Branch Regulation: Low-Overhead Protection from Code Reuse Attacks

Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh and Dmitry Ponomarev
Department of Computer Science
State University of New York at Binghamton
{mkayaalp, mozsoy, nael, dima}@cs.binghamton.edu

Abstract

Code reuse attacks (CRAs) are recent security exploits that allow attackers to execute arbitrary code on a compromised machine. CRAs, exemplified by return-oriented and jump-oriented programming approaches, reuse fragments of the library code, thus avoiding the need for explicit injection of attack code on the stack. Since the executed code is reused existing code, CRAs bypass current hardware and software security measures that prevent execution from data or stack regions of memory. While software-based full control flow integrity (CFI) checking can protect against CRAs, it includes significant overhead, involves non-trivial effort of constructing a control flow graph, relies on proprietary tools and has potential vulnerabilities due to the presence of unintended branch instructions in architectures such as x86—those branches are not checked by the software CFI. We propose branch regulation *(BR), a lightweight hardware-supported protection mechanism against the CRAs that addresses all limitations of software CFI. BR enforces simple control flow rules in hardware at the function granularity to disallow arbitrary control flow transfers from one function into the middle of another function. This prevents common classes of CRAs without the complexity and run-time overhead of full CFI enforcement. BR incurs a slowdown of about 2% and increases the code footprint by less than 1% on the average for the SPEC 2006 benchmarks.*

1 Introduction

Software security exploits have been steadily increasing in frequency and impact. In the past, one of the most prevalent attack vectors was *code injection*: a buffer overflow vulnerability is used to inject attack code on the stack, while simultaneously overwriting the function return address or a function pointer to point to the entry of the injected code [1, 2]. A number of approaches to protect against such attacks were devised, including software and hardware approaches [3–7]. These efforts have culminated in the recent deployment of hardware memory protection mechanisms that do not allow a memory page to be both writable and executable at the same time (the so called $W \oplus X$ protection). As a result, classical code injection attacks no longer represent feasible threats in modern systems.

In response to the mechanisms to prevent code injection attacks, new attacks have been developed that rely on reusing existing code, without the need for code injection. An early example of code reuse attacks (CRA) is the return-into-libc attack [8] which allows a libc function to be called. While powerful, return-into-libc attacks do not allow the attacker to perform arbitrary computation. A recently proposed CRA, called Return-oriented programming (ROP) [9], allows attackers to execute arbitrary code on victim machines. In this attack, the attacker overflows the stack with a sequence of return addresses that point to specific code snippets (called gadgets) in the program under attack. Each gadget ends with a return instruction to trigger the execution of the next gadget pointed to by the next return address on the stack. ROP was shown to be Turing-complete on a variety of platforms [10–14]. Automated tools have been developed that allow unsophisticated attackers to construct arbitrary malicious programs using ROP [15–18].

Since the introduction of ROP, several defense mechanisms have been proposed. Solutions include monitoring the rate of the return instructions [19], monitoring the pattern of return instructions [20], or rewriting the existing code to eliminate all returns [21]. In response to these defenses, a new class of attacks that does not rely on return instructions has been proposed [22–24]. In these jump-oriented programming (JOP) attacks, the attacker chains the gadgets by using a sequence of indirect jump instructions, rather that return instructions, thus bypassing the defense mechanisms designed for ROP. These attacks represent critical exploit strategies that can compromise any machine running software with a buffer overflow vulnerability. Both ROP and JOP attacks can be mitigated by enforcing full Control Flow Integrity (CFI) [25]. However, the CFI approach is a heavy-weight solution that has several important drawbacks, as we discuss in more detail in Section 2.

In this paper, we propose Branch Regulation (BR) - a new low-overhead solution for defending against the ROP and JOP attacks. BR checks the legitimacy of program control flow in a light-weight fashion, avoiding both the analysis complexity and the run time overhead of full CFI. Specifically, instead of constructing and checking the complete Control Flow Graph (CFG), we check only if a branch instruction targets an address within the same function, or targets the starting address of another function. Thus, arbitrary branches that cross function boundaries are prevented,

severely limiting JOP and ROP attacks.

We demonstrate that BR makes the great majority of functions immune to code reuse attacks because they lack a critical dispatcher gadget (discussed in Section 6.1). We also demonstrate that the remaining functions by themselves are not sufficient for carrying out a successful attack; even though they contain some functional gadgets, they lack one or more of the ingredients needed for the attack. In addition to performance advantages, the hardware-assisted checks used in BR supports the checking of *unintended branches*–branches that occur in the middle of an instruction in variable-length ISA such as the x86. Software approaches such as the CFI implementation by Abadi et al [25] cannot perform checks for such unintended branches.

BR has the following key properties.

- It eliminates all vulnerable code in the examined libraries if the attacker relies on the execution of a system call instruction. Even if the attacker does not rely on a system call, BR reduces the number of available gadgets to 1% of what is available in the entire code base, with significant restrictions on how these remaining gadgets can be used.
- It comes at a performance cost of slightly over 2% and increases the code size by less than 1% for SPEC 2006 benchmarks.
- It does not require complex binary rewriting or construction of a full control flow graph of a program. Instead, BR requires only simple binary annotations that can be readily derived from the symbol tables in the ELF binaries.
- With simple hardware support, BR also performs checks for unintended branches (in variable instruction-length architectures, like x86) thus closing the potential security vulnerability of purely software-based solutions.

The remainder of this paper is organized as follows. Section 2 describes the background and related work. We present the threat model in Section 3. BR is introduced in Section 4, and the implementation details are presented in Section 5. We present our approach for analyzing BR security in Section 6, and evaluate BR's security in Section 7. Section 8 presents the performance evaluation of BR. Finally, Section 9 offers some concluding remarks.

2 Background and Related Work

In this section we provide background related to the evolution of CRAs. We also review related work.

2.1 Buffer Overflow and Code Injection

Buffer overflows are one of the most common software exploits in languages without type safety such as C/C++. Stack smashing is a buffer overflow attack on a stack-allocated buffer [1], allowing the attacker to overwrite the return address of the function with the address of the malicious injected code. Several approaches were developed to defeat stack smashing attacks [3–5, 26–28]. Despite these approaches, buffer overflow vulnerabilities remain prevalent. Hardware solutions have been proposed to protect against stack smashing [6, 29–31]. StackGhost uses the register window feature of the Sun Sparc architecture to verify that return addresses have not been overwritten [32]. Data execution prevention (DEP) prevents code from executing from pages allocated for stack or data [33, 34]. While software implementations of DEP are possible, W $\oplus$ X page protection schemes are now commonly supported by CPUs.

2.2 Return Oriented Programming

Protection mechanisms such as DEP are now available in all major operating systems and make it impossible to perform code injection attacks. Adversaries have reacted by devising new attacks that bypass DEP. Techniques such as *return-to-libc attacks*, where the attacker subverts the control flow to call a function in the standard C library, can not be prevented by DEP because they execute valid code in valid code memory segments. However, return-to-libc does not provide a clear way for arbitrary attack code execution, because only regular library functions can be executed.

