Getting in Control of Your Control Flow with Control-Data Isolation

William Arthur
University of Michigan
warthur@umich.edu

Ben Mehne*
University of California – Berkeley
bmehne@eecs.berkeley.edu

Reetuparna Das Todd Austin
University of Michigan
{reetudas, austin}@umich.edu

Abstract

Computer security has become a central focus in the information age. Though enormous effort has been expended on ensuring secure computation, software exploitation remains a serious threat. The software attack surface provides many avenues for hijacking; however, most exploits ultimately rely on the successful execution of a control-flow attack. This pervasive diversion of control flow is made possible by the pollution of control flow structure with attacker-injected runtime data.

Many control-flow attacks persist because the root of the problem remains: runtime data is allowed to enter the program counter. In this paper, we propose a novel approach: Control-Data Isolation. Our approach provides protection by going to the root of the problem and removing all of the operations that inject runtime data into program control. While previous work relies on CFG edge checking and labeling, these techniques remain vulnerable to attacks such as heap spray, read, or GOT attacks and in some cases suffer high overheads. Rather than addressing control-flow attacks by layering additional complexity, our work takes a subtractive approach; subtracting the primary cause of contemporary control-flow attacks. We demonstrate that control-data isolation can assure the integrity of the programmer's CFG at runtime, while incurring average performance overheads of less than 7% for a wide range of benchmarks.

1. Introduction

The software attack surface constitutes a substantial threat to computer security. Software vulnerabilities facilitate a wide array of security exploits: buffer overflows, heap spray attacks, return-to-libc, integer underflow, code gadgets, and a host of others. Today, the risk of software exploitation has escalated beyond DDOS attacks and amateur attacks such as the *Melissa* virus [11]. In the commercialization of the malware industry, new and more serious threats have emerged such as *Crimeware*, which perpetrate identity theft for the purpose of monetary gain [15]. As most attacks are conducted within the application layer [12]. Control-flow attacks, which permit arbitrary code execution, have emerged as a primary means to exploit software.

Our work drives to the heart of pervasive control-flow attacks by directly attacking the root of the problem: user-data derived control-flow. Contemporary research to protect control-flow has been focused on verifying the user data to be injected into the program counter (PC) [1,6,13,17,26,30,33,34] in an effort to establish trusted user data for control-flow targets. These previous works approach control-flow security by layering additional complexity on top of user data in an effort to shield the vulnerability from attack. In this work we adopt a subtractive approach by removing the actual vulnerability. *We simply do not trust any user data, and instead remove all avenues for such data to be injected into the program counter.*

1.1. Control-Flow Attacks

Control-flow attacks implement the redirection of program execution to attacker-selected code, either injected as user data or existing code in the form of code gadgets. These attacks violate, at runtime, the control flow graph (CFG) of an application by corrupting the PC with user-injected data, thereby allowing a program to execute a control edge not defined by the programmer.

Control-flow attacks exploit an inherent weakness ubiquitous in software development: determination of control-flow target addresses at runtime. It is the enmeshed relationship between the Program Counter and runtime data which creates the fundamental weakness of software to control-flow attacks. The classic example of such an attack is the stack buffer overflow. When input to a buffer exceeds the pre-allocated size on the program stack, the return address in the stack frame may be overwritten. In this case, the user data is used as the target of a return instruction, which can then jump to malicious code including the input buffer on the stack.

As a critical element of software exploitation, considerable effort has been expended to address control-flow attacks. Countermeasures such as stack protection, Address Space Layout Randomization (ASLR), and Non-Executable Data (NXD) have been widely adopted. Though many counter-measures have been devised [1,6,7,26,27,30,34], control-flow attacks remain a pervasive threat to computer security [35] due to the persistence of mixing runtime data with program control. Recently, mitigating techniques such as *Control Flow Integrity* (CFI) [1] and its descendents [33,34], Program Shepherding [17], and taint analysis [13] have been proposed. These techniques, which propose increased security through verification of runtime data, retain several vulnerabilities. Some are susceptible to CFG forgery attacks or allow the PC to target the middle of a basic block (or even the middle of an instruction). They also place constraints on their threat models that weaken their

* Portions of this work were completed while the author was with the University of Michigan.

1-4799-8161-8/15/$31.00 ©2015 IEEE

protections, such as the requirement of non-executable data or the assumption that an attacker cannot read or infer the contents of data memory. Additionally, most works do not address call-graph based control flow (i.e., dynamic library calls and returns). In this work, we relax the constraints of previous work, by assuming that the attacker has free reign over all of data memory (read, write, and execute), while also addressing the important issues of call-graph protection and dynamically introduced code such as shared libraries. The limitations of previous works are discussed further in Section 7 and Table 2.

1.2. Control-Data Isolation

Previous works attempt to mitigate control-flow attacks through *verification* of the runtime data which enters the program counter. Though this additional layer infers increased security, it nevertheless leaves the original, fundamental vulnerability: user data is injected directly into the PC. By contrast, this work eliminates arbitrary control flow by *eliminating the connection that exists between the PC and user data*, a technique which we call Control-Data Isolation (CDI). By disallowing the use of runtime data as control-flow targets, the programmer can ensure that all executions adhere to their specified control-flow graph (CFG).

In this paper, we implement CDI by generating code without the use of return and indirect jump/call instructions, the two types of instructions in modern architectures that connect user data and the PC. This creates some challenges in creating arbitrary code, in particular for calls/returns, indirect function calls, and shared libraries, but we show in Sections 2 and 3 how to implement (and subsequently optimize) these code sequences without the use of indirect control-flow instructions. The programs we create *completely sever the link between the PC and user data, and if the entire system adheres to the principles of control-data isolation, all control changes are limited to valid CFG edges, eliminating the way attackers execute control-flow attacks today.*

1.3. Contributions of this Work

The goal of this work is to identify the common thread of software exploitation and directly address the root cause: control targets derived from user data. In this work, we make the following contributions:

- We present an effective, efficient, and scalable approach to enforcing the CFG of an application at runtime. We implement control-data isolation (CDI) as a compilation-based transformation to existing software applications and library code. We advance the state-of-the-art in control-flow attack protection by targeting and eliminating the root cause: the injection of user data into the program counter.
- We present an LLVM-based compiler implementation that generates control-data isolated code for non-trivial programs and shared libraries, eliminating the use of indirect control flow in compiled programs.
- We analyze a diverse set of programs and design and evaluate targeted, profile-guided optimizations to improve the performance of control-data isolated code.
- We evaluate the efficiency of CDI, showing through detailed experiments that the performance and storage costs are minimal, less than many of the previously proposed control-flow attack mitigation techniques.

The remainder of this paper is organized as follows. Section 2 provides an in-depth analysis of CDI. Section 3 details our implementation approach of eliminating all indirect control flow, while Section 4 addresses dynamic code from shared libraries. Section 5 provides detailed analysis of our LLVM compiler-based implementation, PitBull. Experiments testing our method and a full analysis of results are delivered in Section 6. Finally, Section 7 evaluates related works, and Section 8 highlights conclusions and future work.

