Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing

Bin Zeng
Department of Computer Science and Engineering
Lehigh University
zeb209@lehigh.edu

Gang Tan
Department of Computer Science and Engineering
Lehigh University
gtan@cse.lehigh.edu

Greg Morrisett
School of Engineering and Applied Sciences
Harvard University
greg@eecs.harvard.edu

ABSTRACT

In many software attacks, inducing an illegal control-flow transfer in the target system is one common step. Control-Flow Integrity (CFI [1]) protects a software system by enforcing a pre-determined control-flow graph. In addition to providing strong security, CFI enables static analysis on low-level code. This paper evaluates whether CFI-enabled static analysis can help build efficient and validated data sandboxing. Previous systems generally sandbox memory writes for integrity, but avoid protecting confidentiality due to the high overhead of sandboxing memory reads. To reduce overhead, we have implemented a series of optimizations that remove sandboxing instructions if they are proven unnecessary by static analysis. On top of CFI, our system adds only 2.7% runtime overhead on SPECint2000 for sandboxing memory writes and adds modest 19% for sandboxing both reads and writes. We have also built a principled data-sandboxing verifier based on range analysis. The verifier checks the safety of the results of the optimizer, which removes the need to trust the rewriter and optimizer. Our results show that the combination of CFI and static analysis has the potential of bringing down the cost of general inlined reference monitors, while maintaining strong security.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.4 [**Programming Languages**]: Processors

General Terms

Security, Verification

Keywords

Control-Flow Integrity, Static Analysis, Binary Rewriting, Inlined Reference Monitors

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

1. INTRODUCTION

Software attacks often exploit vulnerabilities such as buffer overflows and format-string handling errors that are common in large software systems. In many attacks, one essential step is to induce an illegal control transfer through, for example, overwriting a return address or a function pointer. The illegal control transfer might jump to new code injected by attackers, as in code-injection attacks; it might jump to code already in the target program, as in return-into-libc attacks. The extreme case is Return-Oriented Programming (ROP [9, 26]), which can induce any malicious behavior by combining code snippets in the program with arbitrary control flow. In all these attacks, the expected control-flow graph of the target program is violated.

Control-Flow Integrity (CFI [1,2]) is a defensive technique that can foil attacks based on illegal control transfers. A program satisfies control-flow integrity if it follows a pre-determined control-flow graph. The expected control-flow graph serves as a specification of control transfers allowed in the program. A software-based CFI implementation inserts runtime checks to enforce the specification. The runtime checks will catch and prevent illegal control transfers attempted by attacks.

It is generally believed that CFI is a principled defense mechanism against Return-Oriented Programming (ROP). Previous research has shown that ad-hoc defenses fail to prevent simple variants of ROP [9]. The inventors of ROP have argued that research should instead focus on comprehensive defenses such as CFI.

An attractive property of CFI is that it can be enforced on almost all software, including legacy C and C++ code, and even assembly code without breaking applications and without requiring special hardware features (e.g., segment registers). Software-based Fault Isolation (SFI) [31] is another policy that can be applied in a language-agnostic way. However, SFI only provides weak integrity for control-flow and as such, does not prevent return-into-libc and ROP attacks completely. Furthermore, the lack of an enforced control-flow graph prevents the use of standard compiler techniques for optimizing SFI enforcement mechanisms. In contrast, the enforced control-flow graph of CFI supports standard dataflow analysis and optimizations. Thus, in addition to protecting applications from control-hijacking attacks, CFI provides a basis for efficient enforcement of any inlined reference monitor. In particular:

- *Optimization.* Optimizers can perform static analysis to eliminate unnecessary security checks if they are statically proven unnecessary. This reduces the runtime overhead of enforcing security.

- *Verification.* Static analysis can be used to verify the result of binary rewriting and optimizations. The verification checks whether the rewritten and optimized code obeys the desired security policy, removing the binary rewriter and the optimizer from the TCB.

The original work on CFI recognized the potential for static analysis, but did not take advantage of it. In this paper, we investigate how CFI-enabled static analysis can be used to cut the overheads of SFI-style data sandboxing. Previous implementations of SFI either rely upon segment registers to enforce data sandboxing, or else use software-based techniques but only instrument memory writes due to the high overhead of sandboxing reads. In contrast, we consider software-only techniques, which are applicable to a much wider class of architectures, and consider sandboxing both memory writes as well as reads.

