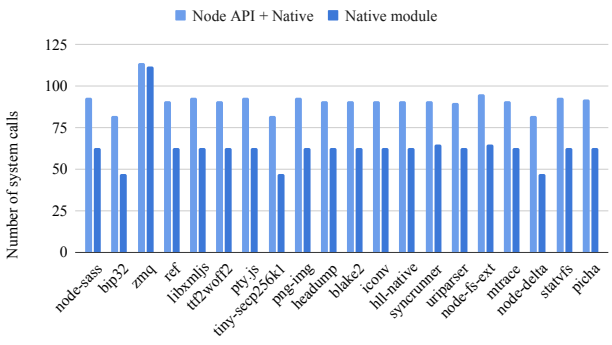


Figure 2: System call set size for the native module and the combined with the Node.js API.

We analysed 20 native modules with sysfilter to (i) extract the set of system calls required by the native module and to (ii) extract the system calls required through Node.js API functions. We found that Node API functions require the same 62 system calls in all native modules. In some native modules, like `node-sass` or `zeromq`, the system call sets also contained `inotify_rm_watch` and `epoll_ctl`. We found that the native module shared object requires mostly the same system calls as the Node.js API functions. The number of system calls inherited from Node.js range from 2 (zeroMQ) to 35 (node-delta). As we can see in Fig. 2 we can safely block more than 2/3 of the available system calls in most cases, except `zeroMQ` which requires 114 system calls due to handling sockets.

Take away: By deploying Sysfilter we can safely deny 2/3 of the available system calls and signifficantly reduce the attack surface.

8.3 Performance Evaluation (Q3)

In this section, we present the performance impact of BinWrap when enabled on third party libraries. We compare the runtime performance of BinWrap against the unmodified version of Node.js runtime environment. We breakdown BinWrap in 3 different parts to measure the performance overhead. The parts are the (i) native function sandbox, the (iii) dynamic analysis privilege checks, and the (iii) combined impact on performance.

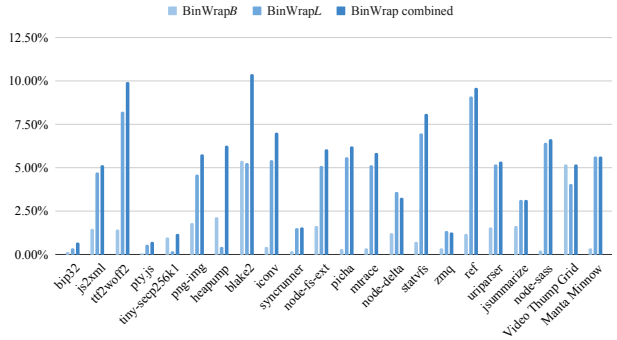
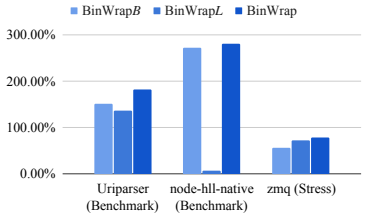


Figure 3: Runtime overhead when deploying BinWrap$_B$, BinWrap$_L$ and BinWrap.

Macro benchmarks We evaluated BinWrap by running the test suite provided by each npm package. We run `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 minimal overhead. The typical workload is execution of JavaScript 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. The overhead originating from the modifications in the native module's shared object is related to the synchronisation between the Node and the native module thread. Since `pkey_set` instructions are executed in userland, and thus changing the rights during domain switches imposes negligible overhead. Interposing system calls and Node.js API functions is also lightweight since the number of extra instructions executed due to Interposition is small.

Micro benchmarks During domain transitions, the native module thread is unlocked and executes the native function. The main thread 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 for incrementing 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 for incrementing the variable unlocks the synchronization variable. When utilising futex's the overhead compared to the benchmark without threads is 240x. When using inline assembly 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 the security mechanisms, we present the results in Fig. 8.3. In the case of `uriparser` the benchmark code is only two loops that parse a `URL` 2M times. The first loop uses the JavaScript implementation of the parser, while the latter uses the natively implemented parser. The majority of the code executed triggers the synchronization mechanism between the Node and the native module 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` benchmark, 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, however a lot of the overhead is amortised due to the execution of network related system calls.



Take away: BinWrap imposes a performance overhead of 3.17% on average.