2. Protecting Control Flow with Control-Data Isolation

Control-flow attacks work by injecting malicious runtime data into the program counter of a susceptible target process. They are a divergence from the programmer-defined CFG of an application, occurring when an attacker creates new control-flow edges from user data at runtime. This can take many forms such as return-oriented programming, heap spray attacks, stack smashing, and even hijacking calls to library functions.

2.1. Threat and Trust Model

The goal of a control-flow attack is to subvert the control flow of a vulnerable process and execute code of the attacker's choosing. In this work, we consider the attacker to play a powerful role. An adversary is assumed to possess arbitrary read, write, and execute privilege to data memory, including the stack and heap. That is, we start from the position that an attacker controls all of data memory. In traditional compilation techniques, many control-flow target addresses are derived from or stored in data memory; hence, once an attacker gains some level of read/write/execute control over data memory, there are typically many avenues to direct program flow to code of their choosing. This is precisely how control flow attacks are currently accomplished.

We do make the assumption that the attacker cannot arbitrarily overwrite executing code segments at runtime. We see this assumption of non-writable code (NWC) as a fundamental element of security. Without this one protection, the attacker could simply substitute their own code for that of the application, obviating the need for control-flow attacks. Similarly, the program loader is trusted, as a compromised loader could simply replace system code with malicious code at load time. It is important to note, however, that the loader can be protected against attacks with CDI, in the same way as other applications.

An important aspect of our relaxed threat model is the assumption that data segments, specifically the heap, may contain executable code. As long as the non-writable code requirement is met, an application may execute code in the heap with full CDI protections. Previous works including all works based on CFI [1], expressly forbid the execution of code on the heap. This requirement is due to their

susceptibility of forgery attacks. As they rely on labels placed at target locations, a heap spray attack could create forged labels which fraudulently identify malicious code as acceptable targets for an indirect call or jump. Our work is not susceptible to this attack, as all targets are embedded into the existing programmer-specified and loader-blessed instructions, eliminating the need to trust destination labels.

The key element of both our threat model and CDI principle is that user data expressly cannot be trusted. An important distinction between CDI and previous works such as CFI [1], and its descendents [7,33], is their use of a shadow stack [24] to secure all `return` instructions. As the shadow stack is resident in data memory, it is inherently susceptible to attack and requires additional protection measures, increasing the potential attack surface. CDI provides the same protection against control-flow attack for all indirect instructions, obviating the need to trust or shield user data.

2.2. CDI Threat Protection

The implementation of CDI eliminates the possibility of any runtime data being used as a control-flow target address. In this work we accomplish this goal by disallowing the execution of indirect control-flow instructions. Simply put, an indirect jump, call, or return will never be executed. This eliminates the critical element pervasive to control-flow attacks. Without these instructions, stack smashing, heap spray, buffer-overflows, return-to-GOT, and return-to-libc attacks are crippled, as they all rely on the ability to derail the control-flow of a process, currently achieved by polluting the data value of indirect control-flow targets. Further, the availability of useful code gadgets and any remaining control-flow attacks are diminished to legal traversals of the program's CFG, since it is not possible to jump to the middle of a basic block (or instruction). By addressing, and removing, the root of the problem we can significantly reduce the software attack surface by limiting control to the programmer-specified CFG. The extent of the protection is determined by the degree to which the code running on the machine adheres to CDI principles. If all code running utilizes CDI, then user-injected data cannot find its way into the PC, and the system is hardened against control-flow attacks. To facilitate this ultimate goal, we focus on CDI-based compilation for applications, libraries, and dynamically introduced code objects, such as shared libraries.

In our relaxed threat model, we enable code to be executed in data space. This supports the use of a prevalent technology previous works have not: just-in-time compilation (JIT). JITted code presents challenges to CDI implementation, such as jump tables for loop unrolling. However, problems analogous to this have already been addressed by our work for similar structures such as the global offset table (GOT). Though we do not inhibit just-in-time compilation, it is beyond the scope of this work and remains a prime target for future work.

2.3. Achieving Higher Levels of Protection by Isolating Control and Data

The threat model defined above creates many opportunities for ambitious attackers to achieve arbitrary code execution. Some instructions, namely indirect control flow instructions, derive control flow, in whole or part, on runtime data. When an attacker gains some level of control over data memory, this runtime data can be manipulated in a malicious manner, permitting an attacker to use (and abuse) the programmer's indirect jumps at will. This can be observed in attacks such as code gadgets, heap sprays, and buffer overflows. These attacks must, at some point, rely on a control-flow target derived from user data which may be injected by an attacker.

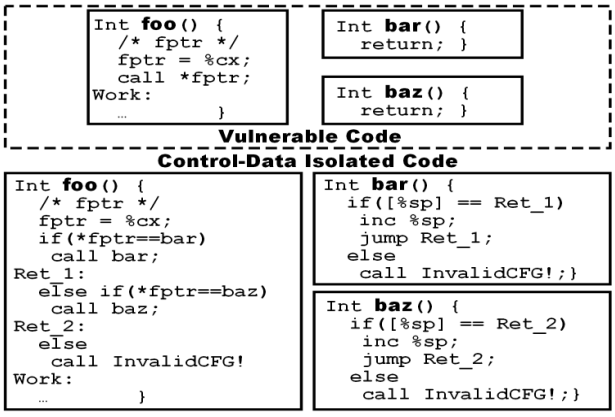

Figure 1 - CDI Control Flow Protection. Indirect branches are converted to direct conditional branches, severing the link between potentially malicious runtime data and the program counter.

To assure that the program's execution adheres to the CFG defined by the programmer, we isolate control-flow from runtime data. To achieve this end, we focus on all control flow decisions at runtime including those which are encapsulated in the executable code objects and control transfers in between. Thus with CDI, all valid edges in the CFG of an application are encoded in the programmer-specified and loader-blessed instructions of an application. This CFG functions as the golden model which completely defines the valid control flow of an application. That is to say, any dynamic paths which adhere to the CFG are potentially secure, but any paths which violate the CFG are explicitly insecure. By embedding all control-flow targets within programmer-written instructions, rather than derived from user data, we eliminate the weakness in software which enables all control-flow attacks.

Figure 1 depicts a simple code sequence vulnerable to control-flow attacks, and an equivalent code sequence constructed with CDI that is protected from control-flow attacks (the full details of this process are discussed in Section 3). *To prevent exploitation of indirect control flow instructions, we simply remove them from software.* The indirect branches are replaced by direct branches, which only allow predetermined know-valid edges. The permissible targets of these instructions, i.e., `Ret_1`, `Ret_2`, `bar`, and `baz`, are identified via CFG discovery.

The control-data isolated code has no avenue for potentially malicious runtime data to be injected into the PC. As such, all target addresses of control-flow come from the programmer-specified text segment of an application. By eliminating the use of indirect instructions, attacks such as Stack Smashing become impossible to implement directly on the programmer's CDI-protected code. Similarly, attacks such as Heap Spray attacks rely on the execution of a control-flow instruction which derives its target from data memory. Even rootkits, where 96% of *Linux* rootkits integrate control-flow attacks [22], rely on subverting data which is injected into the program counter. Additionally, return-oriented programming (ROP) attacks, including those without any function calls, are defeated as these attacks rely on an initial derailment of the control-flow from the CFG by user data injected into the PC.