We highlight key contributions of the paper as follows:

- We describe a series of optimizations for efficient data sandboxing in Sec. 4. These optimizations utilize static analysis including liveness analysis and range analysis to eliminate unneeded security checks. Our implementation on x86-32 adds modest runtime overheads on top of CFI (an average of 2.7% for write protection and 19% for read-and-write protection on SPECint2000).

- We propose to use range analysis as a principled way to verify data-sandboxing optimizations in Sec. 5. A single verifier can verify all our implemented optimizations and beyond.

- Since CFI is used as the basis for static analysis, reducing the overhead of the CFI enforcement itself is beneficial. In Sec. 6, we describe two simple optimizations for faster CFI. The optimizations cut the runtime cost of CFI by more than half.

- Our ideas have been implemented in LLVM and fully evaluated using benchmark programs. Sec. 7 describes our implementation and evaluation.

2. RELATED WORK

We divide closely related work into three categories.

SFI. Software-based fault isolation (SFI, [12,14,21,25,28, 31,34]) rewrites unsafe code to insert sandboxing instructions before memory and control-transfer instructions. The inserted instructions prevent the code from accessing memory outside of a designated data region and executing instructions outside of a designated code region. For avoiding high overhead, most SFI systems sandbox memory writes but not memory reads. NaCl x86-32 [34] restrains memory access through hardware segmentation that applies to memory reads as well as writes. Hardware segmentation, however, is unavailable on x86-64 and ARM. As a result, NaCl's implementations on these processors rely on SFI to sandbox memory writes [25] (or both reads and writes with a significant performance penalty [6]). Our SFI support reduces overhead through static analysis and in principle applies to processors with or without segmentation.

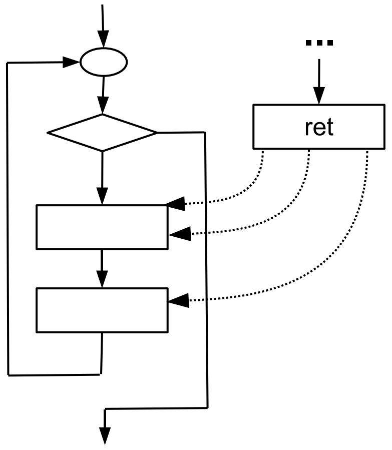

Figure 1: The insufficiency of chunk-based control-flow integrity for static analysis.

One subtle requirement of SFI is that some form of control-flow integrity should be enforced so that inserted checks cannot be bypassed. PittSFIeld [21] and NaCl [25,34] adopt the *chunk-based control-flow integrity.* Code is divided into atomic chunks of fixed sizes such as 16 bytes. A computed jump is restricted by runtime checks to target only beginnings of chunks. Checks before memory operations cannot be bypassed as long as they are in the same chunk.

The chunk-based control-flow integrity guarantees only an imprecise control-flow graph and is insufficient for most static analyses. To illustrate this point, we use loop optimizations as an example. The control-flow graph in Fig. 1 would contain a loop if those dotted edges do not exist. In chunk-based control flow integrity, a `ret` instruction can jump to any chunk beginnings (because the instruction makes a control transfer according to a value from the untrusted stack). These control transfers are represented by dotted edges in the figure. Therefore, basic blocks in the loop body have to be broken into chunks. Worse, the control-flow graph in the figure is no longer a loop because by definition there should be no edges from nodes outside of the loop to the loop's internal nodes. Consequently, loop optimizations cannot be performed.

SFI traditionally provides coarse-grained data sandboxing with respect to one big, contiguous data region. Recent work [4,5,8] extends SFI to provide fine-grained data integrity, which accommodates multiple data regions of various granularity (e.g., at the byte level). However, confidentiality is not supported. Furthermore, the implementation seems to rely upon re-compiling the source, whereas our rewriter and verifier work directly on the assembly language. Consequently, the compiler is not in the TCB in our system. Finally, their techniques do not work for multi-threaded applications; our system is thread safe since it assumes a concurrent attack model. We believe the general idea of using static analysis for optimization and verification should also benefit fine-grained data sandboxing.