9 RELATED WORK

Intra-process isolation operating systems focus on process isolation (virtual memory, etc.) to prevent process's from arbitrarily interrfering with between them. Intra-process isolation, is required in applications that need to isolate components in the same process. For example, web browsers isolate the execution of different pages in order to prevent malicious pages from accessing sensitive data. A notable family of Intra-process isolation techniques is Software Fault Isolation (SFI), SFI instruments memory operationa in order restrict memory access beyond a designated area. Other instrumentation approaches ensure that out-of-bound pointers are transformed into in-bound. Research efforts focus on in-process techniques, that offer isolation guarantees with minimum cost [5, 49].

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 Memory Protection Keys from Intel to differentiate access rights on memory when accessed from the trusted and untrusted parts of the Node.js applications. Several other research efforts, also leverage MPK for intra-process isolation. ERIM [51] and Hodor [19] 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 occurences of MPK instructions. Regarding system calls ERIM only intercepts memory management system calls, while Hodor denies any system call originating from untrusted domains by modifying the operating system. Unfortunatelly, recent publications [10, 56] present attacks on ERIM and Hodor, extended system call filtering with `ptrace` in ERIM solves some issues but incurs substantial overhead [44].

Donky [45] modifies a RISC-V processor, enabling protection keys and user-level interrupts. Domain transitions and memory management system calls are managed by a per-process monitor. Only the monitor has acccess to the protection key registers. Jenny [44] resolves several limitations of Donky, notably more complete system call filtering. However, Jenny and Donky cannot be directly applied in x86 and require custom hardware.

PKRU-Safe[25] polices inter-domain data flows in MPK based sandboxes. The authors do not address any of the security issues presented in [10, 56] (i.e., system call misuse, stray MPK instructions). Thus, 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 modules). Cerberus [56] aims to address the issues with MPK-based sandboxes presented in [10] paper and present 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. However, since the security monitor is implemented in the OS, it is not able to thwart `sigreturn` based attacks.

10 CONCLUSION

We presented BinWrap: a framework that applies acrosss the whole stack of Node.js applications in order to isolate the execution of potentially vulnerable third party applications. We studied the Node.js ecosysystem and understood how native add-ons are used. By identifying these trends we implemented a hybrid that wraps both the native and high-level language component of a library. Next, we evaluated the security our framework against real-world exploits in real-world applications. Finally, we evaluated in BinWrap in terms of performance and presented an average overhead of 3.17%. We believe that BinWrap is a practical framework that can protect Node.js applications in the presence of unsafe native libraries.

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 *Proceedings of the 28th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '12)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2420950.2420952
[2] Apache. 2022. *Couch-DB*. https://docs.couchdb.org/en/stable/
[3] ARM. 2018. *ARM memory 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. *ARM community* (2018).
[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*. Springer, Cham, 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] Bugtraq. [n. d.]. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63.
[9] 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.
[10] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. {PKU} Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1409–1426.
[11] Willem De Groef, Fabio Massacci, and Frank Piessens. 2014. NodeSentry: Least-privilege Library Integration for Server-side JavaScript. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA)

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 [10]	✗	✗	✗	✓	✓[4]	✗	✓
New PKU Pitfalls Protection [56]	✗	✗	✗	✓	✓	✗	✓
Performance Overhead	Low	Moderate	Low	Moderate	Low	Low	Low

(ACSAC '14). ACM, New York, NY, USA, 446–455. https://doi.org/10.1145/2664243.2664276

[12] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. 2020. Sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*. 459–474.

[13] Google. 2017. *Orinoco: young generation garbage collection*. Google. https://v8.dev/blog/orinoco-parallel-scavenger

[14] Google. 2018. *V8 Garbage Collector*. Google. https://github.com/thlorenz/v8-perf/blob/master/gc.md

[15] Google. 2022. *Ignition Interpreter*. Google. https://v8.dev/ignition

[16] Google. 2022. *Sparkplug JavaScript Compiler*. Google. https://v8.dev/blog/sparkplug

[17] Google. 2022. *V8 JavaScript Engine*. Google. https://v8.dev

[18] Google. 2022. *V8's public API*. Google. https://v8.dev/docs/api

[19] 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 *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 489–504.

[20] IBM. 2022. *Kernel Storage Protection Keys*. https://www.ibm.com/docs/en/aix/7.1?topic=concepts-kernel-storage-protection-keys

[21] Intel. 2000. *IA64 software development manual*. http://refspecs.linux-foundation.org/IA64-softdevman-vol2.pdf

[22] Intel. 2022. Intel Memory Protection Keys. https://www.kernel.org/doc/html/latest/core-api/protection-keys.html.

