RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity

Ben Niu
Lehigh University
19 Memorial Dr West
Bethlehem, PA, 18015
ben210@lehigh.edu

Gang Tan
Lehigh University
19 Memorial Dr West
Bethlehem, PA, 18015
gtan@cse.lehigh.edu

ABSTRACT

Managed languages such as JavaScript are popular. For performance, modern implementations of managed languages adopt Just-In-Time (JIT) compilation. The danger to a JIT compiler is that an attacker can often control the input program and use it to trigger a vulnerability in the JIT compiler to launch code injection or JIT spraying attacks. In this paper, we propose a general approach called RockJIT to securing JIT compilers through Control-Flow Integrity (CFI). RockJIT builds a fine-grained control-flow graph from the source code of the JIT compiler and dynamically updates the control-flow policy when new code is generated on the fly. Through evaluation on Google's V8 JavaScript engine, we demonstrate that RockJIT can enforce strong security on a JIT compiler, while incurring only modest performance overhead (14.6% on V8) and requiring a small amount of changes to V8's code. Key contributions of RockJIT are a general architecture for securing JIT compilers and a method for generating fine-grained control-flow graphs from C++ code.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*

Keywords

Control-Flow Integrity; Just-In-Time Compilation; Modularity

1. INTRODUCTION

Programming languages with managed runtime systems are becoming increasingly popular during the last two decades. Such languages include JavaScript, Java, C#, Python, PHP, and Lua. The use of managed languages is in general beneficial to software security. Managed environments provide a natural place to deploy a range of security mechanisms to constrain untrusted code execution. For instance, Java and .NET virtual machines implement security sandboxes and bytecode verification. As another example, a JavaScript engine enforces dynamic typing, making execution of JavaScript much more secure than native-code-based ActiveX.

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660281.

For performance, modern managed language implementations adopt Just-In-Time (JIT) compilation. Instead of performing pure interpretation, a JIT compiler dynamically compiles programs into native code and performs optimization on the fly based on information collected through runtime profiling. JIT compilation in managed languages is the key to high performance, which is often the only metric when comparing JIT engines, as seen in the case of JavaScript. Hereafter, we use the term *JITted code* for native code that is dynamically generated by a JIT compiler, and *code heap* for memory pages that hold JITted code.

In terms of security, JIT brings its own set of challenges. First, a JIT compiler is large and usually written in C/C++, which lacks memory safety. It contains sophisticated components such as a parser that parses untrusted input programs, an optimizing compiler that generates optimized native code, and a garbage collector. There are always security-critical bugs in such a large and complicated C/C++ code base. For instance, several buffer-overflow vulnerabilities have been found in Google's V8 JavaScript engine [1].

Second, the code heap used in JIT compilation is often made both writable and executable to facilitate online code modification, which is used in crucial optimization techniques such as inline caching [15, 19] and on-stack replacement [20]. Consequently, Data Execution Prevention (DEP) cannot be applied to memory pages of the code heap. By exploiting a bug such as a heap overflow, an attacker can inject and execute new code in those pages.

Finally, even without code injection, a class of attacks called *JIT spraying* [8] enables an attacker to craft input programs with special embedded constants, influence a JIT compiler to generate native code with those constants embedded, and hijack the control flow to execute those constants as malicious code; more information about the JIT spraying is in the background section (Sec. 3).

Research progress has been made in improving the security of JIT compilation [6, 21, 34, 11, 35, 10]. NaCl-JIT [6] puts a JIT compiler and its JITted code in a sandbox based on Software-based Fault Isolation (SFI [32]). Software diversification techniques such as librando [21] are used to diversify JITted code to mitigate JIT spraying attacks. However, past techniques suffer from two main drawbacks. First, existing systems provide only loose security. NaCl-JIT enforces coarse-grained control-flow integrity, which cannot prevent advanced Return-Oriented Programming (ROP) attacks [18, 13, 9]. Software diversification techniques improve security in a probabilistic sense; a lucky or determined attacker might be able to defeat such schemes. Second, some systems impose a large performance overhead. For instance, NaCl-JIT imposes around 51% overhead on the V8 JavaScript x86-64 engine. More detailed discussion about related work is in Sec. 2.

In this paper, we propose an approach entitled RockJIT for securing JIT compilation. Our starting idea for security is to enforce fine-

grained Control-Flow Integrity (CFI), in which a high-precision Control-Flow Graph (CFG) is statically extracted from the source code of a JIT compiler and enforced during runtime. The distinction between fine-grained and coarse-grained CFI will be discussed in Sec. 2, but at a high level fine-grained CFI is stronger and is an effective defense against control-flow hijacking attacks, including code injection, return-to-libc, and ROP attacks.

Enforcing fine-grained CFI on a JIT compiler is not enough, however, because it dynamically generates new code, whose CFG also needs to be considered. RockJIT is built upon *Modular Control-Flow Integrity* (MCFI [26]), a new CFI system that supports dynamic linking. In MCFI, a program is divided into multiple modules, each of which carries its own CFG. When modules are linked dynamically, individual CFGs are combined to form the CFG of the combined module. During runtime, CFGs are represented as tables, which are accessed and updated through lightweight software transactional memory. The support for modularity is necessary for a JIT environment because each piece of newly generated code is essentially a new module, whose CFG needs to be combined with the CFG of existing code.

Nevertheless, significant challenges remain when hardening a JIT compiler through MCFI. The first challenge is about performance. MCFI was designed to support dynamic linking of libraries. Dynamic linking happens infrequently during program execution. A JIT compiler, however, generates and updates code frequently. For instance, the V8 JavaScript engine installs new code about 50 times per second. It is unclear how MCFI's scheme scale in the context of JIT with frequent code generation and modification. The second challenge is about the generation of a fine-grained CFG from C++ source code. The MCFI work proposes how to generate a fine-grained CFG for C code, while most JIT compilers have a large body of C++ code. The C++ language contains advanced control-flow features such as exceptions and virtual methods, which complicate the process of CFG generation.

RockJIT addresses the aforementioned challenges and is a general JIT compilation hardening approach. It makes the following contributions:

- RockJIT hardens both the JIT compiler and JITted code, but by enforcing different levels of CFG precision on the JIT compiler and JITted code, its overhead is much smaller than previous work and its security is stronger. Our evaluation on the V8 engine shows that RockJIT-hardened V8 can remove over 99.97% functionality-irrelevant indirect branch edges from NaCl-JIT-hardened V8, and is only 14.6% slower than the vanilla V8.

- We propose a method for generating high-precision CFGs for C++ programs. Our extensive experience on over one million lines of code shows our method is practical: C++ programs can be made compatible with our method with a small amount of changes to source code.

- Our evaluation on the V8 engine shows that our scheme for JIT protection requires only minimal changes to a JIT compiler. We changed only around 800 lines of source code, about 0.14% of V8's code base. Furthermore, our investigation leads us to believe our scheme can be easily adopted to other JIT compilers.

2. RELATED WORK

For related work, we discuss coarse-grained and fine-grained CFI as well as past schemes for protecting JIT compilation.

2.1 Coarse-grained and fine-grained CFI

CFI [3] enforces a pre-determined Control-Flow Graph (CFG) on a program. Indirect branches (i.e., returns, indirect calls, and indirect jumps) are instrumented to ensure that their targets are consistent with the specified CFG.[1] A CFG is a static over-approximation of a program's dynamic control flow and therefore a program can have many CFGs. Depending on the precision of enforced CFGs, we can broadly classify CFI techniques into coarse-grained and fine-grained ones.

In coarse-grained CFI, an imprecise CFG is enforced on indirect branches. In such a CFG, all indirect branches share a common set or a few sets of possible targets. For instance, an indirect call (e.g., a call via a function pointer) can be allowed to call all functions (even though the call may actually target a small subset of functions during runtime). CCFIR [37] and binCFI [38] are two typical coarse-grained CFI systems. PittSFIeld [23], NaCl [36, 29], and MIP [25] are designed to sandbox native code and they also enforce coarse-grained CFI to prevent inlined sandboxing instructions from being bypassed. The precision of coarse-grained CFI is lower and ROP attacks are still possible without violating the imprecise CFG. Recent work [18, 13, 9] demonstrates how to mount ROP attacks on systems hardened with coarse-grained CFI. The benefits of coarse-grained CFI are that the imprecise CFGs are easier to build and its performance overhead is usually lower.

