librando: Transparent Code Randomization for Just-in-Time Compilers

Andrei Homescu
University of California, Irvine
Irvine, CA, USA

Stefan Brunthaler
University of California, Irvine
Irvine, CA, USA

Per Larsen
University of California, Irvine
Irvine, CA, USA

Michael Franz
University of California, Irvine
Irvine, CA, USA

ABSTRACT

Just-in-time compilers (JITs) are here to stay. Unfortunately, they also provide new capabilities to cyber attackers, namely the ability to supply input programs (in languages such as JavaScript) that will then be compiled to executable code. Once this code is placed and marked as executable, it can then be leveraged by the attacker.

Randomization techniques such as constant blinding raise the cost to the attacker, but they significantly add to the burden of implementing a JIT. There are a great many JITs in use today, but not even all of the most commonly used ones randomize their outputs.

We present librando, the first comprehensive technique to harden JIT compilers in a completely generic manner by randomizing their output transparently ex post facto. We implement this approach as a system-wide service that can simultaneously harden multiple running JITs. It hooks into the memory protections of the target OS and randomizes newly generated code on the fly when marked as executable.

In order to provide "black box" JIT hardening, librando needs to be extremely conservative. For example, it completely preserves the contents of the calling stack, presenting each JIT with the illusion that it is executing its own generated code. Yet in spite of the heavy lifting that librando performs behind the scenes, the performance impact is surprisingly low. For Java (HotSpot), we measured slowdowns by a factor of 1.15×, and for compute-intensive JavaScript (V8) benchmarks, a slowdown of 3.5×. For many applications, this overhead is low enough to be practical for general use today.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Code generation*; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Languages, Security, Performance

CCS'13, November 4–8, 2013, Berlin, Germany.

ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516675.

Keywords

JIT compilers, diversification, security, JIT spraying, code reuse attacks, return-oriented programming, binary rewriting, randomization

1. MOTIVATION

Our computing infrastructure depends on high performance delivered by just-in-time (JIT) compilers to a large degree. Efficiently executing JavaScript is a prerequisite for complex Web 2.0 applications. Similarly, Java's success rests on performance delivered by efficient dynamic code generation. From the early beginnings, JIT compilers had to focus on producing code quickly. Usually, they achieve this by optimizing the common case and forgoing time-intensive optimizations altogether. As a result, this leads to highly predictable code generation, which attackers exploit for malicious purposes. This is evidenced by the rising threats of *JIT spraying* [2] and similar attacks on sandboxes in JITs. The former is particularly interesting: JIT spraying relies on JIT compilers emitting constants present in input source code directly into binary code. Due to variable length encoding, attackers can encode and subsequently divert control flow to arbitrary malicious code arranged this way.

This attack vector is innate and specific to JIT compilers. From a security perspective, the state-of-the-art in the field is to address JIT spraying by encrypting and decrypting constants. This addresses the code injection part of JIT spraying, but attackers can fall back on code-reuse techniques. Specifically, return-oriented programming [26] for JIT compiled code is problematic. Instead of finding gadgets in statically generated code (as they would do an a generic return-oriented programming attack), an attacker uses the JIT compiler to create new binary code containing the necessary gadgets by supplying specially crafted source code. The ubiquity of JIT compilers amplifies this security risk to such a degree that JITs become a liability.

Recent work [9, 14, 11, 10, 22, 29] addresses code-reuse attacks by attacking its foundation: the software monoculture. By diversifying the binary code, attackers cannot construct reliable attack code, because the binary code layout differs for each end-user. Consequently, diversity increases the costs for attackers, ultimately rendering them too costly. Unfortunately, existing approaches to artificial software diversity do not protect dynamically emitted code from a just-in-time compiler. We address this challenge by describing the first fully automatic technique to diversify existing JIT compilers in a black-box fashion. Similar to the successful frontend-backend separation in traditional compilers, our proposed black-box

approach has the advantage over a white-box solution—where developers would manually add diversification directly to JIT compiler source code—of not requiring duplicated work for every existing JIT compiler. Besides the obvious savings in implementation time, the black-box approach allows for faster time-to-market for patches, without having to rely on vendors to supply patches to known vulnerabilities. Another benefit is the added security for legacy JIT compilers available only in binary form, where extra defenses cannot be added by changing the source code.

Summing up, we make the following contributions:

- We present librando, the first automated software diversity solution for hardening existing JIT compilers in a black box fashion. Our solution implements two popular defensive techniques: NOP insertion and constant blinding.
- We describe two optimizations (the *Return Address Map* and optional white box diversification—taking advantage of compiler cooperation) to improve the performance of librando.
- We demonstrate applicability of black box diversification on two pervasive industrial-strength JIT compilers: Oracle's HotSpot (used in the Java Virtual Machine) and Google's V8 (used in the Chrome web browser). We then report the results of our analysis of librando performance. We show that we successfully protect:

  HotSpot (a JIT compiler for Java—a statically-typed language) with an overhead of 15%.

  V8 (a JIT compiler for JavaScript—a dynamically-typed language) with a slowdown of 3.5×.

2. BACKGROUND

A JIT compiler transforms a program written in a high-level language (HLL), generating native code at program run-time. The compiler emits native code into a code cache, after parsing and optimizing HLL source code. The compiler itself is written in another programming language, which we call the host language. JIT compilers also contain a language runtime, which is a library of functions that are written in the host language and provide or manage access to system resources. Some examples of such resources are files, networks, operating system threads and complex data structures (maps, trees). When compiling a HLL program, the JIT compiler emits native calls into the runtime whenever the program uses these resources. Figure 1 shows a high-level structure of a JIT compiler and its interactions with the generated code. After emitting all or part of the native code (usually enough code to start execution of the program), the compiler branches to the entry point of the HLL program. The generated code continues execution, calling into other generated functions or the language runtime. The HLL program continues until termination, making repeated calls into the runtime whenever needed.

From a security point of view, JIT compilers have one characteristic that is important in our context: predictability. As JIT compilers usually optimize code for performance, there are only a few optimal translations of HLL code to native code, and a JIT compiler emits one of these. When presented with the same HLL code many times repeatedly, a compiler will emit the same native code; attackers can use this characteristic to their advantage. This is not a problem specific to JIT compilers, but compilers in general; however, predictability of JIT compilers has not been fully explored.