Implementing CDI requires validation of all control targets, which in turn requires complete knowledge of the CFG. Indirect control flow instructions such as function pointers make control flow graph discovery a challenge. In spite of this, previous works have demonstrated that the task of CFG discovery is achievable [1,7,28,31,34,33]. Our CFG discovery approach is addressed in-depth in Section 3.1. Another key challenge, often overlooked by previous works, is control flow transfer between dynamically-linked objects such as shared libraries. Our work solves this issue, as detailed in Section 4.

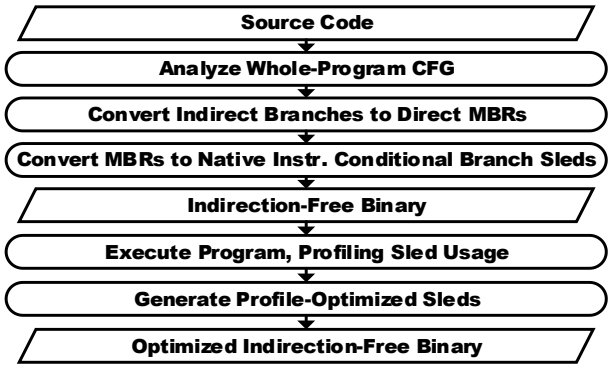

Figure 2 - CDI Compilation Flow. CDI-protected, indirection-free code is generated from application source code. This process converts indirect control-flow to direct branching, which is then profiled to optimize runtime performance of CDI-hardened code.

Indirect control flow is an intrinsic part of modern software, so its removal has the potential to adversely impact the performance of programs. We address this concern by leveraging profile-guided code generation to efficiently select validated targets, which is detailed in Section 3.3. We develop an efficient, effective CDI software implementation which assures the runtime integrity of a program's CFG, demonstrated in Section 5. At first glance, it may appear that eliminating indirect control flow will inherently result in program slowdowns. However, previous research into *devirtualization* demonstrates that such a process is utilized to facilitate program speedups [3,14,16]. Devirtualization is the process by which dynamic virtual function calls are replaced with object test and direct calls, similar to the process depicted for `fptr` in Figure 1. By leveraging superior branch prediction, devirtualization has been proven to improve execution speed in the object-oriented languages to which it has been applied.

3. CDI via Elimination of Indirect Control Flow

The work of creating software free of indirect control flow can be accomplished at varying stages in software development. In this work we propose a combination of a compile-time and load-time solutions that eliminate the use of indirect instructions in binaries. To achieve this, we must discover the CFG of an application and from it identify the indirect branching instructions and their control-flow targets. This information is used in eliminating indirection by substituting hard-coded, direct control flow into the target application. We also implement and identify several optimizations to apply when creating applications free of indirect control-flow.

An overview of this approach is shown in Figure 2. The CDI process begins by discovering the CFG of an application, and subsequently identifying all indirect control flow instructions, i.e., returns and indirect jumps and calls. These are then converted to multi-way branches (MBRs) and a complete target set for each MBR is then identified. A *sled* of conditional branch/direct jump pairs, one for each target, is substituted for each MBR. The sled does the work of converting indirect jumps to direct ones, by comparing the proposed target one-by-one with all of the validated potential targets of the indirect jump. An example of a sled is depicted in Figure 1 by the instructions

```
if(*fptr==bar) call bar;
else if(*fptr==baz) call baz;
```

which are substituted for the vulnerable indirect call. When a matching target is found, a direct jump is made to the validated target; otherwise, an invalid control-flow decision is declared. The resulting code is dynamically profiled and optimized for performance. This process is studied in further detail in Section 3.3.

3.1. CFG Discovery

To enforce the golden-model CFG at runtime, a complete CFG which encapsulates all possible paths through a program for non-trivial software applications must be determined. Lifting binary code to determine the CFG of an application is both an active and well researched topic [2,4,5,28]. However, such a task is made difficult specifically due to indirect control flow. Previous works such as CFI have been able to determine the precise CFG from binary analysis, while in this work we obtain such information from our *LLVM*-based compilation flow.

We shall only consider indirect control flow instructions for CFG analysis, as direct control flow instructions are both trivial for building a CFG and they are not subject to code injection attacks (given the non-writable code assumption of our threat model). The key issue, then, to constructing a runtime invariant CFG is to determine the set of all possible targets for each and every indirect control flow instruction, as shown in Figure 3.

```
for each instruction inst in application
 if type(inst) == return
  target_set(inst)=all instruction after
       call_sites
 elseif type(inst) == indirect_call
  target_set(inst) = all function where
       function_type(function) == call_type(inst)
 elseif type(inst) ==indirect_jump
  target_set(inst) = all instruction where
       instruction == target(inst)
 elseif type(inst) == virtual_call
  target_set(inst) = all function where
       vptr(inst) ∈ vtable(function)
 elseif type(inst) == optimized_switch
  target_set(inst) = all instruction where
       instruction == case(inst)
 elseif type(inst) == function_pointer_call
  target_set(inst) = all function_ptr where
       function_type(function_ptr) ==
       function_type(inst)
 replace inst with multi-way_branch mbr where
  target_set(mbr) == target_set(inst)
```

Figure 3 - Indirect Instruction Target Set for CFG Construction. For each individual indirect jump, call, or return, all allowable control flow edges must be determined prior to executing the code.

Considering software at a low level, indirect control flow may be categorized into three groups: jumps, calls, and returns. Indirect jumps, such as those arising from `switch` statements, are implemented for performance when the `case` set for a `switch` statement is large. At compile time, the target set of basic blocks for the `case` statements is known, making resolution of control flow edges simple. Other indirect jumps often have but a single target, e.g., process linkage table (PLT) entries. These must be resolved at load time for shared library linking. In any case, the exact address will be known at least by load time, thus, the potential targets of indirect jumps is knowable before execution begins.

Direct function calls and returns may be resolved from the call graph for an application. Indirect calls and their returns, however, are a special challenge which arises from programming constructs like function pointers. Pointer analysis in general is difficult for compilers, limiting optimization possibilities. However, in terms of CFG construction, function pointer analysis has distinct advantages over conventional pointer analysis. Most compilers, including *gcc* and *g++*, enforce function pointer assignment by argument and return types. We leverage this knowledge for greater precision in call-graph CFG analysis. There are special conditions which can work to defeat efficient function pointer analysis, such as function pointer casting and return type casting, using data types such as `void *`. However, a complete and correct (but perhaps conservatively constructed) CFG remains determinable. In the worst-case analysis, a function pointer may be assumed to reach any function. Performing function pointer analysis provides a more concise CFG, which further reduces potential code gadgets. Concurrently, this also improves runtime performance by reducing the size of conditional branch sleds for indirect function calls.

Virtual functions are implemented as indirect calls via the `vptr` attribute. Previous work has shown that these may be converted to direct calls by source code rewriting [18]. During compilation however, the same essential information for vtable implementation, i.e. class inheritance and overriding, is leveraged to derive a valid target set for a `vptr` directed call.