In fine-grained CFI, each indirect branch can have its own set of targets and as a result its precision is much higher. It significantly enhances security because it restrains an attacker's ability to link ROP gadgets. We believe it is probably the best defense against ROP attacks. Various techniques for fine-grained CFI have been proposed in the literature [16, 33, 4, 12, 27]. However, none of them is modular, meaning that dynamically linked libraries cannot be instrumented once and reused across programs. Each program has to come with its own instrumented version of libraries. Tice *et al.* [31] proposed an approach to fine-grained CFI with modularity support. However, it does not protect return instructions, and its modularity support introduces time windows for attacks during dynamic module linking. RockJIT is based on MCFI [26], which secures all indirect branches and supports dynamic linking; we will present background information about MCFI in Sec. 3.3.

One obstacle to fine-grained CFI is the difficulty of building a fine-grained CFG. This is technically possible with the access to source code. However, advanced control-flow features in C++ make the building of a CFG difficult. We propose a method for generating high precision CFGs for C++ programs and it covers all C++ control-flow features.

2.2 JIT protection mechanisms

RockJIT's goal of improving the security of JIT compilation is shared by several other systems. Perhaps the closest work is NaCl-JIT [6], which applies Software-based Fault Isolation (SFI) to constraining both a JIT compiler and JITted code. To prevent SFI checks from being bypassed, NaCl-JIT enforces aligned-chunk CFI, which is coarse grained. In aligned-chunk CFI, the code is divided into fixed-size chunks (e.g., 32 bytes) and indirect branches are restricted by address masking to target only chunk beginnings. This form of CFI is easy to implement, but brings several disadvantages. First, it provides weaker security. As discussed, coarse-grained CFI allows an indirect branch to target many more addresses than necessary. In contrast, RockJIT applies fine-grained CFI on the JIT compiler and therefore provides stronger security. As our experi-

[1]The targets of direct branches are statically computable and their control flow is checked statically in CFI. Therefore, only indirect branches need to be instrumented with dynamic checks.

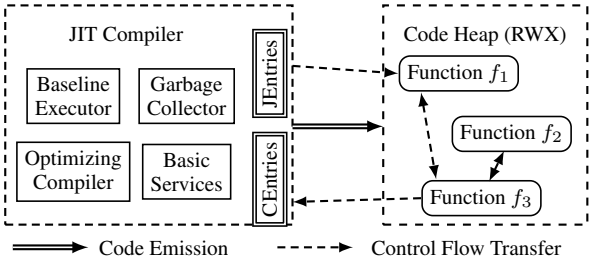

Figure 1: The architecture of modern JIT compilers.

ments show, RockJIT can eliminate 99.97% indirect branch targets from NaCl-JIT's enforced CFG. Second, NaCl-JIT has high performance overhead. Its aligned-chunk CFI requires insertion of many no-op instructions to make indirect-branch targets 32-byte aligned. NaCl-JIT reports no-ops account for half of the sandboxing cost. Largely because of this, its performance overhead is around 51%. By contrast, RockJIT's overhead is 14.6%.

Software diversification has been used to mitigate JIT spraying attacks. The librando system [21] inserts a random amount of no-ops in the JITted code. In addition, it uses a technique called constant blinding: it replaces instructions that have constant operands with other equivalent instruction sequences because JIT spraying attacks inject malicious logic in constants that are operands of instructions such as `xor`. Due to its black-box implementation, librando has to disassemble the JITted code, modify the code, and reassemble the new code. It incurs a significant overhead (265.8%). Other systems including INSeRT [34], JITSafe [11], and RIM [35] also employ diversification techniques similar to librando's. These diversification-based systems provide only probabilistic defenses; a determined attacker can enumerate all possibilities if the search space is small; a lucky attacker might also defeat the defense. Furthermore, they protect only JITted code, not the JIT compiler. In comparison, RockJIT can eliminate JIT spraying attacks and enforces CFI on both the JIT compiler and JITted code. On the other hand, since software-diversification techniques are orthogonal to CFI, it is perhaps beneficial to deploy both defenses in a JIT compiler, following the principle of defense in depth.

Finally, another mitigation mechanism is to separate the write permission from the execution permission for the code heap. For instance, JITDefender [10] and JITSafe [11] drop the write permission of the code heap whenever it is not needed. However, before dropping the permission, those code pages may have already been modified by the attacker for arbitrary code execution. More importantly, they cannot prevent JIT spraying attacks, which do not require modifying the code heap.

3. BACKGROUND

We briefly present the background information for RockJIT, including the general architecture of JIT compilers, how JIT spraying attacks work, and MCFI.

3.1 The architecture of JIT compilers

We investigated a range of JIT compilers, including Google V8 (JavaScript), Mozilla TraceMonkey (JavaScript), Oracle HotSpot (Java), Facebook HHVM (PHP), and LuaJIT (Lua). We found that their architectures share many commonalities and can all be represented by the diagram in Figure 1. A JIT compiler emits JITted code in the code heap and executes it. The code heap is readable (R), writable (W), and executable (X). A typical JIT compiler contains the following major components:

Baseline Executor. When a program starts running, its execution is the job of the baseline executor. Oftentimes, the baseline executor is an interpreter, which is easy to implement but slow. For instance, HotSpot has an interpreter that interprets Java bytecode. The baseline executor may have a different implementation from an interpreter. For example, the baseline executor of V8 compiles JavaScript source code directly to unoptimized native code.

Optimizing Compiler. During the execution of a program by the baseline executor, the JIT compiler performs runtime profiling to identify hot code and to identify types in the case of dynamically typed languages. Based on the runtime profile, the optimizing compiler generates optimized native code. JIT engines can have quite different designs for the optimizing compiler. For example, V8 profiles method execution and optimizes a whole method at a time. However, TraceMonkey profiles execution paths (e.g., a hot loop) and performs trace-based optimizations.

Garbage Collector. Managed languages provide automatic memory management, which is supported by a garbage collector. Most garbage collectors implement common algorithms such as concurrent mark and sweep.

Basic Services. The JIT compiler also provides runtime services, including support for debugging, access to internal states for performance tuning, foreign function interfaces for enabling interoperability between managed languages and native code.

For performance, all JIT compilers we inspected are developed in C/C++. Since the calling convention of C/C++ is different from that of JITted code, which is JIT-compiler specific, JIT compilers introduce interfaces to allow context switches between the code of the compiler and JITted code. In Figure 1, the interfaces are depicted as `JEntries` and `CEntries`; both are essentially indirect branches. `JEntries` transfer control to JITted code and `CEntries` transfer control to the JIT compiler. As an example of `JEntries` in V8, the initial control transfer from the JIT compiler to the code heap is through an indirect call (`JEntry`) in a code stub called `JSEntryStub`. As an example of `CEntries`, V8 provides services (or functions) such as JavaScript object creation and object property access. When JITted code invokes these services, the control is first transferred to a stub called `CEntryStub` with a register containing the address of the target service function. Within `CEntryStub`, an indirect call (`CEntry`) through the register is executed to transfer the control to the service function.

3.2 JIT spraying attacks

JIT compilers have large attack surfaces, since the input program can be fully controlled by an attacker. Specifically, a JIT spraying attack takes advantage of the often predictable code-emission logic in the JIT compiler. The attacker crafts an input program with special embedded constants and uses a vulnerability in the JIT compiler to hijack the control flow to execute those constants as malicious code. To illustrate this point using JavaScript, suppose the input code is "`x = x ∧ 0x3C909090`", where ∧ is JavaScript's `xor` operator. A JavaScript compiler then generates native code for implementing the `xor` operation. Suppose, in its generated code, the constant `0x3C909090` is encoded literally. Then the byte sequence on x86 for encoding an `xor` operation is as follows, assuming `%eax` holds the value for `x`.

```
35 90 90 90 3c:   xorl 0x3C909090, %eax
```

Now suppose the JavaScript compiler has a vulnerability that enables the attacker to control the program counter. Because x86 has a variable-length instruction set, the attacker can then change the control flow to point to the middle of the above instruction and start the execution of a totally different instruction stream from the intended. For example, if the program counter is changed to point to the first `0x90` in the above, then the next instruction to execute is a no-op (the encoding of `0x90`), followed by other instructions not intended in the original program. Note that the constant `0x3C909090` above is under the control of the attacker, who can put any constant there to determine what code to execute.