JIT spraying [2] is one recent attack that relies on predictability of JIT compilers. This attack is a form of code injection targeted at dynamically generated code. In its original form, it relies on

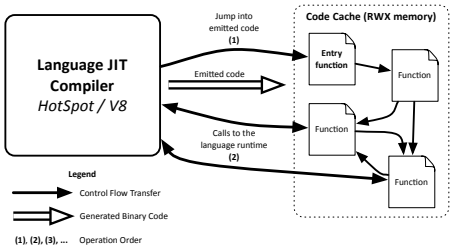

Figure 1: High-level structure of a JIT compiler.

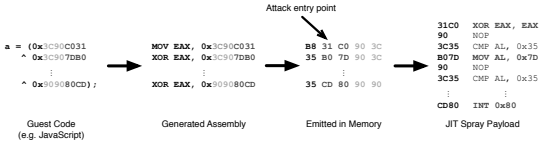

Figure 2: JIT spraying example. The 32-bit constants from HLL code (shown in light and dark grey) appear inside native code, in little-endian form.

one unintended behavior of many JIT compilers: HLL program constants reach native code unmodified, therefore becoming part of the executable code. The attacker injects short sequences (32-bit constants in the original paper), and later jumps to the injected sequence through a separate attack vector. Figure 2 shows an example of code injected using constants. For the attack to work, the attacker must also predict the remaining bytes inserted between the controlled sequences, and use those bytes as part of the payload; this is often possible in practice, due to the predictability of the compiler. This allows the attacker to execute arbitrary native code, even when running on a compiler that runs the generated code in a sandbox (with restricted access to memory, for example).

For many years, most arbitrary code execution attacks used the same method of gaining control of the program: code injection attacks. To prevent these, most operating systems now forbid the same page to be both writable and executable at the same time. Sidestepping this measure, a new class of attacks against applications surfaced and gained popularity: code reuse attacks. Instead of adding new executable code to an application, code reuse attacks locate reusable code sequences inside the application, then thread these sequences into a program written by the attacker. Shacham described one of the first versions of this attack, called Return-Oriented Programming (ROP) [26]; he named the code sequences *gadgets*. A gadget is simply a valid sequence of binary code that the attacker can execute successfully (the gadget decodes correctly and does not contain invalid instructions); a gadget can start anywhere inside the generated code (including in the middle of a proper instruction) and spans one or more of the original instructions emitted by the compiler. ROP uses only gadgets that end in a `RET` instruction (encoded by the `C3` byte on x86); the attacker places addresses of gadgets on the stack on consecutive stack slots, so that each gadget proceeds to the next one using a return. Later work [3] extends this idea to other indirect branch instructions.

This attack is even more potent in the presence of a JIT compiler, as an attacker that controls HLL code can emit an arbitrary amount of native code containing gadgets (by emitting as much HLL code as needed to generate all the gadgets for the attack); Figure 3 illustrates

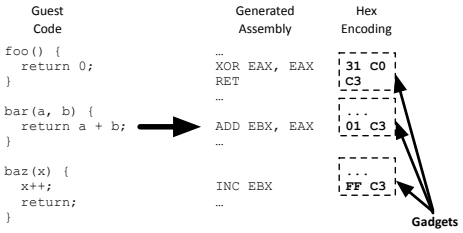

Figure 3: Code reuse example. The compiler transforms HLL code into native code containing ROP gadgets. The `C3` byte encodes the `RET` instruction at the end of a gadget.

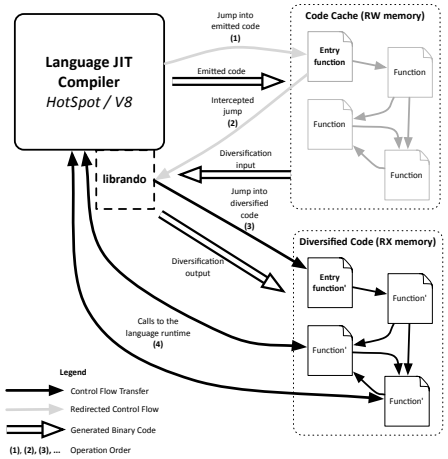

Figure 4: A JIT compiler with librando attached, with the control flow graph of the dynamically generated program. Greyed out edges represent branches that are redirected or never taken after diversification.

how HLL code sequences become gadgets. For example, this can be a problem for web browsers that include a JavaScript compiler, as many web pages include JavaScript code from unreliable (or hostile) sources. Another problem is that current anti-ROP defenses [16, 20, 22, 10, 14, 11] target ahead-of-time compilers or binary rewriters, but do not offer protection to dynamically generated code.

3. DESIGN

The librando library diversifies code generated by a JIT compiler. It reads all code emitted by the compiler, dissassembles the code, randomizes it, and then writes the randomized output to memory. Figure 4 shows the structure of dynamically generated code, as a function call graph.

The library diversifies dynamically-generated code under one of the following models (illustrated in Figure 5):

Black box diversification with no assistance from the compiler (the compiler is a black box and the library has no knowledge of compiler internals). The library attaches to the compiler and intercepts all branches into and out of dynamically-generated code, without requiring any changes to compiler internals.

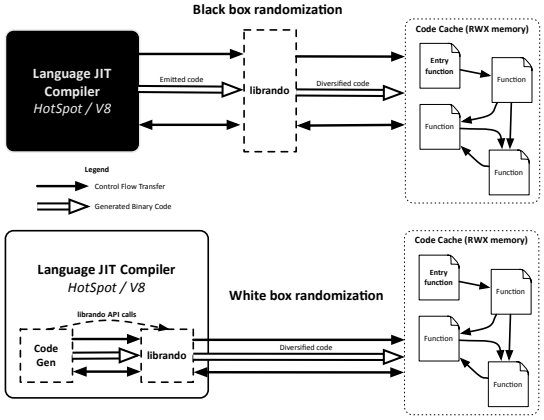

Figure 5: Black and white box diversification architecture.

White box diversification with some assistance from the compiler (the library has some knowledge of compiler internals). The code emitter notifies librando through an API when it starts running undiversified code. The library provides the diversified code addresses to the compiler, and the compiler executes diversified code directly. We change all compiler branches into emitted code to use the addresses returned by librando. This approach is intended as a middle ground between the previous model and a manual implementation of randomization for each compiler, and it requires that compiler source code is available.

As the first security measure, librando prevents execution of all code generated dynamically by the compiler. Instead, the library disassembles the code into a control flow graph, diversifies every basic block in this graph, and then writes the diversified blocks to a separate executable area. All branches (including function calls and returns) to the original undiversified code are redirected to the diversified code (Figure 6 shows an example of a diversified block). While the undiversified code remains available and writable, the compiler or HLL program cannot execute it anymore. The library intercepts all memory allocation functions (the `mmap` function family on Linux) that return executable memory, then removes the executable flag on all intercepted allocation requests. The library also keeps an internal list of all memory allocated by these requests, and uses this list to distinguish between accesses to undiversified code and all other memory accesses.

