Verified and Optimized Inlined Reference Monitors

by

Bin Zeng

A written document submitted in partial fulfillment
of the requirements for the general exam

in
Computer Science and Engineering Department
Lehigh University
Bethlehem, PA 18015

September 2013

Abstract

Ever since Morris worm was released into the wild from MIT by Robert Morris in 1988, computer security has gained increasing attention from both end users and programmers. However, most programmers are still not capable of writing secure code, partially because security incurs extra burden and distracts programmers from application logic. Computer security is still considered to be the job of a few security experts.

Unsafe language is one of the banes of software attacks. Current system softwares are mostly written in unsafe languages such as C and C++ for a variety of reasons. Software attacks sabotage program executions by inducing abnormal control flow transfers to injected shellcode, existing libraries, and code snippets already in application programs, or manipulating data pointers to steal or corrupt sensitive information. By embedding security checks into subject programs during compilation, Inlined Reference Monitors (IRMs) can prevent many attacks effectively. Many software attacks can be foiled effectually because they violate the security properties IRMs check at runtime. However, traditional implementations of IRMs either incur heavy runtime overhead, or lack retargetability and reusability. In addition, their trustworthiness is questionable without verification. In this thesis, we propose solutions to these problems. To improve efficiency, we apply aggressive optimizations to eliminate unnecessary embedded security checks. Retargetability is achieved through modeling security checks in machine-independent representations. The trustworthiness is ensured by verification. We have built a framework on top of the industrial-strength compiler LLVM for IRMs. Experimental results show that the trustworthiness, efficiency, and practicality are all improved significantly over previous work. In the end, we summarize the work and discuss future research.

1 Introduction

In recent years, computer attacks have increased steadily even if a large amount of resources have been invested into bug hunting, intrusion detection, vulnerability mitigations, and other defenses. Software is the weak link in the security chain between end users and their valuable assets due to the complexity of modern software. Software attacks continue to plague computer industry and end users. As a recent example, Twitter, Facebook, Apple, and Microsoft all have been hacked successfully in 2012 or 2013 [28]. Private user information has been stolen and users were even forced to change their passwords afterwards. The victims including the companies and the end users suffered enormous economical loss and intangible reputation damage [28].

One of the major culprits is the ubiquitous use of unsafe languages such as C and C++. Although many type-safe languages such as Java, C#, Python and Ruby et al. gained popularity and started to replace weakly-typed languages in some areas, many software systems are still written in C and C++ for various reasons. A full transition to safe languages is unlikely to happen any time soon. Even for many safe languages, their trustworthiness is questionable because their compilers, virtual machines, interpreters, and runtime system are mostly written in C and C++. For example, Java has many security issues with its virtual machine and runtime system even if the language is considered to be type-safe [34].

Through embedding security checks into programs written in unsafe languages, IRMs can mitigate many vulnerabilities. IRMs can insert security checks into programs at source level, at IR (Intermediate Representation) level, at assembly level, at binary level after linking, or even at runtime during dynamic binary translation. IRMs can be fully automatic and does not require much intervention from programmers, thus incurring almost no extra burden on programmers. Furthermore, IRMs do not distract programmers from their program logic because they require little programmer intervention. In this thesis, we focus on instrumentation during compilation because we believe compilers serve as the center for program analyses and transformations.

IRMs can foil many software attacks. As a concrete example, Control-Flow Integrity (CFI) [3] inserts dynamic checks before computed control flow transfers to ensure that execution paths follow predetermined control flow graphs, which can be constructed by source-level static analyses, binary analyses or even program profiling. Many attacks including traditional buffer overflows and other arc-injection attacks can be effectively foiled by CFI because their execution paths violate expected control flow graphs. As a simple example, conventional stack-based buffer overflow attacks overwrite return addresses, pushed onto the stack by call instructions, to point to injected shellcode, thus inducing abnormal control flow transfers. CFI ensures that return instructions can only return to the instructions after their calls, thus preventing buffer overflow attacks. As another instance, return-to-libc attacks cannot happen with CFI enforced because the attacker-induced control flow transfers to libc functions do not exist in the predetermined control flow graph.

Another example is Software-based Fault Isolation (SFI) [23, 29, 35, 40]. SFI is widely used to isolate untrusted programs from trusted environment. A carefully designed interface is the only pathway through which the untrusted programs can interact with the trusted environment, thus preventing untrusted programs from corrupting the trusted environment.

IRMs for unsafe languages does not come without challenges. Some previous work has focused on enforcements of IRMs for type-safe platforms such as JVML. IRMs for unsafe languages bring additional challenges because no memory safety or control safety is guaranteed for programs written in unsafe languages. The metadata for IRMs has to be carefully placed and protected from being corrupted. A certain level of control flow restriction is also needed so that the embedded security checks cannot be circumvented by adversaries. For type-safe languages such as Java, the language itself enforces type safety which implies memory safety and control safety needed by IRMs. The metadata for IRMs cannot be corrupted by adversaries without breaking the type system first. Similarly, embedded checks cannot be circumvented because control flow cannot jump to the middle of a basic block with type safety enforced. In addition, IRMs for unsafe languages such as C and C++ are often enforced on low-level representations, thus exposing all the nasty details that need to be handled carefully.

1.1 Metrics for Software Security

While some security mechanisms are widely deployed such as stack guard [13], others see little adoption in the industry because they are either untrustworthy, inefficient, impractical or a combination of them. One of the main reasons why it is hard to evaluate security mechanisms is because there is a serious lack of practical metrics. In general, security mechanisms are evaluated according to three key metrics:

- **Trustworthiness**. How trustworthily the security mechanism defends against a wide vector of security attacks. No panacea defense exists for all software attacks. A specific security mechanism target certain computer attacks for a certain period. For example, CFI can mitigate arc-injection attacks but cannot prevent data corruption attacks. No security mechanisms can stay ironclad forever. Given sufficient time and resources, all security mechanisms can be broken. Some security mechanisms can foil attacks temporarily but can be bypassed or circumvented when encountered with well-resourced and patient adversaries, given sufficient incentives. As a simple example, Address Space Layout Randomization (ASLR) [9, 10, 20] is effective against many attacks that require the exact memory locations of memory objects. However, it can be circumvented or even broken through brutal-force attacks, or information leakage if the subject programs are poorly-randomized and the entropy is low [4, 17, 31].

- **Efficiency**. How much cost the security mechanism incurs. While we wish security mechanisms incur zero overhead, everything comes with a cost. Some security mechanisms are lightweight while others incur heavy runtime overhead. Heavyweight security mechanisms can be used in security-critical systems such as the backend of banking systems while they see little adoption in performance-critical systems. Mere ten percent of performance improvement can save thousands of server racks in a typical data center.

- **Practicality**. How easy it is to deploy the security mechanism in current production systems. Many security mechanisms are incompatible with legacy code or require even major hardware modifications. Some security mechanisms are intrusive while others are composable and can be added as an additional layer to the current system or even embedded into the programs with little extra efforts. Some even require rewriting the whole system in a different language.

Traditional implementations of IRMs do not satisfy all the key metrics simultaneously, or striking an unsatisfactory tradeoff among them, thus seeing poor adoption in production systems. For example, the original CFI implementation ensures that execution paths follow a control flow graph through binary instrumentation [3]. The implementation incurs 16% slowdown on SPECint2000 on average. The runtime overhead is tolerable for some systems but hardly acceptable for performance-critical systems. We have proposed a few optimizations such as jump table check optimization, jumping over prefetch instruction, and efficient ID encoding instruction etc. to cut down the runtime overhead of CFI. Experimental results demonstrate that the runtime overhead of our implementation is significantly lower than previously published work.

As another example, conventional implementations of SFI either leverage hardware segmentation to enforce data sandboxing, or adopt software-based approaches by inserting security checks, modeled as instruction sequences, into subject programs. However, memory reads are not checked because they are much more common than memory writes and thus incur heavy runtime overhead. In contrast, we use a generic software-based approach, which can be applied to various architectures, and sandbox both memory writes for integrity and memory reads for confidentiality. The novel approach enables optimizations to reduce the runtime overhead of sandboxing both memory reads and writes to a new low level.

1.2 Thesis Statement

Unfortunately, none of the existing implementations of IRMs satisfies all of the aforementioned metrics. Either they are untrustworthy, inefficient, or their practicality is questionable. This dissertation research addresses the problems with current implementations of IRMs to meet the aforementioned metrics.

Programs written in unsafe languages such as C and C++ are inherently vulnerable. IRMs can mitigate numerous attacks. How do we improve IRMs' trustworthiness, efficiency and portability through compiler-based optimizations and transformations?

1.3 Summary of Contributions

During the investigations and explorations of novel techniques to enhance current IRMs towards aforementioned metrics, this thesis makes the following high-level contributions.

1. A reusable and retargetable framework is proposed, built and evaluated to enforce low-level IRMs on a high-level IR. It is built on top of the industrical-strength compiler LLVM to enforce various IRMs at a machine-independent level, i.e. LLVM IR, which is also language-agnostic and optimization-friendly. It enables portability and improves efficiency of IRMs.

2. A series of aggressive optimizations are applied to reduce the runtime overhead of IRMs without weakening the trustworthiness of IRMs. These optimizations remove provably unnecessary security checks to increase the runtime performance of IRMs. According to our experimental results, the runtime overhead of our framework is cut down to a new level.

3. A general mechanism is introduced to verify the soundness of software-based sandboxing techniques. Based on range analysis, the verifier can statically validate whether certain sensitive operations are safe. The verifier is powerful to verify the results of our optimizations and many more.

4. A novel constraint language to encode the constraints that should be respected by program optimizations and transformations in order to preserve security is invented. Constraints in the constraint language can be carried across program transformations and verified at the very end to validate whether the security assumptions are still sound.

5. A compiled-based framework for dynamic taint tracking is proposed. The novel framework achieves language agnosticism and machine-independency through IR-level instrumentation. To the best of our knowledge, this is the first published framework to do taint tracking in a compiler IR.

This thesis is organized as follows. In Sec. 2 we introduce closely related work and and analyze their deficiencies. Then we describe the attack model and security policies in Sec. 3. To improve runtime efficiency and trustworthiness, we introduce a series of optimizations to cut down the runtime overhead and a verifier to validate the correctness of instrumentation and optimizations in Sec. 4. To enhance the reusability and portability of our framework, we introduce the first implementation of target-independent framework to enforce IRMs for unsafe languages in Sec. 5. Finally, we conclude and enumerate future research directions in Sec. 7.

2 Background and Related Work

Next, we introduce closely related security mechanisms and IRMs including CFI, SFI et al. and analyze their strengths and deficiencies.

IRMs insert checks to guard sensitive operations. Positioning checks before sensitive operations is not new. As a common practice to improve software reliability, developers insert assertion statements into source programs to validate the state of program executions. At intermediate representation level, researchers have been doing bounds checking to ensure memory safety [6, 8, 16]. At machine code level, machine instruction sequences are inserted to enforce certain security policies such as CFI [2, 3, 36, 41] and SFI [23, 29, 35, 40, 41]. Clearly, this is a promising and well-studied research area. Subsequently, we will discuss SFI, CFI and other IRMs.

2.1 Software-based Fault Isolation

SFI, also known as sandboxing, isolates untrusted and faulty modules from trusted system to ensure coarse-grained memory safety [23, 29, 32, 35, 40]. A carefully designed interface is the only pathway through which untrusted modules can interact with the trusted system. The untrusted modules and the trusted system usually live in the same virtual address space. For programs written in C and C++, an arbitrary value such as an integer can be cast to a pointer and then used to access memory. Thus, it poses a challenge to ensure untrusted modules do not access memory outside their sandbox.

The original SFI by Wahbe etc. was proposed for software extension isolation and later on it was adopted for software security [35]. Oftentimes large softwares enrich their functionalities through plug-ins, add-ons or other extensions [23, 33]. Those extensions may come from untrusted sources or contain vulnerabilities that render the whole system weak, thus need to be isolated so that their failures do not destabilize the whole system. Their communications with the rest of the system has to be regulated through a carefully designed interface so that only a few allowed operations can escape the sandbox.

SFI can be implemented either with the assistance of hardware or through a mostly software-based approach. For instance, NaCl x86-32 isolates mobile native code from untrusted sources through rarely used hardware segmentation in x86-32 [29, 40]. Hardware segmentation was deprecated in x86-64 and unavailable on other prevalent Instruction Set Architectures (ISAs) such as ARM and MIPS. Instead, NaCl x86-64 adopts a software-based approach by inserting security checks, modeled as machine instruction sequences, into subject programs to make sure that memory addresses do not fall outside the sandbox. In order to improve efficiency, a combination of static verification and dynamic checks are adopted. For direct memory accesses and direct branches, their memory addresses are explicitly encoded in the instructions and can be statically verified. No dynamic checks are needed. For computed memory accesses, their memory addresses are unknown statically, thus dynamic checks are positioned before them to ensure that their memory addresses stay in the sandbox.