Returns are the most prevalent of all indirect instructions. In theory, the potential set of targets for any return can be determined by identifying all call sites for a function. In practice this does not always hold true, as programming constructs such as tail calls must be detected to reveal the true target. By reverse CFG walking, all reachable paths are found to determine possible return targets.

Position independent code (PIC) are code objects where the resolved address of any instruction is not known until the library is loaded. This presents a special challenge to discovering the CFG when considering objects compiled with PIC. However, the CFG for this code is fully discoverable at compile-time, as the underlying information about target sets for multi-way branches is available without dependence on addressing information.

3.2. Indirection Elimination

Elimination of indirect control-flow is the heart of this work. This severs the link between potentially insecure data and the program counter. Once a complete CFG has been constructed for an application, indirect control flow is no longer necessary for correct execution.

Indirection elimination is the process by which indirect control flow is replaced by direct control flow. The most straightforward approach is to replace an indirect branch with an equivalent set of conditional branches. This construct, called a sled, tests a potential target address against the known set of valid targets identified by CFG discovery. For example, a `return` statement would be replaced by a series of `if...then` statements, where each `if` statement tested a potential known-valid return address, which if matched would lead to a direct jump to the valid target. This process is depicted in returning from functions `bar()` and `baz()` in Figure 1. After complete indirection conversion has been achieved, all targets are reached by direct jumps or calls. Consider the event where an attacker is able to corrupt the data for a return, i.e., stack smash. All potential valid targets will be tested against the tainted value, which will fail to redirect control flow. At the end of any sled, a direct call to an abort function is inserted. This allows for the graceful exit of the program under attack, which can also be used to collect information on the attack.

Though elegant, CDI may introduce inefficiencies to runtime performance. Some instructions, particularly returns, may have a large set of valid control transfer targets. Performance implications are explored in detail in Section 6.

3.3. CDI Performance Optimizations

Assessing potential runtime implications of CDI, there are two major elements which may contribute to a degradation in performance. The first is the number of targets for each multi-way branch. A large set of valid targets will generate a correspondingly large sled of conditional branches. This creates both a larger binary and the potential to execute a greater number of instructions before taking the intended edge. There are several ways to address this concern.

Multi-Way Branch Target Ordering. A significant optimization is the profile-guided ordering of conditional branches in multi-way branch sleds, the process of which is shown in Figure 4. Dynamic profiling of edge counts can dictate insertion order of conditional branches. Complex orderings could be envisioned, such as tuning for branch prediction accuracy. However, the simple method of ordering edges by descending execution frequency provides a highly effective way to minimize the average number of not-taken branches which must be executed before arriving at the correct edge.

Single Target Set Reduction. The simplest optimization is the reduction of single-target indirect instructions to unconditional, direct jumps.

Frequent Function Cloning. Another simple optimization is function duplication for frequently called functions, which can proportionately reduce the set of valid return targets for each individual function clone. This optimization works well for small functions with many call sites.

Large Target Set Resolution. This optimization replaces a series of conditional branches with another mechanism which has either constant or logarithmic time complexity, e.g., a binary search tree. Any search method would incur some overheads, creating a minimum threshold to seek an alternate for a series of conditional branches. For example, a long series of conditional branches where the first is almost exclusively taken will execute faster than a search over the same targets in the average case.

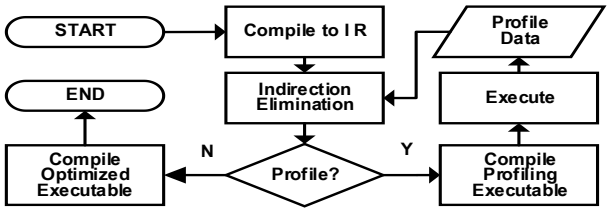

Figure 4 - Dynamic Profiling for CDI Optimization. The runtime performance of multi-way branches implemented with direct conditional branches is greatly impacted by target ordering. By profiling target execution counts, we leverage inherent branch bias and order conditional branches by execution frequency.

The second major performance factor in indirection elimination is branch prediction performance for the inserted conditional branches. Branch mispredictions have non-trivial impact on runtime performance of applications. As such, addressing the predictability of the extra branches inserted to eliminate indirection is a concern. The only controllable dimension to conditional branch insertion is their ordering. Choosing an ordering by execution frequency, ascending or descending, provides the average-case performance benefit of executing the most predictable branches first. Both the overwhelmingly taken and never taken branches will be nearly perfectly predictable. However, ordering with the most oft-taken branches first provides the added benefit of executing less untaken branches in the best and average case.

3.4. Detecting Attacks in CDI Protected Programs

When a control flow attack occurs on a CDI protected program, the realized effect is to exhaust the list of allowable targets in a conditional branch sled without taking any edge. This will also happen in the event of a non-malicious data corruption bug affecting a potential control-flow target. When this happens, the application will instead directly call a handler routine which gracefully exits the program. This handler can aid in debug/diagnosis by obtaining information about the crash, in the form of a unique ID for the call and the offending target address. This data can then be analyzed to determine the nature of the unexpected control edge.

To prevent control-flow attacks, it is essential to disallow any control flow which violates the predetermined CFG. All control flow is classified as either authorized or illegal, to facilitate our relaxed attack model (only a single illegal edge is needed to perform a heap spray attack). By disallowing all illegal CFG edges, we remove the essential element of control-flow attacks, thereby hardening software against them.

4. CDI Implementation for Shared Libraries

Not all control flow edges originate and terminate within a target binary. Many applications make calls to functions in dynamically-linked libraries at runtime. In order to provide protection for any application, the library code it calls should also adhere to the principle of CDI. To achieve this, we extend the use of indirection elimination to shared libraries.

4.1. Dynamic Nature of Shared Libraries

Shared libraries are referred to as such because a single copy of the library can be loaded once into physical memory and shared at multiple start addresses by multiple processes running concurrently. Further, they are dynamically linked when an application is loaded into memory. The dynamic nature of shared libraries make them a natural match for indirect control-flow. However, this also creates a natural vulnerability to control-flow attacks as well. An example of this is the return-to-libc attack [8], which circumvents non-executable stack protection to call attacker-desired functions in libc.

The dynamic nature of shared libraries, and their pervasive use of indirect control flow, presents new challenges for implementing CDI. These challenges include position independent code (PIC), the use of indirect jumps in the PLT in conjunction with the global offset table (GOT), and returns to potentially many different applications from a shared function in a library. Here we demonstrate the process of CDI in the context of shared libraries on *Linux*

systems, though similar methods would be applicable to other approaches such as Dynamically Linked Libraries (*DLL's*) for *Windows*.

The current implementation of dynamically-linked shared libraries on Linux operating systems works as follows. Shared library code is compiled separately from application code and linked together when the application is executed. This linking is accomplished by the resolution of shared symbols in the symbol table of all linked objects. Each function call to a shared library is facilitated by the PLT and the GOT. When a function is called, the application executes a direct call to the PLT entry in the application code associated with the shared library function. The PLT entry then executes an indirect jump to the function, the target of which is stored in the GOT. When the library function completes execution, control returns to the original call site.

