Foundations and Trends® in Privacy and Secruity

Principles and Implementation Techniques of Software-Based Fault Isolation

Suggested Citation: Gang Tan (2017), "Principles and Implementation Techniques of Software-Based Fault Isolation", Foundations and Trends® in Privacy and Secruity: Vol. 1, No. 3, pp 137–198. DOI: 10.1561/3300000013.

Gang Tan
The Pennsylvania State University
gtan@cse.psu.edu

now
the essence of knowledge
Boston — Delft

Contents

Principles and Implementation Techniques of Software-Based Fault Isolation

Gang Tan[1]

[1] *Pennsylvania State University; gtan@cse.psu.edu*

ABSTRACT

When protecting a computer system, it is often necessary to isolate an untrusted component into a separate protection domain and provide only controlled interaction between the domain and the rest of the system. Software-based Fault Isolation (SFI) establishes a logical protection domain by inserting dynamic checks before memory and control-transfer instructions. Compared to other isolation mechanisms, it enjoys the benefits of high efficiency (with less than 5% performance overhead), being readily applicable to legacy native code, and not relying on special hardware or OS support. SFI has been successfully applied in many applications, including isolating OS kernel extensions, isolating plug-ins in browsers, and isolating native libraries in the Java Virtual Machine. In this survey article, we will discuss the SFI policy, its main implementation and optimization techniques, as well as an SFI formalization on an idealized assembly language.

Gang Tan (2017), "Principles and Implementation Techniques of Software-Based Fault Isolation", Foundations and Trends® in Privacy and Secruity: Vol. 1, No. 3, pp 137–198. DOI: 10.1561/3300000013.

1

Introduction

One fundamental idea in protecting a computer system is to have multiple *protection domains* in the system (Lampson, 1974). Each domain has its own capabilities, according to the domain's trustworthiness. Since the introduction of protection rings and virtual memory in Multics (Schroeder and Saltzer, 1972; Saltzer, 1974), all modern operating systems are structured to have an OS protection domain, also known as the kernel mode, and multiple user-application domains, which are processes in the user mode; the OS domain can execute privileged instructions, set up virtual memory protection, and perform access control on resources; a user-application domain has to go through the OS domain via the system-call interface to perform privileged operations. Domains can communicate by message passing or via shared objects. The boundaries between protection domains ensure that errors in one domain do not affect other domains.

It is natural to use protection domains to isolate untrusted components of a system. For instance, a web browser should isolate browser plug-ins so that their malfunctioning would not lead to the crash or leakage of sensitive information of the browser. In the same vein, an operating system should isolate device drivers, which are often devel-

oped by third-party vendors and have a higher bug rate than the OS kernel. In many such situations, it is highly desirable to isolate untrusted components in separate protection domains, grant them a minimum set of privileges, and allow only controlled interaction between them and privileged protection domains (Provos *et al.*, 2003; Brumley and Song, 2004).

Many mechanisms are possible for implementing protection domains. Table 1.1 provides a comparison among common mechanisms that can provide application-level protection domains. Hardware-based virtualization puts components into separate *virtual machines* and relies on virtual machine boundaries for fault toleration and resource control. Process-based separation puts components into separate *OS processes* and relies on hardware-backed virtual memory for isolating processes. In both hardware-based virtualization and process-based separation, user-level instructions run unmodified at native speed and they are fully transparent in that no special compiler is needed to recompile applications, nor do they require developers to port their code. However, their downside is the high-performance overhead for context switching between domains. For instance, a process context switch may require the flushing of the Translation Lookaside Buffer (TLB), which is the cache for the translation from virtual to physical addresses; it also brings adverse effect to data and instruction caches. A virtual machine context switch is even more costly as it involves the switching between two OSes. Therefore, when components are tightly coupled and require frequent domain crossings, separating them via virtual machines or processes is often not adopted due to the high cost of context switches.