One subtle requirement of IRMs and therefore SFI is that some form of control-flow integrity be enforced so that inserted checks can not be bypassed by adversaries. PittSFIeld [23] and NaCl [29, 40] enforce the *chunk-granularity control-flow integrity*. Code section is partitioned into atomic chunks of fixed length such as 16 bytes and 32 bytes. Control flow can only transfer to the beginnings of a chunk and exit a chunk through the last branch instruction or the last instruction. A computed branch is restricted by runtime checks to target only the starting addresses of chunks. Checks for memory accesses cannot be bypassed as long as they stay in the same chunk.

In other words, the chunk-granularity control-flow integrity enforces an extremely imprecise control flow graph, which is a complete graph of all the nodes each of which corresponds to a chunk. The imprecise control flow graph is insufficient for effective static analyses, as the scope of static analyses is limited to a single chunk. The power of static analyses grows with their scope.

For efficiency considerations, traditional SFI implementations only sandbox memory writes to preserve integrity but leave memory reads unsandboxed because memory reads are much more common than memory writes and sandboxing them require many more dynamic checks, thus hurting runtime performance. However, sandboxing memory reads is needed for protecting confidentiality. For some situations, confidentiality is even more important than integrity. For example, online bank passwords, credit card number, private user profiles and other sensitive information are critical and it would cause enormous economical loss and inconvenience if it is leaked. Confidentiality should be preserved. In addition, traditional implementations do not work for multi-threaded programs because they assume a single-threaded attack model. A single-threaded attack model can be problematic in multi-threaded world. We propose a novel implementation by combining control-flow integrity and data sandboxing to sandbox both memory reads and writes. In order to decrease the runtime overhead, we propose and evaluate a series of optimizations to remove unnecessary checks.

Finally, traditional SFI implementations are performed through binary rewriting, assembly instrumentation or transformations on equivalent low-level code. As a result, they are tightly tied to a specific target machine and difficult to port to other ISAs. One advantage of low-level rewriting is that it holds the promise of rewriting without source code being available. However, most previous SFI implementations still ask for the cooperation of the code producer by requiring assembly code or a customized compiler to be used. A recent SFI system [37] makes substantial progress towards an implementation through pure binary rewriting; it remains to be seen whether the system can be generalized to IRMs other than SFI and how optimizations can be performed. We propose a framework to do sandboxing at a machine-independent level. The majority of the framework can be reused for various targets supported by the industrial strength compiler LLVM.

2.2 Control-Flow Integrity

Control-flow integrity is a vulnerability mitigation that can foil attacks which require inducing abnormal control flow transfers. As a special IRM, CFI ensures that runtime control flow transfers follow a predetermined control flow graph even if the whole data memory is under the control of adversaries [2, 3]. The enforced

control flow graph can be constructed through source-level static analysis, IR-level program analysis, binary analysis or even program profiling.

CFI can foil many attacks including stack-based buffer overflows, heap overflows etc. One of the most pressing type of attacks is code injection attacks. Adversaries inject arbitrary code into vulnerable programs and then manipulate code pointers to jump there in order to carry out the functionality intended by the adversaries. CFI can prevent these attacks because they require an essential step of inducing control flow transfer to injected code and it violates the predetermined control flow graph enforced by CFI. Even sophisticated attacks like return-to-libc and Return-Oriented Programming (ROP [12, 30]), which can bypass many existing mitigations such as Data Execution Protection (DEP [24]), can be mitigated by CFI.

One subtle requirement for IRMs is that embedded checks not be bypassed by adversaries, thus needing some form of control flow restriction. Conventional IRMs have to restrict control flows through code pointer alignment by padding with *nop*s, which incur additional overhead. CFI guarantees a stronger control flow restriction than IRMs require even assuming that data memory is at attackers' full disposal. Thus it can serve as the foundation for SFI and other IRMs. In addition, CFI assumes a multi-threaded attack model. Adversaries can control the whole data region through a separate thread. Thus, CFI is well-suited to protect multi-threaded programs.

In general, CFI can be implemented in two different approaches. One approach is to insert a bit pattern, or ID at each branch target and an ID-check before each branch instruction to ensure that the runtime destination has the proper ID [3]. The ID is encoded in a side-effect-free instruction such as *prefetch*. Similar to SFI, only computed control flow transfers need to be instrumented for efficiency. Direct branches can be statically verified, as their branch targets are encoded in the instructions as immediate values. Another approach to enforce CFI is through defunctionalization [36]. For a computed control flow transfer, its potential targets are encoded in a write-protected table and an index into the table is used to retrieve the target address. Before the computed control flow transfer, the index is checked to make sure that it falls into the table before it is used to fetch the target address.

Although CFI can defend against many attacks, its implementations incur significant overhead. According to the seminal paper [3], CFI incurs approximately 20% slowdown on SPECint2000 on average. It can be problematic for performance-critical systems. In order to reduce the runtime overhead even further towards industrial adoption, we have proposed a series of optimizations such as jumping over prefetch instructions and jump table check optimization to reduce the number of dynamic checks. Furthermore, we have discovered more efficient instructions to encode CFI ID's than *prefetch*. These optimizations have brought down the runtime cost of CFI significantly, as demonstrated by our experimental results.

Combining CFI with Other IRMs. CFI can serve as the foundation for other IRMs because IRMs require a certain form of control-flow integrity. On top of CFI, XFI employs a protected shadow stack to store function call return addresses [18]. XFI and CFI enhance each other with XFI promoting control flow precision from Deterministic Finite Automata (DFA) to Pushdown Automata (PDA) and CFI providing XFI with the guarantee of control flow restrictions [3]. However, XFI is platform specific and and its heavy runtime overhead might be problematic for production systems.

3 Attack Model and Security Policy

Every security mechanism assumes an explicit or implicit attack model. Some security mechanisms adopt optimistic attack models while we adopt the pessimistic CFI attack model. It is both conceptually simple and realistic. It assumes separate code and data regions for an untrusted program. The code region is non-writable and the data region is non-executable. [1] An attacker is modeled as a separate thread that runs in parallel with the subject program. The concurrent attacker thread can overwrite any part of the data region, including the stack, the heap, and the global data region, between any two instructions. This rather pessimistic assumption captures real attack scenarios and is also amenable to formal analysis [3].

One implication of the attack model is that the code region and registers cannot be changed by the attacker directly. Note this assumption by itself does not prevent indirect changes induced by attackers to

[1] This assumption can be discharged either by the hardware Non-eXecute (NX) protection or by a pure software-based approach; the software approach sandboxes indirect branches so that control always stays in the code region for single-threaded programs.

the code region and registers. For instance, if the program loads into a register some contents from the data region, the register can afterwards hold any value supplied by the attacker as he/she controls the data region. As another example, an unconstrained memory write in the program could possibly change the code region—one goal of data sandboxing is to prevent this from happening by sandboxing memory writes.

3.1 Control-flow security policy

The CFI policy for a program enforces that execution paths follow a predetermined graph whose nodes consist of addresses of basic blocks, and whose edges connect control transfer instructions (i.e., jumps and branches) to allowed destination basic blocks. Within a function or method, this corresponds directly to a basic control flow graph. For dynamic control flow (i.e., a jump through a register, a return, or other computed jumps), the outgoing edges correspond to the possible destinations where control is allowed to transfer to.

Definition 1 *Program P respects the CFI policy C if and only if when executed, all control flow transfers in P respect the graph G.*

CFI policies are in essence NDFA's or regular expressions denoting sets of possible control flow transfer traces. In contrast, XFI provides a richer language of policies (corresponding to push-down automata), which can ensure that functions only return to the code that called them. CFI can only ensure that functions return to *some* possible caller. On the other hand, XFI requires a high-integrity stack to store return addresses (which is broken by our strong attack model) and introduces issues with setjmp, exceptions, continuations, and other unconventional control transfers. Thus, CFI has the attractive property that it breaks fewer applications than XFI, supports a strong, concurrent attack model, and provides tighter bounds on control flow than SFI.

3.2 Data sandboxing policy

The data sandboxing policy specifies that any memory access in the untrusted program must be confined to the allowed data region, i.e. the sandbox. Consequently, integrity and confidentiality of memory outside of the data region are ensured. Integrity is usually more important for security than confidentiality. But in highly sensitive applications such as bank account information, protecting confidentiality can be just as vital, if not more important.

We next formalize the data sandboxing policy. We assume there is a large, contiguous memory region that untrusted code can read from and write to. We assume the data region begins with the address `DB` (Data Begin) and ends with the address `DE` (Data End). That is, the address range of the data region is `[DB, DE]`, inclusive. As an optimization for performance, we set up *guard zones* of size `GSize` right below and above the data region, similar to previous work. The size of the guard zone can vary depending on the host policy and the usage of virtual address space. It is further assumed that memory accesses to locations in guard zones can be efficiently trapped. [2] As we will discuss later, guard zones enable many optimizations.

Definition 2 *A memory access is allowed if the address is within the range `[DB-GSize, DE+GSize]`.*[3]

4 Enforcing Confidentiality by Combining CFI and Data Sandboxing

Conventional implementations of SFI either rely upon special segment registers to enforce data sandboxing, or else use pure software-based techniques but only sandbox memory writes due to the high runtime overhead of sandboxing reads. In a typical program, memory reads are much more common than memory writes and

[2] This can be conveniently discharged through page protection using `mprotect` function.

[3] The size of the memory access is irrelevant because of the presence of the guard zones and the assumption that accesses to guard zones is trapped. Furthermore, the policy in our implementation also allows reading the code region because the inserted CFI checks read IDs from the code region. This is an unimportant exception and will not be discussed further in the rest of the thesis.

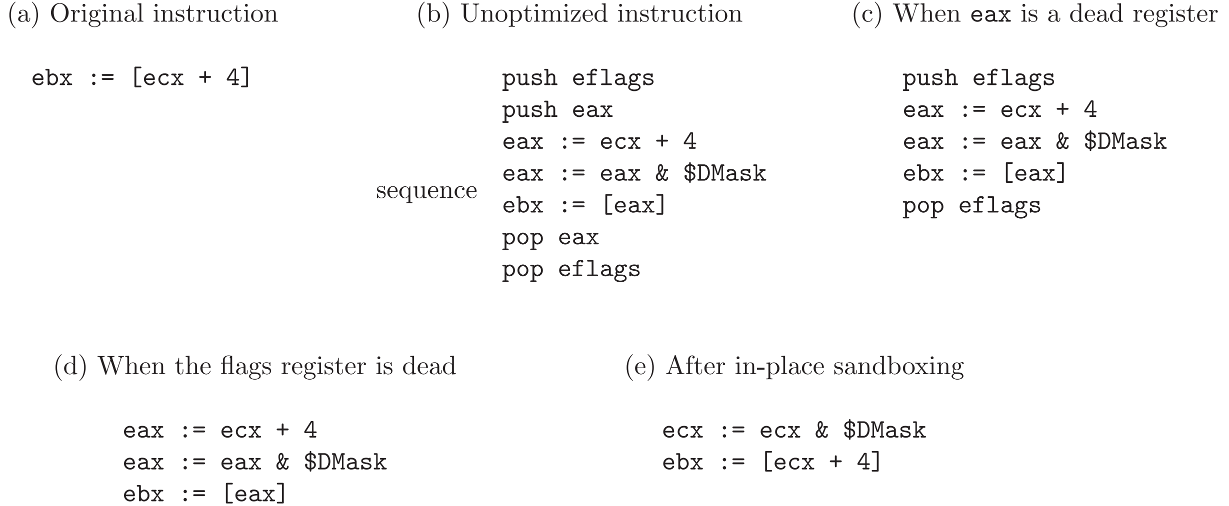

Figure 1: A running example for illustrating optimizations. `DMask` is the data-region mask.

require many more security checks. As a result, confidentiality is sacrificed for runtime efficiency. As discussed above, in many situations, confidenciality is more important than integrity. In contrast, we focus on software-only techniques, which are applicable to virtually all available architectures, and enforce both confidentiality and integrity by sandboxing both memory writes and reads efficiently with aggressive optimizations enabled by CFI. In order to preserve the trustworthiness, we propose a novel verifier to validate the final result of our optimizations and transformations.