XFI. Extended Fault Isolation (XFI, [11]) is a closely related system in its approach. Similar to our system, XFI builds on CFI and exploits the control-flow graph for stronger security and for performance. On the one hand, XFI's goal is to provide a more comprehensive protection system for loading untrusted code. For instance, it also provides a high-integrity stack for protecting return addresses. On the other hand, our system explores more aggressive optimizations such as loop optimizations. The static analysis enables our system to have more efficient support for data sandboxing (more comparison with XFI will be in the following sections).

Program shepherding. Instead of static rewriting, program shepherding [18] (as well as systems in [22, 24]) relies on dynamic binary rewriting to enforce security. One downside of dynamic rewriting is the whole dynamic optimization and monitoring framework is in the TCB, whereas only a verifier needs to be trusted in static rewriting. Furthermore, program shepherding relies on page protection for memory protection; it cannot prevent untrusted application code from reading outside its data region (e.g., the dynamic rewriter's own code and data regions are readable).

Static analysis on low-level code. Analyzing low-level code such as assembly code is more difficult than analyzing source code due to the lack of structured information. Despite this difficulty, there has been plenty of research that applies static analysis to low-level code for various purposes (e.g., [7, 29, 33]). We use static analysis to reduce runtime overhead of SFI-style data sandboxing. Compared with previous low-level code analysis, our static analysis is simpler and more specialized. First, since our attack model (described in Sec. 3) assumes data memory can change between instructions, our static analysis only needs to track properties of registers. Second, some of our static analysis such as range analysis is geared toward the purpose of optimizing checks for guaranteeing safe access within data memory.

Code sandboxing. Also related is the general idea of sandboxing. SFI-style sandboxing, the topic of this paper, is a particular form of sandboxing untrusted code. Isolating untrusted code in a trusted environment has long been a goal of computer-security research. This line of work includes systems that monitor and restrict OS system calls [15–17, 23], systems that isolate device drivers from kernel code [30], systems that isolate web applications from browsers [10, 34], and systems that isolate native code from language virtual machines [19, 27].

3. ATTACK MODEL AND SECURITY POLICY

We adopt the CFI attack model. It is both conceptually simple and realistic. It assumes separate code and data regions for an untrusted program. The data region is not executable.[1] An attacker is then modeled as a thread that runs in parallel with the program. The concurrent attacker thread can overwrite any part of the data region, including the stack, the heap, and global data. This model effectively assumes that contents in the data region can arbitrarily change between any two instructions in the program. This rather pessimistic assumption captures real attack scenarios and is also amenable to formal analysis [2].

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 the attacker to 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.

Control-flow security policy. A CFI policy for a program is a graph whose nodes consist of addresses of basic blocks, and whose edges connect control instructions (i.e., jumps and branches) to allowed destination basic blocks. Within a procedure, 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 addresses where control is allowed to transfer.

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

CFI policies are in essence NDFA's or regular expressions denoting sets of possible control traces. In contrast, XFI provides a richer language of policies (corresponding to pushdown 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.

Data sandboxing policy. This policy dictates that any memory access in the untrusted program must be within the data region. 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 military applications, protecting confidentiality can be as important.

We next formalize the data sandboxing policy. We assume there is a large, contiguous region of data memory that untrusted code can read and write. We assume the data region begins with the address `DB` (Data Begin) and ends with the address `DL` (Data Limit). That is, the address range of the data region is `[DB, DL]`, inclusive. Following previous SFI systems, we set up *guard zones* of size `GSize` before and after the data region. The size of a guard zone can vary depending on the host policy. It is further assumed that memory access to locations in guard zones can be efficiently trapped (through page protection). Fig. 2 depicts the data region, the guard zones, and relevant parameters.

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

[1] This assumption can be discharged either by the hardware No-eXecute (NX) protection or by a pure software approach; the software approach sandboxes indirect jumps so that control always stays in the code region.

[2] The size of the memory access is irrelevant because of the presence of the guard zones and the assumption that access 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 a special case and will be ignored in the rest of the paper.