Modern operating systems deploy Address Space Layout Randomization (ASLR), which makes it hard for the attacker to guess the absolute addresses of constants in instructions such as xor. The attacker, however, can spray many copies of the same code in memory to increase the chance of a successful attack on the JIT; this is why it is called JIT spraying. Real JIT spraying attacks have been demonstrated, for example, on the JavaScript engine of the Safari browser [30] and Adobe's Flash Player [7].

One observation about JIT spraying attacks is they involve both the JITted code and the JIT compiler. The attacker takes advantage of the fact that the JITted code is often predictable for a given piece of source code. Furthermore, there must be a vulnerability in the compiler so that the control can be transferred to the middle of an instruction to start an unexpected and harmful code sequence. Given this observation, one natural defense is to randomize code generation to make the generated native code less predictable. This approach has been explored by systems such as librando [21] and others. The downside is that it provides only a probabilistic defense. Instead, we explore an alternative approach. We harden the JIT compiler (and the JITted code) using control-flow integrity so that it is impossible to transfer the control to the middle of instructions; as a result, unexpected instructions can never be executed.

3.3 Modular Control-Flow Integrity (MCFI)

MCFI [26] is a new fine-grained CFI system with low performance overhead (around 5%) and modularity support. Past CFI techniques do not support dynamic linking, because they require all code to be available during static instrumentation. In MCFI, a program is divided into multiple modules. Each module contains not only code and data, but also auxiliary information that is used to generate a new CFG when linking with other modules. Code of a module is instrumented separately for CFI, without considering other modules. When a new module is dynamically linked, a new CFG is generated based on the new module's and the existing module's auxiliary information. The new control-flow policy may allow an indirect branch to target more destinations.

The auxiliary information carried in MCFI modules is type information, specifically, the types of its functions and function pointers. MCFI takes a module's source code and compiles it using a modified LLVM toolchain to acquire the type information. MCFI includes a simple yet effective way of generating CFGs for C programs (but not C++ programs) based on a type-matching method. This method is efficient and can be used during dynamic linking. It generates relatively precise CFGs, and requires only small changes to the source code.

For CFI enforcement, MCFI represents the CFG in tables during runtime. It designs thread-safe table transactions for accessing and updating the tables. When a new module is dynamically linked, a table-update transaction is executed to update the tables with information about the new CFG. Indirect branches are instrumented to first load the target into a register r, run a table-check transaction to see if the target is allowed, and, if so, perform an indirect jump through r. The tables are stored in memory and need protection. MCFI instruments indirect memory writes in the untrusted program to first mask the target address before the write; this is a form of Software-based Fault Isolation and ensures the integrity of the tables. Finally, indirect branch targets are aligned at four-byte aligned addresses so that an entry in MCFI tables can be retrieved atomically in one memory-access instruction.

4. RockJIT OVERVIEW

In this section, we discuss RockJIT's threat model, its main defense mechanisms, and its security strength.

4.1 Threat model

RockJIT's threat model is the same as the strong model in the original CFI work [3]. An attacker is modeled as a concurrent user-level thread, running in parallel with other threads in the JIT compiler. The attacker thread can read and write any memory, subject to memory page protection. Therefore, in this model, any writable memory can change because of the attacker thread between any two instructions in the user program. CPU registers of a thread are assumed not writable directly by the attacker thread. However, the attacker can indirectly affect registers of other threads by corrupting memory. For example, if one JIT-compiler thread loads from writable memory to a register, then the register's value is controlled by the attacker because she/he controls the writable memory.

We assume the JIT compiler's code is benign, but may contain vulnerabilities. The JITted code can contain malicious logic since it is compiled from source code that might be provided by the attacker. The malicious logic aims to launch attacks such as code injection and JIT spraying attacks.

We further make two assumptions about the JIT compiler. First, we assume context switches between the JIT compiler and JITted code are through a set of interfaces; that is, only through one of those `JEntries` and `CEntries` in Figure 1 can the control transfer between the JIT compiler and JITted code. This assumption enables different CFG precision on the JIT compiler and JITted code. Second, we assume JITted code, when executed normally (i.e., no jumps to the middle of instructions), does not contain direct system-call invocations and privileged instructions. The JITted code can, however, invoke one of the `CEntries` to request services such as OS system calls from the JIT compiler (after appropriate security checking by the compiler). These two assumptions are true in all the JIT compilers we have inspected. Even if a certain JIT compiler violates these assumptions, it should be easy to modify it to make the assumptions hold.

4.2 Defenses in RockJIT

RockJIT's architecture is visualized in Figure 2. It provides services to a JIT compiler and monitors its security. An existing JIT compiler such as V8 is modified slightly to cooperate with RockJIT. It is then compiled and instrumented by RockJIT's compilation toolchain to generate an MCFI module. The module is loaded by RockJIT into a sandbox. After loading, RockJIT generates a control-flow graph for the JIT compiler based on the auxiliary type information in the module, constructs MCFI tables that encode the control-flow graph, and starts execution of the JIT compiler.

The sandbox around the JIT compiler and JITted code restricts their control flow according to the tables and also restricts their memory access to be inside the sandbox. The JIT compiler can request services provided by RockJIT via a set of well-defined interface functions. For example, to prevent code in the sandbox from changing memory protection arbitrarily, all direct system calls for changing memory mapping and memory pages' protection bits are

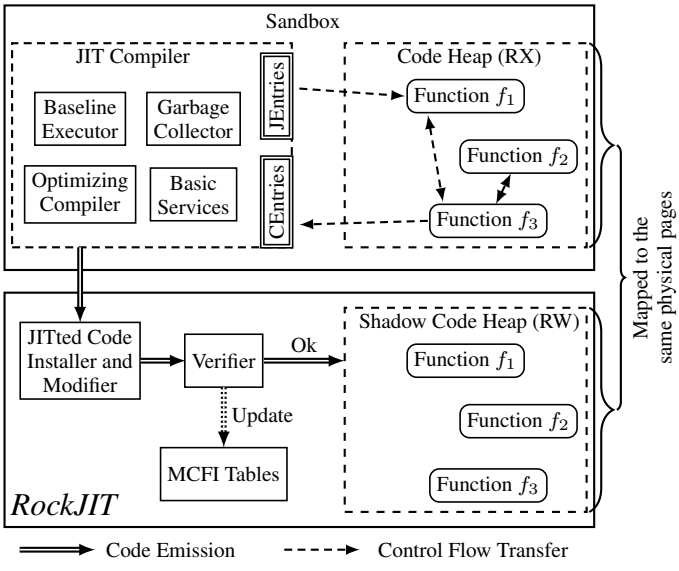

Figure 2: The architecture of RockJIT.

forbidden; instead, the code can invoke services provided by RockJIT to issue such system calls in a managed way.

To rule out code injection attacks, RockJIT guarantees that no memory pages are writable and executable at the same time, similar to Data Execution Protection (DEP). One thorny issue we mentioned before is that the code heap (i.e., memory pages that hold JITted code) is made both writable and executable in typical JIT compilers.

To address this issue, RockJIT uses a *shadow code heap*, similar to what NaCl-JIT does [5]. It takes advantage of the virtual memory mechanism available to user-space programs. The shadow code heap is outside of the JIT compiler's sandbox and in RockJIT's private memory. It is mapped to the same physical pages as the code heap in the sandbox but with *different permissions*. This can be achieved by shared memory mechanisms provided by OSes (e.g., `shm_open`, `ftruncate` and `mmap` libc calls on Linux). In particular, the code heap in the sandbox is made readable and executable, but not writable. The shadow code heap is made readable and writable, but not executable.[2] Because memory access of the JIT compiler is restricted to be inside the sandbox, the JIT compiler cannot directly modify the shadow code heap for runtime code manipulation. Instead, it invokes services of RockJIT to install new native code or modify existing native code. RockJIT performs *verification* on the native code to check a set of properties (detailed in Sec. 5) for security. If the verification succeeds, RockJIT installs the new code in the shadow code heap and updates MCFI tables using a new control-flow graph that takes the new code into account. Since the shadow code heap maps to the same physical pages as the in-sandbox code heap, the code heap is filled with the same code, which can then be invoked by the JIT compiler.