A CRA that allows arbitrary code execution was recently devised. Return-oriented programming (ROP) [9] attacks are mounted as follows. The attacker identifies *gadgets*, which are sequences of instructions in the victim program (including any linked in libraries) that end with a return. Sufficient gadgets can be identified to allow the composition of arbitrary attack code. The attacker uses a buffer overflow vulnerability to inject a sequence of return addresses corresponding to a sequence of gadgets. When the function returns, it returns to the location of the first gadget. As that gadget terminates with a return, the return address is that of the next gadget, and so on. ROP executes instructions only from the code segment and therefore is not prevented by DEP. A Turing-complete set of gadgets has been demonstrated on a number of architectures and operating systems [10–14]. Compilers have been built to ease the development of ROP attacks, making them accessible to unsophisticated attackers.

Because of the seriousness of ROP attacks, a number of mitigation techniques have already been proposed. Davi et al. proposed the use of a reference monitor to detect the repeated execution of a small number of instruction sequences followed by a return [19]. Chen et al. monitor return properties to detect possible ROP attacks [20]. Li et al propose rewriting binaries to eliminate the use of returns, completely preventing return-oriented attacks [21]. Bania proposed a number of compiler and binary rewriting approaches to protect from ROP [35]; the proposed ideas are preliminary, and most of the proposed techniques rely on validating the call-return behavior.

An interesting and effective recent approach was developed by Onarlioglu et al. who rewrite away all unintended control flow paths [36]. They further protect intended branches through a combination of pointer encryption and stack cookies. Stack cookies are function-specific markers pushed on the frame. Additional code is inserted after every branch to check this cookie. Thus, a branch to another function is possible, but subsequent indirect branches would be detected (allowing return-to-libc attacks, but preventing basic ROP and JOP attacks). However, if gadgets are available in a function to replace the cookie before leaving, this protection can be defeated. The approach cannot protect legacy

binaries; it also increases the code footprint by over 25%.

In order to preserve the integrity of return addresses written to the data stack by return instructions, several previous efforts targeted separation of return addresses from the data stack and enforced address matching for call-return pairs [6, 29–31]. When deployed, these techniques are able to thwart ROP attacks, but are not sufficient to protect against other types of CRAs, particularly the ones described in the next subsection.

2.3 Jump-Oriented Programming

The above solutions detect or prevent ROP attacks by monitoring the behavior of call and return instructions. Recently, a variation of the ROP attack was proposed that instead uses branch instructions (jumps) to transfer control between the gadgets. This attacking technique is called *jump-oriented programming* (JOP) [22–24]. JOP requires a critical dispatcher gadget that orchestrates the sequence in which other gadgets are called. The authors demonstrated that dispatcher gadget is fairly common in standard libraries, allowing Turing-complete functionality to be composed using this approach. Since no known defenses against ROP prevent JOP attacks, there is a critical need for techniques that prevent JOP attacks with low overhead.

2.4 Control Flow Integrity (CFI)

Our work is most related to recent work on using a reference monitor to track and enforce control flow integrity [25]. In particular, Abadi et al. observed that most rootkits and other malware change the control flow graph (CFG) of the original program [25]. Thus, CFI attempts to prevent these attacks by ensuring that any control flow change is consistent with the original program CFG; this is the invariant it attempts to preserve. If a branch that is not present in the CFG is encountered, the system stops the process and signals a CFI violation thwarting the attack. Enforcing full CFI at the branch level should completely protect from ROP and JOP attacks; this is a promising approach to address the problem. With aggressive optimization, the CFI performance overhead is non-negligible, but moderate: an average of 16% performance penalty for SPEC 2000 benchmarks was reported in [25]. Our own behavioral model of CFI shows 22% performance loss for a larger set of benchmarks from SPEC 2006 suite.

Unfortunately, CFI requires expensive static analysis, profiling, or deep binary analysis to build the CFG. No practical tool that analyzes a binary to form a complete CFG (including "Vulcan", reported to be used in [25]) is publicly available to the best of our knowledge. In addition, it is unclear how CFI will operate in a system with shared libraries and where functions may be bound late or even dynamically. Furthermore, the existing software implementation of CFI also suffers a weakness on x86 and similar ISAs that feature variable instruction lengths: unintended control flow instructions can occur in the middle of multi-byte ISA instructions. These unintended instructions are not protected because no inline code can be inserted to check them. We show in Section 4.2 that for libc, these unintended branches far outnumber the legitimate ones. The technique presented in this paper uses hardware-supported checks to address this issue.

2.5 Other Defense Approaches

A number of other solutions have been proposed to prevent or limit attacks on software vulnerabilities. Memory bounds checking (MBC) techniques attempt to eliminate buffer overflow vulnerabilities by checking every memory access against the base and bound of the associated data structure [37–39]. MBC is a promising solution that addresses the root problem. However, the hardware costs or performance overhead of MBC schemes is significant. In addition, MBC affords limited protection for legacy binaries and externally linked or loaded components. Other issues related to memory aliasing and handling of variable length argument lists also hinder the practical adoption of MBC.

Data flow integrity [40] uses static, compile-time analysis to infer the data flow graph of a program and instrument the program to enforce conformance with this graph. Using similar analysis, WIT [41] associates instructions with their allowed target objects and enforces integrity of each write operation.

Dynamic Information Flow Tracking (DIFT) taints the information coming from insecure sources, and dynamically tracks and propagates the taint through processor registers and memory locations. If a tainted address is used for writing into the stack, a security exception is raised. The drawback is that DIFT is a heavy-weight approach that entails a significant redesign of the processor datapath and memory system if implemented in hardware [42–44], or incurs a substantial performance overhead if implemented in software [45, 46]. In addition, DIFT solutions may suffer from false positives, where the tainted state of the system rapidly expands in a domino fashion.

A different approach to protection against code injection attacks uses randomization. Address space layout randomization (ASLR) [47] positions the program's memory at a random offset in memory. ASLR (and other optimized heap allocation models [48, 49] make it difficult for an attacker to guess the correct address of the malicious code. However, successful attacks can be mounted if the addresses are leaked (e.g., format string attacks allow attackers to peek at the stack [50]), or by creating many copies of the malicious code to increase the chance of reaching it (heap/stack spraying) [51].

Related to our work, Champagne and Lee [52] introduce the idea of valid function entries as part of the Bastion framework for software security within a virtualized environment. The function entries are setup at startup time and used to ensure legal control flow at runtime. Although the approach is similar, we believe the context and application are quite different. Moreover, it is interesting that a similar approach is useful for two different security applications.

3 Threat Model and Assumptions

We assume that the attacker has full control over data and stack memory regions. This assumption is consistent with published CRAs that make use of stack/heap buffer overflows and/or string formatting bugs to overwrite a return address, a function pointer or a non-local jump buffer (that is placed in memory by a call to `setjmp` [53]). In

addition, we assume that the attacker is able to modify the program counter as well as other registers upon the initiation of the attack, as it can be achieved by overwriting a non-local jump buffer. We also assume execution prevention for writable memory that forces the attacker to reuse existing code.

We assume that the vulnerability exploited to initiate the attack does not lead to a privilege escalation. This assumption is necessary because any defense mechanism could be thwarted by an adversary with sufficient privileges. Rather, the attacker's objective is to use the CRA to achieve privilege escalation.

We do not consider protection for the `setjmp` and `longjmp` C calls. These calls can be more naturally supported in the operating system. Specifically, BR will generate an exception for these instructions which can be checked and allowed by the OS. Similarly, we did not consider the possibility of functions with multiple entry points (a small number of such functions occur in the standard C library). It is possible with a small amount of effort to support such functions either by chaining annotations or handling the lookup for them in the OS after BR raises the exception.