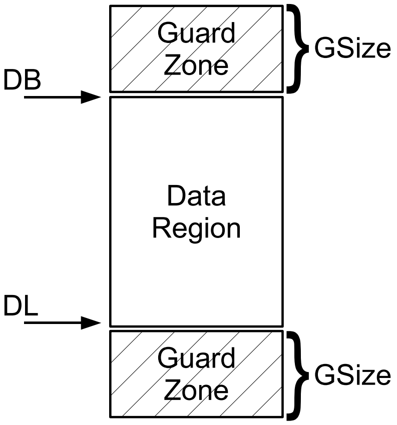

Figure 2: Data region and guard zones.

4. DATA SANDBOXING OPTIMIZATIONS

We next present a series of optimizations that significantly cut the data-sandboxing cost. The data-sandboxing optimizations utilize static analysis to identify optimization opportunities. The optimizations are similar to those performed in an optimizing compiler, except they need to not only improve performance but also maintain the security.

Note this section assumes CFI is already in place; Sec. 6 will discuss CFI and its own optimizations. One immediate benefit of CFI is it obviates the need for chunk-based control-flow integrity [21, 25]. CFI provides better security as its control-flow graph is more precise. Moreover, it also avoids the large number of no-ops that have to be inserted for instruction alignment. The extra no-ops add both spatial and temporal overheads. PittSFIeld [21] reports that inserted no-ops account for about half of the runtime overhead of enforcing data integrity. NaCl [25] 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 illustrates how the optimizations work.

A running example. Fig. 3 presents the example. For clarity, this and other programs in the paper 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. 3(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. 3(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 `$DMask` denotes the data-region mask. Following PittSFIeld [21], a single bitwise-and sandboxing instruction is used to ensure that the resulting address is in the data region. For instance, if the data region starts from 0x20000000 and is of 16MB size, then `$DMask` is `0x20ffffff`. The sandboxing instruction will also be called a check in the rest of the paper.

4.1 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. 3(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 [21] 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. 3, 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. 3(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. 3(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

(a) Original instruction

```
ebx := [ecx + 4]
```

(b) Unoptimized instruction sequence

```
push eflags
push eax
eax := ecx + 4
eax := eax & $DMask
ebx := [eax]
pop eax
pop eflags
```

(c) When `eax` is a dead register

```
push eflags
eax := ecx + 4
eax := eax & $DMask
ebx := [eax]
pop eflags
```

(d) When the flags register is dead

```
eax := ecx + 4
eax := eax & $DMask
ebx := [eax]
```

(e) After in-place sandboxing

```
ecx := ecx & $DMask
ebx := [ecx + 4]
```

Figure 3: A running example for illustrating optimizations. `DMask` is the data-region mask.

the end of a basic block). This seems to be an assumption used by compilers and has been confirmed by our experiments on SPECint2000.[3]

PittSFIeld [21] 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.

4.2 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 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. 3 uses address `ecx+4`. Therefore, the sequence in Fig. 3(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 `GSize` ≥ 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.

4.3 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.

```
              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 4: An example demonstrating redundant check elimination.

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

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

(a) Unoptimized sequence

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

Figure 5: An example demonstrating loop check hoisting.

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. 4. 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. 5 presents an example program showing how static analysis enables 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) {
  sum = sum + *p;
  p = p + 1;
}
```

In Fig. 5, `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. 5. 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. 5(b).

It is instructive to understand why the code in Fig. 5(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` ∈ `[DB, DL+4]` is a loop invariant. The condition is clearly true at the beginning of the loop since the sandboxing instruction gives `esi` ∈ `[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` ∈ `[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.

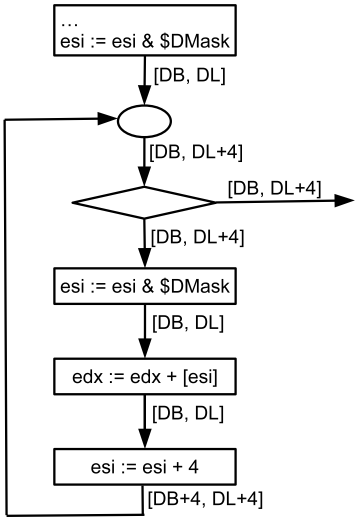

Figure 6: Range analysis result for the program in Figure 5(a), after the check is duplicated at the loop pre-head. Only the ranges of `esi` are included.

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 [3].