4.1 Data Sandboxing Optimizations

We next present the series of optimizations that we propose to cut the runtime cost of data sandboxing. The data-sandboxing optimizations utilize static analyses to identify optimization opportunities without sacrificing security. The optimizations are similar to those performed in an optimizing compiler, except that they need to not only preserve program semantics but also maintain the security properties.

Note this section assumes CFI is already in place; Sec. 4.2 will discuss CFI and its own optimizations. One immediate benefit of CFI is that it obviates the need for chunk-granularity control-flow integrity [23, 29]. CFI provides better security as its control-flow graph is more precise. Moreover, it also avoids the large number of `nops` that have to be inserted for instruction alignment. The extra `nops` add both spatial and temporal overheads. PittSFIeld [23] reports that inserted `nops` account for about half of the runtime overhead of enforcing data integrity. NaCl [29] reports that instruction alignment accounts for most of its performance overhead. In addition, CFI enables many other optimizations. Before we discuss these optimizations, we present a running example that will be used to illustrate how the optimizations work.

A running example Fig. 1 presents the example. For clarity, this and other programs in the thesis use a pseudo-assembly syntax whose notation is described as follows. We use "`:=`" for an assignment. When an operand represents a memory address, it is put into square brackets. For instance, `[esp]` denotes a memory-address operand with the address in `esp`, while `esp` (without square brackets) represents a register operand. We will also use the syntax "`if ... goto ...`" to represent a comparison followed by a conditional jump instruction (i.e., `jcc`).

Fig. 1(a) shows an instruction that transfers the contents at memory location `ecx+4` to `ebx`. For protecting confidentiality, the address has to be sandboxed before the memory read. Fig. 1(b) presents an unoptimized sequence of instructions that performs the sandboxing. In the sequence, `eax` is used as a scratch register for holding the intermediate value. Since the old value of `eax` might be needed afterwards, the sequence pushes `eax` onto the stack and later restores its value. The flags register, which stores status flags such as the overflow flag, also needs to be saved and restored since the bitwise-and instruction changes the status flags and the old flags might be needed for the subsequent computation. The constant `$DMask` denotes the

data-region mask. Following PittSFIeld [23], a single bitwise-and sandboxing instruction is used to ensure that the resulting address is in the data region. For instance, if the data region starts from 0x20000000 and is of 16MB size, then `$DMask` is `0x20ffffff`. The sandboxing instruction will also be called a check in the rest of the thesis.

4.1.1 Liveness analysis

Our system performs liveness analysis on registers and the flags register to remove operations that save and restore old values when they are unnecessary.

Register liveness analysis Oftentimes inlined reference monitors require the use of scratch registers for storing intermediate results. For instance, `eax` is used as a scratch register in Fig. 1(b). One simple approach to avoiding the overhead of saving and restoring scratch registers is to reserve a dedicated scratch register. As an example, PittSFIeld [23] reserves `ebx` as the scratch register. This approach has the downside of increasing the register pressure, especially on machines with few general-purpose registers.

Our alternative approach relies on register liveness analysis. Liveness analysis is a classic compiler analysis technique. At each program point, the liveness analysis calculates the set of live registers; that is, those registers whose values are used in the future. A register is dead if it is not live. At a program point, a dead register can be used as the scratch register without saving its old value (since the old value will no longer be needed). When no dead register is available, we can resort to the old way of saving the scratch register on the stack.

For the running example in Fig. 1, if register liveness analysis determines that `eax` is dead after the instruction `ebx := [ecx + 4]`, then there is no need to save and restore its old value; the sequence in Fig. 1(c) is then sufficient.

We implemented an intra-procedural register liveness analysis with extra assumptions about the calling convention. It is a backward dataflow analysis and uses a standard worklist algorithm. The analysis takes advantage of the `cdecl` calling convention to deal with function calls and returns. In particular, the live-out of a return instruction is those callee-saved registers (including `ebx`, `esi`, and `edi`) together with registers that contain the return value. The live-in of a call instruction is the live-out subtracted by the set of caller-saved registers (including `eax`, `ecx`, and `edx`).

We note the correctness of liveness analysis (including the assumption about the calling convention) does not affect security. When liveness analysis produced wrong results, some registers would not be properly saved and restored; this would affect the correctness of the program, but not the security of its memory operations.

Flags-register liveness analysis We also perform the flags-register liveness analysis to remove unnecessary operations for saving and restoring the flags register. Saving and restoring the status flags are costly on modern processors. For the running example, if the analysis determines that the flags register is dead after the instruction `ebx := [ecx + 4]`, then the sequence in Fig. 1(d) can be used.

For simplicity, we perform the flags-register liveness analysis only within basic blocks. We assume flags are not used across basic blocks (meaning the register is assumed dead at the end of a basic block). This seems to be an assumption used by compilers and has been confirmed by our experiments on SPECint2000.[4]

PittSFIeld [23] avoids saving and restoring the flags register by disabling instruction scheduling in compilers. It prevents compilers from moving comparisons away from their corresponding branching instructions. In contrast, our approach allows instruction scheduling within basic blocks.

4.1.2 In-place sandboxing

Thanks to the guard zones before and after the data region, there are special cases where we can avoid using a scratch register. A commonly used address pattern in memory operations is a base register plus a small displacement (which is a static constant value). For instance, this pattern is used to access fields of a data structure; the base register holds the base address of the data structure and the displacement is the offset of

[4] This might be broken by hand-written assembly code. But similar to register liveness analysis, the correctness of this analysis does not affect security.

```
        ecx ∈ [−∞,+∞]
ecx := ecx & $DMask
        ecx ∈ [DB, DL]
eax := [ecx + 4]
        ecx ∈ [DB, DL]
... // assume ecx not changed in between
        ecx ∈ [DB, DL]
ecx := ecx & $DMask
        ecx ∈ [DB, DL]
ebx := [ecx + 8]
        ecx ∈ [DB, DL]
```

Figure 2: An example demonstrating redundant check elimination.

(a) Unoptimized sequence

```
  esi := eax
  ecx := eax + ebx * 4
  edx := 0
loop:
  if  esi≥u ecx goto end
  esi := esi & $DMask
  edx := edx + [esi]
  esi := esi + 4
  jmp loop
end:
```

(b) Hoisting checks outside of the loop

```
  esi := eax
  ecx := eax + ebx * 4
  edx := 0
  esi := esi & $DMask
loop:
  if  esi≥u ecx goto end
  edx := edx + [esi]
  esi := esi + 4
  jmp loop
end:
```

Figure 3: An example demonstrating loop check hoisting.

a field. When a memory address of this pattern is used, we can perform the optimization that sandboxes just the base register. The running example in Fig. 1 uses address `ecx+4`. Therefore, the sequence in Fig. 1(e) sandboxes the base register `ecx` directly.

The safety of this transformation is straightforward to see. After "`ecx := ecx & $DMask`", register `ecx` is constrained within the data region. Consequently, `ecx+4` must be within the data region plus the guard zones (assuming $\texttt{GSize} \geq 4$). The memory operation is then allowed according to the data security policy.

We call this optimization *in-place sandboxing* since it sandboxes the base register directly and avoids the use of an extra scratch register. Additionally, it has the benefit of making it convenient to remove redundant checks, as shown next.

4.1.3 Optimizations based on range analysis

According to the data-sandboxing policy, a memory access is allowed if the address range is within the valid range of the data region plus guard zones. This definition naturally leads to a strategy of removing unnecessary checks: if the address range of a memory access can be statically determined to be within the valid range, then it is unnecessary to have a check before the memory access.

To realize this idea, we have implemented *range analysis* on low-level code. At each program point, the range analysis determines the ranges of values in registers. The range $[-\infty, +\infty]$ is the universe and gives no information. For instance, after an operation that loads contents from the data region into a register, the register's range becomes $[-\infty, +\infty]$; this reflects that the attack model allows arbitrary changes to the data region. In many other situations, a more accurate range can be obtained. For instance, after "`ecx := ecx & $DMask`", the range of `ecx` is `[DB, DL]` (i.e., the data region).

We have implemented two optimizations that take advantage of range analysis. We discuss them next.

Redundant check elimination This optimization takes an input program with checks embedded in and aims to eliminate redundant checks. It is performed in two steps. In the first step, range analysis is performed on the input program. In the second step, it uses the results of range analysis and heuristics to decide whether a check can be eliminated. For instance, if the range of `r` is within the data region before "`r := r & $DMask`", then the check is equivalent to a no-op and thus unnecessary. As another example, suppose (1) the instruction sequence is "`r := r & $DMask`" immediately followed by a memory dereference through `r`; (2) the range of `r` before the sequence is within the data region plus the guard zones, then the check can also be removed because the memory dereference is safe without the check. The general criterion for a removal is if it can be statically determined that the removal will not result in unsafe access in the following memory dereference.

We next examine a simple example in Fig. 2. Imagine `ecx` is the base address of a C struct. The program then loads two fields from the struct. Each memory read is preceded by a check. The figure also shows the ranges of `ecx` at each program point. With range analysis, the optimizer can tell that the second check can be removed because the range of `ecx` is already in the data region before the check.

Loop check hoisting This optimization hoists checks so that one single check outside the loop is sufficient to guarantee safety of all memory access in the loop.

Fig. 3 presents an example program showing how static analyses enable hoisting checks outside of loops. The assembly program in the figure calculates the sum of an integer array (with base address `a` and length `len`) and roughly corresponds to the following C program:

```
sum = 0;
int *p = a;
while (p < a + len) {   c
  sum = sum + *p;
  p = p + 1;
}
```

In Fig. 3, `eax` holds the initial address of the array, `ebx` holds the length, and `esi` holds the pointer value `p`. Without optimization, `esi` needs to be sandboxed within the loop body. The sandboxing instruction is underlined in Fig. 3. That sandboxing instruction can actually be moved outside of the loop, avoiding the per-iteration sandboxing (how the optimization is achieved will be discussed shortly). The optimized code is shown in Fig. 3(b).

It is instructive to understand why the code in Fig. 3(b) is safe even though it sandboxes only the beginning address of the array and there is no restriction on the array length. To show its safety, it is sufficient to show that `esi` $\in$ `[DB, DL+4]` is a loop invariant. The condition is clearly true at the beginning of the loop since the sandboxing instruction gives `esi` $\in$ `[DB, DL]`. Next, assuming the condition holds at the beginning of the loop body, we try to re-establish it at the end of the loop body. The key step in the reasoning is that `esi` $\in$ `[DB, DL]` holds after `edx := edx + [esi]`—a hardware trap would be generated if `esi` were in guard zones. With that result, the following add-by-four operation clearly re-establishes the loop invariant. What has been exploited is the following observation: since access to guard zones can be efficiently trapped, a successful (untrapped) memory access actually serves as a "check" and narrows the range down to the data region.

The loop optimization is implemented in multiple steps, outlined as follows:

(1) *Dominator trees* are used to identify loops in assembly code. Backward edges in a dominator tree is then used to locate loops. Calculation of the dominator tree and loop identification are standard techniques in an optimizing compiler [5].

(2) Given an input program, any check that appears in the loop body is duplicated at the beginning of a loop. For the program in Fig. 3(a), this step results in the program depicted in Fig. 4. Notice the instruction "`esi := esi & $DMask`" is duplicated before the loop. One worry of eagerly sandboxing before the loop is it might change the program behavior. However, if we assume good code will always

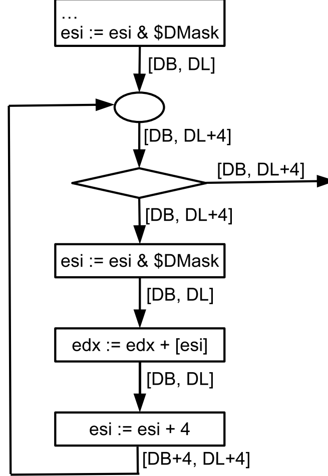

Figure 4: Range analysis result for the program in Figure 3(a), after the check is duplicated at the loop pre-head. Only the ranges of `esi` are included.

have pointers that point into the data region, then eager sandboxing should be an idempotent operation for good code and it breaks only programs that would violate the policy.[5]

(3) A range analysis is then performed to decide if any check is unnecessary. Fig. 4 also presents the ranges of `esi` at each program point. Notice the range of `esi` is within the data region plus guard zones (assuming `GSize` ≥ 4). This enables the optimizer to remove the check in the loop body. After its removal, we get the optimized program in Fig. 3(b). There is also a possibility that the optimizer decides that the check in the loop cannot be removed; in this case, the corresponding one that was added before the loop is removed.