RockJIT enforces control-flow integrity on both the JIT compiler and JITted code, but applies different levels of precision on those two parts. For the JIT compiler, RockJIT applies a C++ CFG generation strategy detailed in Sec. 6 to produce a relatively fine-grained CFG offline; it takes into consideration C++ semantics such as virtual method calls. In contrast, the CFG for JITted code is coarse-grained in the sense that all its indirect branches share a common set of targets. The JIT compiler is modified to emit not only native code, but also information about indirect-branch targets. The verifier then deduces the coarse-grained CFG for the new code and combines it with the old CFG.

The approach of hybrid CFI precision in RockJIT is the result of a careful consideration of both security and performance. First, the JIT compiler's code is mostly where the majority of the code is and contains dangerous system call invocations. Since its code is statically available, constructing a fine-grained CFG offline for the JIT compiler increases security substantially as recent work has shown that coarse-grained CFI can still be attacked by ROP attacks [18, 13, 9]. On the other hand, JITted code is frequently generated on the fly and for performance it is important that verification and new CFG generation do not have high performance overhead. Verification and CFG-generation algorithms for coarse-grained CFI run much faster. Some readers may wonder whether coarse-grained CFI for JITted code might jeopardize security. We do not believe that is the case because one of our assumptions is that JITted code cannot contain dangerous instructions such as system calls, a property that is enforced by RockJIT's verifier; such instructions are required in an attack. JITted code can still request system-call services from the JIT compiler, but the JIT compiler is hardened through fine-grained CFI: security is maintained as long as sufficient checks are placed before system calls for the set of control-flow paths in a fine-grained CFG, which is a much smaller set than the one in a coarse-grained CFG.

4.3 Security benefits

Despite the threats described in the threat model, RockJIT's defense provides the following security benefits:

- No JIT spraying attacks. JIT spraying attacks inject malicious instruction sequences in seemingly benign JITted code and jump to the malicious code by hijacking the control-flow. RockJIT enforces control-flow integrity and it is not possible to execute instructions that are not in the original JITted code. Therefore, JIT spraying attacks are prevented.
- The execution of the JIT compiler respects a fine-grained CFG and there are no known ROP attacks that can attack a system with fine-grained CFI. Furthermore, no memory pages in the JIT compiler and JITted code are both writable and executable, preventing code injection attacks.

One point worth mentioning is that, thanks to the verifier, the JIT compiler is not in the TCB even though it performs runtime code manipulation. The native code generated by the JIT compiler is first checked to obey a set of safety properties before installed. The verifier is in the TCB but it is much smaller than the JIT compiler.

5. SECURING JITTED CODE

The code heap maintained by a JIT compiler is where code is dynamically managed. It consists of multiple code regions. A JIT compiler dynamically installs, deletes, and modifies code regions. New code regions are frequently generated by the compiler and installed in the code heap. When a code region is no longer needed, the JIT compiler can delete it from the code heap and reuse its memory for future code installation. Runtime code modification is mostly used in performance-critical optimizations. As an example, *inline caching* [15, 19] is a technique that is used in JIT compilers to speed up access to object properties. In this technique, a JIT compiler modifies native code to embed an object property such as a member offset after the property has been accessed for the first

[2]Since the shadow code heap is controlled by trusted RockJIT, whether it is executable or not does not affect security; we make it not executable, following the principle of least privilege.

time, avoiding expensive object-property access operations in the future. Another example of runtime code modification happens in V8 during code optimization. V8 profiles function and loop execution to identify hot functions and loops. It performs optimization on the hot code to generate an optimized version. Afterwards, runtime code patching is performed on the unoptimized code to transfer its control to the optimized version through a process called on-stack replacement [20].

Since RockJIT enforces CFI, it is necessary to check security for each step of runtime code installation, deletion, and modification. In Sec. 5.1, we present how verification is performed when a new piece of code is installed. The process for code deletion and modification has only small differences; we leave their discussion to Sec. 5.2 when we discuss the detailed steps for runtime code manipulation. In Sec. 5.3, we discuss how to modify a JIT compiler to cooperate with RockJIT for secure native code execution using V8 as an example.

5.1 Verification

The verifier maintains three sets of addresses that are code addresses in the code heap:

- **Pseudo-instruction start addresses (**PSA**).** This address set remembers the start addresses of all *pseudo-instructions*. We define a pseudo-instruction as: (1) a *checked indirect branch*, which is MCFI's table-based instruction sequence for checking a register r immediately followed by an indirect branch through r; or (2) a *masked memory write*, which is MCFI's mask on a register r immediately followed by a memory write through r; or (3) an instruction that is neither an indirect branch nor an indirect memory write.

- **Indirect branch targets (**IBT**).** This address set remembers all possible indirect branch targets. All such targets are four-byte aligned.

- **Direct branch targets (**DBT**).** This address set remembers all direct branch targets.

The critical invariant of the three sets is $\mathrm{IBT} \cup \mathrm{DBT} \subseteq \mathrm{PSA}$. That is, all indirect and direct branch targets must be start addresses of pseudo-instructions. With this invariant, it is impossible to jump to the middle of an instruction, which is necessary for JIT spraying attacks. Furthermore, it is impossible to transfer the control to an indirect branch or a memory write without executing its preceded MCFI check, which is necessary for CFI and SFI.

The three address sets are built incrementally with the installation of new code. Initially, they are all empty sets when the code heap contains no code. When a new code region is installed, the verifier updates the three address sets; that is, compute PSA', IBT', and DBT' after taking new code into consideration. Our approach for computing these new sets is through a combination of information already in the code and meta information emitted by a modified JIT compiler. For instance, direct branch targets (DBT') can be computed from the code alone. For PSA', our verifier for V8 takes the start address of the new code and identifies pseudo-instruction boundaries by following a process similar to sequential disassembly (however, no full disassembly is performed; only the boundaries are identified; see the DFA approach later). This is sufficient for V8. If a JIT compiler mixes code and data in JITted code, then we could modify the JIT compiler to emit also instruction boundary information. For IBT', since V8 installs the native code of one function at a time, new indirect branch targets include the start address of the function and the addresses after direct/indirect calls in the function. In addition, there are indirect branch targets related to exception handling and optimization (e.g., on-stack replacement entry points in an optimized function). We modified V8 to emit these additional indirect branch targets along with code.

With the new address sets, the verifier checks $\mathrm{IBT}' \cup \mathrm{DBT}' \subseteq \mathrm{PSA}'$ and the following constraints on the new code:

C1 Indirect branches and memory-write instructions are appropriately instrumented. In particular, only checked indirect branches and masked memory writes are allowed.

C2 Direct branches jump to addresses in DBT'. This ensures that the new code respects DBT'.

C3 The code contains only instructions that are used for a particular JIT compiler. This set of instructions is usually a small subset of the native instruction set and can be easily derived by inspecting the code-emission logic of a JIT compiler. Importantly, this subset cannot contain system calls and privileged instructions—one of our assumptions.

Next we present some implementation details about a verifier we constructed for V8. First, the address sets are implemented by bitmaps for fast look-ups and updates. Each bitmap maps a code address to one if and only if that address belongs to the corresponding set, otherwise zero.

Second, the speed of verification is of practical importance. Since V8 performs frequent code installation, a slow verifier can impact the performance nontrivially. For example, NaCl-JIT includes a disassembly-based verifier and it reports 5% overhead for the verification alone. We adopt an approach based on Deterministic Finite Automata (DFA) following RockSalt [24]. It performs address-set updates and constraint checking in one phase. Our verifier incurs only 1.7% overhead for the verification.