In the black box diversification model, the library transparently intercepts all branches to protected blocks. To do this, we protect all undiversified code pages against execution, then catch all attempts to execute these protected pages. The library installs a handler for the segmentation fault signal (`SIGSEGV` in Linux), which is raised whenever a processor instruction attempts to access memory it does not have permission for. Whenever the processor attempts to execute a non-executable page, it triggers a page fault in the MMU. The operating system handles this fault and calls our `SIGSEGV` signal handler. The library then redirects execution to diversified code.

While static disassembly of binary code accurately is impossible in the general case [6] (due to the need to distinguish code from data), dynamic disassembly is much simpler for one reason: we only

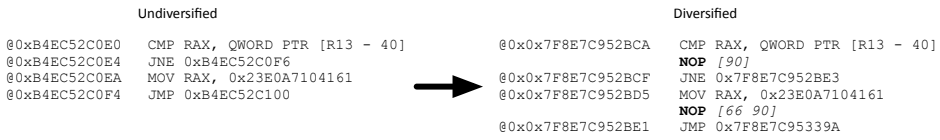

Figure 6: Block contents and diversification example.

disassemble bytes that we are certain represent code. Our use of signals guarantees this: the signal handler is always called when the JIT compiler executes an undiversified instruction at some address, so the bytes at that address (and the entire block starting at that address) are guaranteed to be code. The same is true for all blocks in the control flow graph containing that instruction, since we follow direct branches to find more code.

We make a few simplifying assumptions about the emitted code that reduced our implementation effort significantly:

No stack pointer reuse On the x86 architecture, the stack pointer register (`RSP`) is available as a general-purpose register. The x86 64-bit ABI reserves this register for its intended purpose, as stack register. However, compilers only need to follow the ABI when calling into external libraries and system code; they can ignore its guidelines inside emitted code. Code emitted by a JIT compiler might use another register to keep track of the HLL stack, and re-use `RSP` for another purpose. Generally, JIT compilers do not do this in practice, so we assume that this register always points to the top of a valid native stack. This allows both code emitted by the compiler and by librando to use several stack manipulation instructions, such as `PUSH`, `CALL` and `RET`.

No self-modifying code While it is possible for a JIT compiler to emit code that modifies itself, this is usually not the case. We assume that only the compiler can modify code. Once emitted code starts executing, it remains unchanged. This assumption simplifies our implementation, as we only have to detect code changes originating in the compiler itself; therefore, we can safely discard old versions of modified code, without the risk of having to continue execution inside discarded code.

Calls paired with returns Compilers often use the `CALL` x86 instruction for calls and `RET` for the corresponding returns, but not always. The former pushes the return address on the native stack and branches to a function, while the latter pops the return address and branches to it. There are equivalent instruction sequences with the same behavior which a compiler can use instead, perhaps to push/pop the return address to another stack. However, there is a performance penalty from using these sequences, as modern processors use an internal return address stack to improve branch prediction for `CALL`/`RET` pairs, and only for those pairs. A compiler optimized for performance always uses this combination for function calls, simplifying our implementation as well. Consequently, librando only needs to analyze and rewrite these instructions when redirecting function calls.

Our technique diversifies code by relocating emitted code blocks, rewriting the code, and inserting instructions. To preserve HLL program semantics, a diversification library must be transparent to both the compiler and the compiled program. We identified several pieces of program state which librando must preserve when rewriting code (related work [4, 17, 25] identifies a superset of these for general-purpose dynamic binary rewriters):

Processor register contents including the flags register. The library inserts instructions that hold intermediate values in registers. Also, some inserted instructions (like `ADD` and `XOR`) modify the processor flags register. The library attempts to only add instructions that do not change any registers or flags; where this is not possible, it temporarily saves the register(s) to the stack, performs the computation, then restores the register(s).

Native stack contents Some JIT compilers use the native stack for HLL code, while others switch to a separate stack when branching to HLL code. To support the former, the library must preserve all contents of the native stack inside diversified code. This must be true not only during execution of dynamically generated code, but also during calls into the runtime, as the runtime itself may read or write to the native stack. For example, the V8 runtime walks the native stack during garbage collection to find all pointers to data and code. Changing any such pointer or any other data on the stack leads to program errors and crashes. Therefore, native stack contents must be identical when running with and without librando. This includes return addresses; when a diversified function calls another diversified function, the former must push the undiversified address on the stack. Figure 7 illustrates this transformation. The original call pushes the address of the next instruction on the stack, then jumps to the called function. We replace it with a `PUSH`/`JMP` pair that pushes the undiversified return address. There is one exception to this restriction: we consider any memory past the top of the stack (below `RSP`) to be unused, so we use it to save registers.

Undiversified code instructions JIT compilers frequently emit code with temporary placeholders, then later replace them with other instructions. The compiler usually uses fixed instruction sequences for these placeholders, so it first checks their contents before patching. In other cases, certain instruction sequences are used to flag properties of emitted code. The library cannot modify any of the original, undiversified code in-place, as it cannot distinguish between regular code and these placeholders. Even if the compiler never reads back the former, it may read back and validate the contents of the latter, then crash or execute incorrectly. The library is required to preserve the undiversified code at all times, as originally emitted by the compiler.

POSIX signals Some JIT compilers (such as HotSpot) intercept POSIX signals and install their own handlers. We intercept the `SIGSEGV` signal to catch execution and write attempts on undiversified code, but pass all other signals (including `SIGSEGV` we do not handle) back to the JIT compiler.

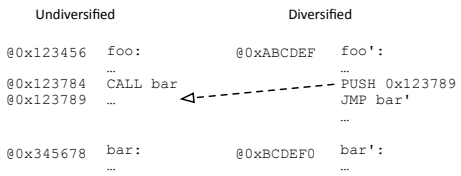

Figure 7: Rewriting the `CALL` instruction.

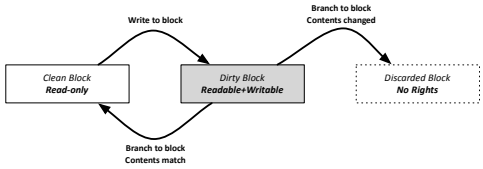

Figure 8: Per-block finite state machine that handles block changes.