The above strategy of loop optimizations has the benefit of performing only one round of range analysis even if the program has multiple loops or nested loops. However, it does not capture every loop-hoisting opportunity. Another strategy is to hoist checks outside the loop one by one, and use the verifier (discussed in the next section) to check the safety of intermediate results; but it involves backtracking and performing a range analysis after each intermediate step.

4.1.4 Validating data sandboxing

Of the three SFI optimizations we have discussed, liveness analysis is not security critical: security would not be affected even if it produced wrong results. On the other hand, the other two optimizations change checks, remove checks, or move checks to a different place. Security would be affected if they were wrong. Since low-level optimizations are extremely error prone, it is always a good idea to have a separate verifier to verify the results of optimizations, instead of trusting the optimizer.

Previous SFI verifiers assume checks appear immediately before memory operations. With that assumption, a simple linear scan of the code is sufficient to check the code's safety. Since our optimizer eliminates checks and moves checks away from memory operations, a more complex verifier is needed to check the result of optimizations. This does increase the size of the TCB, but seems unavoidable when verifying the optimization results.

We have implemented a verifier based on range analysis which can check the results of our optimizations. The basic idea is to perform range analysis over the optimized program and determine the range of addresses used in memory operations; the program is verified if every such address is statically determined to be within `[DB-GSize, DL+GSize]`.

[5]Certain programming practices use pointers that are outside the object bounds, but those pointers should stay inside the data region. For instance, pointers that are one element past the end of an array in the heap stay in the data region if the stack is in the upper portion of the data region.

Original code	Code after instrumentation	Comment
`call fun`	`call fun`	
	`prefetchnta [$ID]`	a side-effect free instruction with an ID embedded
`ret`	`ecx := [esp]`	retrieve the return address
	`esp := esp + 4`	adjust the stack pointer
	`if  [ecx+3]≠$ID goto error`	check ID; ecx+3 is the address of the ID since the opcode of the prefetch instruction takes three bytes
	`jmp ecx`	transfer the control flow

Figure 5: A CFI example.

The following example is the code in Fig. 2 after the optimizer has removed the second check. Range analysis determines that the address range in the first memory access is `[DB+4, DL+4]` and the address range in the second memory access is `[DB+8, DL+8]`. Assuming `GSize` ≥ 8, both are safe according to the policy.

```
        ecx ∈ [−∞, +∞]
ecx := ecx & $DMask
        ecx ∈ [DB, DL]
eax := [ecx + 4]
        ecx ∈ [DB, DL]
... // assume ecx not changed in between
        ecx ∈ [DB, DL]
ebx := [ecx + 8]
        ecx ∈ [DB, DL]
```

In a similar fashion, range analysis can determine the safety of the program in Fig. 3(b), which is the result after loop optimizations.

Our verifier is robust in the sense it can verify many more optimizations, including those we have not implemented. New optimizations, including more aggressive loop optimizations, can be verified by the same verifier. Another use of the verifier is for *speculative optimizations*. The optimizer can eliminate a check or move a check to a different place even when it is not clear whether that transformation would result in a safe program. After the transformation, the verifier can be used to check the safety of the resulting program; if the verifier fails, the optimizer can resort to the old program. We have not tried any speculative optimizations.

4.2 More efficient control-flow sandboxing

We describe two simple CFI optimizations that result in a more efficient implementation. This reduces the overhead of protection schemes that build on CFI, including our data-sandboxing scheme.

4.2.1 Original CFI instrumentation

As background information, we discuss CFI's original mechanism for ensuring that a program's execution follows a pre-determined control-flow graph [1, 3]. CFI uses a combination of static verification and dynamic instrumentation for enforcement. For a direct jump instruction, a static verifier can easily check that the target is allowed by the control-flow graph, without incurring any runtime overhead. For a computed jump, CFI inserts runtime checks into the program being protected to ensure that the control flow transfer is consistent with the control-flow graph.

Fig. 5 presents an example illustrating how CFI checks are performed. It shows how a direct function call is instrumented in CFI. Instruction "`call fun`" invokes a known function with name `fun`. The direct call itself does not need instrumentation—whether this is legitimate is checked statically. The return instruction

Original code	Code after instrumentation
`call fun`	`call fun` `prefetchnta [$ID]`

`ret`	`ecx := [esp]`
	`esp := esp + 4`
	`if [ecx+3]≠$ID goto error`
	<u>`ecx := ecx + 7`</u>
	`jmp ecx`

Figure 6: New CFI instrumentation by skipping over prefetch instructions.

in the body of `fun`, however, needs instrumentation since it takes from the data region a return address, which may be corrupted by the attacker.

The instrumentation is performed in two steps. First, an ID is inserted after the call instruction (note the same ID is inserted after all possible call sites to `fun`). Second, the return instruction is changed to a sequence of instructions that checks the correct ID is at the target before the control flow transfer. The ID is embedded in a side-effect free `prefetch` instruction. The instruction takes a memory location as its operand, and moves data from memory closer to a location in the cache hierarchy. It is a hint to the processor and does not affect program semantics. Instrumentation of other computed jumps is similar: IDs are inserted at the allowed targets and runtime checks ensure correct IDs are there before control flow transfers. Note that IDs are inserted into the code region and cannot be changed by the attacker. Furthermore, since the data region is not executable, the attacker cannot manufacture new code in the data region with the correct ID in it and jump to it for execution.

4.2.2 CFI optimizations

We next describe two simple CFI optimizations.

Jumping over prefetch instructions The original CFI implementation inserts prefetch instructions at targets of computed jumps. However, prefetch instructions incur significant overhead by fetching data from memory and increasing cache pressure. Therefore, the first optimization jumps over prefetch instructions to avoid their execution. Fig. 6 presents the new code sequence after the optimization. The only difference is a new instruction (underlined) that adds seven to the register that holds the target address.[6] Since the size of a prefetch instruction is seven, it is skipped over. This optimization trades a `prefetch` instruction for a cheaper `add` instruction. Our evaluation shows this alternative significantly cuts the runtime overhead of CFI (from 24.90% to 7.74%). Designers of the original CFI mentioned this alternative, but it seems it has not been evaluated for performance comparison.

Jump table check optimization Computed jumps are often used for efficient compilation of switch statements. Most compilers generate a jump table for a switch statement. The starting address of each branch of the switch statement is stored as an entry in the jump table. Fig. 7 presents an example. The first column presents the typical sequence of instructions used by compilers to transfer the control flow to a branch of a switch statement. It assumes `edx` stores an index into the jump table and `JT` is a constant denoting the start address of the jump table. `edx` is scaled by a factor of four when loading an entry from the jump table because each entry is assumed to be a four-byte address.

The middle column lists the code sequence after the CFI instrumentation. Because "`jmp ecx`" is a computed jump, the instruction checks that there is a correct ID at the target.

Jump tables are usually stored in read-only sections of object code. If we assume jump tables cannot be

[6] The sequence takes care of only control-flow sandboxing. In our implementation for data confidentiality, `esp` is sandboxed to stay in the data region and `ecx` is sandboxed to stay in the code region.

Original code	CFI instrumentation	After optimization

`ecx := [$JT + edx*4]`
`jmp ecx`
	`ecx := [$JT + edx*4]`
	`if [ecx+3]≠$ID goto error`
	`ecx := ecx + 7`
	`jmp ecx`
		`if edx>u15 goto error`
		`ecx := [$JT + edx*4]`
		`ecx := ecx + 7`
		`jmp ecx`

Figure 7: Jump table check optimization. Assume `JT` is a constant denoting the start address of the jump table, `edx` holds the index into the jump table, and the jump table is of size 16.

modified by attackers[7], then control-flow integrity is satisfied if (1) the index into the jump table is within bounds and (2) all jump targets in the jump table are legal according to the control-flow graph. The second condition can be checked statically and the first condition needs a bounds check. The last column in Fig. 7 presents the new sequence with the bounds check, assuming the size of the jump table is 16. We use $>_u$ for the unsigned comparison so that large numbers would not be treated as negative. The bounds check involves only register values and avoids retrieving IDs from memory. Furthermore, it turns out that the LLVM compiler, in which our prototype implementation is developed, already inserts bounds checks before using a jump table. This further simplifies the work of our CFI rewriter. Note our system does not depend on the assumption that LLVM emits bounds checks since the CFI verifier would complain if it did not.

This jump-table check optimization is essentially a special case of the idea of encoding target tables for computed jumps, as implemented in HyperSafe [36]. The effectiveness of this optimization depends on how often switch statements are used in programs.

4.3 Implementation and evaluation

We next discuss our prototype implementation and evaluation of the implementation on benchmark programs.

4.3.1 Prototype implementation

Our implementation is built in LLVM 2.8 [22], a widely used compiler infrastructure. We inserted a pass for CFI rewriting, a pass for data sandboxing rewriting and optimization, and a pass for CFI and data-sandboxing verification. All these passes are inserted right before the code-emission pass. There are approximately 14,000 lines of C++ code in total added to LLVM (including comments and code for dumping debugging information). Our rewriters essentially perform assembly-level rewriting. We chose LLVM as our implementation platform because LLVM preserves helpful meta-information at assembly level (such as the control flow graph). It also provides a clean representation of compiled programs, which benefits instrumentation and optimization. In addition, it is easy to extend LLVM since inserting an extra pass into its compilation process requires nothing more than a registration of the pass.

Control flow graph Control-flow graphs are constructed with the help of LLVM. In particular, the LLVM compiler preserves meta-information so that a precise intra-procedural control-flow graph can be reconstructed at the assembly level. The inter-procedural control-flow graph (or the call graph) is conservatively estimated by allowing a computed call instruction to target any function. The precision of the call graph could certainly be improved through further static analyses such as inter-procedural control flow analysis and it would benefit security. On the other hand, since all our optimizations are intra-procedural, the precision of the call graph is not critical to these optimizations. In fact, we suspect inter-procedural static analyses would not result in significant performance improvement; the attack model assumes the data region can arbitrarily change between instructions and thus inter-procedural analysis on the data region such as shape analysis would be inapplicable.

Verifier We have implemented a CFI and a data sandboxing verifier. The CFI verifier is similar to previous CFI verifiers and checks whether IDs and checks are inserted at appropriate places. We do not elaborate

[7] This assumption can be discharged by either putting jump tables into code region as in HyperSafe [36] or have read-only data write-protected through page protection.

	gzip	vpr	gcc	mcf	crafty	gap	vortex	bzip2	twolf	average
CFI (%)	3.47	1.05	2.52	0.09	11.47	13.87	26.78	6.36	4.04	7.74
CFI.jt.no-skip (%)	14.72	3.23	4.14	0.14	25.28	82.03	67.66	22.18	4.74	24.90
CFI.no-jt.skip (%)	3.84	1.01	2.46	0.04	15.28	13.61	28.39	6.32	3.18	8.24
CFI.no-jt.no-skip (%)	14.45	3.77	5.73	0.09	34.11	81.73	72.96	22.29	5.68	26.76

Table 1: CFI runtime overheads for SPECint2000.

on its details. The implementation of the data-sandboxing verifier contains approximately 7,000 lines of C++ code. The majority of the code is a large switch statement that calculates the ranges of registers for machine instructions. There are over 3261 distinct machine opcodes inside LLVM including opcodes and pseudo opcodes for IA-32, IA-64, x87 FPU, SSE, SSE2, SSE3, and others. The verifier could be shortened by grouping instructions into cases and omitting instructions that IA-32 does not support. Given its large size, its own trustworthiness should be independently validated (e.g., by testing or by developing its correctness proof in a theorem prover); we leave this to future work.

At the beginning of a function and any basic block that a computed jump might target, the ranges of general-purpose registers (`eax`, `ebx`, `ecx`, `edx`, `esi` and `edi`) are assumed to be the universe ($[-\infty, +\infty]$) and the ranges of `esp` and `ebp` to be the data region.[8] For each instruction, the verifier updates the ranges of the registers that are defined by the instruction or used to compute a memory location. For example, after "`movl (%ebx), %eax`", the range of `ebx` is narrowed down to the data region if the old range of `ebx` is within the data region plus guard zones; furthermore, the range of `eax` is set to be the universe because it is loaded from the untrusted data region.

Since the lattice in range analysis is of infinite height, its termination is not guaranteed unless some widening strategy is adopted. Our implementation uses a simple one: if a node has been processed more than a constant number of times, the ranges of registers that have not been stabilized are set to be the universe. Other than this aspect, our implementation of range analysis follows a standard worklist algorithm.

During the development, the verifier helped us catch several implementation errors in early versions of the optimizer; these errors would be hard to find by hand.