To facilitate the sharing of libraries, the address of a shared library function in the virtual address space must be resolved, as this is typically a randomized location in the memory space due to ASLR. When a function is called for the first time, the target address of the PLT jump in the GOT will not target the desired library function, but instead the next instruction in the PLT entry. This is a direct jump to a helper function which will determine the actual address of the desired function, via the program loader using the symbol tables of the code objects. Once the target address is established, the corresponding entry in the GOT is overwritten with the actual address of the desired function. This process is called *binding*, typically seen as *lazy binding* where the binding between objects is done at runtime upon the first invocation of a library function. This introduces an inherent weakness to control-flow attack, as the GOT table of function addresses could be overwritten with data at runtime which is then directly injected into the PC at the next shared library function invocation. Attacks on the GOT due to this weakness have been demonstrated [9,25].

4.2. Enforcing CDI for Shared Libraries

Elimination of indirect control-flow removes the need to establish trust in user data. Target set resolution for MBRs remains the same process regardless of whether code is static, relocatable or position-independent. However, PIC code cannot contain absolute address references. To remedy this, all conditional branch/direct jump sleds are comprised of PC-relative address references. This allows all jumps and calls within PIC code to be implemented as direct jumps and calls.

In order to enforce CDI for shared library calls, our work eliminates the use of all indirect jumps implemented in the current structure using the PLT and GOT. An overview of our shared library implementation is depicted in Figure 5. Shared libraries remain separately compiled and linked by the program loader when an application is executed. As before, the PLT is used to invoke the library function. However, with CDI the program loader will overwrite the indirect jump instruction in the PLT entry with a direct call to the address of the library function, which was previously being written into the GOT as an indirect target. This is depicted in the application in Figure 5. We enforce dynamic linking at load time (i.e., non-lazy binding) before any runtime data is encountered. Thus, all control transfer targets are derived from programmer-specified instructions and the program loader, side-stepping any need to trust runtime data.

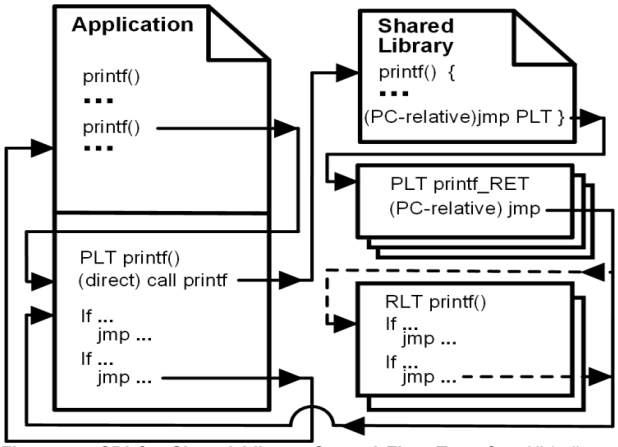

Figure 5 - CDI for Shared Library Control Flow Transfer. All indirect control flow is replaced by direct calls and jumps, resolved at load time and written to the PLT, obviating the need for the GOT in function calls. In the case where more than one object may invoke a library function within the same process, an RLT entry is created, which executes a sled to return execution to the calling code object PLT entry. This then selects the correct return point in the application. Each process has a unique copy of the PLT and RLT while continuing to share the library code.

The task of returning from a shared library call is the last challenge in eliminating indirect control flow, and it requires eliminating the use of the `return` instruction. Here we leverage the same mechanism used to call the library function: the PLT. The return instruction is replaced with a direct, PC-relative jump to a new PLT entry whose purpose is to return control-flow back to the calling code object. This PLT entry then contains a direct jump to one of two locations. In the case where only one dynamically-linked code object in a process address space may call a given function, the PLT of the called function contains a direct jump back to the PLT of the calling function. If there is more than one code object in a process address space which may call the library function (e.g., malloc() is called by both the application and library other than libc) then the single direct jump from the PLT will prove insufficient. In this case, a new code object is defined, referred to as the return linkage table (RLT). An RLT entry holds a conditional branch/direct jump sled which contains the return target addresses for all of the possible calling code objects within the address space of the process calling the library function. The PLT entry in the called function will then directly jump to its respective RLT entry. When the prospective return address is compared to the list of allowable targets and a match is found, the RLT then executes a direct jump to the target. The RLT is depicted in the shared library object in Figure 5.

The inclusion of direct jumps in the PLT and RLT require that they not be shared in memory (as they will differ for

each application). Thus, they are aligned on page boundaries immediately following the shared PIC code of the library. This facilitates the ability to reach the PLT for each application by the same instruction in the library function. Consequently, all the benefits of shared libraries are retained such as dynamic linking and a single copy of large libraries like glibc. Additionally, the elimination of indirection in the implementation of shared libraries effectively removes the ability to perpetrate GOT-based attacks and any attack which exploits a `return` instruction, as not a single `return` instruction will remain in any code executed by a process.

It should be noted that the elimination of indirection in PIC is greatly aided by PC-relative instructions in the x86-64 and ARM ISAs. In other ISAs such as 32-bit x86, PIC implementation is more complicated by lack of PC-relative jump instructions. In such a case, CDI can still be readily achieved. To accomplish this, sharing would be disallowed, and libraries would be implemented as relocatable code, which is identical in implementing CDI as application code.

5. PitBull: Compiler-Based CDI

To validate our control-data isolation enforcement via indirection conversion, PitBull (**P**ositive **I**ndirection elimination **B**y **LL**VM) was built. PitBull is a compiler optimization utilizing the LLVM compiler infrastructure [19]. The set goal was to establish feasibility for indirection-free executables. Of equal importance, this also facilitates the analysis of runtime performance implications of CDI.

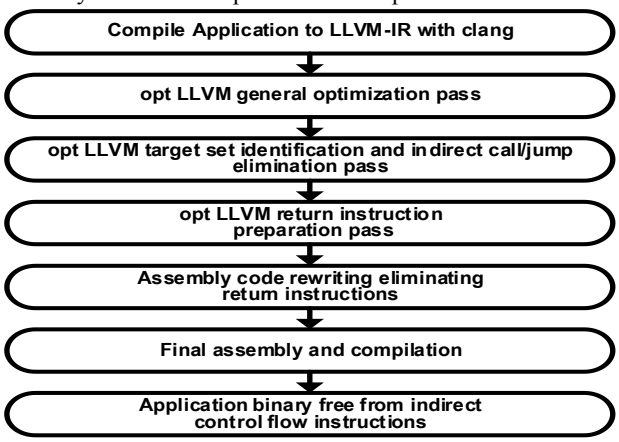

Figure 6 - PitBull Compilation Flow. Applications are compiled to be free of indirect control flow instructions. Leveraging the LLVM compiler infrastructure, optimization passes identify all valid targets of indirect control flow and insert conditional branches in replacement.