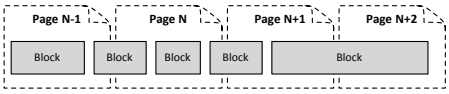

Figure 9: Blocks spanning several memory pages and crossing page boundaries.

3.1 Code Relocation

In our control flow graph, a block is a maximal continuous run of instructions, so that the block ends in an unconditional branch instruction, but contains no such instruction inside. We also break blocks at function calls (the `CALL` instruction) and returns, but not at conditional branches; a basic block can have one or more conditional branches inside. However, we use one heuristic to split blocks: whenever a block contains a number of consecutive zeroes over a given threshold, we break the block after a small number of those zeroes. This is needed because compilers sometimes emit code only partially (or lazily), initializing the remainder with zeroes. The compiler emits the rest of the code at a later moment, after some event triggers generation of the missing code. We implemented this heuristic to support the V8 compiler, which uses lazy code generation.

Many JIT compilers perform garbage collection on generated code, discarding unused code and reusing that space for new code. The compiler will write and later execute this new code in place of the old version. This happens frequently in the modern JIT compilers we investigated. We detect such changes, discard the diversified versions of old blocks, then read the new blocks and integrate them into the existing control flow graph. To detect all changes to a block, librando marks it read-only after diversification using the `mprotect` function on Linux. If the compiler writes to the block later, the library allows the write to succeed, but marks the block as dirty.

The `mprotect` function has one significant restriction: it can only change access rights on zones aligned to hardware pages; on x86, a page is 4096 bytes by default. There are several issues to consider when using `mprotect` to protect a block. First, a single page may contain more than one block. Second, a block may be spread across several consecutive pages. Third, for each page, one or two blocks might cross its boundaries, one at each end. Figure 9 shows examples of all three cases. Instead of marking a block as read-only, librando actually marks all pages containing that block as read-only. However, those pages may contain other blocks that the compiler might never modify.

After librando marks a page as read-only, the operating system starts notifying the library of all writes to read-only pages. We also use POSIX signals to receive these notifications. The operating system calls the `SIGSEGV` signal handler each time a processor instruction tries to write to a protected page, as long as the page is protected. However, this occurs before the actual instruction writes the data; we have to either emulate the instruction itself (while keeping the page read-only), or make the page writable and allow the original instruction to execute. As emulating x86 instructions correctly requires significant development effort, we choose the latter solution. However, once we allow writes to a page, there is no way to catch each write to that page separately; the processor will allow all writes to succeed without generating a page fault. At this point, the only information available to librando is that at least one byte in that page may have been modified, but nothing more. Therefore, we mark all blocks contained in a writable page as dirty. The next time librando intercepts a branch to undiversified code, it checks all dirty blocks to verify that the VM has actually modified their contents. The library only discards and re-diversifies changed blocks, keeping unchanged blocks as they are. Figure 8 shows this process per block, in the form of a finite state machine.

To check whether a block has been modified, the library compares the new contents of the block against its original contents. However, this requires that two copies of each block be stored: the original and current code. To save memory space, we do not store both copies in memory; instead, each block stores a hash code of its original contents. Every time librando checks if a block has been modified, it hashes the current contents of the block and compares the hash code to the one stored in the block. If the hash codes match, librando assumes that the block has not been changed; otherwise, it discards the block. While there is a very small possibility of collision (where the contents of a block change, but the hash remains the same), we performed all our experiments and benchmarks successfully with this optimization. A librando user has the option of disabling hashing and verifying the entire contents of blocks, which guarantees correctness.

The user attaches librando to a JIT compiler in one of several ways: either by linking it (statically or dynamically) to the compiler, or through the `LD_PRELOAD` environment variable on Linux. The latter mechanism preloads a library into a process at program start-up time, allowing the library to override some of the program's symbols. We used the latter approach in our evaluation, but the former is preferred in an actual deployment for increased security.

3.2 Diversification

The main security benefits from librando come from rewriting the generated code. We propose two rewriting techniques that harden the generated code against attacks.

NOP Insertion. Code reuse attacks rely on the code having predictable location (address) and contents. We implement one fine-grained instruction-level code layout randomization technique: *NOP insertion*. To randomize code layout, we randomly insert NOP instructions (instructions without effects) into the diversified blocks, between the existing instructions. This technique has been used successfully in other work to change instruction or block align-

Data: Input block B. Probability p of NOP insertion.
Result: Block B with NOPs inserted.
begin
  for *each instruction* $i \in B$ **do**
    Pick a random real number in $[0.0, 1.0]$
    $v \leftarrow random()$
    if $v < p$ **then**
      $i \rightarrow insert(randomNOP())$
    end
  end
end

Algorithm 1: NOP insertion algorithm.

ment to improve performance [12], security [18, 31, 14, 11], or provide contention mitigation [28]. NOP insertion pushes each proper instruction forward by a random offset (the total length of all preceding inserted NOPs), making the location of each instruction more difficult to predict. Since the total length of NOPs accumulates as more instructions are added, uncertainty of code addresses increases as more code is generated. This technique makes it difficult for an attacker to predict not only addresses of known code, but also distances between known locations. The attacker might try to learn the address of some known code object, and access all other code relatively to this address. This requires that the attacker know relative locations of code in advance; NOP insertion makes this much less likely, since NOPs displace instructions randomly.

We implemented the algorithm (shown in Algorithm 1) as a single linear-time pass over the instructions in each block. After each proper instruction, the algorithm decides whether to insert a NOP or not by coin toss. In our tests, we set the probability of NOP insertion to $p = 0.5$, but this can be adjusted for better performance or more security. If the algorithm inserts a NOP, the next step picks a NOP randomly from a set of candidates. We used the smallest three NOP instructions from the set of canonical NOPs recommended by the Intel architecture manual [13]: `90`, `66 90`, and `0F 1F 00`.

Constant Blinding. We previously described JIT spraying as an example of a code injection attack against JIT compilers. In general, these attacks rely on the compiler emitting native code that contains some binary sequence from the source-program as-is. In the particular case of JIT spraying, this sequence is a set of 32-bit constants emitted as immediate operands to x86 instructions. Recently, JavaScript compilers have started to implement their own defensive measures particular to this attack [24]. There are two ways that a HLL program value winds up in the executable code region: the compiler stores the value either close to the code (without executing it as code), or as an immediate instruction operand. In the former case, NOP insertion shifts the location of the value by a random offset, making its location hard to guess. For the latter case, there is one simple solution based on obfuscation: emit each immediate operand in an encrypted form, then decrypt its value at run-time using a few extra instructions. We pick a random value (a *cookie*) for every operand, blind the operand using the cookie, emit the original instruction with the blinded immediate, then emit the decryption code. While other implementations use an XOR operation for encryption, we do not use it since it alters the arithmetic flags, so librando would have to save them. Fortunately, the processor provides an addition in struction that leaves the flags intact: `LEA`. Therefore, we blind the original value by subtracting the cookie from it, then add an immediate-operand `LEA` to add the cookie back. Figure 10 shows an example of this transformation.