Instead of focusing on preventing attacks altogether, our goal is to render any arbitrary code reuse attack that the attacker manages to initiate useless. With a severely reduced number of available gadgets, and reduced ability to exploit these gadgets, arbitrary computation can be avoided. For example, unless the attacker can find gadgets to execute system calls, the damage from any attack is limited to the compromised process.

4 Branch Regulation (BR)

In this paper, we propose and investigate Branch Regulation, a technique that defends against CRAs by enforcing simple control flow invariants present in function-based programming languages. BR requires little analysis of the program code, and can be derived directly from the binary, allowing it to protect legacy code. Its simplicity allows for practical implementations with small overheads in storage and execution time. Moreover, by providing simple hardware support, our scheme protects against attacks that exploit unintended instructions (those starting in the middle of a legal instructions in variable-length ISAs such as x86).

Figure 1-a shows a simple JOP attack scenario that uses one dispatcher gadget, two additional gadgets and a system call instruction (`int 80h` in x86) that are spread across two functions. Suppose that the attacker has taken full control over writable memory, initialized some of the registers, diverted the execution to the start of the dispatcher gadget and the goal of the attack is to execute a system call. For this example, we can assume that `esi` points to the dispatcher gadget, `edx` points to the system call instruction and `ecx` points to a memory location where addresses of gadgets 1 and 2 are stored contiguously. Furthermore, the parameters for the system call are assumed to be written in appropriate memory locations by the attacker.

As the dispatcher gadget is executed, the address of the next gadget to execute is fetched into `eax` from the array indexed by `ecx` and the control flow is diverted to that address by the indirect jump instruction. According to the attacker-initiated values, control flow follows the numbered arrows on the left of the figure. After the first hop, gadget 1 sets `ebx` as the system call parameter from memory under the attacker's control. Similarly, after the third hop, gadget 2 sets `eax` as the system call ID and jumps to the system call instruction.

Figure 1-b shows the impact of BR on this attack. Under BR, the third hop causes an exception since the destination of the indirect jump initiated in function `f1` is in the middle of another function (`f2`), which is an illegitimate control flow transfer in our scheme.

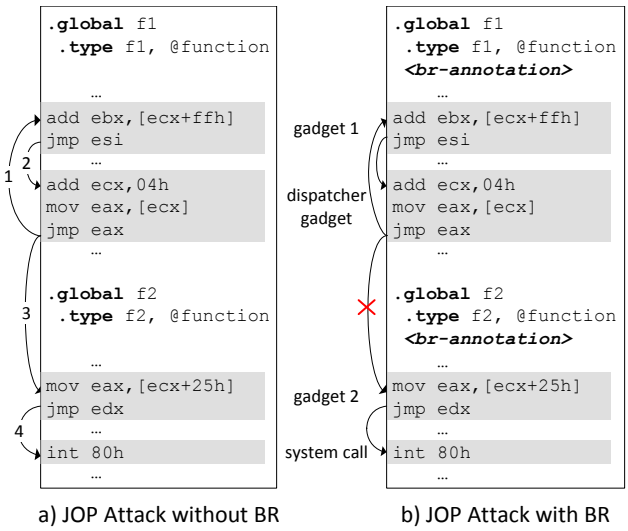


Figure 1. Impact of BR on CRAs

4.1 Enforcing BR Rules

BR relies on the observation that legal control flow changes (for simplicity called branches henceforth) target an address satisfying one of the following cases: (1) an address within the same function, as would occur with loops and conditional statements; (2) the entry point to a new function, as would occur with a function call; or (3) a return address generated by a legitimate prior call as in number (2) above. Thus, BR works by enforcing the following set of rules on branches (see Figure 2):

RET: A return instruction should point to the address that is saved by a corresponding call instruction.

Indirect JMP: A branch instruction that uses a computed target should point to either an entry point of a function, or a location inside the same function.

CALL: A call instruction should divert execution to the entry point of a function.

To support the first rule, we propose maintaining a call stack for the hardware context being executed, that can only be modified by call and return instructions. This stack, called the Secure Call Stack (SCS), can be used to match the computed return addresses against the stored ones. This aspect of BR is similar to some prior approaches to protect against ROP attacks [6, 29–31, 52]. In order to check for the last two rules, we need to determine where a function begins and ends. To support this capability, we augment each stack entry with function bounds. Therefore, a single

structure can support all three rules. The entry at the top of the stack is examined on every BR check.

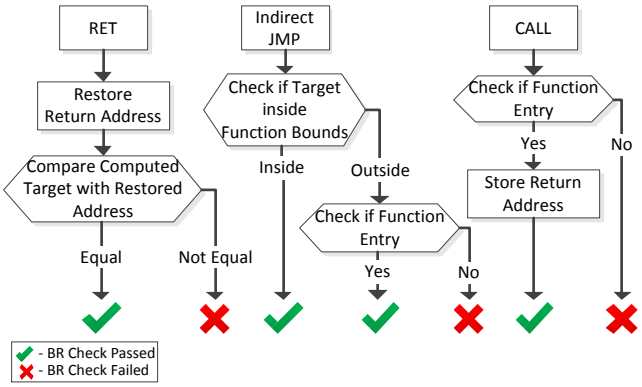


Figure 2. Branch Regulation Checking Flow

4.2 Unintended Branches

We pursue hardware support for BR, for performance and binary compatibility reasons, but more importantly for security reasons. As we show in this section, an attacker can find many unintended instructions in the binary image of an ISA with variable size instructions such as x86. Specifically, these are instructions that start at a byte in the middle of a multi-byte instructions. We show that these instructions account for a large number of the gadgets exploitable by attackers. Software approaches do not typically protect from exploits that use these instructions; for example, an unintended branch will not appear in the CFG and will not be checked by the software CFI implementation developed by Abadi et al [25]. Attackers may bypass the software enforcement completely by using gadgets consisting of unintended instructions. Hardware support that checks every branch closes this security vulnerability.

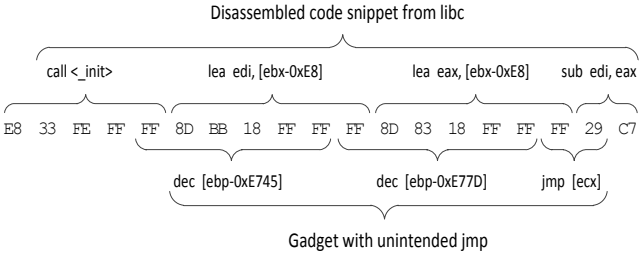


Figure 3. Example Gadget with Unintended Branch

To illustrate the concept of unintended branches, a portion of the disassembly of the `__libc_csu_init` function is shown on the top part of Figure 3. If the decoding starts after skipping the first four bytes, a different instruction sequence can be decoded as shown at the bottom of Figure 3, containing an indirect jump that the programmer did not intend to execute. One particular property of indirect jump instructions that makes them easier to discover in an instruction sequence is that they start with `FF` —a common byte used in immediate values (e.g. bit-masks and sign bits of negative values). In the second `lea` instruction of the intended code example, the negative immediate value `-0xE8` is encoded as `FF FF FF 18` in two's complement with little endian byte order. The last one of these `FF` bytes is adjacent to the opcode of the `sub` instruction which is `29`. These two bytes can be decoded as an indirect jump which might be used by an attacker to jump to the memory location that `ecx` points to.