Another approach is through *language-based isolation*, which relies on safe high-level languages for isolation. This approach fine-grained, portable, and flexible. The Java Virtual Machine (JVM) and the Common Language Runtime (CLR, *Common Language Infrastructure (CLI)* 2006) enforce type-based isolation through a combination of static and dynamic checks. Languages such as E (Miller, 2006) and Joe-E (Mettler *et al.*, 2010; Krishnamurthy *et al.*, 2010) enforce a stronger level of isolation than Java through an object-capability model. Their downside is an overall loss of performance caused by dynamic checks. Techniques

Table 1.1: Comparison of isolation mechanisms.

	Context-switch overhead	Per-instruction overhead	Compiler support	Software engineering effort
Virtual machines	very high	none	no	none
OS processes	high	none	no	none
Language-based isolation	low	medium (dynamic) or none (static)	yes	high
SFI	low	low	maybe	none or medium

using pure static types (e.g., Morrisett *et al.*, 1999) have no runtime overhead, but require nontrivial support from developers and compilers. One significant downside of language-based isolation is that a single language model has to be adopted, meaning that the software-engineering effort to rewrite legacy C/C++ code is significant.

Software-based Fault Isolation (SFI) is a software-instrumentation technique at the machine-code level for establishing logical protection domains within a process. The main idea is to designate a memory region for an untrusted component and instrument dangerous instructions in the component to constrain its memory access and control transfer behavior; it is sometimes referred to as code *sandboxing*. In SFI, protection domains stay within the same process, incurring low overhead when switching between domains. As a result, it is especially attractive in situations when domain crossings are frequent (e.g., the interaction between a browser and a plug-in, or the interaction between an OS and a device driver). As we will discuss, SFI can be implemented in many ways: in a machine-code interpreter, in a machine-code rewriter, or inside a compiler. When SFI is implemented in a machine-code interpreter or rewriter, applications can run without porting by developers. In

contrast, some porting effort may be required when SFI is implemented inside a compiler, as is the case with NaCl (Yee *et al.*, 2009).

First proposed by Wahbe *et al.* (1993), SFI has enjoyed many successes thanks to its runtime efficiency, strong guarantee, and ease of implementation. It has been implemented in several architectures, including MIPS (Wahbe *et al.*, 1993), x86-32 (Small, 1997; McCamant and Morrisett, 2006; Ford and Cox, 2008; Yee *et al.*, 2009; Zeng *et al.*, 2011; Zeng *et al.*, 2013), x86-64 (Sehr *et al.*, 2010; Deng *et al.*, 2015), and ARM (Sehr *et al.*, 2010; Zhao *et al.*, 2011; Zhou *et al.*, 2014). It has also been used in many applications, including isolating OS kernel modules (Small, 1997; Erlingsson *et al.*, 2006; Mao *et al.*, 2011; Castro *et al.*, 2009), isolating plug-ins in the Chrome browser (Yee *et al.*, 2009; Sehr *et al.*, 2010), and isolating native libraries in the Java Virtual Machine (Siefers *et al.*, 2010; Sun and Tan, 2012).

In this survey article on SFI, we will focus on the principles and common implementation techniques behind many SFI systems. Chapter 2 will give a concise definition of the SFI policy. The bulk of the article will be in chapter 3, which presents the implementation and optimization techniques that enforce the SFI policy. In chapter 4, we will formalize the main constraints enforced by SFI, through a formalization of an SFI verifier; a correctness proof of the verifier will also be discussed. We will briefly discuss future research directions in chapter 5 and cover stronger policies than fault isolation in chapter 6.