4.3.2 Performance evaluation

To evaluate our implementation, we conducted experiments to test its runtime overhead on SPECint2000. Experiments were conducted on a Linux CentOS 5.3 box with Intel Xeon X5550 CPU at 2.66 GHz and 12GB of RAM. All experiments were averaged over six runs. Three benchmark programs in SPECint2000 could not be compiled by LLVM: `eon` is written in C++ and LLVM's front end (clang) does not support the version of the standard C++ library in CentOS 5.3; `perlbmk` and `parser` could not be compiled with the optimization level 3 and seem incompatible with LLVM. All other benchmark programs were compiled with the optimization level 3.

Table 1 presents the runtime percentage increases of CFI compared to uninstrumented programs for SPECint2000. The CFI row reports the results of our CFI implementation. On average, it adds 7.74% runtime overhead. The *CFI.jt.no-skip* row shows the results of disabling the optimization of jumping over prefetch instructions. Disabling this optimization results in significant performance degradation: the overhead shoots up to almost 25%. This is due to the execution of costly prefetch instructions. The *CFI.no-jt.skip* row reports the results when the optimization for jump-table checks is disabled. The performance improvement by this optimization is modest, suggesting opportunities for this optimization in SPECint2000 are limited. *CFI.no-jt.no-skip* is the same as the original CFI implementation by Abadi *et al* [3]. They reported an average overhead of 16% on SPECint2000 on an older system (Pentium 4 x86 processor at 1.8 GHz with 512 MB of memory and Windows XP SP2). The experiments show that our CFI implementation is efficient even though we have not yet implemented sophisticated CFI optimizations.

Table 2 presents the runtime percentage increases for data sandboxing. All numbers in the table include the CFI overhead because our data-sandboxing optimizations build on top of CFI. The row of *DS-w.CFI* contains the numbers when sandboxing only the writes. The average overhead is 10.40%, which means it

[8] Before a function call and a computed jump, the verifier checks that `esp` and `ebp` are indeed in the data region.

	gzip	vpr	gcc	mcf	crafty	gap	vortex	bzip2	twolf	average
DS-W.CFI (%)	6.21	5.82	2.29	1.80	12.80	13.16	29.77	14.18	7.55	10.40
DS-RW.CFI.no-opt (%)	384.34	429.38	129.36	452.51	523.66	1092.98	690.73	748.27	476.25	547.50
DS-RW.live (%)	24.38	28.06	6.30	8.15	39.67	58.02	43.04	23.58	52.91	31.57
DS-RW.live.in-place (%)	24.29	26.69	5.81	5.13	39.12	49.23	39.08	23.65	48.69	29.08
DS-RW.CFI (%)	23.66	25.12	4.96	4.53	37.70	44.49	34.55	23.41	45.94	27.15

Table 2: Runtime overheads of data sandboxing plus CFI for SPECint2000.

adds roughly 2.7% on top of CFI. The overhead is low considering it sandboxes memory writes and enforces CFI.

Table 2 also presents the overheads when sandboxing both reads and writes. To understand the overhead reduction of the three data-sandboxing optimizations, it presents the overheads incrementally with respect to the optimizations. The row of *DS-RW.CFI.no-opt* contains the numbers when all optimizations are disabled. In this case, a check is inserted before every memory access; scratch registers and the flags register are saved on and restored from the stack. Overheads are high because the saving and restoring registers and the flags register are costly. The row of "DS-RW.CFI.live" contains the numbers after performing liveness analysis to remove unnecessary saving and restoring operations. After this optimization, the overheads are significantly lower. The row of "DS-RW.CFI.live.in-place" contains the numbers with both liveness analysis and the technique of in-place sandboxing; this drives down about 2% of the overhead. Finally, the last row contains numbers when optimizations based on range analysis are turned on; they cut down another 2% of the overhead. When all optimizations are turned on, data sandboxing adds about 19% on top of CFI. The overhead for protecting both reads and writes is modest and it is acceptable for applications where confidentiality is of great concern.

5 Enforcing IRMs at a Machine-independent Level

Next we introduce how to achieve retargetability through modeling security checks on a machine-independent level. Low-level IRMs such as SFI and CFI are usually implemented through low-level rewriting, either by performing binary instrumentation, assembly-code instrumentation, or by modifying a compilation toolchain's backend to emit code with embedded checks. One key benefit of rewriting at the low level is that a separate verifier can be built to check the result of rewriting. The separate verifier removes the rewriter outside of the TCB. Furthermore, in a server-client environment, only the verifier needs to be installed at the client's side. The verifier checks the security of untrusted, remotely downloaded modules. The security architecture of NaCl follows the separation between the rewriter and the verifier.

On the other hand, low-level rewriting is tightly coupled with a certain ISA, resulting in poor reusability and retargetability, and hindering optimizations. It is non-trivial to port a low-level IRM to another ISA and existing parts are hard to reuse. For instance, NaCl's initial implementation was on x86-32 and its port to x86-64 and ARM involved significant effort in design and implementation [29]. One reason for the nontrivial effort is the differences among ISAs, including the instruction set, the available hardware features, the number and size of registers, and others. In addition, many components need to be built from scratch. A typical example is optimizations. Any decent IRM implementation requires optimizations to bring down the runtime cost. However, those optimizations are tied to an ISA and hard to reuse.

We explore the building of a retargetable framework for low-level IRMs on a high-level compiler intermediate representation; in particular, the LLVM IR [21]. The framework, called Strato[9], is general in the sense that various security policies can be conveniently enforced and much code can be reused among them and that inlined high-level checks for a specific policy can be lowered into distinct machine-code sequences. In Strato, we have enforced CFI and data sandboxing for both memory writes and reads.

IR-level rewriting comes with several benefits. First, it is retargetable. Security checks are inserted into the high-level representation. The check-insertion component is shared by all target ISAs the compiler supports. Optimizations that operate on the IR are also reused among different targets. To support a new

[9]The name Strato comes from stratosphere, which is an intermediate layer of Earth's atmosphere that contains ozone absorbing ultraviolet light from the Sun.

target ISA, only the lowering from high-level checks to machine-instruction sequences needs to be changed. Even for the same ISA, it is easy to explore different machine-instruction sequences that implement the same high-level check since the lowering part can be easily changed. Our framework was originally built to support x86-32 and then extended to support x86-64; only a small amount of code was altered to retarget it for x86-64.

The second benefit of IR-level rewriting is that optimizations are easier to implement and more optimizations can be supported. An IR usually carries a wealth of structured information and attains many properties that are amenable to program analyses and optimizations. For instance, LLVM IR is in the Static Single Assignment (SSA) form [14, 15], making analysis easier to implement. In addition, LLVM IR preserves type information, loop information, and dominator-tree information, which facilitate analyses and optimizations. Finally, a high-level representation contains many fewer instructions than a typical target machine (e.g., in LLVM 2.9, LLVM IR has only 54 instructions while the x86-64 target has over 3,500). All these benefits make it easier to implement optimizers that remove or hoist security checks; we call these optimizers *security-check optimizers*.

However, the downside of pure IR-level rewriting is that it results in a larger TCB compared to low-level rewriting. The compiler backend performs sophisticated transformations to generate low-level code, including instruction selection, ISA-specific optimizations, and register allocation. Those transformations can invalidate the hypotheses assumed by a security mechanism at the high level. First, a bug in a transformation can produce insecure low-level code. A more subtle issue is that those transformations may assume a machine model different from the attack model of a low-level IRM. As a simple example, a backend transformation might assume that a variable holds the last stored value after it is loaded back from the memory location into which the variable is spilled. However, many low-level IRMs such as CFI assume the memory may change arbitrarily between any two instructions because of memory-corruption attacks. Under this attack model, a spilled variable cannot be assumed to hold the same value. Consequently, the transformation may produce insecure code according to the attack model.

Therefore, the challenge is how to perform IR-level rewriting while still preserve low-level security. Strato adopts a twofold approach. First, it includes a novel constraint encoding and checking process to propagate assumptions required by security-check optimizers. The optimizers do not remove checks; instead, they mark them as removable and attach constraints to them. After the backend transformations, Strato checks whether the constraints have been invalidated by the backend. If they are not, the unnecessary security checks are removed or hoisted. Otherwise, the checks are left intact to preserve the security of the low-level code. In other words, the security-check optimizers mark optimizations at the IR level, but the effect is taken only at the low-level code after ensuring constraints are not violated. To further ensure the trustworthiness, we implement an independent verifier to validate the final low-level code, thus removing all the instrumentations, transformations, optimizations, and constraint checking out of the TCB. The verifier helped us uncover 35 critical bugs in early versions of Strato.

5.1 Overview of Strato

This section elaborates on the workflow of Strato. We will discuss where checks are inserted and optimized, and where constraint checking and verification happen. We have used Strato to implement CFI and data sandboxing, two specific IRMs. Therefore, we first discuss those IRMs' attack model and security policies.

The data-sandboxing policy restricts memory reads and writes. Following previous work [23, 35, 41], we place a *guard zone* immediately before the data region and another guard zone immediately after the data region. We use `gz_size` to denote the size of both guard zones. We assume that access to guard zones is hardware trapped (through page protection). Guard zones facilitate optimizations of data sandboxing. With the guard zones, a memory read or write is safe with respect to the data-sandboxing policy if its address is either in the data region or in the guard zones.

Workflow. In order for an IRM-implementation framework to be retargetable, the majority of instrumentation and optimizations need to be decoupled from a specific ISA. A high-level representation provides such a vehicle. A remaining question is where to insert the IRM-instrumentation phase inside a compiler. A typical optimizing compiler has many layers that transform an IR program to another IR program with simplified semantics or better performance. Therefore, the IRM-instrumentation phase can be scheduled at any stage between IR generation and the backend.

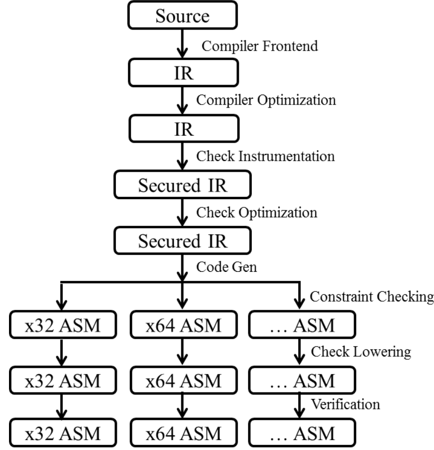

Figure 8: Workflow of Strato

On one end of the spectrum, we can schedule the IRM-instrumentation pass right after the compiler frontend; that is, after the IR is generated by the frontend. The benefit is that it can reuse a large number of existing IR-level optimization passes, which can optimize away unnecessary security checks. However, it has two drawbacks. First, since the security of low-level code generated by the compiler is what we are interested in, we need a way to ensure those IR-level optimizations do not wrongly optimize away security checks. One way to accomplish this would be to modify the optimizations to carry enough information to the low level for certification, similar to proof-carrying code. However, modifying complex compiler optimizations is non-trivial. The second and more serious drawback of scheduling the IRM-instrumentation pass right after the frontend is that existing optimizations may not be safe according to the attack model of an IRM. As we discussed before, the CFI attack model assumes that data memory is untrusted. However, a typical IR assumes a much different machine model. As an example, LLVM IR adopts an Unlimited Register Machine (URM) model in the SSA form [21]. A real machine has a limited number of registers and so LLVM IR variables may be spilled into untrusted memory locations. Therefore, if an LLVM IR optimization depends on the URM model for correctness, then the optimized result may not be safe according to CFI's attack model.

On the other end of the spectrum, the IRM-instrumentation phase can be scheduled right before the compiler backend for code generation. This is the design adopted in Strato. The downside is that existing compiler optimizations are not reused and we have to develop our own optimizations to optimize IRM checks to improve efficiency. But optimizations for security checks can be implemented straightforwardly at the LLVM IR level, which has a small number of instructions and is in the SSA form. For the policy of data sandboxing, we implemented three optimizations with ease, shared by all targets supported by LLVM. With this design, there is no need to trust or modify a large number of existing IR-level compiler optimizations. Optimizing compilers have a large code base and bugs are unavoidable [38].

Fig. 8 presents the workflow of Strato, which is implemented as extra passes added to the LLVM compiler. We next explain the steps of how source code is translated to low-level code through Strato-augmented LLVM. We add stars to those steps that are added in Strato to distinguish them from those steps already in LLVM.

(1) Compiler frontend. LLVM's clang frontend generates the IR code.

(2) Compiler optimizations. LLVM's transformations and optimizations change the IR code to simpler and optimized code.