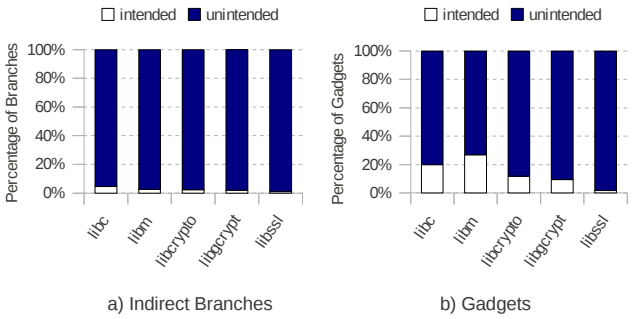


Figure 4. Number of Indirect Branches in the Binary and Gadgets Generated with Indirect Branches

As shown in Figure 4, the unintended branch instructions (and the gadgets constructed using them) far outnumber the intended branches. For example, 80% of all gadgets that can be constructed in libc use unintended instructions. The percentages are even higher for other libraries that we considered.

5 BR Implementation Details

In this section, we describe the details of the BR implementation. We first discuss how function annotations are implemented, and then describe the hardware support for BR.

5.1 Function Annotations

Function limits are easily found in symbol tables of executable formats such as ELF. We propose annotating these points using a simple binary rewriting scheme to inform the architecture about the function bound information. We then check control altering instructions to verify that they target legitimate destinations per the BR rules.

BR annotation starts with a prefetch instruction to a special address to ensure that the annotated code retains binary compatibility; this is the same approach for implementing annotations used by CFI [25]. This instruction is followed by the size of the function. It is unlikely to encounter this sequence of bytes inside regular program code. Even though it is possible for an attacker to write this annotation sequence inside the program stack, it is not possible to use this sequence as an annotation if the system has $W \oplus X$ protection for the data segment.

Figure 5 explains the use of BR annotations. Each function starts with the annotation. In the first two cases shown, the BR checks pass and the access is allowed. In the third case, the violation is detected and a BR exception is raised.

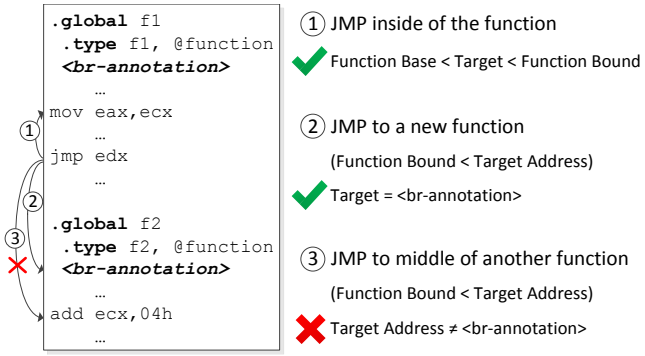


Figure 5. Branch Regulation Checks

The second branch represents a legal use of indirect jumps since the branch target is an annotation even though the destination address is out of function bounds.

5.2 Architectural Support for BR

At runtime, BR checks are performed in hardware at the execution stage of the pipeline after the target address of an indirect branch is computed. Figure 6 shows the architectural layout for BR. To maintain the function bounds metadata, BR relies on a Secure Call Stack (SCS), which is a portion of memory allocated by the OS and accessed by the hardware upon each call or return instruction. It is similar to the architected page table mechanism, in the sense that it serves as a virtually unbounded storage accessible by the hardware. The metadata maintained in the SCS (base, size, current index) is part of a process context and is therefore saved and restored upon a context switch.

The additional hardware includes a new structure, which we call the Function Bounds Stack (FBS), to serve as a cache for the legal return addresses and function bounds that are stored in the SCS. Note that the only purpose of the FBS is to improve performance. Indeed, BR architecture can be implemented without hardware FBS by looking up all bounds information in the SCS —we analyze the performance impact of FBS in Section 8. New entries that are pushed onto the FBS cause the eviction of the oldest entries and a miss to the FBS occurs only when all FBS entries are popped off the stack by the return instructions; in this case, we bring the next function bounds from the top of the SCS stored in memory.

In addition, BR architecture requires the checking logic that is used for comparison of target addresses with the base and bound of the current function. While the base and bound information of the current function is stored at the top of FBS, return addresses and function bounds for previous legitimate call instructions are stored in the rest of the stack. An FBS entry for a 32 bit architecture is composed of two 32-bit values, base and bound, and a 32-bit return address.

By the time a `call` instruction gets to the execution stage, the next instruction has already been fetched. The first 8 bytes of the next instruction are compared against the BR annotation in order to decide if this is a function call. If this check succeeds, then the current function's base and bound values are pushed onto the FBS along with the return address for this call instruction. The base and bound values

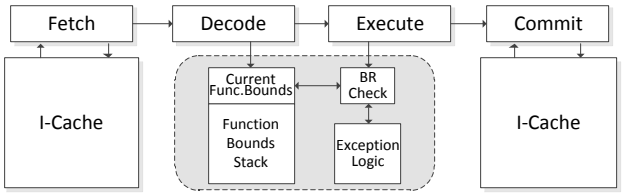


Figure 6. Processor Pipeline with Branch Regulation Hardware

of the newly called function are updated with the information from the BR annotation.

When an indirect jump instruction is executed, its target address is first compared against the current function bounds. The jump is allowed to take place if the target address is within the function boundary, as the first hop shown in Figure 5. A failed check signifies a jump outside of the function boundary. In this case, an additional check is required to determine if the jump is the same as the call instruction check described above. However, there is no need to store the return address for a jump instruction.

For return instructions, when the return address is computed, it is compared against the value at the top of the FBS; if the check passes, the bounds for the current function are popped. If the FBS is empty, then a memory request is generated to access the SCS (to recover older bounds that were pushed to the SCS when the FBS was full). The entry that is retrieved from the memory has the base and bound information of the caller function, along with the return address. The equality of the two return addresses are then checked by the BR logic.

6 Security Analysis of BR

BR uses the Secure Call Stack (SCS) to track function bounds and call points allowing us to directly match returns to calls: this technique defeats return based attacks such as return-to-libc and ROP. Thus, our concern is Jump-oriented Programming (JOP) attacks [22–24]. BR limits the attacker from selecting arbitrary gadgets across the entire code base to construct exploit code. Instead, on a system with BR, the attacker is confined to stay within a function. As a result, the security analysis can be applied on a function by function basis. To carry out a JOP attack, the attacker must find all the necessary ingredients to launch and construct the attack *within the same function*; these ingredients include an exploit to initiate the attack, a Turing-complete set of gadgets to construct the attack, and a syscall to attack the system. There is a legitimate concern that a large function may still contain all these ingredients, allowing the attacker to mount a successful CRA.

Since it is very difficult to prove that a set of gadgets is not Turing-complete, we approach the problem by identifying the critical ingredients for a JOP attack and showing that they do not exist within a single function in the examined code bases. We focus our analysis first on identifying the critical dispatcher gadget needed for a JOP attack. We derive the conditions necessary for a dispatcher

gadget. All functions that do not have these conditions are guaranteed to be safe due to the absence of a dispatcher; we call the remaining functions *Dispatcher-Gadget Potential (DGP)* functions. The presence of the conditions does not necessarily mean that there is a dispatcher. Thus, we developed a tool to identify dispatcher gadgets; the functions that contain dispatchers are called *Dispatcher-Gadget Confirmed (DGC)* functions. Note that additional DGP functions may contain a dispatcher since the detection algorithm may miss some elaborate dispatcher gadgets. We focus further analysis on DGC functions only.