In detail, we followed Seaborn's approach [28] of using a trie structure [17] to enumerate all possible allowed instruction encoding. Then the trie is converted to a DFA. The DFA has 257 states. It has multiple acceptance states: one for recognizing a checked indirect branch; one for recognizing a masked memory write; one for recognizing a direct branch; one for recognizing all other V8-allowed instructions. The verifier iterates through all instructions recognized by the DFA. When a direct branch is matched, it records its jump target; when a checked indirect branch, a masked memory write, or one allowed instruction is matched, it moves forward. In the above cases, the pseudo-instruction boundaries are also recorded. The verification fails when the DFA reaches a failure state (e.g., due to an illegal instruction). After all code bytes have been matched, the verifier updates the address sets and checks that $\mathrm{IBT}' \cup \mathrm{DBT}' \subseteq \mathrm{PSA}'$. When the verification succeeds, constraints C1–C3 are respected by the code.

Recall that our threat model does allow attackers to write arbitrary memory pages in the sandbox that are writable, so it is possible that after the code is emitted in the sandbox and before it is copied outside of the sandbox for verification, the attackers might corrupt it. However, the corrupted code still needs to pass the verification. Once it passes the verification, the security benefits mentioned in Sec. 4.3 are still valid.

5.2 JITted code installation, deletion, and modification

In RockJIT, a JIT compiler cannot directly manipulate the code heap, which does not have the writable permission. Instead, RockJIT provides interface functions to the JIT compiler for code instal-

lation, deletion, and modification. One worry for runtime code manipulation is thread safety: one thread is manipulating code, while another thread may see partially manipulated code. We next discuss the detailed steps involved in RockJIT's code manipulation and how thread safety is achieved.

Code installation. For code installation, the JIT compiler invokes RockJIT's code installation service and sends a piece of native code, the target address where the native code should be installed, and meta information about the code for constructing new address sets. The code-installation service then performs the following steps:

1. The verifier performs verification on the code and updates the address sets to PSA', IBT', and DBT'.

2. If the verification succeeds, the code is copied to the shadow code heap at an address computed from the start address where the code should be installed. There is a fixed correspondence between addresses of the code heap in the sandbox and addresses of the shadow code heap.

3. The runtime tables used by MCFI are updated to take into account the new code. Since coarse-grained CFI is enforced on JITted code, only information in IBT' is needed to update the tables.

There are a couple of notes worth mentioning about the above steps. First, the verification of benign programs is expected to succeed if there are no bugs in the JIT compiler. A verification failure indicates a bug that should be fixed. Second, it is important that the MCFI tables are updated after copying the code, not before. During the copying process, the code becomes partially visible to the JIT compiler as the code heap is mapped to the same physical pages as the shadow code heap. However, since the MCFI tables have not been updated yet, no branches can jump to the new code, avoiding the situation in which one thread is installing some new code and another thread branches to partially installed code.

Code deletion. In a multi-threaded JIT compiler, one thread may request the deletion of a code region, while another thread may be executing in the middle of that code region due to JIT compiler bugs or attacks. For safety, the code region shall not be deleted until all threads exit the code region. The following steps are performed when one thread invokes the code-deletion service to delete code region cr:

1. Check that direct branches outside cr do not target any instruction in cr. If this check fails, deleting cr would break the critical invariant mentioned before; this would imply either a bug in the JIT compiler or an attack and therefore RockJIT simply terminates the JIT compiler in this case.

2. Remove cr-related entries in the MCFI tables to prevent all indirect branches from targeting cr. After this step, no thread can enter cr simply because no direct or indirect branch in the JITted code can target cr.

3. Check that there are no threads running (or sleeping) in cr. To achieve this, RockJIT waits until it observes that each thread has entered the code in RockJIT's runtime at least once after the update to the MCFI tables. Once a thread enters the code in RockJIT's runtime, it can no longer execute instructions in cr thanks to the update to the MCFI tables.

   In detail, RockJIT maintains a local counter for each thread. The counter for a thread is atomically incremented by one each time when the thread enters the code in RockJIT's runtime. When handling a code-deletion request, RockJIT atomically reads all threads' counters, associates them with cr, and returns without removing cr. At a later time (e.g., in the next invocation of code deletion), RockJIT checks that *each thread's current counter value is not equal to the thread's old counter value associated with* cr. If the condition holds, it means that after the code deletion request, each thread has executed RockJIT's code at least once and therefore no thread can possibly run code in cr; so it can be safely deleted.

Compared to NaCl-JIT, which supports only a finite number of code deletion operations, our code deletion supports an arbitrary number of code deletion operations. Readers may wonder whether some code region may never be deleted in our asynchronous scheme if a thread is executing a long-running loop in the JITted code. However, modern JIT engines implement mechanisms to interrupt JITted code execution (e.g., V8 inserts extra code to each function's prologue and each loop to interrupt the execution to support optimization and deoptimization). Therefore, even if the JITted code is running in a loop, its execution can be interrupted.

Code modification. If the new code region has the same internal pseudo-instruction boundaries and native instruction boundaries as the old code region and the new code passes verification, RockJIT follows NaCl-JIT's approach to replace the old code with the new code. Otherwise, code modification is implemented as a code deletion followed by a code installation.

5.3 Modification to a JIT compiler

An existing JIT compiler needs to be modified to work with RockJIT. We next report our experience of adapting Google's V8 JavaScript engine (v3.25.28.3). To adapt V8's x86-64 source, we modified only 811 lines of its source code: 801 lines were changed to make it generate MCFI-compatible code and invoke RockJIT's services for runtime code manipulation; 10 lines were added for CFG generation, which will be discussed in the next section. This experience partly demonstrates that modifying an existing JIT compiler to work with RockJIT requires only modest effort. Most of the changes to V8 were in its code-emission logic to make the generated code compatible with MCFI:

- Code-emission functions that generate indirect branches were modified to generate checked indirect branches. A checked indirect branch in MCFI requires two scratch registers to hold intermediate values. Since V8 reserves `r10` for its internal use, `r10` is used as one scratch register. In addition, V8 reserves `r12` to always hold a constant representing integer one. We use `r12` as the second scratch register and restore its constant value after a checked indirect branch.

- Code-emission functions for indirect memory writes were modified to generate masked memory writes. The sandbox resides in the [0, 4GB) memory. Therefore, an indirect memory write is turned into two instructions: the first loads the target address into a scratch register r and clears the upper 32 bits; the second writes data to the address in r.

- Code-emission functions for procedure calls were modified to align the addresses immediately following the calls to four-byte aligned addresses. All indirect branch targets need to be four-byte aligned to allow atomic table access. The address alignment is achieved by inserting multi-byte no-op instructions before call instructions.

Another part we modified was to accommodate online code patching. When V8 emits certain optimized native code, it reserves some bytes in the code in anticipation of future code patching (for a process called deoptimization). The original V8 reserves 13 bytes for such purpose. RockJIT needs more bytes because of extra MCFI checks; we had to reserve 44 bytes instead.

Finally, changes were made to V8 to invoke code installation, deletion, and modification services provided by RockJIT at appropriate places.

RockJIT changes much less code than NaCl-JIT, which changed over 5,000 lines of code for the x86-64 version of V8. NaCl-JIT requires more changes because: (1) it disallows the mix of code and data in V8's code and V8 has to be changed to separate code and data; RockJIT's CFI allows the mix of code and data as long as data cannot be reached from code in control flow; (2) NaCl-JIT uses the ILP32 programming model on x86-64, while the native V8 uses LP64 model; therefore, it has to change nearly the entire code-emission logic.

6. C++ CFG GENERATION

RockJIT secures a JIT compiler's code by enforcing fine-grained CFI. Since all JIT compilers are developed in C/C++ for performance, we need a general methodology for generating fine-grained CFGs from C/C++ programs. The MCFI work presents such a method for C programs, but not C++ programs. SafeDispatch [22] performs Class Hierarchy Analysis (CHA [14]) to identify all possible targets of virtual method calls in C++ programs, but it does not cover other control-flow features including exceptions and indirect calls via function pointers. We next discuss RockJIT's method for generating high precision CFGs, which covers all C++ control-flow features.