(3) *Check insertion. Security checks are inserted before dangerous instructions to generate secured IR code. The dangerous instructions and checks inserted depend on the security policy. Since the current policy is CFI and data sandboxing, security checks are inserted before memory loads and stores as well as computed jumps. Note this step inserts more checks than necessary. Later steps will remove

unnecessary checks. Security checks are inserted as LLVM intrinsic functions, which will be lowered to machine-instruction sequences at a later step (if they are not optimized away).

(4) *Check optimization. After check insertion, custom optimizations for removing security checks are performed on the IR code. We implement three effective optimizations to demonstrate the amenability of high-level IR to optimizations: redundant check elimination, sequential memory access optimization, and loop-based check optimization. Our optimizations differ from traditional ones in that no security checks are removed or moved around at this step. A check that is deemed unnecessary is marked as removable and constraints are attached to it. The check will be removed only after the constraints are checked to be valid at a later step. If those constraints are violated by later steps of the compilation, the check will not be removed.

(5) Code generation. The compiler backend performs instruction selection, instruction scheduling, ISA-specific optimizations, and register allocation. Low-level assembly code is generated as the result.

(6) *Constraint checking. If the constraints for a check are invalidated during compiler transformations and optimizations, the check is kept intact. Otherwise, it is removed.

(7) *Check lowering. Security checks are lowered to machine-instruction sequences. Usually a security check can be implemented by multiple machine-instruction sequences. This step therefore provides a convenient place to experiment with different sequences to evaluate which one produces the best performance.

(8) *Verification. An independent verifier is run to check the low-level code. If the verification fails, then the code is rejected.

The above design makes it straightforward to adapt to a different ISA. Steps including check insertion, check optimization, and constraint checking can be reused across ISAs. The check-lowering and the verifier components need to be tailored for a new ISA. As we will discuss, the amount of effort involved to retarget Strato's implementation from x86-32 to x86-64 is small.

5.2 Check Instrumentation, Optimizations and the Constraint Language

The goal of security-check instrumentation is to guard dangerous operations with checks so that they cannot be abused by adversaries. For CFI, IDs are inserted before control-flow targets. Furthermore, checks are inserted before computed jumps, including indirect calls, indirect jumps and return instructions[10]; these checks ensure that the expected IDs are there at the targets of control flow transfers [1].

For data sandboxing, Strato inserts a check before each load and store instruction; the check ensures that the instruction's memory address is within the data region. In addition, after a definition of a pointer variable, a check is inserted to ensure that the pointer is within the data region. A check is also inserted at the entry of a function for a pointer parameter. In this step, Strato inserts more checks than necessary.

Since checks are inserted at the IR level, the same protection strategy is adopted for all machine targets, including x86-32 and x86-64. This provides uniformity and enables most of the code to be shared between targets. In contrast, NaCl adopts very different protection strategies for x86-32, x86-64, and ARM (segmentation on x86-32, large addresses and guard regions on x86-64, and address masking on ARM).

After check instrumentation, optimizations are run on the secured IR to mark unnecessary checks. To demonstrate the ease of implementing optimizations at the IR level, we have implemented three optimizations for removing unnecessary data-sandboxing checks: redundant check elimination, sequential memory access optimization, and loop-based optimization. In these optimizations, checks are not removed. Rather, checks that are deemed removable are marked and constraints are attached to them. Checks whose constraints are still valid after the backend processing are removed in the later constraint-checking step.

Redundant Check Elimination. Since the check-instrumentation step inserts a check after the definition of a pointer variable and also before the use of the variable via a load or store, the checks before the uses are redundant at the IR level. Fig. 9 presents an example. Column (a) presents the original C code and column (b) presents the LLVM IR code before check instrumentation. During check instrumentation, three

[10] Return instructions are changed to a sequence of pop, check, and indirect jump instructions to prevent a concurrent attacker from modifying the stack after the check.

(a) Original C code

```
int foo
(int v, int *ptr)
{
    int tmp = 0;
    if (v > 47)
       *ptr = v;
    else
       tmp = *ptr;
    return tmp;
}
```

(b) Unsecured IR code

```
entry:
   tmp = 0
   if(v > 47) goto then
else:
   tmp = load *ptr
   goto end
then:
   store v, *ptr
end:
   ret tmp
```

(c) Secured and optimized IR code

```
entry:
    ptr.safe = call guard(ptr) // check1
    tmp = 0
    if(v > 47) goto then
else:
    ptr.safe1 = call guard(ptr.safe) // check2
    # noSpill(ptr.safe, check1, check2)
    tmp = load *ptr.safe1
    goto end
then:
    ptr.safe2 = call guard(ptr.safe) // check3
    # noSpill(ptr.safe, check1, check3)
    store v, *ptr.safe2
end:
    ret tmp
```

Figure 9: An example for illustrating redundant check elimination. `guard` is the security check to ensure that the input pointer is in the data region, which is implemented as an LLVM intrinsic function.

(a) Original C code

```
struct s
{
  long x;
  long y;
};
int sum
(struct s *p)
{
  return p->x + p->y;
}
```

(b) Unsecured IR code for `sum`

```
x = gep p, 0, 0
tmp1 = load *x
y = gep p, 0, 1
tmp2 = load *y
sum = add tmp1, tmp2
ret sum
```

(c) Secured and optimized IR code

```
p.safe = call guard(p) // check1
x = gep p.safe, 0, 0
x.safe = call guard(x) // check2
# noSpill(p.safe, check1, check2)
# sizeof(struct s)*0 + sizeof(long)*0 < gz_size
tmp1 = load *x.safe
y = gep p.safe, 0, 1
y.safe = call guard(y) // check3
# noSpill(p.safe, check1, check3)
# sizeof(struct s)*0 + sizeof(long)*1 < gz_size
tmp2 = load *y.safe
sum = add tmp1, tmp2
ret sum
```

Figure 10: An example for illustrating sequential memory access optimization.

checks are inserted. First, a check is placed at the beginning of the function for the pointer parameter `ptr`. Second, since there are a load in block labeled `else` and a store in block labeled `then`, checks need to be placed before them. `check2` and `check3` are unnecessary at the IR level. However, they cannot be removed in the IR code because `ptr.safe` might be spilled into the untrusted stack during register allocation. Instead, `check2` is marked as removable and a constraint is attached, specifying that the check can be removed if and only if `ptr.safe` is not spilled between `check1` and `check2`. In Fig. 9(c), constraints are at lines starting with the # symbol. `check3` in block `then` is attached with a similar constraint. After register allocation, the constraint checker checks whether `ptr.safe` has been spilled. If not, the two checks are removed. Otherwise, they are kept intact.

LLVM IR is in the SSA form, making it easy to implement the above optimization. First, the def-use chain is explicit in the SSA form. Furthermore, the SSA form ensures that `ptr.safe` is not modified between `check1` and `check2`. By contrast, if carried out on machine code, the optimizer would have to perform dataflow analysis to determine whether a pointer has been guarded and modified.

Sequential Memory Access Optimization. Most programming languages support aggregate types such as structs in C and classes in C++. A common pattern exists in member accesses: a base pointer plus a constant offset is used to visit a specific member. With the guard zones before and after the data region, a memory access with a base pointer and a constant offset is safe as long as the base pointer is within the data region and the offset is smaller than the guard-zone size. This observation can be exploited to remove

(a) Original C code

```
long sum
(long *ar, long len)
{
  long rst = 0, i;
  for (i=0; i<len; ++i)
   rst += ar[i];
  return rst;
}
```

(b) Unsecured IR code

```
rst = 0
i = 0
if (len <= 0) goto end
for.body:
  ptr = gep ar, i
  tmp = load *ptr
  rst += tmp
  i += 1
  if (i >= len) goto end
  goto for.body:
end:
  ret rst
```

(c) Secured and optimized IR code

```
rst = 0
i = 0
ar.safe = call guard(ar) // check1
if (len <= 0) goto end
for.body:
  ptr = gep ar.safe, i
  ptr.safe = call guard(ptr) // check2
  # noSpill (ar.safe, check1, check2)
  # noSpill (i, check1, check2)
  # sizeof(long) * 1 < gz_size
  tmp = load *ptr.safe
  rst += tmp
  i += 1
  if (i >= len) goto end
  goto for.body:
end:
  ret rst
```

Figure 11: An example for illustrating loop optimization.

checks if members of an object are visited sequentially and the base pointer is shared by multiple visits.

In LLVM IR, the `getelementptr` instruction takes a base pointer and multiple indices as operands and is used to compute the address of a sub-element of an aggregate data structure. If the base pointer has been guarded, the offset is a constant, and the offset is determined to be smaller than the guard zone size, then the pointer computed by `getelementptr` does not need to be guarded again. However, the size of each member cannot be determined at the IR level because it may be target dependent. For example, type `long` takes 4 bytes in x86-32 and 8 bytes in x86-64. As a result, the check after the definition of a pointer variable through `getelementptr` can be marked as removable and attached with constraints specifying that the base pointer cannot be spilled and the final offset from the base pointer should be less than the guard-zone size.

Fig. 10 presents an example. In column (a), `struct s` contains two `long` members and the function `sum` computes their sum. In column (c), Strato inserts `check1` at the entry to function `sum` and `check2` and `check3` after each `getelementptr` instruction (abbreviated as `gep` in the figure). The constraints for `check2` specify that it can be removed if there is no spill between `check1` and `check2` for pointer `p.safe` and the offset `sizeof(struct s)*0 + sizeof(long)*0` is smaller than the guard-zone size (this condition can be determined to be true even at the IR level because it is always zero; however, values of other expressions may be target dependent). Similar constraints are attached for `check3`.

Loop-based check optimization. Loop optimization is important because programs tend to spend the majority of runtime in loops. Performance is improved if a security check inside a loop can be hoisted outside the loop. For example, if a pointer is not modified inside the loop, then a check for the pointer can be hoisted. As another example, if a pointer is incremented or decremented for a small stride (less than the guard-zone size) inside the loop, and there is a memory access through the pointer in the loop, then the check can be hoisted. The reason this is safe is because access to the guard zones is trapped; if the initial value of the pointer is checked to be inside the data region, then the change to the pointer in one loop iteration will make the pointer to be either in the data region or in guard zones and the access through the pointer serves as a check. This optimization follows the loop optimization described by Zeng et al. [41]; please refer to that paper for detailed analysis of the soundness of the optimization. As another optimization example, if there is a pointer that is calculated from the induction variable of the loop, the increment to the induction variable is a small stride (less than the guard zone size), and there is a memory access through the pointer in the loop, then the check can be hoisted. LLVM IR encodes loop information explicitly, making it easy to detect induction variables and strides.

In our optimizations, a hoistable check is not moved. Instead, a new check is inserted into the loop preheader and the old check is marked as removable and attached with constraints. An important constraint is that the relevant pointer cannot be spilled. Fig. 11 presents a concrete example. The program in column

```
constraint ::= noSpill [var, src, dst]
             | term comparator term
term ::= term + term | term * term |
         var | constant | gz_size
comparator ::= < | > | == | >= | <=
```

Figure 12: Syntax of the constraint language.

(a) adds elements in array `ar`. The program visits memory once per iteration. The check can be hoisted if and only if the induction variable is the only variable used to calculate the memory location and the stride is smaller than the guard-zone size and there is a memory visit using the memory location in every path inside the loop.

Summary about optimizations. The three optimizations demonstrate that a high-level IR can simplify the implementations of optimizations, which are reused among target ISAs. Additional optimizations enabled by a high-level IR can further decrease the runtime cost of Strato.

The constraint language. For completeness, we present the syntax of the constraint language in Fig. 12. A check may be attached with one or more constraints. All constraints need to be satisfied in order for the check to be removed; that is, there is an implicit conjunction when interpreting a list of constraints.

The constraint `noSpill [var, src, dst]` denotes that `var` cannot be spilled between `src` and `dst`, where `src` and `dst` are program locations. Note the semantics of this constraint is that the variable cannot be spilled along *every* control-flow path from `src` to `dst`. Therefore, the `noSpill` constraints in the example of Fig. 12 effectively require that `ar.safe` and `i` cannot be spilled in the entire loop. The *comparison constraint* "`term1 comparator term2`" represents a relation between `term1` and `term2`. It can be used to encode the constraint that a constant offset should be less than the guard-zone size, as in Fig. 10.

The design of the constraint language depends on what optimizations Strato supports and also LLVM's backend. For instance, the `noSpill` constraint is there because LLVM's backend may break this assumption by spilling variables to memory. An IR-level optimization may check more conditions. But if there is no possibility for the backend to break a condition, that condition does not need to be encoded and propagated. For instance, in loop optimizations, the condition that there must be a memory access through the pointer in question can be checked at the IR-level alone. Another point is that it is possible new optimizations may require adding new predicates to the constraint language. We believe the constraint language can be extended straightforwardly.