The LLVM optimization-based implementation of PitBull, shown in Figure 6, works as follows. The target applications are first compiled to LLVM-IR with the *clang* compiler. All IR files are then linked by the `llvm-link` tool. A standard optimization pass is then performed by the LLVM tool `opt`. At this point the target executable has been compiled into LLVM-IR and is ready for our indirection conversion optimization passes, invoked again with the LLVM tool `opt`. The primary pass first identifies the nodes and edges of the CFG relevant to indirect calls, jumps, and returns. Function pointer analysis is performed to identify control flow edges not readily available from the standard `-dot-callgraph` LLVM `opt` pass. Once the targets of indirect control flow instructions have been identified, indirect call and jump instructions are replaced with a series of `if..then (icmp..br)` statements. For each allowable target, a compare is made to the candidate target, followed by a direct jump to the allowable target. A second pass then aggregates call and return data in preparation for the ensuing assembly-level rewriting passes. The transformed LLVM-IR is then compiled to assembly via the LLVM `llc` tool.

At this point, indirect calls and jumps have been eliminated from the target application. Returns are then handled by assembly code rewriting. First, a label is placed after each call of the program. Next, all return statements are replaced with a series of compares and direct jumps. The set of valid return targets, provided by the first `opt` pass, have their corresponding newly inserted labels compared to the stack pointer. Each compare is followed by a conditional direct jump to the compared label. After all compares and conditional jumps are inserted, a terminating call to the attack handling routine is inserted. Since we substitute direct jumps for returns, the stack pointer is incremented before any compares, which then compares to the stack pointer minus one word.

All indirect instructions have now been replaced by compares and direct jumps. The code is then assembled and an executable is produced. We note that the process for relocatable code is exactly the same as that for static application code. When the code is relocated at load time, the relocation table will update all absolute address references with their new location, for both the compares and direct jump instructions.

To facilitate shared libraries, we also eliminate indirect control flow instructions in PIC code. This requires a more nuanced approach than static or relocatable code. The target sets for indirect branching instructions are identified exactly as before. However, the inserted sleds must be PIC-compliant and may not contain absolute address references. To implement this, our system instead utilizes PC-relative instructions. For each allowable target, a comparison is made to the candidate target, followed by a PC-relative direct jump.

In our framework, we do not implement load-time functionality. This means that our current LLVM compiler-based implementation does not provide our proposed CDI protection against GOT-based attacks. However, we do model overheads associated with dynamic library implementation, including non-lazy binding. Further, the additional sled added to the calling application PLT, depicted in Figure 5, is simulated by adding the sled as padding to the return location of the called library function. The execution of library calls would be nearly identical in performance, save the last step would be a direct call from the PLT. As stated for devirtualization in Section 2.3, this is expected to improve execution time for calls. Returns from libraries would potentially suffer from additional RLT entry traversal. However, this impact would be minimal, as in our benchmarks only 2% of control transfer from library calls

Benchmark	Optimized Runtime Overhead	Naïve Runtime Overhead	Binary Size Increase	Valid Control Flow Edges per Indirect Instruction							
				Static				Dynamic			
				Max	Mean	Median	Mode	Max	Mean	Median	Mode
gzip	0.7%	4.3%	14%	45	5.7	2	2	15	3.0	2	1
vpr	1.0%	2.5%	33%	116	5.1	3	2	20	2.7	2	1
gcc	34.4%	59.2%	112%	1946	30.3	8	2	648	5.5	2	0
mcf	0%	0%	10%	2	1.0	1	1	2	1.0	1	1
crafty	4.0%	10.0%	29%	33	5.8	3	2	16	2.4	2	2
parser	5.3%	39.3%	46%	216	8.3	3	2	151	6.1	2	2
eon	22.8%	51.2%	73%	114	7.2	2	0	33	0.5	0	0
perlbmk	20.9%	189.1%	148%	537	31.4	18	18	134	3.8	1	0
vortex	20.1%	143.0%	42%	192	11.7	4	3	158	8.7	2	1
bzip2	0.9%	1.5%	9%	3	2.1	2	2	3	1.4	1	1
twolf	0.7%	1.1%	24%	18	3.8	2	2	8	1.5	1	1
md5sum	0%	0%	25%	8	2.6	2	2	3	1.0	1	0
sha1sum	0%	0%	22%	8	2.6	2	2	3	1.0	1	0
sha256sum	0%	0%	20%	8	2.6	2	2	3	1.0	1	0
sha512sum	0%	0%	16%	8	2.6	2	2	3	1.0	1	0
bftpd	0%	0%	81%	109	6.8	2	2	41	1.5	0	0
tcpdump	0%	1.2%	174%	400	75.5	65	65	14	0.1	0	0
SPEC Avg.	10%	45.6%	49%	134	10.2	4.4	2	108	3.7	1.5	1
Average	6.5%	29.6%	52%	293	10.2	4.4	2	74	2.5	1.2	0

Table 1 - Control/Data Isolation Performance. The optimized runtime overhead from CDI appears in the first column. The last 8 columns detail indirect control flow edges metrics for benchmarks. The static details the properties of the CFG related to indirect control flow instructions. Dynamic data reflects the runtime control edges seen during execution. Together, these contrast dynamic and static properties of control flow. Runtime overhead can be seen to positively correlate with control flow edges.

require an RLT entry, while the remainder would jump directly from the PLT of the shared library to the PLT of the application.

6. Experimental Evaluation

To fully understand the runtime implications of CDI, the performance of our compiler implementation was evaluated. The testing platform consists of 64-bit x86 workstations running *Ubuntu* 12.04 LTS *Precise Pangolin* with Linux kernel 3.5.0-39-generic. Compilation and optimization is accomplished with clang and LLVM, both release version 3.3. All optimization passes are registered LLVM passes, while the assembly code rewriting is performed using *Perl*.

6.1. Benchmark Applications

Several security-sensitive and network-facing applications were chosen to evaluate runtime performance. These include sha1sum, sha256sum, sha512sum, and md5sum from the *GNU Coreutils* suite, as well as *tcpdump*, a popular network packet analyzer and *bftpd*, an ftp server. The *SPECINT2000* benchmarks were also included to allow a direct comparison between our work and earlier works such as CFI [1]. We further implemented CDI for the *musl libc* library [20], due to a known lack of compatibility between *clang* and *glibc*.

6.2. Performance Evaluation

SPEC benchmarks were executed with the standard runspec interface. Other benchmarks were executed while processing as input large, 45GB network capture files. Results are timed and averaged over 5 runs, shown in Table 1. The runtime overheads shown reflect the increase in runtime for benchmarks relative to the original, unmodified applications. Default compilation parameters are held constant for both original and modified binaries. The naïve runtime overheads represent the performance overhead without any subsequent optimizations. Ranging from almost zero to nearly 2X slowdown, the naïve implementation averages about 45% for all benchmarks. When optimizations are applied (as detailed in Section 3.3), we see a dramatic decline in the execution overheads for all benchmarks, where over half have no perceivable overhead at all. There is also a noticeable difference in runtime overheads between SPEC benchmarks and the network-facing applications. SPEC benchmarks, by design, are generally compute-intensive workloads. However, the remaining applications, such as tcpdump, typically have performance which is I/O bound. For these workloads, which are a prime candidate for CDI protection, the cost for such protection is hidden by I/O overhead.

6.3. Impact of Optimization

As observed in Table 1, optimization has a considerable impact on performance. There are two main factors which influence this; the heavily biased nature of dynamic branch execution, and the execution frequency of indirect control flow instructions. In our experiments, we implemented an optimization based on execution frequency of indirect branch targets. Benchmark applications were profiled to collect edge counts for the MBR edges. This information is then fed back into a second compilation. Edge counts are utilized to order conditional branch insertion for indirection conversion, by descending order of execution frequency. This yields the optimized performance shown in Table 1.