We analyze DGC functions for the presence of other ingredients. In general, BR provides security because it substantially limits the scope of the attack, making it extremely difficult (impossible for the libraries that we studied) to find all the ingredients necessary for an attack. For example, we analyze DGC functions for the presence of system call instructions without which the damage from the attack is limited to the compromised program. We show that none of the DGC functions in the libraries we analyzed has a system call. Another important ingredient is the presence of sufficient gadgets to build interesting attacks. We analyze this property and show that most DGC functions include only a small number of gadgets. Moreover, many of the gadgets in these functions are not usable for reasons such as the presence of side effects in the gadget that interfere with the dispatcher or interrupt the control flow, or because unintended instructions, which account for the majority of gadgets, are inherently more difficult to use. Another critical ingredient is the presence of a vulnerability to initiate the attack in the same function (we did not study the prevalence of vulnerabilities).

6.1 Dispatcher Gadgets

In ROP and JOP attacks, the attack proceeds as a sequence of gadgets separated by returns in ROP or indirect branches in JOP. We may view the stack pointer as a Gadget-Level Program Counter (GLPC) which is incremented to point to the address of the next gadget. In a JOP attack, an indirect jump is used to divert the control flow according to the GLPC. However, an indirect jump instruction cannot advance the GLPC to the next gadget address. Thus, a dispatcher gadget must have the necessary instructions to both increment the GLPC and to execute an indirect branch to the new gadget address. Without a dispatcher gadget, a JOP attack cannot be mounted [22–24].

Formally, a dispatcher gadget includes **an iterator** that advances the GLPC to point to the address of the next gadget, then either **a loader** that loads the address into a register and **a register indirect branch** which jumps to the address held in that register; or **a memory indirect branch** which jumps to the address held in the memory location pointed to the GLPC. Note that, the necessary operations defined here can be carried out in many possible ways. For example, a `pop` instruction acts both as an iterator and a loader since it modifies the stack pointer and loads from the stack. A dispatcher gadget may also include extra data movement instructions, called **conveyors**, that set the value of a register using another register or a memory location. Extra dereferencing, loading or copying operations are not of importance as long as the GLPC is altered in a predictable way so that the attacker knows where and how to place the gadget addresses in the memory. We use two different approaches for evaluating whether a function has a dispatcher gadget, which are explained in the following two subsections.

6.1.1 Identifying Dispatcher-Gadget Potential (DGP) Functions

For each function, it is safe to assume that a dispatcher gadget does not exist if at least one of the following two conditions is true: (i) there are no indirect branches or no valid instruction preceding an indirect branch: in this case, there are no viable gadgets; or (ii) there are no instructions that modify the targets of any of the indirect branches: in this case, there can be no dispatcher gadget.

As stated previously in the threat model, the attacker has full control over the register contents and data memory at the initiation of the attack. However, after initiating the attack, if none of the registers used by indirect branches found in the function can be modified or no memory location can be modified (as per the second condition above), then the number of executable gadgets is limited by the initial values. A limited number of executable gadgets means that for any initial state, it is known whether it will halt or not and therefore it is not Turing-complete. Note that the presence of the above conditions does not necessarily mean that DGP functions have dispatcher gadgets. Therefore, as a second approach we attempt to detect actual dispatcher gadgets inside DGP functions.

6.1.2 Identifying Dispatcher-Gadget Confirmed (DGC) functions

We adapted the methodology used by prior works to discover dispatcher gadgets by looking for specific dispatcher patterns. We extend prior algorithms for dispatcher gadget discovery to construct a more flexible discovery algorithm that searches for more complicated dispatcher gadgets that have no limits on the number of instructions or on the number of registers used. This approach covers the *trampolines* introduced in [23], all cases explained in [22] (as well as multi-register schemes) and *control gadgets* presented in [24].

The dispatcher discovery algorithm is shown in Algorithm 1. It starts by building the gadget trie as described in [9]. In a gadget trie, indirect jump instructions are represented as immediate children of a dummy root node. A child node of an indirect jump represents a possible decoding of an instruction preceding the parent instruction (an example gadget trie with its root at the bottom is given in Figure 8). Once the trie is constructed, the algorithm traverses the nodes starting with an indirect branch toward its children, trying to identify possible dispatchers as described in Section 6.1. For memory indirect branches, the algorithm looks for an iterator that advances the registers used in addressing the memory. In case of register indirect branches,

an instruction that loads to the register is needed first. If found, the algorithm then looks for an iterator for the source registers of loaders. Loading can occur either directly from memory or through conveying a value from other registers that are in turn loaded from memory (Algorithm 2). `find_setter` function returns the closest instructions on each path from a node to its descendants that sets a given target register. Iteration is achieved through arithmetic instructions that use the destination also as a source operand. Addition/subtraction instructions or `lea` instructions that explicitly specify the source register as their destination are considered as potential iterators. `find_iterator` function is identical to the `find_loader` algorithm, except instead of checking for a load operation, it checks if the setter is an instruction that iterates its target (such as `add`/`sub`/`lea`) and calls itself recursively.

Algorithm 1: find_dispatcher_gadgets($function$)

```
1  D ← {}                                              // dispatchers
   /* for trie building see Galileo in [9]                        */
2  trie ← build_gadget_trie(function)
3  foreach indirect jump: j in trie do
4      if j is a register indirect jump then
5          t ← target register of j
6          foreach loader: l in find_loaders(j, t) do
7              if l is also an iterator then                // e.g. pop
8                  D ← D ∪ {l}
9              else
10                 foreach source register: r that l uses do
11                     D ← D ∪ find_iterators(l, r)
12                 end
13             end
14         end
15     else if j is a memory indirect jump then
16         foreach source register: r that j uses do
17             D ← D ∪ find_iterators(j, r)
18         end
19     end
20 end
21 return D
```

Algorithm 2: find_loaders($node$, $target$)

```
1  L ← {}                                                  // loaders
2  S ← find_setters(node, target)                          // setters
3  foreach setter:s in S do
4      if s loads from memory then
5          L ← L ∪ {s}
6      else if s is a conveyor then                        // mov
7          foreach source register: r that s uses do
8              L ← L ∪ find_loaders(n, s)
9          end
10     end
11 end
12 return L
```

6.2 Other Attack Considerations

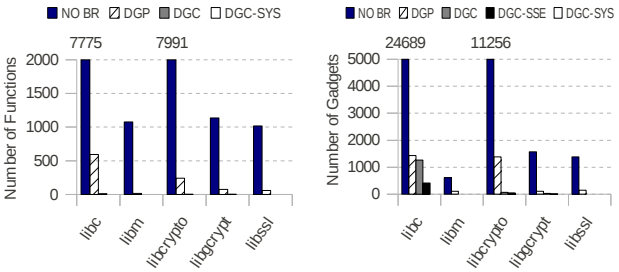
Another ingredient needed for the attack is a system call: without a system call, the attack's damage is limited to the process. For example, a system call is needed to achieve privilege escalation or to communicate critical data.