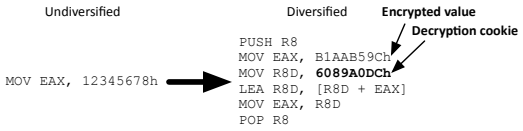

Figure 10: Example of blinding the immediate operand of an instruction. `12345678h` is replaced with `B1AAB59Ch`+`6089A0DCh` (modulo 2^{32}).

Operation	Form	Opcode
MOV	MOV EAX, imm32	B8
	MOV ECX, imm32	B9
	...	
	MOV EDI, imm32	BF
	MOV reg, imm32	C7
PUSH	PUSH imm32	68
IMUL	IMUL reg, reg, imm32	69
TEST	TEST EAX, imm32	A9
	TEST reg, imm32	F7
Arithmetic	op reg, imm32	81
	ADD EAX, imm32	05
	OR EAX, imm32	0D
	ADC EAX, imm32	15
	...	
	CMP EAX, imm32	3D

Table 1: x86 instructions with 32-bit immediate operands. The arithmetic class contains `ADD`, `ADC`, `SUB`, `SBB`, `AND`, `OR`, `XOR` and `CMP`.

The x86 architecture supports 8-, 16-, 32- and 64-bit immediate values for many instructions. We encountered only the latter two types in most code we analyzed, so we implemented only these sizes; the others can be trivially added. Table 1 shows all x86 instructions that accept a 32-bit immediate. There is only a single instruction that accepts a 64-bit immediate (the `REX.W + B8` encoding of `MOV`). Everytime librando encounters one of these instructions, it transforms the instruction to the equivalent encrypted sequence. Each instruction from Table 1 was sufficiently different from the others, so we implemented different blinding code manually for each type of instruction.

3.3 Optimizations

Intercepting branches to all generated code and rewriting their contents has a cost in program performance, as our evaluation will show. We propose several optimizations to our approach to reduce this cost as much as possible.

The Return Address Map. One of the restrictions of our design was transparency of the native stack. Even when executing diversified code, the native stack must have the same contents as it would have under undiversified code. We meet this requirement through one change: we rewrite call instructions to push undiversified return addresses on the stack, as shown in Figure 7. However, this has one significant drawback: the later `RET` matching the replaced call pops the undiversified address and branches to it. Since this address is now non-executable, this generates a page fault and a call to our signal handler. As the `SIGSEGV` handler is called after the processor interrupts the execution of another instruction, the operating system saves a lot of processor state before calling the handler; this makes signal handlers very slow. For this reason, `RET` instructions in diversified code are also expensive.

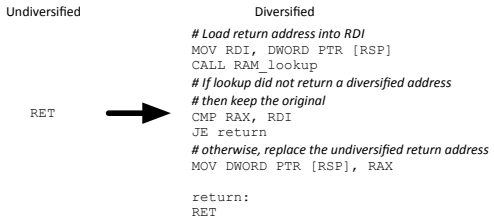

Figure 11: Rewriting the `RET` instruction to use the *Return Address Map*.

To improve performance, we prevent the `SIGSEGV` signal from being triggered. We rewrite return instructions, adding code to handle the case when the return address is in undiversified code that also has a corresponding diversified address. In as few instructions as possible, the diversified `RET` instruction now looks for the return address in a data structure that maps undiversified to diversified addresses (we call this data structure the *Return Address Map*). If an entry is found, execution continues in diversified code; otherwise, the original address is used as-is. Figure 11 shows how this optimization rewrites the `RET` instruction.

We store the address mapping in a hash map. We require a data structure that supports three operations: lookup, insertion and removal. The library adds the addresses of a block to the map whenever it diversifies the block, then removes the addresses when it discards the block. One significant factor in the choice of data structure is that lookups are far more frequent than the other operations (every `RET` performs a lookup), so the data structure implementation must perform the former as efficiently as possible. We use a cuckoo hash map for this goal [21], as it provides both fast constant-time lookups (we implemented a lookup function in 28 lines of assembly code) and good memory utilization.

As returns are essentially just indirect branches (a pop followed by a branch to the popped value), we also use this data structure to optimize all other indirect branches. We extend the map to all basic block addresses (not just return addresses), then prepend hash map lookups to all branches with unknown targets (where the target is not a direct address, but one loaded from a register or memory).

White Box Diversification. With the *Return Address Map* handling all diversified-to-diversified-code indirect branches, and the rewriting of all direct branches to target diversified code, the signal handler only intercepts branches entering or exiting the diversified code (mainly the compiler and the language runtime). Some HLL programs make many calls to the runtime (either explicitly as function calls, or implicitly through language features or inline caches), so the overhead from the signal handler remains significant. This overhead is difficult to reduce without making changes to the compiler itself, so the next step in optimization is **white box diversification**. Under this model, librando provides an interface to the compiler, which the latter uses to notify the former of branches to generated code, with the goal of avoiding the signal handler. JIT compilers frequently have one or a few centralized places in their source code that all jumps to generated code pass through; by manually inserting calls to librando in these few places, we can significantly reduce the overhead of compiler-to-undiversified-code jumps. For example, we identified a single function in V8 (called `FUNCTION_CAST`) that returns the memory address of a JavaScript function; by inserting a single line that calls librando from `FUNCTION_CAST`, we completely intercepted all indirect jumps and function calls from V8 to generated code.

In many cases, functions in the language runtime are implemented in a host language such as C; generated code calls these functions directly (using the `CALL` instruction), and control flow returns from the runtime to generated code not through explicit branches, but through function returns. A host language compiler (gcc, for example) generates these returns automatically, so we cannot manually insert calls to librando for all of them. Also, programs written in a higher-level language usually do not have direct access to the native stack, so they cannot change the return address through host language code. For this reason, we implemented a compiler-level extension for LLVM that adds a new function attribute–`rando_hash_return`. The compiler adds calls to librando at the return sites of all functions marked with this attribute; whenever such a function returns, librando checks whether the function returns to undiversified code or not. As there is no automatic analysis that can determine whether a runtime function is called from the runtime or not, the librando user has to find all functions that can be called from generated code and manually add this flag to all of them[1].