In CFI, a binary-level CFG is enforced. In such a CFG, nodes represent machine instructions and there is a directed edge between two instructions if the control can possibly reach the second instruction after the execution of the first. The edges out of non-indirect-branch instructions can be statically computed. The difficulty is about indirect branches. However, we can statically compute a superset of their possible targets for approximation. In a C++ program, indirect branches are compiled from code that uses features such as virtual method calls and exceptions. We next discuss those C++ features and how RockJIT approximates the resulting indirect branches' targets by static analysis. It should be noted that C++ compilation is ABI-dependent, and our CFG generation targets binaries conforming to the mainstream Itanium C++ ABI [2] supported by LLVM and GCC. The approach has been tested using a modified LLVM compiler (version 3.3).

Virtual method calls. C++ supports multiple inheritance and virtual methods. A virtual method call through an object is compiled to an indirect call (or an indirect jump with tail call optimization). A virtual call on an object is resolved during runtime through dynamic dispatch. Which method it invokes depends on the actual class of the object. Similar to SafeDispatch, RockJIT performs CHA on C++ code. This analysis tracks the class hierarchy of a C++ program and determines, for each class C and each virtual method of C, the set of methods that can be invoked when calling the virtual method through an object of class C; these methods might be defined in C's subclasses. RockJIT simply allows a virtual method call to target all methods determined by the CHA analysis.

It should be pointed out that CHA is usually a whole-program analysis. To support separate compilation, our implementation emits a class hierarchy for each module and combines modules' class hierarchies at link time.

```
1   typedef int (*Fp)();
2   Fp fp = &getpagesize;
3   std::cout << (*fp)();
4   ...
5   typedef int (Animal::*memFp)() const;
6   Animal *animal = new Pigeon();
7   memFp memfp = &Animal::age;
8   std::cout << (animal->*memfp)();
```

Figure 3: An example about C++ function pointers.

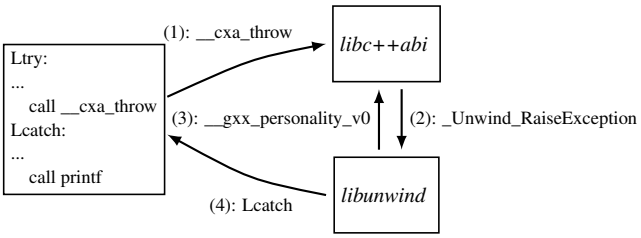

Figure 4: Control transfers during C++ table-based exception handling.

Function pointers. C++ supports two kinds of function pointers: (1) those that point to global functions or static member methods; (2) those that point to non-static member methods. Function pointers in these two kinds have different static types. Their target sets are disjoint and they are handled differently by compilers. Figure 3 shows a code example about the two kinds of function pointers.

Function pointer `fp` is of the first kind. It is assigned to the address of a global function `getpagesize` at line 2. At line 3, the function pointer is invoked via an indirect call (or indirect jump if it is a tail call). To identify its targets, RockJIT adopts a type-matching method that is similar to our previous MCFI work: an indirect branch via a function pointer of type $\tau*$ can target any global function or static member method whose static type is equivalent to τ and whose address is taken in the code.

Function pointer `memfp` at line 7 is of the second kind. The code assumes `Pigeon` is a subclass of `Animal`. According to the C++ semantics, we allow an indirect branch through such a function pointer of type $\tau*$ to target any member method defined in the same class whose type is equivalent to τ and whose address is taken. Further, for each matched virtual member method, we search the class hierarchy to find in derived classes all virtual methods whose types match and add those functions to the target set.

Exception handling. We first discuss how C++ exceptions are handled by LLVM that implements the Itanium C++ ABI. In this ABI, C++ exception handling is a joint work of the compiler, a C++-specific exception handling library such as `libc++abi` and a C++-agnostic stack-unwinding library such as `libunwind`.

When a compiler compiles a C++ program, it emits sufficient information for stack unwinding, since every stack frame needs to be searched to find a matching catch clause for a thrown exception object. Such data is emitted as metadata (e.g., the `eh_frame` and `gcc_except_table` sections in an ELF file) during compilation. Figure 4 depicts the runtime control flow when an exception object is thrown. It assumes `libc++abi` and `libunwind` are used; the control flow would be the same when other libraries are used as long as they obey the Itanium C++ ABI.

The left box in Figure 4 shows some assembly code, where the `Ltry` label starts a C++ `try` statement and `Lcatch` implements a `catch` statement. A C++ `throw` statement is translated to a direct call to `libc++abi`'s `__cxa_throw`, which takes three arguments: the heap-allocated exception object, its type information, and destructor. It performs initialization and invokes `_Unwind_RaiseException` in `libunwind`, which extracts the code address where the exception is thrown and walks through each stack frame by consulting the `eh_frame` section. In each stack frame, `_Unwind_RaiseException` uses an indirect call to invoke a C++-specific routine called `__gxx_personality_v0`. It is defined in `libc++abi` and searches for catch clauses in that frame by consulting `gcc_except_table`. Two cases can happen. If a type-matching catch clause is found in the current frame, then control is transferred to the catch clause via an indirect branch, which we call `CatchBranch`. If a type-matching catch is not found, the stack unwinding should be resumed. However, if there is a clean-up routine that is used to deallocate objects allocated in `try` statements, then the clean-up routine needs to execute before the unwinding continues. It turns out that the same indirect branch (`CatchBranch`) is used to transfer the control to the clean-up routine, but with a different target address.

All control-flow edges in Figure 4, except for the edges out of `CatchBranch`, can be handled using the strategies we have discussed (CHA analysis and the type-matching method). For the `CatchBranch`, our implementation connects it to all catch clauses and cleanup routines. To support separate compilation, RockJIT's modified LLVM compiler emits a table recording addresses of all catch clauses and cleanup routines in each module, and these tables are combined during linking.

If an exception object is caught, but not rethrown, `libc++abi` also invokes the object's destructor, which is registered when calling `__cxa_throw`. The invocation is through an indirect call. Possible targets of this call in a module can be statically computed by tracking `__cxa_throw` invocations. As a result, RockJIT's C++ compiler also remembers these target addresses for each module and combines them at link time.

Global constructors and destructors. The constructors of global and local static objects' are invoked before the main function of a C++ program, and their destructors are called after the main function returns. LLVM handles such cases by generating stub code for each such object. The stub code directly invokes the constructor and registers the destructor using either `__cxa_atexit` or `atexit` defined in `libc`. The addresses of the stub code are arranged in the binary and iterated through by an indirect call (called *CtorCall*) in `libc` before `main`. After `main`, another `libc` indirect call (called *DtorCall*) iterates through the registered destructors to destroy objects. Both CtorCall and DtorCall's targets are statically computable by analyzing the compiler-generated stub code.

Other control-flow features. Return instructions are handled in the same way as MCFI. By analyzing the targets of call instructions, we first construct a call graph. Then a return instruction in a function can return to any address immediately following a call that can invoke the function according to the call graph.

Switch and indirect goto statements are typically compiled to jump-table based indirect jumps; their targets can be statically extracted from read-only jump tables. These indirect jumps are subject to static verification and do not need instrumentation.

Lambda functions are available in C++11, whose related control-flow edges are also supported by our CFG generation. Compilers automatically convert lambda functions to functors, which are classes with `operator()` methods. Therefore, control-flow edges related to the `operator()` methods in such structures can be approximated using our discussed CFG generation method.

Conditions for our C++ CFG-generation method. Our method for CFG generation is largely type based. Indirect calls through a function pointer to a global function is allowed to call any global function whose type matches the function pointer's type. The class hierarchy analysis, which is used to resolve virtual method calls, is also based on static types. As a result, if a C++ program misuses types using features such as arbitrary type casts, then our CFG-generation method may construct a CFG whose edges do not cover all dynamic control flow of the program; enforcement of such a CFG would break the program's execution. On the other hand, we believe our method will not break a C++ program's execution if the following conditions are met: (1) no type cast to or from function pointer types; (2) no C-style type cast or `reinterpret_cast` from or to classes with virtual member methods; (3) no inlined assemblies. These conditions are similar to the ones in MCFI's CFG construction for C programs.