2

The SFI Policy

To sandbox a piece of code, SFI constructs a logical address space within a process's memory and constrains the code's data-access and control-flow behavior. The logical address space is sometimes referred to as the *fault domain* and is visualized in Figure 2.1. In particular,

- There is a data region where runtime data is stored. Assume the data region starts at address *DB* (abbreviation for Data Begin) and ends at address *DL* (Data Limit). The data region holds all data needed by the code, including stack data and heap data.
- There is a code region where code is loaded into. Assume the code region starts at address *CB* (Code Begin) and ends at address *CL* (Code Limit).
- There is a set of safe external code addresses in *SE* (Safe External) outside of the fault domain. Control transfers to *SE* addresses allow interaction between the sandboxed code and trusted system services implemented outside the sandbox; without allowing transfers to external addresses, the sandboxed code would be totally isolated from the rest of the system.[1] External code addresses can

[1] Interaction with the outside work can also be achieved through shared memory.

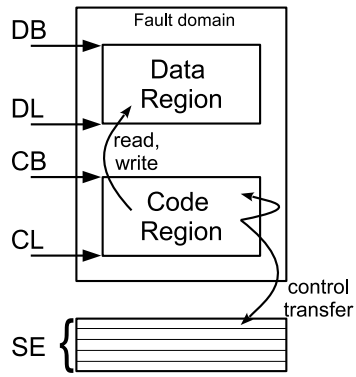

Figure 2.1: The SFI policy.

host trusted services that require higher privileges than what is given to the sandboxed code.

- The code region is disjoint from the data region, and the *SE* set is also disjoint from the addresses in code and data regions. In other words, the following three sets of addresses are mutually disjoint: $[CB, CL]$, $[DB, DL]$, and SE.

With the aforementioned set up, the SFI policy for a piece of untrusted code running inside an SFI sandbox is stated as follows.

Definition 2.1. The SFI policy has two components:

- *Data-access policy.* All memory reads and writes by the code should be within the data region; that is, the set of memory locations accessed by a read/write instruction should be within the range of $[DB, DL]$.
- *Control-flow policy.* A control-flow transfer by the code must stay within the code region or target a safe external address; that is, a control flow target must be within $[CB, CL] \cup SE$.

However, it is harder to enforce security on shared memory communication and it is rarely used in SFI implementation.

Note that SFI does not need to assume non-writable code and non-executable data as they are outcomes of enforcing the SFI policy. Since all memory writes stay within the data region, the code region cannot be modified; this prevents untrusted code from injecting new code into the code region during execution. Moreover, since control-flow transfers cannot target the data region, it is impossible to execute data in the data region as if it were code.

A practical SFI implementation may enforce a stronger policy for efficiency and for ease of implementation. For instance, an implementation (e.g., NaCl Yee *et al.*, 2009) may disallow a specific class of instructions (e.g., system calls) inside the code region to make it easier to enforce the SFI policy. As another example, on an architecture with an variable-sized instruction set (e.g. x86), an implementation typically further restricts the control flow to disallow jumping into the middle of instructions so that no instructions with overlapping memory addresses can be executed (McCamant and Morrisett, 2006); this restriction facilitates instrumentation and makes it easier to reason about code behavior.

3

SFI Enforcement

We call memory reads, memory writes, and control-transfer instructions *dangerous instructions* as they have the potential of violating the SFI policy. At a high level, an SFI enforcement mechanism checks every dangerous instruction to ensure their safety. When enforcing SFI on binary code, there are in general two main strategies:

- *Dynamic binary translation.* This approach is depicted in Fig. 3.1(a). In essence, it uses an efficient interpreter to interpret instructions in the target program, and for each instruction the interpreter checks that it is safe according to some policy before the instruction is executed. A typical example is program shepherding (Kiriansky *et al.*, 2002), which reuses the highly efficient dynamic optimization framework DynamoRio to enforce a limited form of control-flow integrity; the same approach can be used to enforce other policies including the SFI policy. Other similar systems include software dynamic translation (Scott and Davidson, 2002) and libdetox (Payer and Gross, 2011). VX32 (Ford and Cox, 2008) also relies on dynamic binary translation to confine indirect branches and other dangerous instructions.