(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. 5(a), this step results in the program depicted in Fig. 6. 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 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.[4]

[4] Certain programming practices use pointers that are out-

(3) A range analysis is then performed to decide if any check is unnecessary. Fig. 6 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` $\geq$ 4). This enables the optimizer to remove the check in the loop body. After its removal, we get the optimized program in Fig. 5(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.

5. 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]`.

The following example is the code in Fig. 4 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. 5(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.

6. 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.

6.1 Original CFI instrumentation

As background information, we discuss CFI's original mechanism for ensuring that a program's execution follows a predetermined control-flow graph [1, 2]. 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 transfer is consistent with the control-flow graph.

Fig. 7 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 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 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 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.

6.2 CFI optimizations

We next describe two simple CFI optimizations.

Jumping over prefetch instructions. The original CFI

side 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

Figure 7: A CFI example.

Original code	Code after instrumentation
`call fun`	`call fun`
	`prefetchnta [$ID]`
`ret`	`ecx := [esp]`
	`esp := esp + 4`
	`if  [ecx+3]≠$ID goto error`
	`ecx := ecx + 7`
	`jmp ecx`

Figure 8: New CFI instrumentation by skipping over prefetch instructions.

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. 8 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.[5] 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. 9 presents an example. The first column presents the typical sequence of instructions used by compilers to transfer the control 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 modified by attackers[6], 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. 9 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 [32]. The effectiveness of this optimization depends on how often switch statements are used in programs.

7. IMPLEMENTATION AND EVALUATION

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

7.1 Prototype implementation

Our implementation is built in LLVM 2.8 [20], 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

[5] 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.

[6] This assumption can be discharged by either putting jump tables into code region as in HyperSafe [32] or have read-only data write-protected through page protection.

Original code	CFI instrumentation	After optimization

ecx := [$JT + edx*4]
jmp ecx
	ecx := [$JT + edx*4]
	if [ecx+3]$\neq$$ID goto error
	ecx := ecx + 7
	jmp ecx
		if edx$>_u$15 goto error
		ecx := [$JT + edx*4]
		ecx := ecx + 7
		jmp ecx

Figure 9: 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.

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 analysis 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 analysis 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.

Linker scripts, loader, and libraries. LLVM generates object code with multiple sections (.bss, .data, .text, .rodata, and others). We developed linker scripts to link multiple sections into three sections (code, data and read-only data) at certain start addresses. We modified PittSFIeld's loader to set up code and data regions and load executable code. PittSFIeld's loader does not protect confidentiality and allows sandboxed code to read outside of the sandbox; we modified the loader so that it copies arguments into the sandbox. Furthermore, the three sections are locked down with the desired permissions with `mprotect`: the code section is readable and executable; the data section is readable and writable; the read-only data section is readable (it includes data such as jump tables and string literals). We also reused PittSFIeld's library wrappers and libraries, including its reimplementation of library functions for dynamic memory allocation (malloc, free, and so on).

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 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.[7] 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.

7.2 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* [2]. 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 implemen-

[7] 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
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.

	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.

tation 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 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.

Performance comparison with related systems. We next compare our system with PittSFIeld and XFI, two systems that adopt software-only techniques for protection. PittSFIeld reports an average of 21% for SPECint2000 for sandboxing both memory writes and jumps. Our system has a lower overhead (10.4% for CFI and write protection) and provides stronger control-flow integrity; it can additionally sandbox memory reads with acceptable overheads.

To compare with XFI, we have evaluated our implementation on the Independent JPEG Group's image-decoding reference implementation. The XFI paper reports both fastpath and slowpath overheads for the JPEG program; the XFI fastpath overhead is directly comparable with our implementation. The following table shows the performance overheads of our implementation compared to XFI's fastpath implementation, for images of different sizes. The columns of *DS-W.CFI* and *DS-RW.CFI* report the numbers of our system for write protection and read-write protection, respectively. The columns of *XFI-W* and *XFI-RW* report XFI's numbers for write protection and read-write protection, respectively. In both cases, our implementation reports lesser overheads. It seems that our optimizations are effective at bringing down the overheads. Note the comparison is preliminary as we have tested only on one program and LLVM is a different compiler from the one used in XFI.