We have built in Clang, LLVM's front-end, a static checker to catch violations of the above conditions in C++ programs. Violations reported by the checker on a C++ program can be straightforwardly fixed to be compatible with our CFG-generation method using the wrapper approach described in MCFI. For V8, which has over 555,000 lines of code, we modified only 10 lines of code using the wrapper approach to make it compatible with our CFG-generation method. We also tried our approach on the seven C++ programs in SPECCPU2006 as well as `libc++`, `libc++abi`, and `libunwind` for a total over 620,000 lines of code, only 35 lines of code (all in SPECCPU2006 benchmark 453.povray) need to be changed to generate CFGs using our method. In addition, all the generated CFGs have been tested on data sets that come with those benchmarks and the results are summarized in the appendix.

CFG statistics for C++ programs. Table 1 shows CFG generation statistics for V8 and the C++ benchmarks in SPECCPU2006. They are compiled with the `O3` optimization[3] and are statically linked with dependent libraries including `libc++`, `libc++abi`, `libunwind`, and MUSL `libc`. For each program, the table lists its source lines of code (SLOC) and the number of indirect branches (IB). The table also presents statistics for the CFGs generated using RockJIT's CFG-generation method. The column IBT lists the number of indirect-branch targets in the CFGs. It is the number of functions whose addresses are taken, plus the number of return addresses, plus the number of catch clauses and clean-up routines.

The column EQC presents the number of *equivalence classes* of addresses in the CFG. RockJIT follows the original CFI [3] of using equivalence classes: two addresses are equivalent if there is an indirect branch that can jump to both targets according to the CFG. If the target sets of two indirect branches are not disjoint in the CFG, then the two sets are merged into one equivalence class and the two indirect branches are allowed to jump to any target in the equivalence class. This process results in some loss of CFG precision. However, as we can see from Table 1, V8 still has over 10k equivalence classes of target addresses (note that EQC is upper bounded by IB). This is much stronger than coarse-grained CFI, which enforces only one or several equivalence classes.

7. EVALUATION

We have evaluated RockJIT's security and performance using Google's V8 JavaScript engine. We conducted all experiments on

[3]On V8, tail call optimization is turned off for more equivalence classes of return instructions and yet performance overhead is negligible.

Program	SLOC	IB	IBT	EQC
V8	555,383	34,279	100,497	10,452
444.namd	3,886	598	4,694	287
447.dealII	94,384	11,426	58,930	2,529
450.soplex	28,277	4,554	17,944	1,387
453.povray	78,705	2,247	15,477	1,048
471.omnetpp	19,991	5,672	30,781	1,494
473.astar	4,280	544	3,813	293
483.xalancbmk	267,399	27,397	94,103	6,490

Table 1: CFG statistics for V8 and SPECCPU2006 C++ benchmarks.

a system with x86-64 Ubuntu 14.04, an Intel Core i7-3770 CPU, and 8GB physical memory. All programs tested were compiled to 64-bit binaries at optimization level three.

7.1 Security evaluation

By enforcing fine-grained CFI on V8's code, RockJIT improves its security. ROP attacks are restricted in terms of both the available gadgets and how gadgets can be chained to form an attack. To test the first aspect, we used a ROP-gadget finding tool, `rp++`[4], to find the number of unique gadgets that can be found in the native V8 and RockJIT-hardened V8. In the hardened V8, a potential gadget has to start at a valid indirect branch target (e.g., a return address). Our result shows that RockJIT can eliminate nearly 98.5% gadgets from V8's code base. We also tried the tool on those SPEC C++ benchmarks and RockJIT can eliminate 98.3% gadgets from those benchmarks. A caveat about these numbers is that they depend on a specific gadget-finding tool. Other tools might use different definitions of gadgets.

Fine-grained CFI further improves security by eliminating many more functionality-irrelevant control-flow edges and therefore restricting how gadgets can be chained. The execution of gadget g can be followed by only those gadgets whose start addresses can be targeted by the indirect branch at the end of g. By contrast, coarse-grained CFI allows gadget g to be followed by all other gadgets (assuming only one equivalence class is enforced). As a concrete comparison, the following table lists the number of edges for indirect branches in NaCl-JIT V8's CFG, which enforces coarse-grained CFI, and the number in RockJIT V8's CFG.

V8 defenses	NaCl-JIT V8	RockJIT V8
Total # of indirect-branch edges	7,976,474,777	2,051,600

Since NaCl-JIT allows an indirect branch to target any 32-byte aligned address, the number of indirect-branch edges for NaCl-JIT V8 is computed by $\mathrm{IB} * \mathrm{CodeSize}/32$, where IB is the number of indirect branches in V8 and CodeSize is the size of V8's code. The number for RockJIT V8 is computed by summing the out degrees of indirect-branch nodes in its CFG.

As we can see, RockJIT eliminates 99.97% more edges for indirect branches in V8 compared to NaCl-JIT. Therefore, we believe fine-grained CFI improves security significantly. However, the security evaluation of fine-grained CFI needs much further investigation. It would be ideal to have a metric for fine-grained CFI that is directly related to security than the number of control-flow edges; the topic goes beyond the scope of this paper.

7.2 Performance evaluation

RockJIT incurs performance overhead because of inlined checks and JITted code verification. We ran RockJIT-hardened V8 on the Octane 2 JavaScript benchmarks[5] and measured the performance overhead. Octane 2 consists of 17 tests, which measure different aspects of a JavaScript compiler, from speed of bit operations to compiler latency. Each benchmark was run ten times and the variance was less than 2%. Figure 5 presents the performance overhead of the benchmarks. As a summary, RockJIT imposes 14.6% average overhead over all tests and the maximum overhead is 36.4%.

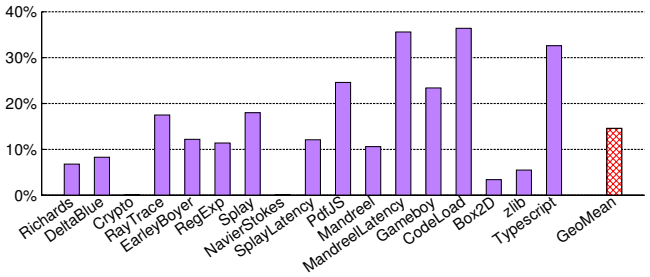

Figure 5: Performance overhead imposed by RockJIT-hardened V8 on Octane 2 benchmarks.

Note that RockJIT's performance overhead varies over different benchmarks. Through V8's internal performance profiler, we found that, the less frequently a benchmark's execution stays in the JITted code, the more overhead RockJIT tends to incur on the benchmark. The reason is that the density of indirect branches in V8's runtime is roughly four times the density of indirect branches in its JITted code. For example, the NavierStokes benchmark (as well as Crypto) reports nearly zero performance overhead, mainly because 96% of its execution is in the JITted code. In the JITted code, over 96% execution is in optimized loops that iterate through numeric arrays, during which indirect branches and indirect memory writes are rare. As an example of the other extreme, the CodeLoad benchmark reports the largest overhead of 36.4%. The benchmark measures the compilation latency and nearly 98% of execution is performed on the V8 runtime for compilation. Other benchmarks with big JavaScript code base such as the PdfJS, Mandreel and Typescript also spend great portions of time in the V8 runtime, incurring relatively large overhead.

In general, RockJIT's performance overhead are due to three major contributors: the inlined checks in V8 runtime's code, the inlined checks in JITted code, and verification. The following table shows the performance overhead of each contributor over Octane 2 benchmarks. These overheads were generated by disabling factors one at a time. Since they are not independent, these overheads cannot be simply added.

Aspects	V8 Runtime Checks	JITted Code Checks	Verification
Overhead	4.3%	8.4%	1.7%

NaCl-JIT and librando tested overheads on a subset of the Octane 2 benchmarks. NaCl-JIT incurs about 51% overhead on average, and librando 265.8%. On the same subset, RockJIT incurs 9.0% overhead. NaCl-JIT is slower mainly because of the following reasons: (1) NaCl-JIT emits many more no-ops for 32-byte alignment, whose execution consumes roughly 37% extra time; RockJIT requires that indirect branch addresses are four-byte aligned and does not insert many no-ops; (2) RockJIT's DFA-based verification (1.7% overhead) is faster than NaCl's disassembly-based verification (5% overhead).