[23] kashif. 2022. *node-cuda provides NVIDIA CUDA™ bindings for Node.js*. https://github.com/kashif/node-cuda

[24] keyhash. 2022. *Cryptonight hashing functions for node.js*. https://github.com/keyhash/node-cryptonight-old-hardware

[25] 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 *Proceedings of the Seventeenth European Conference on Computer Systems*. 132–148.

[26] 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.

[27] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. (2017).

[28] 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*. Springer, 141–160.

[29] MITRE. 2020. *CVE-2020-28248*. MITRE. https://nvd.nist.gov/vuln/detail/CVE-2020-28248

[30] 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.

[31] 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.

[32] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 736–747.

[33] Node.js. 2022. Native Abstractions for Node.js. (2022). https://github.com/nodejs/nan

[34] nodejs. 2022. *What is Node-API?* https://nodejs.github.io/node-addon-examples/about/what/

[35] ohmu. 2022. *The missing POSIX system calls for Node*. https://github.com/ohmu/node-posix

[36] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*. 49–58.

[37] Aleph One. 1996. Smashing The Stack For Fun And Profit. *Phrack Magazine* 7, 49 (1996).

[38] openJS Foundation. 2009. *node.js*. OpenJS Foundation. https://nodejs.org/en/

[39] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 241–254.

[40] Prior99. 2022. *Unofficial bindings for node to libpng*. https://github.com/Prior99/node-libpng

[41] 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.

[42] Jonathan Salwan. 2015. *ROPgadget tool*. Github. https://github.com/JonathanSalwan/ROPgadget

[43] Sascha Schirra. [n. d.]. *Ropper*. https://github.com/sashs/Ropper

[44] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *Proceedings of the 31th USENIX Security Symposium*.

[45] 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 *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1677–1694.

[46] Snyk. 2021. *Snyk Vulnerability Database*. https://snyk.io/vuln?type=npm

[47] Snyk. 2022. *node-sass vulnerabilities*. Snyk Vulnerability Database. https://security.snyk.io/package/npm/node-sass

[48] 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.

[49] Gang Tan et al. 2017. *Principles and implementation techniques of software-based fault isolation*. Now Publishers.

[50] 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*. Springer, 67–83.

[51] 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 *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1221–1238.

[52] Nikos Vasilakis, Grigoris Ntousakis, Veit Heller, and Martin C. Rinard. 2021. Efficient Module-Level Dynamic Analysis for Dynamic Languages with Module Recontextualization. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1202–1213. https://doi.org/10.1145/3468264.3468574

[53] 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 *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 18 pages. https://doi.org/10.1145/3460120.3484535

[54] Victor van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. 2012. Memory Errors: The Past, the Present, and the Future. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*. 86–106.

[55] Verdaccio. 2022. *Lightweight private npm proxy registry.* https://verdaccio.org/docs/what-is-verdaccio
[56] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You shall not (by) pass! practical, secure, and fast PKU-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 266–282.
[57] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with High Risks: A Study of Security Threats in the Npm Ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 995–1010.

A NODE.JS AND V8

Memory Organisatioon 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 [13]. 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 [14]. 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 addons 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.*).

Addons 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 any Node API, or even the more esoteric V8 API [18], addons 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 [34]. Another API designed for Node.js has started providing the NAN (Native Abstractions for Node.js) API [33] as a *portable*, stable API for addon development, given that both the Node-API and V8 API are version- and platform-dependent (and therefore hinder the portability on addon code).

B PROTECTION KEYS

Protection Keys is a relatively common architectural feature, first introduced in IBM System/360. Today, IBM storage protection keys [20] are part of Z architecture systems. 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 [21] are designed to restrict permissions on memory by tagging each virtual page with a unique domain identifier. IBM extended 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 sand-boxes 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 Memory Protection Keys offer 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 `%pkru` register is thread-specific, MPK supports per thread view of the process's memory. For example, different application threads have different access rights 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 to each thread running in the 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.

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 support 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. In a similar manner as 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` 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`). Finally, `rdpkru` instructions returns the contents of the executing thread's `%pkru` register.

C WRAPPER LIBRARY TEMPLATE

```
__attribute__ ((aligned(4096), pure))
void
wrap_node_api_func(){
    asm ("movq 0xdeadcafe, %rax; jmpq *%rax");
}
...
static __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 | 0xb848;
    ...
    mprotect((void*) wrap_node_api_func, 4096, PROT_EXEC);
    write_got = native_module_address +
                node_api_func_got_entry;
    *write_got = wrap_node_api_func;
    ...
}
```

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 fynctions. Finally, the wrapper functions are configured as execute only.

D MODIFICATIONS IN NODE.JS AND V8 API

```
unsigned
enable_access() {
    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, transers to the native module.