Size	DS-W.CFI	DS-RW.CFI	XFI-W	XFI-RW
4k (%)	2.90	15.53	18	78
14k (%)	2.32	13.09	18	80
63k (%)	9.99	25.27	17	75
229k (%)	9.09	14.17	15	68

8. FUTURE WORK

We plan to implement more static-analysis based optimizations. First, more aggressive loop optimizations based on induction variable analysis should further bring down the data sandboxing overhead. Second, CFI can also benefit from static analysis—an ID check for a computed jump is unnecessary if the jump targets can be statically determined to obey the control-flow policy.

Our prototype implementation is built for x86-32 and we have not addressed the portability issue. We plan to port our implementation to newer architectures including x86-64 and ARM. These architectures should benefit more since they do not have the hardware segmentation support.

SFI is a special kind of Inlined Reference Monitors (IRM [12, 13]). IRMs can enforce fine-grained safety properties. Clearly, the methodology of combining CFI with static analysis to reduce the runtime overhead applies to general IRMs. For instance, fine-grained memory protection, which allows access control of multiple data regions of small sizes [4, 5, 8], can also benefit from CFI-enabled optimizations. Dynamic taint tracking is another example.

9. CONCLUSIONS

In this research, we have explored how CFI-enabled static analysis can help build efficient and validated system for data sandboxing, for the case of protecting both integrity and confidentiality. We believe the combination of CFI and static analysis provides a sweet point in design space for enforcing security policies on untrusted or buggy software: it provides strong security, enables sound optimization strategies, is thread safe, and can be easily integrated into the software tool chain. The combination can possibly serve as a foundation for improving efficiency of general inlined reference monitors for enforcing advanced security polices.

Acknowledgments

This research is supported in part by NSF grant CCF-0915157, CCF-0915030, a research grant from Google, and by AFOSR MURI grant FA9550-09-1-0539.

10. REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *12th CCS*, pages 340–353, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13:4:1–4:40, Nov. 2009.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[4] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *IEEE S&P*, pages 263–277, 2008.

[5] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *18th Usenix Security Symposium*, pages 51–66, 2009.

[6] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. Schuff, D. Sehr, C. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *PLDI*, pages 355–366, 2011.

[7] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *13th International Conference on Compiler Construction (CC)*, pages 5–23, 2004.

[8] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *SOSP*, pages 45–58, 2009.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *17th CCS*, pages 559–572, 2010.

[10] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, pages 339–354, 2008.

[11] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address spaces. In *OSDI*, pages 75–88, 2006.

[12] Ú. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 87–95. ACM Press, 1999.

[13] Ú. Erlingsson and F. Schneider. IRM enforcement of Java stack inspection. In *IEEE S&P*, pages 246–255, 2000.

[14] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306, 2008.

[15] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[16] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium*, 1996.

[17] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: a new approach to application security. In *ACM SIGOPS European Workshop*, pages 108–115, 2002.

[18] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th Usenix Security Symposium*, pages 191–206, 2002.

[19] P. Klinkoff, E. Kirda, C. Kruegel, and G. Vigna. Extending .NET security to unmanaged code. *Internation Journal of Information Security*, 6(6):417–428, 2007.

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

[21] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *15th Usenix Security Symposium*, 2006.

[22] M. Payer and T. R. Gross. Fine-grained user-space security through virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE)*, pages 157–168, 2011.

[23] N. Provos. Improving host security with system call policies. In *12th Usenix Security Symposium*, pages 257–272, 2003.

[24] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, pages 209–218, 2002.

[25] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *19th Usenix Security Symposium*, pages 1–12, 2010.

[26] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *14th CCS*, pages 552–561, 2007.

[27] J. Siefers, G. Tan, and G. Morrisett. Robusta: Taming the native beast of the JVM. In *17th CCS*, pages 201–211, 2010.

[28] C. Small. A tool for constructing safe extensible C++ systems. In *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 174–184, 1997.

[29] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper*, Hyderabad, India, Dec. 2008.

[30] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *OSDI*, pages 1–16, 2004.

[31] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *SOSP*, pages 203–216, New York, 1993. ACM Press.

[32] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE S&P*, pages 380–395, 2010.

[33] Z. Xu, B. Miller, and T. Reps. Safety checking of machine code. In *PLDI*, pages 70–82, 2000.

[34] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, May 2009.