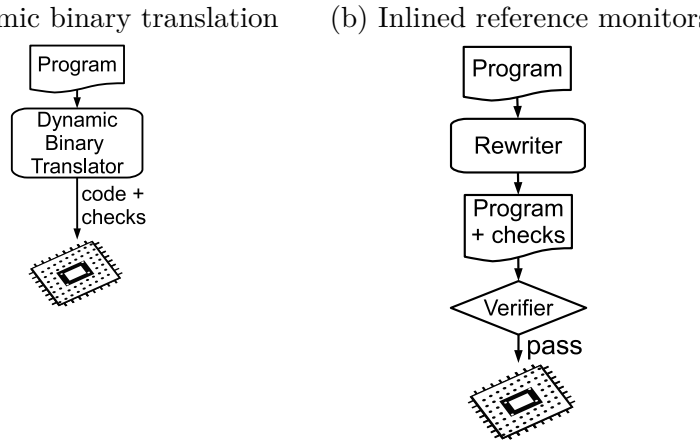

Figure 3.1: Binary-level policy enforcement: dynamic binary translation versus inlined reference monitors.

- *Inlined reference monitors (IRMs).* As visualized in Fig. 3.1(b), this approach requires a static program rewriter, which transforms the input program and outputs a program with checks inlined. When the instrumented program executes, checks before dangerous instructions prevent policy violations.

Enforcement through an IRM is the main SFI implementation strategy (e.g., Wahbe *et al.*, 1993; Small, 1997; McCamant and Morrisett, 2006; Yee *et al.*, 2009; Sehr *et al.*, 2010; Zeng *et al.*, 2011; Zeng *et al.*, 2013) since it offers several benefits. First, it results in a more efficient implementation than dynamic binary translation, which comes with the additional cost of translation and analysis during runtime. Furthermore, for efficiency an IRM can employ sophisticated static analysis to optimize checks (see Sec. 3.1.4), while dynamic binary translation cannot afford expensive analysis at runtime. Another benefit of IRMs is that an additional verifier can be added to remove the rewriter from the Trusted Computing Base (TCB) and to perform some checks statically when possible. The verifier can check the output of the rewriter to ensure sufficient checks are in the code for the policy. In contrast, existing dynamic binary translation systems put the whole dynamic optimization and monitoring engines in the TCB.

Therefore, the rest of this article will focus on techniques for the IRM enforcement of the SFI policy. At the same time, certain techniques are also applicable to the dynamic-binary-instrumentation enforcement; we will comment on the applicability when appropriate.

Implementing IRM rewriters. The rewriter in the IRM approach can be implemented either directly via binary rewriting or inside a compiler. The binary-rewriting approach takes a piece of binary code, performs disassembly, inserts checks, and assembles the instrumented code. The binary-rewriting approach does not require source code and can rewrite even legacy binaries. On the other hand, disassembling binaries statically without meta information including symbol tables can still be challenging for many binaries, including obfuscated binaries. Furthermore, optimizing checks in binaries directly is challenging—it is difficult to perform basic static analysis such as alias analysis due to a lack of structured information such as types in binaries.

The rewriter can also be implemented inside a compiler. This approach requires access to source code. When source code is translated to binary code, the compiler inlines necessary checks into the binary code. The approach of compiler-based rewriting has access to more information about the code (e.g., information about types and loops) and can perform more precise static analysis for optimizing checks, resulting in more efficient enforcement. Moreover, with some effort, the compiler-based rewriting can also be made portable across architectures (Zeng *et al.*, 2013; Donovan *et al.*, 2010; Kroll *et al.*, 2014), while a binary-rewriting framework is specific to some architecture.