Table 2 shows the two operations that librando provides to the compiler. Using only this small API, most (if not all) invocations of the signal handler disappear.

4. EVALUATION

We implemented librando as a dynamic library for Linux and loaded it into compilers using the `LD_PRELOAD` mechanism. We evaluated the performance impact of librando on two JIT compilers: V8 (the JavaScript compiler included in the Google Chrome browser) and HotSpot (the Java client compiler from Oracle). We ran all benchmarks on a 2.2GHz Intel Xeon E5-2660 system with 32 GiB of memory, running Ubuntu Linux with kernel version 3.2.0.

First, we benchmarked V8 using the benchmark suite included with the compiler. These benchmarks stress different parts of the compiler (integer operation optimizations, floating points optimizations, inline caches, regular expression implementation) differently, and the overhead of librando varies accordingly. We sought to determine the performance impact of control flow rewriting (without diversification), our proposed optimizations, as well as the diversification techniques. We first tested librando under the black box model without optimizations or randomizations, and gradually added them one by one in the following order: the `Return Address Map` (RAM), NOP insertion, then constant blinding. We then implemented white box randomization by manually changing V8 to interact with librando (we changed 70 lines of code in total), then ran all the benchmarks again to measure the effects of this interaction.

Figure 12 shows the results of our tests. The benchmark suite reported both a per-test score, as well as the geometric mean of all benchmarks (the `Total` column). The overall impact of librando on the V8 benchmark suite is around 3.5x, with all optimizations and randomizations enabled; without randomizations, this overhead is approximately 2.7x. This overhead is not uniform across all benchmarks; `NavierStokes`, for example, shows an overhead between 1.2× and 1.4×. The fastest benchmarks (as well as the second best-performing one, `Crypto`) have a similar structure where the impact of our approach is small: nested loops of primitive numeric operations, where the types of variables are mostly static (either integers or numbers). Most of the time spent in these benchmarks

[1]Most such functions in V8 are defined using a macro called `RUNTIME_FUNCTION`, so we easily added the flag by searching for all uses of this macro.

Function	Description
`rando_redirect(addr)`	Diversifies (if needed) the code starting at `addr` and returns the diversified address
`rando_hash_return()`	Checks the return address of the current function and replaces it with the diversified equivalent

Table 2: Application Programming Interface provided by librando.

is in a highly-optimized loop that does not branch or call outside the loop. The V8 compiler emits very well optimized code once for such structures, then executes it for a substantial amount of time. Primitive operations are implemented directly as native instructions, so they contain very few calls to the runtime.

Second, we benchmarked the HotSpot client compiler for Java, using the Computer Language Shootout Game benchmarks [8]. Figure 13 shows the per-benchmark slowdown factor for each benchmark, as well as the geometric mean over all four tests. Note that the overall slowdown is smaller for HotSpot; the main cause for this is that Java is a statically-typed object-oriented language, where the data types of values are known ahead-of-time. While Java object instances can have different types at runtime (based on the program class hierarchy), primitive values have fixed types and primitive operations can be emitted very efficiently on the first attempt. HotSpot needs to collect substantially less type information than V8, as so much is already available, so it requires a lot less time to fully optimize a program. In our Java benchmarks, we see the same behavior we saw in the `NavierStokes` test for V8. We see an average slowdown of about 1.08× (a procentual slowdown of 8%) for rewriting and around 1.15× (15%) with full diversification. However, as the impact of rewriting is now much smaller, we start to notice a significant impact from NOP insertion. Since so much of the benchmarks's execution time is now spent in a very small loop (a few instructions in length), every extra instruction starts to count. On `fannkuchredux`, NOP insertion at $p_{\text{NOP}} = 0.5$ almost triples the overhead (from 15% to almost 40%).

In addition to these tests, we also ran the same benchmarks on V8 and HotSpot without the *Return Address Map* optimization. V8 showed an average slowdown factor of 50× (with a maximum of 232× for `EarleyBoyer`), much higher than the 2.7× slowdown for the optimized version. The impact on HotSpot is less severe (about 1.6× slowdown without `RAM` as opposed to 1.08× with it), but still substantial. The `Return Address Map` optimization is crucial to librando performance. However, we leave further enhancements to our choice and implementation of data structure (cuckoo hash with hand-implemented lookup) to future work, as some of our experiments showed that there is still room for some improvements. We ran the Language Shootout benchmark `binarytreesredux` (a recursive binary tree implementation in Java that makes extensive use of function calls) on HotSpot and observed a slowdown of approximately 70x, even with the `RAM` optimization enabled. Further profiling of both this benchmark and all V8 benchmarks showed that around 25% of time spent inside librando takes place inside the `RAM` lookup function, even with our optimized implementation (28 assembly instructions in total).

Overall, the performance impact depends greatly on the structure and goal of the HLL program, but also on features of the HLL. As we have shown, a statically-typed but just-in-time-compiled language has much lower diversification overhead than a dynamically-typed program. Another factor to account for is the ratio of time spent in generated code versus time spent in the compiler or runtime. This technique has higher overhead when new code is generated with high frequency (as is the case with dynamically-typed languages), and on other transitions from generated code to the outside. However, synthetic CPU-bound benchmarks are not completely representative of the average workload of a JIT compiler; these benchmarks perform very little input/output operations, and the time spent in the few such operations is small. In contrast, real-world code interacts much more with the rest of the system, and also makes much more use of external libraries. When the execution time of a program is dominated by input and output, or by calls to libraries, the overhead should be much smaller. Our results show an upper-bound for this overhead. This upper-bound is also not necessarily significant in a real-world application; for example, we linked the Chrome browser to librando and used it to visit various web pages, some only containing static content and some using JavaScript heavily. We observed no noticeable degradation in browsing experience.

5. RANDOMIZATION DISCUSSION

We designed librando as a framework for intercepting and securing dynamically-generated code. While the library only supports two randomization techniques at present, the design allows for more to be easily added. Based on our experience implementing the current randomizations, we believe new ones could be added without much effort. Examples of possible randomizations include (related work [9, 14, 22, 29] also discusses and implements some of these for statically-generated code diversification):

`mmap` Address Randomization On current operating systems, ASLR guarantees that addresses returned by mapping functions (`mmap` and similar) are randomized. However, this required that ASLR is supported and enabled on the system. In many cases, it is possible for the program (or librando) to ask for a mapping at a specific address, instead of letting the operating system pick the address. By asking for a randomized address, librando can add address randomization even on systems where ASLR is not enabled (some systems also allow for ASLR to be disabled on a per-program basis, so librando could be used only on those programs).