When considering the dynamic behavior of branches, it has long been known that branches are heavily biased to one particular branch direction during execution [32]. This

Work	Explicit Dependencies				Susceptible to these Attacks			Relies on Data Memory Security	Eliminates Usage of Indirect Control
	NWC	ASLR	W⊕X	Shadow Stack	Heap Spray	GOT	Read		
This work	Yes	No	No	No	No	No	No	No	Yes
Abadi et al. [1]	Yes	No	Yes	Yes	No	Yes	No	Yes	No
Xia et al. [30]	Yes	No	No	No	Yes	Yes	No	Yes	No
Budiu et al. [7]	Yes	No	Yes	Yes	No	Yes	No	Yes	No
Zhang, Sekar [33]	Yes	No	No	Yes	No	No	No	Yes	No
Kiriansky et al. [17]	Yes	No	No	No	No	No	Yes	Yes	No
Cowan et al. [10]	Yes	No	No	No	Yes	Yes	Yes	Yes	No

Table 2 - Related Works. The first set of columns details what system dependencies are explicitly required to maintain the purported security benefits of a work. The next set details which vulnerabilities a work provides no hardening against. The final column states whether a work eliminates the root cause of contemporary control flow attacks: indirect control-flow . Our work remains as the single one to harden against all control-flow attacks while maintaining only the most fundamental dependency of NWC. NWC=non-writable code, Shadow Stack=separate stack in memory to verify return address targets, GOT=Attacks on calls to libraries, Read=Technique is weakened if attacker can read or infer any data memory contents or locations.

biased property strongly facilitates the high accuracy of modern branch prediction.

Though indirect branches are more difficult to predict [16], they remain highly biased as well [30]. To assess this bias, we profiled execution of benchmarks to determine the distribution of dynamically executed targets, shown in Figure 7. When indirect control flow instructions are broken down into binary branch decisions, the resulting control flow points, taken individually, become more easily predicted than the original indirect branch. As shown in Table 1, the dynamic target set is considerably smaller than the static set. This highlights the crucial factor for runtime overheads; dynamic branching properties, not static, are the driving force behind runtime performance.

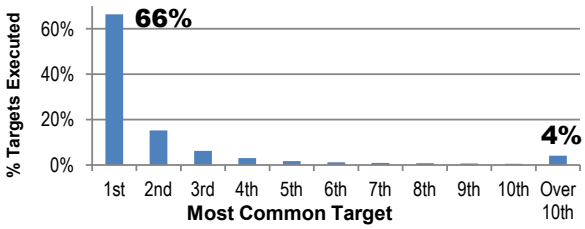

Figure 7 - Indirect Branch Bias. Branching instructions are heavily biased in execution. The percentage of instructions as a function of the most common targets for each indirect branch/call/return are binned by execution frequency. For all benchmarks combined, the most commonly executed target of each MBR accounts for over 66% of all edges executed.

The second property of software which heavily dictates the performance of indirection conversion is the relative execution frequency of indirect control flow. For all benchmark applications, indirect instructions accounted for 1% of the total instructions executed. Coupling this with highly biased branches, it is no surprise that indirection-free transformations incur minimal overheads.

As shown in Table 1, runtime overheads generally tend to be positively correlated with the size of target sets for indirect instructions. One indirect instruction in the *gcc* benchmark had 648 different valid targets executed at runtime (a `return` instruction). That implies that at least once, a function was forced to execute 647 not-taken conditional branches before finding the correct edge for a return. This highlights the opportunity for a low time-complexity alternatives to conditional branch insertion, as discussed in Section 3.3., which is left for future work.

7. Related Work

This work is conducted in light of many techniques which have been devised in an attempt to address control-flow attacks. An abridged list is presented here, divided into software and hardware approaches. Direct comparisons are summarized in Table 2.

7.1. Software Mechanisms

An important work in this area is Control Flow Integrity (CFI) by Abadi et al. [1], which spurred an avalanche of interest in the dynamic enforcement of software CFGs at runtime. The relatively low overhead, simple solution set a high bar for all subsequent efforts. In their work, the authors utilized a labeling system to verify the authenticity of a target address. Return instructions are guarded by a shadow stack, with the code segment register functioning as the shadow stack pointer. Though CFI is an elegant solution, it relies on a more restrictive attack model than our work while incurring greater execution overheads. Further, with reliance on a shadow stack located in data memory, and not addressing shared library calls and returns, there are continued concerns about control-flow attacks. In contrast, CDI's threat model assumes that the attacker fully owns data memory, with read, write, and execute privilege.

A recent work which expands upon CFI is Control Flow Integrity for COTS Binaries [33] by Zhang and Sekar. This work provides a solid implementation for CFI instrumentation for stripped binaries. However, it was not extended to protect control transfer between shared library and application code (e.g., GOT attack [9]). It also retains limitations from binary-rewriting such as not handling dynamic code generation.

G-Free [21] is a compiler-based approach to eliminating ROP attacks. This is a two-pronged approach of excising unintended return or return-like instructions, along with encryption-based verification of the context in which indirect branches are executed (e.g., a `ret` instruction is executed only after the first block of the function in which it resides has been executed). Though this appears to constrain code gadgets, the approach offers no protection from non-code gadget attacks such as heap spray and return-to-GOT.

Another foundational work is Secure Execution Via Program Shepherding by Kiriansky et al. [17]. Utilizing dynamic binary instrumentation, Program Shepherding enforces a security policy by monitoring control flow transfer at runtime. Though Program Shepherding could enforce a policy similar to CDI, it still cannot determine all valid indirect branching targets without nontrivial compilation support (such as what we propose in this work). The CFG as enforced by program shepherding is emblematic of the actual CFG, and therefore cannot offer the same level of protection as CDI. When an application's CFG is discovered at runtime, the targets of indirect jumps cannot be known before they execute, and therefore cannot be verified dynamically. This allows a jump to the middle of an x86-64 instruction, permitting unfettered code gadgets within an application.

Recently, compiler-based solutions have also been proposed. One such work is Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM [29], which instruments applications with CFI checks and labels at compile time. Though Tice *et. al.* achieve low runtime overheads, they do not address `return` instructions, which constitute the majority of indirect control flow. Their approach remains vulnerable to code gadgets, as the range verification for jumping to compiled executables allows jumping to the middle of instructions.

Another compiler-based approach to CFI is Control-Flow Restrictor: Compiler-based CFI for iOS [23]. In their work, Pewny and Holz share the most commonality with our work, applying a similar MBR conversion approach. However, focusing on iOS on an ARM platform, they do not explore large, complex applications with many functions or topics such as PIC or shared libraries.

7.2. Hardware Solutions

A variety of hardware-based solutions have been proposed to address control flow security. One example is Architectural Support for Software-Based Protection [7]. That work is an extension of the original CFI [1] work, with a proposed ISA extension to move CFI label checking into hardware. This work carries with it the same weaknesses as CFI. Another hardware solution is offered in the work CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters by Xia et al. [30]. This work leverages existing hardware in the form of performance counters. Though their solution has low overheads, it has to contend with deficiencies such as false positives and negatives, as well as allowing "suspicious" branch targets to execute, making the technique readily susceptible to heap spray attacks.