The syntax of an assembly language. We will use assembly-code examples to illustrate how the SFI policy is enforced via IRM and what optimizations can be performed for efficient enforcement. To have a uniform syntax for these examples, in Fig. 3.2 we introduce an idealized assembly language. In this language, we use w for a constant word of some fixed size; without loss of generality, we will use 32-bit words in examples. Furthermore, we use r for a register, *op* for an operand (which is either a register or a word), *aop* for a binary arithmetic-logic operator, and *cop* for a binary comparison operator. Operator $\gg$ is

$$
\begin{array}{lrcl}
(\textit{Instr}) & i & ::= & r_d := r_s \ \textit{aop}\ \textit{op} \\
 & & | & r_d := \mathrm{mem}(r_s + w) \ \mid \ \mathrm{mem}(r_d + w) := r_s \\
 & & | & \text{if } (r_s \ \textit{cop}\ \textit{op}) \text{ goto } w \ \mid \ \text{jmp } \textit{op} \\
(\textit{Register}) & r & ::= & \text{r0} \mid \text{r1} \mid \text{r2} \mid \ldots \\
(\textit{Operand}) & \textit{op} & ::= & r \mid w \\
(\textit{ALOp}) & \textit{aop} & ::= & + \mid - \mid \gg \mid \ll \mid \& \mid \text{`}\mid\text{'} \mid \ldots \\
(\textit{CompOp}) & \textit{cop} & ::= & > \mid < \mid \leq \mid \geq \mid = \mid \neq \mid \ldots
\end{array}
$$

Figure 3.2: Syntax of an assembly language.

for logical right shift, $\ll$ for logical left shift, & for the bitwise-and operation, and "|" for the bitwise-or operation.

Instructions in Fig. 3.2 are typical in an assembly language: an arithmetic-logic instruction "$r_d := r_s \ \textit{aop}\ \textit{op}$" performs operation *aop* on r_s and *op* and then stores the result in r_d; a memory-load instruction "$r_d := \mathrm{mem}(r_s + w)$" loads from memory at address $r_s + w$ to register r_d; a memory-store instruction "$\mathrm{mem}(r_d + w) := r_s$" stores the value in r_s into memory at address $r_d + w$; a conditional jump instruction "if $(r_s \ \textit{cop}\ \textit{op})$ goto w" transfers the control to w if the comparison between r_s and *op* using *cop* returns true and otherwise moves the control to the address following the instruction; instruction "jmp *op*" transfers the control to the value in operand *op*.

We next introduce some abbreviations and terminology. We will write $r := r'$ for $r := r' + 0$. In memory load and store instructions, we will use $\mathrm{mem}(r)$ as an abbreviation for $\mathrm{mem}(r + 0)$. Instruction "jmp w" will be called a *direct jump* since its jump target can be statically determined; in contrast, "jmp r" is called an *indirect jump* since its jump target is in a register and cannot always be determined statically.

For simplicity of discussion, the idealized assembly language differs from more realistic assembly languages in several ways that may affect the enforcement of SFI.

First, the idealized language does not include (push/pop) instructions that manipulate the stack. They can easily be included, after adding some special stack registers. On the other hand, the SFI policy

does not differentiate the stack memory from the heap memory. Both are in the data region and the SFI enforcement works the same on stack or heap accesses.[1]

Second, the idealized assembly language includes only indirect jumps through registers. Realistic assembly languages have general *indirect branches*, which refer to branches that use computed addresses. An indirect branch can be an indirect jump (a jump via a register or memory operand), an indirect call (a call via a register or memory operand), or a return instruction. An indirect branch via a memory operand can be translated into a memory-read instruction that reads the destination address into a register, followed by an indirect branch via the register. An indirect call via a register can be translated to a push instruction that moves the return address to the stack, followed by an indirect jump via the register. A return instruction can be translated to a pop instruction that moves the return address to a register, followed by an indirect jump via the register.

Finally, the idealized assembly language assumes a word-addressed memory and a memory operation accesses only a single memory location. A realistic assembly language, in contrast, is based on a byte-addressed memory and a memory instruction can access several consecutive memory locations; e.g., an x86-32 memory instruction can access four consecutive memory addresses to read/write 32-bit words. For enforcing the SFI policy, all memory accesses within a single instruction must stay in the data region. This can be achieved by either requiring additional alignment requirements on data addresses or setting up guard zones around the data region. In the rest of the article, this issue will be ignored and a memory operation is assumed to access only one memory location.