Basic Block Reordering Our diversifier can emit basic blocks in any order (although some orders have better performance than others, due to spatial locality). While we currently impose no order on the blocks, the implementation can be extended to explicitly reorder blocks randomly. This randomization would be complementary to address randomization and NOP insertion; the former randomizes the location of the entire executable region of code, basic block reordering would reorder blocks inside the region and NOP insertion would randomize instructions inside each block. The techniques would operate at different levels of granularity to provide even more randomization of the location of an instruction.

Equivalent Instruction Substitution The x86 instruction set has several possible encodings for some of the most used instructions. Randomly replacing some of these encodings with equivalent ones changes the code without changing its behavior. In other cases, it is possible to change one or more instruction with an equivalent sequence of other instructions, while also maintaining behavior. Making such changes has a similar effect on security as NOP insertion.

Register Re-allocation Register operands are encoded inside one or more bytes of an x86 instruction. By randomizing the

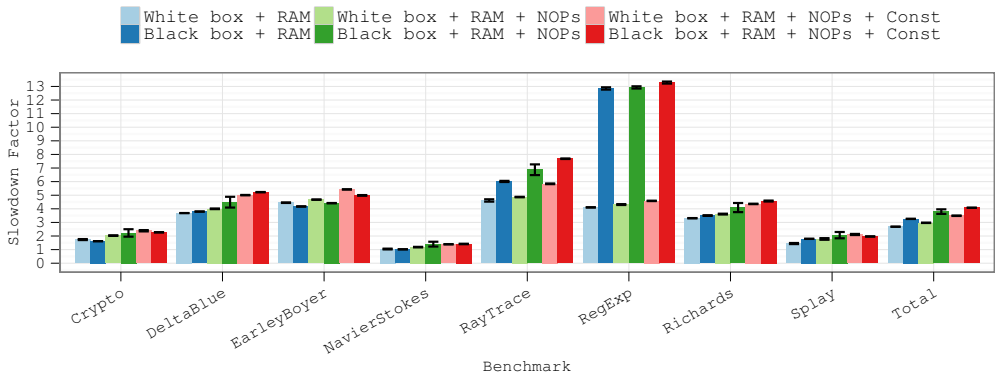

Figure 12: V8 benchmark results.

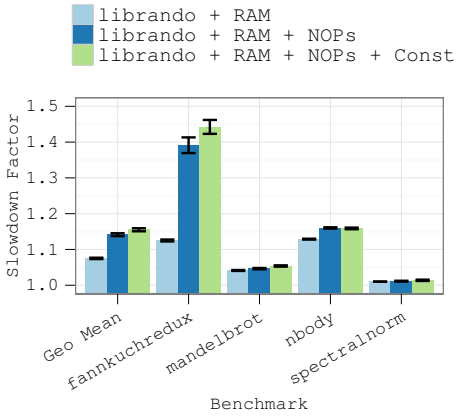

Figure 13: HotSpot benchmark results.

registers allocated to program values, librando can randomize instruction encoding further. However, this requires a dataflow analysis stage to identify the unique values stored in physical registers and their live ranges.

Instruction Reordering As an alternative (or complement) to inserting NOPs, we could reorder the instructions of each basic block. However, this requires that all dataflow dependencies are preserved; an instruction that generates a value cannot be moved after instructions that use the value. This requires running an algorithm to determine all data dependencies.

6. RELATED WORK

6.1 JIT Compiler Protection

One way of improving JIT compiler security is manually adding security checks and diversity by modifying the compiler. JITDefender [5] hinders JIT spraying attacks by marking all HLL code pages as non-executable; the compiler changes these pages to executable on entry to the generated code, and changes them back on return to the compiler or runtime. This effectively prevents an attacker from redirecting any other branch (other than the intended ones) to dynamically generated code. INSeRT [30] adds code randomization (concretely, it randomizes the operand of every emitted instruction) through a white box approach, requiring manual changes to the native code emitter inside the compiler. Both approaches successfully hinder most JIT spraying and code reuse attacks against HLL code, but require changes to the compiler; they are not feasible when source code is not available, or the modified code cannot be deployed. On the other hand, librando can harden a compiler without requiring any cooperation from the compiler itself.

A similar line of research investigates JIT code generation from inside a sandbox (where control flow is restricted, and code must follow certain restrictions). Native Client [31] is a sandboxed execution environment which adds static checks to native code that enforces restrictions on branch targets (such as 16-byte alignment for all targets). The original implementation of the system did not allow dynamic code generation at all, but later work [1] added this feature. The JIT extensions adds an API to the sandbox that a JIT compiler uses to generate new code; before executing that code, the sandbox verifies that the new code respects all restrictions imposed by Native Client, and therefore cannot break out of the sandbox. Another of their contributions was an in-depth analysis on the different types of NOPs to insert, which they use to enforce basic block alignment; here, we use NOPs for randomization. This is, in some ways, similar to our white box diversification approach: librando can be viewed as the sandbox that the JIT compiler runs inside.

6.2 Software Diversity

Cohen [6] first introduced the idea of raising the costs to the attacker by randomization, and presented several possible ways of doing this: equivalent instruction substitution, instruction reordering, extra jump or call insertions and many others. Forrest et al. [7] later described one of the first practical implementations of these ideas, randomizing the layout of the program stack and global variables. They demonstrate that software diversity increases security (preventing stack-based attacks) with a small performance overhead.

Address space layout randomization (ASLR) is an operating system-level randomization technique that places parts of the program at random addresses. While this is not possible for the en-

tire program (executable code is typically compiled as position-dependent code that must be loaded at fixed addresses), it is active for some significant areas like the stack and memory obtained from `mmap`. This technique is enabled on all major operating systems (Windows, Linux, Mac OS) and has significant security benefits. However, it is not a sufficient security measure against code reuse attacks, since only the base address of a region is randomized. If the attacker somehow determines that base address, they gain access to the rest of the code as well, as the contents remain the same and in the same positions. Researchers have shown that ASLR has very low entropy on 32-bit systems [27] (taking just a few minutes to find the base address by brute force), so ASLR by itself does not provide strong security. It may be possible to find the randomized base address even on 64-bit systems through means other than brute force (for example, through some other information leak in the application).