[4] `https://github.com/0vercl0k/rp`

[5] `https://developers.google.com/octane/benchmark`, revision 33.

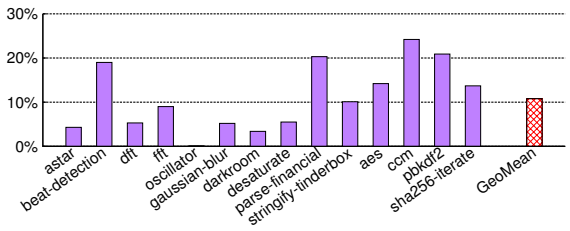

Figure 6: Performance overhead imposed by RockJIT-hardened V8 on Kraken 1.1 benchmarks.

We also tested RockJIT-hardened V8's performance on Mozilla Kraken benchmarks[6], and the results (in Figure 6) show that on the average of ten runs RockJIT incurs 10.8% overhead with less than 1% variance, similar to the results on Octane 2.

Code size increase. With all the libraries linked, the code of the RockJIT-hardened V8 is 37.5% larger than its native counterpart, due to the CFI checks. The execution of all Octane 2 benchmarks generates around 9.9% more code in the code heap than the native V8. The reason why the JITted code has less code-size increase is that it uses less indirect branches. By contrast, V8's code uses virtual method calls heavily.

8. CONCLUSIONS AND FUTURE WORK

We have presented RockJIT, a general approach to securing JIT compilers. RockJIT enforces fine-grained CFI on the JIT compiler and coarse-grained CFI on the JITted code, resulting in much improved security and lower performance overhead than other state-of-the-art systems. The benefits of RockJIT have been empirically demonstrated by a RockJIT-hardened JavaScript compiler. We believe RockJIT can greatly raise the bar of mounting attacks on JIT compilers. As future work, we plan to: (1) port more JIT compilers such as HotSpot, HHVM, and LuaJIT to RockJIT; (2) formally prove the correctness of the verifier following the approach of RockSalt [24]; and (3) integrate RockJIT-hardened V8 engine into Chromium and harden all components in Chromium by MCFI and comprehensively evaluate the security, performance, and engineering cost of fine-grained CFI.

9. ACKNOWLEDGMENTS

We thank anonymous reviewers for their helpful comments. This research is supported by US NSF grants CCF-1217710 and CCF-1149211, China NNSF grant 61272086, and a research award from Google.

10. REFERENCES

[1] Google V8 vulnerabilities. `http://www.cvedetails.com/product/17734/Google-V8.html?vendor_id=1224`.

[2] Itanium c++ abi. `http://mentorembedded.github.io/cxx-abi/abi.html`.

[3] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-Flow Integrity. In *12th ACM Conference on Computer and Communications Security (CCS)* (2005), pp. 340–353.

[4] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing Memory Error Exploits with WIT. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (May 2008), pp. 263–277.

[6] `http://krakenbenchmark.mozilla.org/`

[5] ANSEL, J. Personal communication, March 2014.

[6] ANSEL, J., MARCHENKO, P., ERLINGSSON, Ú., TAYLOR, E., CHEN, B., SCHUFF, D., SEHR, D., BIFFLE, C., AND YEE, B. Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (2011), pp. 355–366.

[7] BADISHI, G. JIT Spraying Primer and CVE-2010-3654. `http://badishi.com/jit-spraying-primer-and-cve-2010-3654/`, 2012.

[8] BLAZAKIS, D. Interpreter Exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies* (2010), WOOT'10, USENIX Association, pp. 1–9.

[9] CARLINI, N., AND WAGNER, D. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security 14)* (Aug. 2014), USENIX Association.

[10] CHEN, P., FANG, Y., MAO, B., AND XIE, L. JITDefender: A Defense against JIT Spraying Attacks. In *26th IFIP International Information Security Conference* (2011), vol. 354, pp. 142–153.

[11] CHEN, P., WU, R., AND MAO, B. JITSafe: A Framework Against Just-In-Time Spraying Attacks. *Information Security, IET 7*, 4 (December 2013), 283–292.

[12] DAVI, L., DMITRIENKO, R., EGELE, M., FISCHER, T., HOLZ, T., HUND, R., NURNBERGER, S., AND SADEGHI, A. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Network and Distributed System Security Symposium (NDSS)* (2012).

[13] DAVI, L., LEHMANN, D., SADEGHI, A., AND MONROSE, F. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)* (Aug. 2014), USENIX Association.

[14] DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming* (1995), ECOOP '95, pp. 77–101.

[15] DEUTSCH, L. P., AND SCHIFFMAN, A. M. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1984), POPL '84, ACM, pp. 297–302.

[16] ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. XFI: Software Guards for System Address Spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 75–88.

[17] FREDKIN, E. Trie Memory. *Communications of ACM 3*, 9 (Sept. 1960), 490–499.

[18] GOKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out Of Control: Overcoming Control-Flow Integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on* (May 2014).

[19] HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, P. America, Ed., vol. 512 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1991, pp. 21–38.

[20] HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (1992), PLDI '92, ACM, pp. 32–43.

[21] HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Librando: Transparent Code Randomization for Just-In-Time Compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security* (2013), CCS '13, ACM, pp. 993–1004.

[22] JANG, D., TATLOCK, Z., AND LERNER, S. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *20th Annual Network and Distributed System Security Symposium* (2014), NDSS '14, The Internet Society.

[23] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (2006), USENIX-SS'06, USENIX Association.

[24] MORRISETT, G., TAN, G., TASSAROTTI, J., TRISTAN, J., AND GAN, E. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2012), PLDI '12, ACM, pp. 395–404.

[25] NIU, B., AND TAN, G. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013), CCS '13, ACM, pp. 199–210.

[26] NIU, B., AND TAN, G. Modular Control-Flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), PLDI '14, ACM, pp. 577–587.

[27] PEWNY, J., AND HOLZ, T. Control-Flow Restrictor: Compiler-based CFI for iOS. In *ACSAC '13: Proceedings of the 2013 Annual Computer Security Applications Conference* (2013).

[28] SEABORN, M. A dfa-based x86-32 validator for native client. `https://github.com/mseaborn/x86-decoder`, 2011.

[29] SEHR, D., MUTH, R., BIFFLE, C., KHIMENKO, V., PASKO, E., SCHIMPF, K., YEE, B., AND CHEN, B. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th Usenix Security Symposium* (2010), pp. 1–12.

[30] SINTSOV, A. Safari JS JITed Shellcode. `http://www.exploit-db.com/exploits/14221/`, 2010.

[31] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)* (Aug. 2014), USENIX Association.

[32] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (1993), SOSP '93, ACM, pp. 203–216.

[33] WANG, Z., AND JIANG, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on* (May 2010), pp. 380–395.

[34] WEI, T., WANG, T., DUAN, L., AND LUO, J. INSeRT: Protect Dynamic Code Generation against Spraying. In *Information Science and Technology (ICIST), 2011 International Conference on* (March 2011), pp. 323–328.

[35] WU, R., CHEN, P., MAO, B., AND XIE, L. RIM: A Method to Defend from JIT Spraying Attack. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on* (Aug 2012), pp. 143–148.

[36] YEE, B., SEHR, D., DARDYK, G., CHEN, J., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Security and Privacy, 2009 IEEE Symposium on* (May 2009), pp. 79–93.

[37] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *Security and Privacy (SP), 2013 IEEE Symposium on* (May 2013), pp. 559–573.

[38] ZHANG, M., AND SEKAR, R. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22Nd USENIX Conference on Security* (2013), SEC'13, USENIX Association, pp. 337–352.

APPENDIX

We measured MCFI's performance overhead on SPECCPU2006 C++ benchmarks. The experiments were conducted in the same environment as mentioned in Sec. 7. All benchmark programs and their dependent libraries were compiled with the `O3` optimization. The results are averaged over three runs and presented in the following figure. The x86-32 and x86-64 bars are results of benchmarks compiled with `-m32` and `-m64` compiler options, respectively. As can be seen, MCFI incurs around 6.8%/15.7% (average/-maximum) performance overhead on C++ benchmarks.