In addition, even if a dispatcher gadget is found in a function, the attacker also needs a set of functional gadgets to be able to perform general computation. These operations include arithmetic and logical operations and control flow instructions. However, a dispatcher gadget that operates on registers uses some registers as source operands. If one of these registers is modified by a functional gadget, it is considered to have a side effect that disrupts the dispatcher gadget, unless it can be patched up by another functional gadget. Since each gadget that is useful for the attack can have multiple unneeded extra instructions before reaching the indirect jump, with each having their own side effects, recovering from side effects is difficult when only a limited number of gadgets is available to the attacker.

Finally, BR does not allow the attacker to move from a vulnerability to an arbitrary location in the code. In other words, the attacker must find a vulnerability in the same function where the code reuse attack will be attempted. This additional requirement further degrades the attacker's ability to launch an attack.

7 Security Evaluation

We implemented a software routine that uses libdisasm to disassemble code sections of ELF binaries and analyze them to identify gadgets. As our codebase, we selected the GNU C Library (libc) and the C mathematics library (libm), which are commonly linked to programs. Also we evaluated GNU's library of cryptographic building blocks (libgcrypt), the cryptographic library of OpenSSL (libcrypto) and the Secure Socket Layer library of OpenSSL (libssl), which are critical in applications communicating over the Internet. We compiled the libraries on an Intel x86 (32-bit) Ubuntu Linux 3.0.0 machine with GCC-4.6.1.



a) Eliminating Functions under BR

b) Eliminating Gadgets under BR

Figure 7. Analysis of Functions and Gadgets

Figure 7 presents the number of vulnerable functions (left) and the maximum number of available gadgets (right) remaining in a vulnerable function. The bars labeled NO BR represent the baseline case where the entire library is available to the attacker. Bars labelled DGP and DGC show the statistics for the DGP and DGC functions respectively. Bars labelled DGC-SYS refer to the DGC functions with system call instructions. The DGC-SEE bar in Figure 7-b shows the number of gadgets left after the gadgets with side effects are eliminated. Almost 95% of the functions are safe. For the remaining functions, we discovered 15

dispatcher gadgets in 12 different functions of libc, 3 dispatcher gadgets in 3 functions of libcrypto and 2 dispatcher gadgets in 2 functions of libgcrypt. As shown in DGC-SYS bar of Figure 7-a, no function that has a dispatcher gadget also has a system call in the libraries examined for this study.

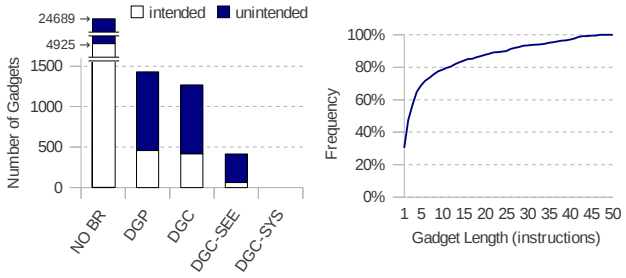
The above analysis demonstrates that BR completely protects all the libraries that we examined, because no DGC functions in these libraries have a syscall instruction. To provide more insight into BR, we continued our analysis of functions to further illustrate the difficulty of attacks even if the requirement of executing a system call is not present. As shown in Figure 7-b, the vulnerable codebase is reduced by 90% even when only the provably secure functions (non-DGP) are omitted. It is further reduced almost in half when we consider only the DGC functions. We examined the remaining gadgets in these functions and eliminated two thirds of them due to their side effects on the program state (see Section 6.2). Even if the system call requirement was not considered, the remaining potentially vulnerable code base shown in DGC-SEE bar, is only 1.2% of the entire code base. Moreover, recall that the attacker must find a vulnerability in the function that it selects for attack, making it further unlikely that these functions can be attacked.

An example gadget trie obtained from the analysis of the `d2i_RSA_NET_2.isra.0` function which is part of the libcrypto binary is depicted in Figure 8. The trie is constructed using an algorithm similar to the Galileo Algorithm defined in [9], except the gadgets here are ending with indirect jump instructions instead of returns. Each node of the trie represents an instruction, while a path from one node all the way down to the root forms a gadget. For example, the dispatcher gadget found in this function is represented as three nodes at the bottom of the middle sub-trie, `pop ecx; jg 0x000E0FF3; jmp ecx`. The attacker has to ensure that the conditional branch instruction (`jg`) is not taken in order not to change the semantics of the dispatcher gadget.

In fixed instruction length architectures, given a byte stream and either a starting or ending point, there is only one way of decoding an instruction sequence. However, for architectures with variable instruction length, a given byte stream and a starting point has only one meaning as we decode forward, but a byte stream with a given ending point may have several different meanings as we decode backwards. These possible different meanings for the same byte stream and ending point correspond to the branches in the trie. This characteristic provides the attacker with more gadgets in variable instruction length architectures.

In the dispatcher gadget shown in the Figure 8, GLPC is the stack pointer (`esp`) and `ecx` is a temporary register which is overwritten with the next gadget address. A `leave` instruction is equivalent to `mov esp, ebp; pop ebp` and when a gadget containing the `leave` instruction is executed, it overwrites the GLPC, leading to a problem in the dispatcher gadget unless the damage is recovered possibly by some other gadgets. The two `leave` instructions shown in Figure 8 are therefore said to have side effects. Since the nodes above these instructions have side effects, these gadgets are not usable, and the analysis of the remainder of this sub-trie can be stopped. The remaining gadgets consist of unintended instructions, which tend to be more difficult to use and often require complementary gadgets to prepare operands and fix side-effects; these complementary gadgets are difficult to find within the same function.

Similar analysis of other functions reveal that unintentional gadgets are of limited use without complementary gadgets not present in the same function. Figure 9-a shows the breakdown of intentional and unintentional gadgets for libc results, previously shown in Figure 7. The unintentional gadgets account for almost 70% of all gadgets. Additionally, we observe that the utility of gadgets is inversely proportional to gadget length. In a gadget that has many instructions, intermediate instructions are more likely to destroy the machine state used by the attacker. Figure 9-b shows the cumulative frequency histogram of gadget lengths found in the function with the maximum number of gadgets for DGC-SEE, which has 415 gadgets. This histogram shows that only 30% of these gadgets have a single instruction before the indirect jump. Therefore, 70% of the gadgets have at least one intermediate instruction and over 30% of the gadgets have at least five intermediate instructions.



a) Intended – Unintended Breakdown

b) Gadget Length Histogram for DGC-SEE

Figure 9. Gadgets for libc-2.14.1 with Intended and Unintended Branches

Figure 10 shows the cumulative percentage of gadgets present, first for all functions in libc, then in DGP, DGC and DGC-SEE functions. Specifically, the functions are sorted in order of the number of gadgets available for exploitation, with the largest number first. The largest number of gadgets are written (as "max.") on each figure. Only a few functions account for most of the remaining gadgets; most functions have very few gadgets and cannot be exploited. For example, even though there are 12 DGC functions, gadgets contained in only four functions with largest number of gadgets accounts for 96% of all gadgets and 91% when gadgets with side effects are eliminated (DGC-SEE).