It should be noted that all previous works may be classified as mitigation techniques. That is, they all seek to guard, verify, or otherwise shield the root of the problem for control-flow attacks: indirect branching. As such, they all rely heavily on a host of security assumptions, which are in turn susceptible to attack. Regardless of the proposed solutions, of which there are many, control-flow attacks persist. In contrast, our work directly addresses the root issue and permanently removes it by isolating control from user data.

8. Conclusions

Computer security has become a dominant topic in the information age. The software attack surface has remained as a chief area of security exploit for years. Though vulnerabilities have been well studied, exploitations persist. Given the continuing nature of these attacks, this work directly addresses and eliminates the prevailing root of the problem: indirect control flow.

In this work we presented a novel approach to software security, called control-data isolation, which eliminates the link between potentially malicious runtime data and program control by eliminating the use of indirect control in generated software. We have shown that eliminating the root cause giving rise to the predominant mode of control-flow attacks is not only feasible, but has minimal impact on runtime performance. Control-data isolation provides a greater level of security than previous proposals while experiencing overheads that are comparable or better. We feel strongly that by directly addressing control-flow attacks, rather than mitigating them, the overall software attack surface can be greatly diminished.

8.1. Future Work

The implementation offered in this work demonstrates the possibility of indirection-free execution. There are many improvements which could be made to further enhance performance. Optimizing indirection conversion where a large number of target edges exist is a prime target for improvement. Due to the dynamic nature of computing, exploring hardware-based acceleration for CDI implementation would also prove a promising extension to enforcing CDI principles.

Acknowledgements

The authors would like to thank the reviewers, whose insights improved this work. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, Control Flow Integrity, *ACM Trans. Inf. Syst. Secur.*, 2009, pp. 1-40.

[2] K. Anand et al., A compiler-level intermediate representation based binary analysis and rewriting system, *Proc. of the 8th ACM European Conf. on Computer Systems*, 2013, pp. 295-308.

[3] D. Bacon and P. Sweeny, Fast Static Analysis of C++ Virtual function Calls, *Proc. of theConf. on Object-Oriented Programming, Systems, Languages, an Applications*, 1996, pp. 324-341.

[4] A. R. Bernat and B. P. Miller, Anywhere, any-time binary instrumentation, *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, 2011, pp. 9-16.

[5] A.R. Bernat and B.P. Miller, Structured Binary Editing with a CFG Transformation Algebra, *19th Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 9-18.

[6] T. Bletsch, X. Jiang, and V. Freeh, Mitigating code-reuse attacks with control-flow locking, *Proc. of the 27th Annual Comp. Sec. App. Conf.*, 2011, pp. 353-362.

[7] M., Erlingsson, U. Budiu and M. Abadi, Architectural support for software-based protection, *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, 2006, pp. 42-51.

[8] c0ntext. (2012, April) Bypassing non-executable stack during exploitation using return-to-libc. [Online]. http://css.csail.mit.edu/6.858/2012/readings/return-to-libc.pdf

[9] c0ntext. (2011) How to Hijack the Global Offset Table with pointers. [Online]. http://www.exploit-db.com/papers/13203/

[10] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, Point-Guard: Protecting pointers from buffer overflow vulnerabilities, *12th USENIX Security Symposium*, 2003.

[11] L. Garber, Melissa Virus Creates a New Type of Threat, *Computer*, vol. 32, no. 6, pp. 16-19, 1999.

[12] Gartner. (2013) Gartner. [Online]. http://www.gartner.com/technology/home.jsp

[13] J.L. Greathouse et al., Testudo: Heavyweight security analysis via statistical sampling, *41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 117-128.

[14] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, A Study of Devirtualization TEchniques for a Java Just-In-Time Compiler, *Proc. of theConf. on Object-Oriented Programming, Systems, Languages, an Applications*, 2000, pp. 294-310.

[15] Kaspersky. (2013) Computer Threats. [Online]. http://www.kaspersky.com/threats

[16] H. Kim et al., VPC prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization, *Proceed. of the Interntl. Symp. on Computer Architecture*, 2007, pp. 424-435.

[17] V. Kiriansky, D. Bruening, and S. Amarasinghe, Secure execution via program shepherding, *Proc. of the USENIX Security Symp.*, 2002.

[18] B. Kuhn and D. Binkley, An enabling optimization for C++ virtual functions, *Proceedings of the 1996 ACM symposium on Applied Computing*, 1996, pp. 420-428.

[19] C. Lattner and V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, 2004.

[20] (2014, February) musl-libc. [Online]. http://www.musl-libc.org/

[21] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries, *Proc. of the Annual Computer Security App. Conf.*, 2010, pp. 49-58.

[22] Jr., N. Petroni and M. Hicks, Automated detection of persistent kernel control-flow attacks, *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 103-115.

[23] J. Pewny and T. Holz, Control-Flow Restrictor: Compiler-based CFI for iOS, *Proc. of the Annual Comp. Security App. Conf.*, 2013, pp. 309-318.

[24] M. Prasad and T. Chiueh, A binary rewriting defense against stack based buffer overflow attacks, *Proc. of Usenix Tech. Conf.*, 2003, pp. 211-224.

[25] M. Ramilli. (2011, November) Global Offset Table Injection Procedure. [Online]. http://marcoramilli.blogspot.com/2011/11/global-offset-table-injection-procedure.html

[26] Y. Shi, S. Dempsey, and G. Lee, Architectural Support for Run-Time Validation of Control Flow Transfer, *International Conference on Computer Design (ICCD)*, 2006, pp. 506-513.

[27] Y. Shi and G. Lee, Augmenting Branch Predictor to Secure Program Execution, *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2007, pp. 10-19.

[28] H. Theiling, Extracting safe and precise control flow from binaries, *Proceedings. Seventh International Conference on Real-Time Computing Systems and Applications*, 2000, pp. 23-30.

[29] C. Tice et al., Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM, *23rd USENIX Security Symposium*, 2014.

[30] Y. Xia, Y. Liu, H. Chen, and B. Zang, CFIMon: Detecting violation of control flow integrity using performance counters, *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012, pp. 1-12.

[31] L. Xu, F. Sun, and Z. Su, Constructing precise control flow graphs from binaries, University of California, Davis, Tech. Rep. 2009.

[32] C. Young, N. Gloy, and M. Smith, A comparative analysis of schemes for correlated branch prediction, *Proceedings of the Iinternational Symp. on Computer Architecture*, 1995, pp. 276-286.

[33] M. Zhang and R. Sekar, Control Flow Integrity for COTS Binaries, *USENIX Security Symposium*, 2013, pp. 337-352.

[34] C. Zhang et al., Practical Control Flow Integrity and Randomization for Binary Executables, *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 559-573.

[35] T. Zimmermann and S. Neuhaus, Security Trend Analysis sith CVE Topic Models, *International Symposium on Software Reliability Engineering (ISSRE)*, 2010, pp. 111-120.