5.3 Constraint Checking and Check Lowering

After LLVM's backend processing, Strato performs constraint checking and check lowering. Constraint checking examines the constraints attached to each check and checks whether they are valid. If the constraints are valid, the check is removed. If they are invalid, the check is lowered to a sequence of machine instructions.

Our constraint language is designed so that constraints can be checked straightforwardly at the low level, with the help of information preserved by LLVM. For example, a comparison constraint becomes constant expressions at the low level because after fixing the ISA sizes of types become constants. To check a `noSpill` constraint, the constraint checker first identifies the correspondence between IR variables and registers (LLVM preserves enough information for this purpose) and uses data-flow analysis to check whether the register that corresponds to the IR variable is moved to memory between the source and destination locations.

Remaining checks are lowered to machine-instruction sequences. A high-level check can be implemented by many machine-instruction sequences. The overhead of different sequences varies. Strato makes it easy to try different machine-instruction sequences—only the check-lowering step needs to be modified. We have evaluated a large number of machine-instruction sequences for checks in CFI and data sandboxing. We discuss examples of possible sequences next, but leave the discussion about the performance overhead of various sequences to the evaluation section.

ID encodings. Strato's CFI instrumentation requires the encoding of IDs at control-flow targets. ID-encoding instructions have to satisfy two conditions. First, the instruction must take a long immediate value as an operand, which is used to encode the ID. Second, the instruction cannot introduce side-effects that

```
        andl $DATA_MASK, %eax
        movl (%eax), %eax
        cmpl $3, %eax
        ja .LBB5_8
        movl *.LJTI(,%eax,4), %eax
.LBB2:
        ...
.LBB8:
        ...
```

Figure 13: An example for illustrating path sensitivity in range analysis.

change the semantics of the program. The original CFI uses `prefetch` instructions for encoding IDs. We have evaluated a large number of alternative instructions that satisfy the two conditions. They are put into three groups. Instructions in the first group take an immediate value and assign it to a machine register. For example, "`movl $ID, %eax`" assigns the immediate value `ID` to register `eax`. It can be used to encode the ID as long as `eax` is dead at the point where the instruction is inserted.[11]. Instructions in the second group perform arithmetic operations on a register with the ID and assign the result to a register. For example, "`add $ID, %eax`" can be used to encode the ID as long as `eax` and the flags register are dead. Instructions in the last group take a register and the ID value and defines only the flags register. For example, "`cmp $ID, %eax`" can be used as long as the flags register is dead at the point of insertion.

5.4 Verification

Compiler optimizations and transformations are untrustworthy because compilers have a large code base and are buggy [27, 39]. Bugs in optimizations that remove security checks are even harder to catch because they do not crash programs but introduce vulnerabilities silently. To remove those optimizations out of the TCB, Strato includes a verifier at the end of the compilation pipeline to validate the assembly output of the compiler.

The verifier checks if the assembly code satisfies the CFI and data sandboxing policy. CFI verification is straightforward. The LLVM assembly preserves enough information to reconstruct a control-flow graph, which is used by the verifier to check if necessary ID-encoding and ID-checking instructions are there in the assembly. Data-sandboxing verification is more challenging because Strato's optimizers may remove or hoist checks. Strato's verifier follows the design of a previous verifier [41] to implement range analysis for data-sandboxing verification (with improvements; see below). The basic idea is to compute the ranges of registers at all program points and check if the ranges of memory addresses fall into the data region plus guard zones. The calculation of ranges uses a standard iterative algorithm until a fixed point is reached.

Strato's verifier improves the previous range-analysis verifier by adding path sensitivity. Ranges of registers may be different along different paths after a sequence of comparison and jump instructions. As an example, the assembly snippet in AT&T syntax in Fig. 13 is extracted from 175.vpr in SPECint2000. The range of register `eax` shrinks down to the data region after the `andl` masking, where `$DATA_MASK` is the constant mask for the data region. The `movl` instruction expands the range of `eax` to `[bottom, top]` because it loads from untrusted memory.[12] Without path sensitivity, the range of `eax` would remain `[bottom, top]` at the entry to the two successor blocks labeled with `.LBB2` and `.LBB8`; consequently, the verifier would report an out-of-range error on the `movl` instruction because its memory address is "`.LJTI + eax*4`", where .LJTI is a constant in the data region. With path sensitivity, the verifier computes the range of `eax` to be `[0, 3]` before `movl` and successfully validates that the address is within the data region plus guard zones. The verifier in the previous system [41] used instruction pattern matching to verify this pattern. By adopting

[11] `eax` is a caller-saved register and is dead at the entry to a function. Furthermore, dead registers can be identified through liveness analysis

[12] In our implementation, bottom is 0 and top is the maximum unsigned integer, which is $2^{32} - 1$ for x86-32 and $2^{64} - 1$ for x86-64.

```
popl %ecx
cmpl $ID, 1(%ecx)
jne error
jmpl *%ecx
```

Figure 14: A wrong CFI sequence for return instructions.

a path-sensitive analysis, Strato's verifier is more general and can verify all security-check optimizations we have presented.

With the help of the verifier, we discovered 35 subtle bugs in early versions of Strato. Those bugs would be hard to discover otherwise. We classify the bugs into three groups as follows.

- Bugs in CFI instrumentation code. CFI implementation inserts IDs at branch targets and check instructions before computed jumps. The check instructions need to load the IDs at target locations. Therefore, they visit memory and need to be sandboxed as well according to data sandboxing. As an example, the snippet in Fig. 14 contains an early version of a CFI check sequence for return instructions. The return address is popped into register `ecx`, which is then used to load the ID for comparison. The `cmpl` instruction visits the code region using address `ecx+1`, which is unsafe because it is from the untrusted stack. Our verifier successfully caught this bug and we fixed the sequence by inserting a data-masking instruction on `ecx` before `cmpl`.

- Bugs in the source program. Our verifier even found a bug in the source code of `253.perlbmk`, a program in SPECint2000. The bug is a possible null-pointer dereference. The `perlbmk` program has its own `malloc` function, which can return a null pointer when a memory allocation fails and the `malloc` is inlined by the compiler. A null pointer is represented as value 0 and is outside the range of data region plus guard zones. It was caught by the verifier. We modified the source code of `perlbmk` to fix this bug.

- Bugs in LLVM intrinsic functions such as `llvm.memset` and `llvm.memcpy`. LLVM synthesizes programs into intrinsic function calls as an optimization. These intrinsic function calls can be lowered into a sequence of machine instructions or direct calls to the library functions depending on the tradeoff between code size and the function-call overhead. At the IR level, there is no way to predict how an intrinsic function will be lowered. If they are lowered into machine-instruction sequences, then their pointer arguments need to be sandboxed as they may visit memory. Our verifier caught these bugs and we fixed those machine-instruction sequences to insert data-masking instructions for pointers.

The combined verifier for x86-32 and x86-64 contains approximately 7k LOC including white space lines, comments, and debug statements. The majority of the code is a giant switch table for all the machine instructions that the verifier has to support (In LLVM 2.9, x86-64 target contains 3,747 machine instructions). The size of the verifier is a concern and we leave its verification for future work [25].

Finally, we note that our verifier performs verification at the assembly level. We use it mostly to catch bugs in the Strato compiler so that the compiler is out of the TCB. A more desirable design is to implement a verifier for binaries directly. This is actually a matter of engineering: we just need to modify the assembler to encode the control-flow graph as extra information in a binary so that the binary-level verifier can disassemble the binary reliably; the kinds of verification tasks involved are the same after disassembly.

5.5 Implementation and Evaluation

We have implemented Strato on top of LLVM 2.9. The check-instrumentation step is implemented as a pass and scheduled after LLVM's IR optimization passes. The security-check optimizations are performed right after check instrumentation. The constraint checking and check lowering are implemented in one pass in the compiler backend after register allocation. The range-analysis based verifier is scheduled at the very end in the backend. Constraints are encoded as LLVM *metadata*. In total, the instrumentation and optimizations

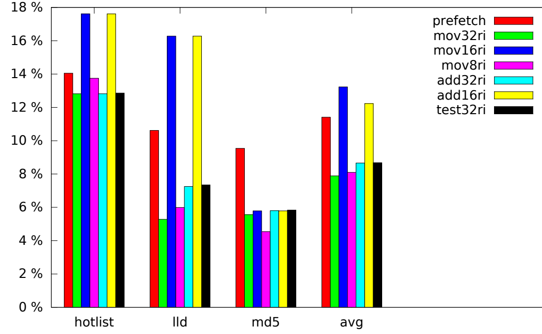

Figure 15: Performance overhead for CFI with various ID-encoding instructions on bakeoff programs.

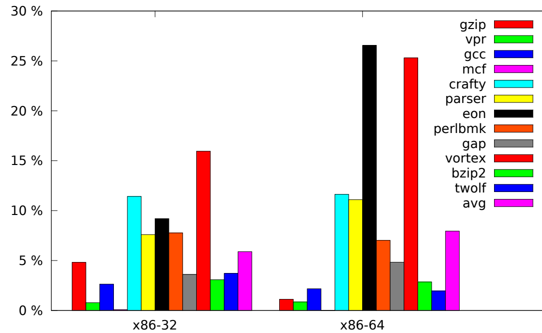

Figure 16: Performance overhead for CFI with `movl` as the ID-encoding instruction on SPECint2000.

consist of approximately 3,750 lines of C++ code, shared between x86-32 and x86-64. The constraint checking and check-lowering component has 1,420 lines of C++ code with an additional 180 lines added for x86-64. The verifier includes 6,960 lines of C++ code, with 1,240 lines added for x86-64.

The object code after data-sandboxing and CFI instrumentation cannot run directly and needs specialized linker scripts and a specialized loader. We have developed linker scripts for both C and C++ programs targeting x86-32 and x86-64. The linker scripts link object code generated by LLVM to three sections (including code, data, and read-only data) at specified addresses. We have also developed a loader that loads various sections in a binary at specified addresses in the address space and sets up appropriate protection for those sections using the `mprotect` system call. We reused PittSFIeld's library wrappers and libraries for x86-32 [23]; we also adapted them for x86-64.

For evaluation, we have built and run the benchmark suites bakeoff and SPECint2000 in Strato. The bakeoff benchmark suite contains three programs: hotlist, lld, and md5. It has been used by previous code-sandboxing frameworks for evaluation [18, 19, 32]. SPECint2000 contains twelve computation-intensive programs and is widely used for compiler evaluation. All benchmark programs in bakeoff and SPECint2000 can be successfully compiled in Strato. All programs were compiled with the `-O3` full optimization level except for 254.gap in SPECint2000, which ran correctly only with `-O0` enabled due to bugs in LLVM 2.9's optimizations. All experiments were conducted on a Ubuntu 11.10 box with an Intel Core 2 Duo CPU at 3.16GHz and 8GB of RAM. Experiments were averaged over three runs and the standard deviation was less than two percent of the arithmetic mean.

Performance evaluation. IRMs insert runtime checks into programs and slow down program execution. We present the performance overhead as the percentage of execution-time increase of instrumented programs compared with uninstrumented programs.

We first evaluated the performance implication of alternative machine-instruction sequences that implement the same high-level checks. We have tested a large number of alternative ID-encoding instructions, classified into three groups discussed in Sec 5.3. Fig. 15 presents the runtime overhead of various ID-encoding instructions on bakeoff programs for x86-32 when enforcing the CFI policy. In the figure, color bars are used for different ID encodings. The figure presents only a subset of what we have tried due to space limit. In

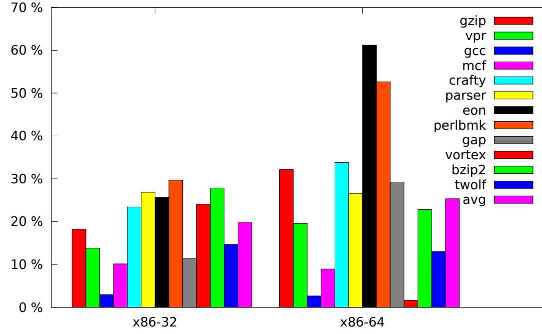

Figure 17: Performance overhead for CFI combined with data sandboxing for both reads and writes on SPECint2000.

addition, we evaluated different ID lengths such as 32-bit IDs, 16-bit IDs and 8-bit IDs. Shorter IDs do not necessarily have better performance and they shrink the space for IDs. In the figure, we use `mov32ri` to represent the case of using a 32-bit `mov` instruction that moves a 32-bit immediate value to a general register. As can be seen from the figure, most ID-encoding instructions are more efficient than `prefetch`, the one used in the original CFI implementation.[13] The case of 32-bit `movl` instruction has the lowest runtime overhead.