A more fine-grained defense against code reuse attacks is code randomization. Recent years have seen a resurgence of this approach; related work randomizes code layout at the compiler level [9, 14, 11], virtual machine level [10] or through static binary rewriting [22, 29]. These implementations use various randomization techniques to diversify code: instruction and basic block reordering, register re-allocation, instruction substitution and NOP insertion (we discuss these techniques in Section 5). Much progress has been made in this area in a short while, showing that diversification has a positive impact on software security with negligible impact on performance. However, these defenses only diversify code available to the compiler or rewriter; even virtual machine-based solutions [10] rely on some rewriting of the main program binary itself. To our knowledge, no existing code layout randomization implementation handles dynamically generated code.

6.3 Binary Rewriting

There is significant research into dynamic rewriting of program code at run-time. DynamoRIO [4], PIN [17], Valgrind [19], and Strata [25] all intercept and rewrite program code at run-time, for purposes such as instrumentation, profiling, identifying program errors and increasing security The transparency restrictions in their designs are very similar to the restrictions we described for librando. However, one difference that sets librando apart from other rewriters is the limited scope of code rewriting: librando only intercepts and rewrites dynamically generated code, whereas all other rewriters handle all of the application code. For static program code, we envision that a compiler-level diversification solution is used on statically generated code; librando is only meant to diversify code which ahead-of-time diversification solution cannot protect. Our approach allows the JIT compiler and language runtime to run natively, only intercepting dynamically generated code; all other solutions would intercept and rewrite everything, starting with the first instruction in the JIT compiler. Another difference is that dynamic rewriters are designed as frameworks that allow arbitrary changes to the intercepted code, which increases their complexity significantly, whereas the magnitude of our changes to the code is very small and security-focused (adding NOPs and rewriting instruction operands). For example, Valgrind disassembles executed code and converts it to an intermediate representation (IR), which all code transformations operate on. After all transformations are done, Valgrind converts this IR back to x86 code; this two-way translation is not needed by librando.

Existing dynamic binary rewriters have been used for security. Program Shepherding [15] (implemented using DynamoRIO) and libdetox [23] add security checks before branches, function calls and system calls by intercepting and rewriting program blocks. These checks only allow branches to allowed code addresses, determined algorithmically or from a given list. This effectively defends against code reuse attacks (and more), albeit using a different approach from diversification. Our proposed solution has similar goals, but different methods: we randomize code layout, restricting whatever knowledge the attacker possesses of program code. While deterministic techniques are successful at hardening applications, they are also vulnerable in one regard: if an attacker finds a flaw in such a technique, they are able to attack all hardened targets using this flaw. Randomization, on the other hand, does not assume safety of the defense itself; instead, the goal is to restrict any successful attack to only a small subset of possible targets.

7. CONCLUSIONS

Traditional code injection attacks are becomming increasingly hard. This prompted the evolution of attacks targeted at JITs: JIT spraying and code reuse attacks. We describe librando, a binary rewriting library that hardens JIT compilers, and generally any software that generates new code at run-time, against these attacks. Our library supports randomization of code from a JIT compiler without requiring any internal changes to the compiler. This approach is portable to any existing or new compiler, providing increased security while saving man-months or -years of development effort; instead of having to redo the effort of implementing security measures on every JIT, developers may opt to use our library to secure their compiler. The library can also be used as an interim measure, providing security at a temporary performance penalty until security measures are implemented more efficiently in the compiler itself. In cases where compiler source code is not available, or where recompilation and reinstallation are not feasible, this penalty is preferable to the loss in security. If compiler source code is available, compiler developers can improve diversification performance through white box diversification, by adding calls from the compiler to librando. We also presented an optimization with substantial impact on performance: the *Return Address Map*. We successfully tested our work on two industrial-strength JIT compilers, both widely used at present. Our evaluation showed that the impact of diversification depends greatly on the workload; librando provides great security benefits at low performance cost (around 15%) for statically-typed languages (where it can be enabled at all times), and at a larger cost (a 3-4× slowdown) for dynamically-typed languages.

We have extended the reach of automated software diversity to also encompass code generated by a JIT compiler. While we are sure that we have not seen the last of JIT specific attacks, librando gives defenders a quick and comprehensive response.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

9. REFERENCES

[1] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee.

Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 355–366, 2011.

[2] D. Blazakis. Interpreter exploitation. In *Proceedings of the 4th USENIX Workshop on Offensive technologies*, WOOT'10, 2010.

[3] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, 2011.

[4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 2003 International Symposium on Code Generation and Optimisation*, CGO '03, 2003.

[5] P. Chen, Y. Fang, B. Mao, and L. Xie. JITDefender: A defense against JIT spraying attacks. In *Proceedings of the 26th IFIP TC 11 International Information Security Conference*, SEC '11, pages 142–153, 2011.

[6] F. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):565–584, Oct. 1993.

[7] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, HotOS '97, pages 67–72, 1997.

[8] B. Fulgham. The computer language benchmarks game. `http://shootout.alioth.debian.org/`, 2012.

[9] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Security Symposium*, pages 475–490, 2012.

[10] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P '12, pages 571–585, 2012.

[11] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization*, CGO '13, 2013.

[12] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao – an extensible micro-architectural optimizer. In *Proceedings of the 9th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 1–10, 2011.

[13] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, August 2012.

[14] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. In S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense*, volume 54 of *Advances in Information Security*, pages 77–98. Springer New York, 2011.

[15] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, 2002.

[16] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 195–208, 2010.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[18] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, pages 209–224, 2006.

[19] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 2003.

[20] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, 2010.

[21] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[22] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P '12, pages 601–615, 2012.

[23] M. Payer and T. R. Gross. Fine-grained user-space security through virtualization. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 157–168, 2011.

[24] C. Rohlf and Y. Ivnitskiy. Attacking clientside JIT compilers. In *Black Hat USA*, 2011.

[25] K. Scott, N. Kumar, S. Velusamy, B. R. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the 2003 International Symposium on Code Generation and Optimisation*, CGO '03, 2003.

[26] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, 2007.

[27] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, 2004.

[28] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: mitigating contention for QoS in warehouse scale computers. In *Proceedings of the 10th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '12, pages 1–12, 2012.

[29] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, 2012.

[30] T. Wei, T. Wang, L. Duan, and J. Luo. INSeRT: Protect dynamic code generation against spraying. In *Proceedings of the 2011 International Conference on Information Science and Technology*, ICIST '11, pages 323–328, 2011.

[31] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, S&P '09, pages 79–93, 2009.