8 Performance Evaluation of BR: Methodology and Results

For evaluating the performance of BR, we used PTLsim [54] - a cycle-accurate x86 processor simulator. We simulated a 4-wide issue out-of-order core with 64KB L1 data and instruction caches, 512KB L2 cache and 2 MB L3 cache. Memory latency was assumed to be 100 cycles. We

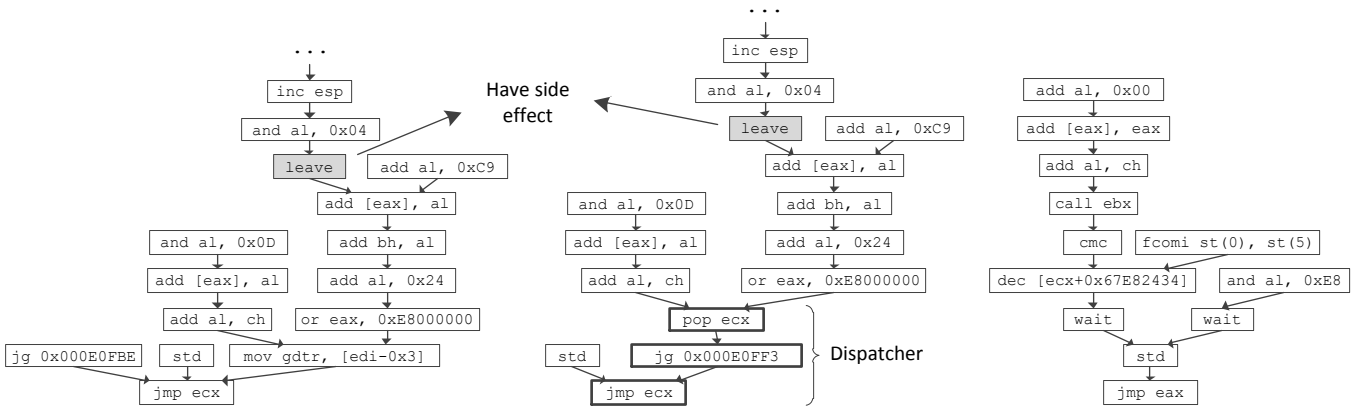


Figure 8. Gadget Trie of `d2i_RSA_NET_2.isra.0` Function

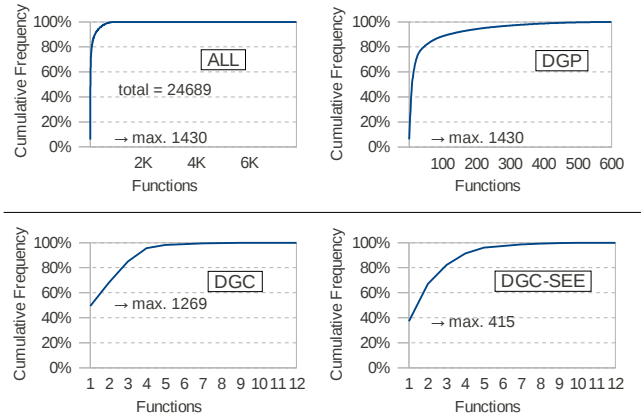


Figure 10. Gadget Distribution in Functions in libc-2.14.1

used 18 C and C++ SPEC CPU2006 [55] benchmarks for our experiments. We performed assembly-level instrumentation of the binaries for both BR and CFI to insert the additional instructions needed to perform the checks for both schemes. Note, that due to the high complexity of CFI technique, we only performed behavioral simulation of it for comparison purposes, without keeping track of the actual meta-data or constructing a full CFG. The benchmarks were compiled using GCC-4.2 compiler on a x86 machine running Ubuntu with kernel version 2.6.24.

Each benchmark was simulated for 2 billion committed instructions after fast-forwarding for the first 100 million instructions. Since Control Flow Integrity [25] technique relies on the use of Vulcan binary rewriting tool [56] which is not publicly available, we simulated CFI in the following manner. We generated the assembly files (using -S flag) and instrumented them using a script that inserts the extra control flow checking instructions described in [25]. Our goal was to only support behavioral simulation of CFI to measure its performance and binary overhead compared to BR. The CFI performance results obtained in this manner are consistent with what was reported in [25]. For evaluating BR, we similarly inserted the annotations to the assembly files to model the increase in the code size and also simulated the hardware structures to model the performance impact of BR.

The binary size increase of CFI and BR is compared in Figure 11. On the average across all benchmarks, CFI has about 4% increase in the binary size, while BR has less than 1% increase. While xalancbmk and omnetpp benchmarks exhibit about 9% overhead, bzip2 and mcf have no overhead for both CFI and BR.

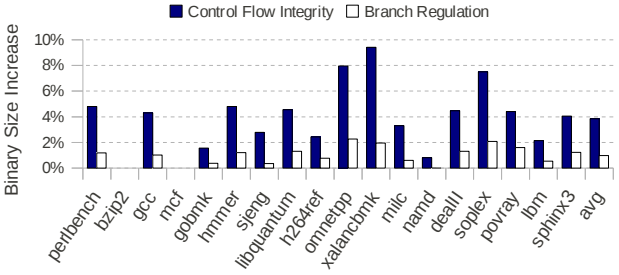


Figure 11. Executable Size Increase

Next, we evaluate the performance impact of BR and compare it against that of CFI. For BR, the performance is sensitive to the size of the hardware Function Bounds Stack (FBS in 6) - the smaller stack will result in more FBS misses for the return instructions and more memory accesses. Performance of BR as a function of FBS size is shown in Figure 12.

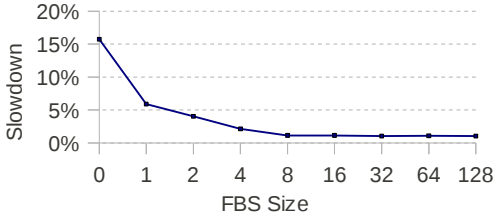


Figure 12. Effect of FBS Size on Performance

As expected, when the FBS size increases, performance impact due to BR overhead become smaller. On the average, the degradation is about 2% for the FBS size of 4 entries and it is about 1% for the FBS size of 8 entries and

above. The remaining degradation in performance is due to slightly higher number of misses in the instruction and data cache. The individual benchmark results are presented in Figure 13 for a system with a 4-entry FBS compared to the performance overhead of CFI.

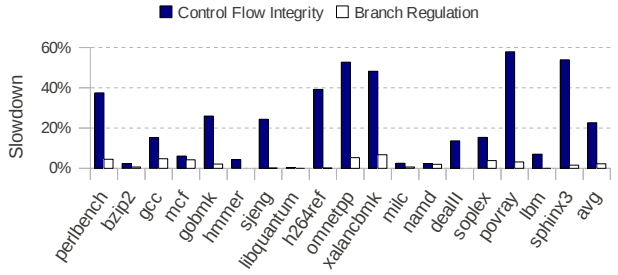


Figure 13. Performance Overhead of Branch Regulation

For a 4-entry FBS, performance overhead of BR is 2.14% on the average and it is less than 7% for all benchmarks. This compares favorably to 22% performance overhead for CFI on the average across all benchmarks. Figure 14 shows the percentage of relevant control flow instructions in the dynamic instruction stream. As shown, performance overhead strongly correlates with the number of control instructions. However, sphinx3, dealII, namd and lbm do not behave similarly since FBS performs better for these benchmarks. Because of the high FBS hit rate in these benchmarks, it is less likely that a memory access for obtaining the function bounds will be required.