Fig. 16 presents the performance overhead of CFI alone on SPECint2000 with `movl` as the ID-encoding instruction. CFI incurs an average of 5.89% and 7.95% slowdown on x86-32 and x86-64, respectively. Our CFI implementation is competitive with previous CFI systems. The original CFI work [3] has 15% overhead and our own previous work [41] has 7.7% overhead on x86-32.

Fig. 17 presents the overhead of enforcing both CFI and data sandboxing for SPECint2000. Both cases for x86-32 and x86-64 are presented. The numbers are with respect to the case of using the `move32ri` instruction as the ID encoding in CFI, and the `and` instruction for sandboxing memory addresses. On average, Strato incurs 37.7% on x86-32 and 39.3% on x86-64 for SPECint2000 programs.

In summary, Strato provides competitive performance and provides retargetability, lacked by previous systems.

6 Improving the Efficiency of Dynamic Taint Tracking

In this section, we discuss improving the runtime efficiency of dynamic taint tracking through enforcement in a compiler.

6.1 Introduction

Taint tracking, also referred to as taint analyis, is a type of information flow analysis. It can also be used as a powerful security mitigation against control hijacking attacks, memory corruption attacks etc. It can detect whether malicious users have manipulated control flow and data pointers, thus preventing buffer overflow attacks. Dynamic taint tracking tracks down the information flow from untrusted taint source, to sensitive operations, i.e. taint sinks at runtime. The taint sources can be files such as pdf, mp3 or html files, network packets, keyboard, mouse and touchscreen input, webcam et al. The taint sinks can be sensitive operations such as control flow transfers, system calls etc. During program execution, the taint tracker propagates taints to keep track of tainted information according to a set of rules, called taint propagation rules. If an operation uses the value of some tainted object, say X, to compute the value of another object, say Y, then object Y becomes tainted. Taint propagation is transitive.

Dynamic taint tracking can be used for attack detection and prevention. Current software attacks usually arrive as user input from a remote communication channel, and once resident in a program memory, triggers pre-existing software flaws and overwrites control data to hijack program execution. Taint tracking can track

[13]Since the original CFI work, some versions of Intel and AMD hardware changed the behavior of `prefetch`; it becomes more expensive, since it pulls in TLB entries. As a result, choice of IDs used in `prefetch` greatly affects its runtime cost.

down the information flow from untrusted users to sensitive operations including control flow transfers. It can detect whether user-controlled data reach sensitive operations such as control flow transfers.

Unfortunately, current dynamic taint tracking incurs prohibitively heavy overhead. Current implementations are so slow that it sees little adoption in production systems. Even the fastest dynamic taint tracker doubles the program execution time. For example, Argos, a dynamic taint tracker based on QEMU, incurs 15x-26x slowdown depending on the benchmark program [26]. As another example, Minemu, a fast dynamic taint tracker, incurs a slowdown of 1.5x-3x [11]. Their performance is unsatisfactory and prohibits them from being adopted in production systems.

The major reason for the slowdown is that current dynamic taint trackers are mostly built on top of emulators and simulators using dynamic translation which itself is expensive. For instance, Argos incurs 15x-26x slowdown, most of which is caused by the emulator QEMU [7]. Minemu incurs 2.4x slowdown on SPECint 2006 on average, of which 1.4x is caused by dynamic translator [11], albeit designed specifically for taint tracking. In addition, emulators dynamically translate binary code which usually contain no symbol tables, no debug information, and little structured information that are necessary for aggressive program analyses and optimizations to reduce the runtime overhead. In an optimizing compiler, the whole framework is designed to assist program analyses and optimizations. Many optimizations can be applied to dynamic taint tracking conveniently. For instance, LLVM IR is in SSA form which makes it trivial to find the definition and the uses of a variable. Also, the type information, control flow information etc. existing in intermediate representation is useful for program analyses. Many optimizations that would be difficult if not impossible to do on the binary level can be implemented conveniently on LLVM IR.

Furthermore, LLVM IR has a limited amount of instructions compared with a modern physical ISA. LLVM IR assumes a Unlimited Register Machine (URM) model based on the design philosophies of RISC architecture. For example, in lastest LLVM 3.3, there are approximately 60 instructions while in x86-64 there are more than 3,500 machine instructions. The instrumenter is much simpler and easier to reason about than on binary level.

Dynamic taint tracking implemented as a special IRM during compilation provides the opportunity to greatly lower the heavy overhead. First, no heavyweight dynamic binary translation is required. Taint propagation operations are inlined within the subject programs during compilation, thus obviating dynamic binary translation. Second, many optimizations enabled by the abundance of structured information and a compiler framework can be adopted to eliminate unnecessary taint propagation operations. A compiler has many infrastructures and facilities to assist program analyses and optimizations. In addition, the intermediate representation contains the abundance of structured information which is amenable to program analysis and transformations.

As a simple example, an operation takes two objects, say A and B, and computes the result C. If it is statically proved that A and B cannot be tainted by user input through static analysis, then there is no need to propagate taint to object C. On the other hand, if A or B is tainted, then C is tainted statically. We can taint C together with A or B in one operation. In other words, many taint propagation operations can be conducted statically during compilation instead of dynamically to improve runtime efficiency. As another example, if an object in a loop is tainted, there is no need to taint it for every iteration. In other words, we can hoist it up to the loop preheader or sink it down.

6.2 Design and Implementation

Next, we talk about the design and implementation of dynamic taint tracking in the compiler framework LLVM. For each memory unit, the taint tracker needs a special memory location, to store the metadata, also known as taint tags. These special memory locations reside in a memory region, referred to as shadow memory. In order to protect shadow memory from being corrupted by untrusted user programs, we need to isolate shadow memory somehow. To separate shadow memory from program memory, we can use software-based fault isolation to ensure that subject programs cannot tamper with taint tags. In addition, control flow integrity can be employed to make sure that embedded taint propagation operations and SFI checks cannot be bypassed by adversaries. Our SFI and CFI implementation in previous work incurs low overhead compared with dynamic binary translation [41].

The framework is built on top of industrial strength compiler LLVM. A function-level pass is scheduled in LLVM IR pipeline to do the instrumentation. The pass is inserted at the end of LLVM IR pipeline

	hotlist	lld	md5	average
slowdown(%)	5.34	3.19	4.79	4.44
code (%)	21.94	20.22	90.67	44.28
data (%)	1466.67	920.00	294.74	893.80
bss (%)	0.00	0.00	40.00	13.33
binary (%)	23.01	0.23	91.39	38.21

Table 3: Runtime overhead of dynamic taint tracking on bakeoff

	DES	PC1	RC4	CRC32	average
slowdown(%)	98.18	0.20	4.82	7.84	27.76
code (%)	128.68	36.18	43.13	39.96	61.92
data (%)	7.60	209.09	25.00	0.58	60.57
bss (%)	0.89	0.85	44.44	0.59	11.69
binary (%)	102.20	29.27	41.94	8.18	45.40

Table 4: Runtime overhead of dynamic taint tracking on encryption algorithms

because the LLVM IR code has been optimized, simplified and canonicalized by the LLVM optimizer. The instrumenter pass on IR level is much simpler and easier to reason about than on binary level. For each LLVM IR instruction, we define an instrumentation rule to guide the instrumenter to insert taint propagation operations modeled as LLVM IR instruction sequences. After the instrumentation pass, the LLVM IR code is lowered into machine code by LLVM backend.

6.3 Evaluation and Performance

We have built a proof-of-concept prototype on LLVM 3.3, the latest official release. The instrumenter, implemented as a function-level pass, is scheduled at the end of IR pipeline after compiler optimizations. We have evaluated the runtime performance of our prototype on bakeoff, which is a widely used benchmark suite for sandboxing [18, 19, 32] and a few widely adopted encryption algorithms including DES, PC1, RC4, and CRC32 on x86-64. The preliminary experimental results are encouraging.

The experiments were conducted on a 64-bit Ubuntu 11.10 box with an Intel Core 2 Duo CPU running at 3.16GHz and 6GB of RAM. All the experiments were averaged over three runs and the standard deviation was less than one percent of the arithmetic mean. The taint sources include the command line options passed to the programs. The taint sinks contain printf statements, malloc call arguments, and free call arguments etc. Fig. 3 and Fig. 4 show the measurements of our prototype on bakeoff and the aforementioned encryption algorithms. For bakeoff, the average slowdown is about 4.44 percent, significantly lower than previous work. Previous work for dynamic taint tracking incurs as much as twice runtime slowdown. We also measured the code section size, data section size, bss section size and binary size of the executables compiled by our prototype, as shown on the 3rd to 6th rows in Fig. 3 and Fig. 4 respectively. Our prototype increase the code size by about 44.28% on bakeoff and 61.92% on encryption algorithms on average. The data size is increased by a large amount because the data section is very small in these binaries.

Althought these preliminary performance numbers look promising, we are still building other enhancements such as shadow stack, which can increase the overhead significantly, and customized optimizations which can reduce the runtime overhead. We will report further experimental results after we completed the enhancements. We plan to make a conference submission by the end of 2013.

7 Conclusions and Future Work

In this thesis, we studied prevalent software attacks and their existing vulnerability mitigations. Although buffer overflow attacks have been around for more than 25 years and many ingenious mitigations have been invented, they still pose serious threats to software industry and end users. Recently, buffer overflows have

morphed into other forms and present new challenges to security research. We investigated the current vulnerability mitigations and studied their shortfalls. Few mitigations meet the aforementioned requirements for security defenses. For example, IRMs are promising defense techniques because they are powerful enough to enforce many policies, but they are not retargetable. Their runtime overhead is still too high for performance-critical systems and their trustworthiness is questionable without verifiers. We have proposed solutions to tackle the problems towards industrial adoption. First, traditional sandboxing implementations target only memory writes but not reads due to runtime performance considerations. We have combined CFI with data sandboxing and proposed optimizations to cut down the overhead to a new low level. Also, current IRMs are enforced through binary rewriting or other low-level transformations. We have proposed a framework to enforce IRMs at an intermediate representation level, i.e. LLVM IR, which enables language agnosticism, retargetability, and aggressive optimizations. Furthermore, current taint tracking is enforced either by language-dependent source-level transformation or through ISA-specific binary rewriting. They incur prohibitively heavy runtime overhead. We proposed a new approach to do taint tracking on LLVM IR, thus enabling language-independency, retargetability, and optimizations.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In Kung-Kiu Lau and Richard Banach, editors, *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 2005.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

[4] Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM Trans. Inf. Syst. Secur.*, 15(2):8:1–8:29, July 2012.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[8] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[9] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[10] Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *Proceedings of the fourth ACM conference on Wireless network security*, WiSec '11, pages 127–138, New York, NY, USA, 2011. ACM.

[11] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: the world's fastest taint tracker. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, RAID'11, pages 1–20, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[13] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with stackguard. In *In Linux Expo*, 1999.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

[15] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[16] Dinakar Dhurjati and Vikram S. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE*, pages 162–171, 2006.

[17] Tyler Durden. Bypassing pax aslr protection, 2002.

[18] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[19] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: a retrospective. In *Proceedings of the 1999 workshop on New security paradigms*, NSPW '99, pages 87–95, New York, NY, USA, 2000. ACM.

[20] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[22] LLVM 2.8. http://llvm.org.

[23] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[24] microsoft. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003, September 2006.

[25] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: Better, faster, stronger SFI for the x86. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 395–404, 2012.

[26] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, April 2006.

[27] John Regehr. The future of compiler correctness, August 2010.

[28] Mattew J. Schwartz. Microsoft hacked: Joins apple, facebook, twitter. http://www.informationweek.com/security/attacks/microsoft-hacked-joins-apple-facebook-tw/240149323, 2013.

[29] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[30] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[31] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[32] Christopher Small. A tool for constructing safe extensible c++ systems. In *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, COOTS'97, pages 13–13, Berkeley, CA, USA, 1997. USENIX Association.

[33] Christopher Small and Margo Seltzer. A comparison of os extension technologies. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[34] Sophos. Security threat report 2013. http://www.sophos.com/en-us/medialibrary/PDFs/other/sophossecuritythreatreport2013.pdf, 2013.

[35] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[36] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 299–308, 2012.

[38] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

[39] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[40] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Orm, Shiki Okasaka, Neha Narula, Nicholas Fullagar, and Google Inc. Native client: A sandbox for portable, untrusted x86 native code. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2009.

[41] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *18th ACM Conference on Computer and Communications Security*. ACM, October 2011.