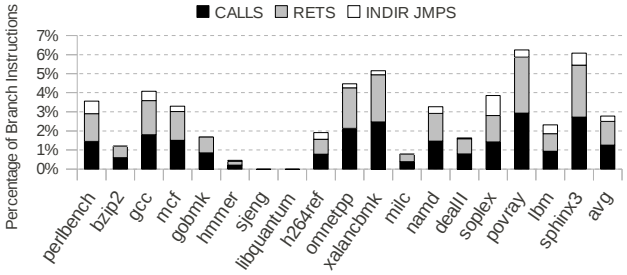


Figure 14. Prevalence of Calls, Returns and Indirect Jumps

9 Concluding Remarks

In this paper, we presented Branch Regulation (BR), a new low-overhead defense mechanism against Code Reuse Attacks (CRAs). BR limits the target addresses of branches to be either within the same function or at the start of another function, with the exception of return statements that are matched to prior calls. By preserving these simple invariants, we show that BR dramatically reduces (to 1% of the original number) the ability of the attacker to find exploitable gadgets needed for the CRA. In addition, we demonstrated that no CRA (including return and jump-oriented programming attack) is possible with BR in place for five sample libraries, including the standard C library and several cryptographic libraries, when the attacker has to use a system call instruction as part of the attack.

We demonstrated that the security benefits of BR are achieved at a very modest cost: about 2% performance loss, about 1% binary size increase, simple hardware at the execution stage of the pipeline, and simple binary annotations based on the information that is readily available from the symbol tables.

10 Acknowledgements

We would like to thank our shepherd Ruby Lee for her insight and suggestions, and the anonymous reviewers for their useful comments; the paper is significantly improved thanks to them. This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-09-1-0137 and by National Science Foundation grants CNS-1018496 and CNS-0958501. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of Air Force Research Laboratory, National Science Foundation, or the U.S. Government.

References

[1] Aleph One. Smashing the stack for fun and profit, Nov. 1996.

[2] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2:20–27, July 2004.

[3] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conf.*, pages 251–262, 2000.

[4] T. cker Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDCS'01*, 2001.

[5] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of USENIX Security*, volume 7, 1998.

[6] J. McGregor, D. Karig, Z. Shi, and R. Lee. A processor architecture defense against buffer overflow attacks. In *Proceedings of ITRE*, pages 243 – 250, aug. 2003.

[7] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX Annual Technical Conf.*, pages 211–224, 2003.

[8] S. Designer. "return-to-libc" attack, 1997. Bugtraq.

[9] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS*, pages 552–61. ACM Press, Oct. 2007.

[10] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of CCS*, pages 27–38. ACM, 2008.

[11] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the avc advantage. In *Proceedings of EVT/WOTE*. USENIX/ACCURATE/IAVoSS, aug 2009.

[12] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-Oriented Programming without returns on ARM. Technical report, System Security Lab - Ruhr University Bochum, 2010.

[13] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of CCS*, 2008.

[14] F. Lindner. Developments in cisco ios forensics. confidence 2.0. presentation, 2009. http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf.

[15] T. Dullien and T. Kornau. A framework for automated architecture-independent gadget search, 2010.

[16] T. A. Edward J. Schwartz and D. Brumle. Q: Exploit hardening made easy. In *Proceedings of USENIX Security*, 2011.

[17] R. Hund, T. Holz, and F. C. Freiling. Returnoriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of Usenix Security*, 2009.

[18] R. G. Roemer. Finding the bad in good code: Automated return-oriented programming exploit discovery. Master's thesis, University of California, San Diego, 2009.

[19] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of ACM STC*, pages 49–54. ACM, 2009.

[20] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of ICISS*, pages 163–177. Springer-Verlag, 2009.

[21] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of EuroSys*, pages 195–208, New York, NY, USA, 2010. ACM.

[22] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of ASIACCS*, pages 30–40. ACM, 2011.

[23] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of CCS*, pages 559–72. ACM Press, oct 2010.

[24] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of ASIACCS*, pages 20–29. ACM, 2011.

[25] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of CCS*, pages 340–353. ACM, 2005.

[26] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguardtm: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of USENIX Security*, pages 7–7. USENIX Association, 2003.

[27] H. Etoh and K. Yoda. Propolice: Improved stack-smashing attack detection. *IPSJ SIG notes on computer security*, Oct 2001.

[28] Vendicator. Stack shield technical info file v0.7, January 2001. http://www.angelfire.com/sk/stackshield/.

[29] R. B. Lee, D. Karig, J. P. McGregor, and Z. Shi. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of SPC*, pages 237–252, 2003.

[30] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[31] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer. Architecture support for defending against buffer overflow attacks. In *Proceedings of Workshop on Evaluating and Architecting Systems for Dependability*, 2002.

[32] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of USENIX Security*, pages 5–5. USENIX Association, 2001.

[33] P. Team. Pax non-executable pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt.

[34] S. Andersen. Part 3: Memory protection technologies. In *Changes to Functionality in Microsoft Windows XP Service Pack 2*. Microsoft Corp., 2004. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[35] P. Bania. Security mitigations for return-oriented programming attacks. *CoRR*, abs/1008.4099, 2010.

[36] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. Gfree: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of ACSAC*, pages 49–58, 2010.

[37] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of ICSE*, pages 162–171. ACM, 2006.

[38] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. In *Proceedings of the ASPLOS*, pages 103–114, New York, NY, USA, 2008. ACM.

[39] S. Ghose, L. Gilgeous, P. Dudnik, A. Aggarwal, and C. Waxman. Architectural support for low overhead detection of memory violations. In *Proceedings of DATE*, 2009.

[40] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of OSDI*, pages 147–160. USENIX Association, 2006.

[41] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. *Security and Privacy, IEEE Symposium on*, 0:263–277, 2008.

[42] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of ISCA*, pages 482–493. ACM, 2007.

[43] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri. SIFT: A low-overhead dynamic information flow tracking architecture for smt processors. In *Proceedings of CF*, May 2011.

[44] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of ASPLOS*, pages 85–96. ACM, 2004.

[45] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of NDSS*, feb 2005.

[46] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of MICRO*, pages 135–148. IEEE Computer Society, 2006.

[47] P. Team. Pax address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt.

[48] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of PLDI*, pages 158–168. ACM, 2006.

[49] O. Whitehouse. An analysis of address space layout randomization on windows vista, 2007.

[50] T. Newsham. Format string attacks, September 2000. http://julianor.tripod.com/bc/tn-usfs.pdf.

[51] A. Sotirov and M. Dowd. Bypassing browser memory protections. In *In Proceedings of BlackHat*, 2008.

[52] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *Proceedings of HPCA*, pages 1–12. IEEE Computer Society, 2010.

[53] T. O. Group. IEEE Std 1003.1, 2004. `http://pubs.opengroup.org/onlinepubs/009695399/functions/setjmp.html`.

[54] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of ISPASS*, pages 23–34, 2007.

[55] C. D. Spradling. Spec cpu2006 benchmark tools. *SIGARCH Comput. Archit. News*, 35(1):130–134, 2007.

[56] A. Edwards, H. Vo, A. Srivastava, and A. Srivastava. Vulcan binary transformation in a distributed environment. Technical report, MSR-TR-2001-50 Microsoft Research, 2001.