3.1 Enforcing the data-access policy

For SFI's data-access policy, a naive enforcement can insert address checks before memory instructions to ensure that they access memory

[1] We note that there are SFI optimizations that take advantage of usage patterns of stack accesses; see Sec. 3.1.5. Also, there are stronger policies that target the safety of the stack (Erlingsson *et al.*, 2006; Kuznetsov *et al.*, 2014).

within the data region. As an example, instruction "mem(r1 + 12) := r2" is instrumented into

$$\begin{aligned}&\text{r10} := \text{r1} + 12\\&\text{if } (\text{r10} < DB) \text{ goto error}\\&\text{if } (\text{r10} > DL) \text{ goto error}\\&\text{mem(r10)} := \text{r2}\end{aligned}$$

The above sequence uses r10 as a scratch register to store the address of r1 + 12 and checks that the address falls within $[DB, DL]$. If the checks fail, the code jumps to a special error label where an appropriate response can be issued (e.g., terminating the application).

The naive implementation enforces the data-access policy, but it incurs a high runtime overhead because two checks are inserted before every memory access.[2] A practical SFI implementation has to implement optimizations to drive down the overhead. In the rest of this section, we discuss a set of optimizations used in SFI implementations and illustrate them using examples.

We will start with common optimizations that are architecture independent (from Sec. 3.1.1 to 3.1.6); these optimizations can be freely combined in a specific SFI implementation. In Sec. 3.1.7, we discuss optimizations that are architecture dependent. In Sec. 3.1.8, we comment on how the presented optimizations can be used in SFI enforcement by dynamic binary instrumentation.

3.1.1 Data region specialization

The first common optimization is to use special bit patterns for addresses in the data region so that address checks can be performed more efficiently. One idea that was proposed in the first SFI system (Wahbe *et al.*, 1993) and was adopted in all subsequent SFI systems was to have the upper bits the same for all addresses in the data region. These upper bits are called the *data region ID*. As an example, let DB be `0x12340000`

[2]The need for optimizations depends on the frequency of memory-access instructions in an application. Jaleel (2010) performed measurements on SPEC2000 and SPEC2006 benchmarks and found that around 50% of instructions executed dynamically on x86 performed some memory access and around 10% performed memory writes.

and let *DL* be `0x1234FFFF`; the data region ID is then `0x1234`, since all addresses in the data region have `0x1234` in their upper 16 bits. With this specialization on the data region, the address check before a memory operation is simplified to check that the memory address has the correct data region ID. A memory store "mem(r1 + 12) := r2" is then instrumented as

$$
\begin{aligned}
&\text{r10} := \text{r1} + 12\\
&\text{r11} := \text{r10} \gg 16\\
&\text{if } (\text{r11} \neq \texttt{0x1234}) \text{ goto error}\\
&\text{mem(r10)} := \text{r2}
\end{aligned}
$$

The above sequence uses a right-shift instruction "r11 := $r10 \gg 16$" to get the data region ID of the address in r10 and checks if it is the expected ID. On architectures such as x86, a right-shift instruction is more efficient than a conditional jump and therefore the above sequence is more efficient than the one in the naive implementation.[3]

3.1.2 Integrity-only isolation

An observation about a typical program is that it performs many more memory-read operations than memory-write operations. Therefore, one way that can significantly reduce the overhead is to weaken SFI's data-access policy by checking only memory writes but not memory reads. This guarantees the integrity of the memory outside the data region, but ignores confidentiality. The integrity guarantee can be sufficient for some applications. For instance, a dynamic taint-tracking system stores taints of the program being tracked in a memory region; the program's memory writes can then be instrumented to ensure the integrity of the region that stores the taints; confidentiality of taints is not a concern.

Wahbe *et al.* (1993) reported that on a 1993 RISC machine the performance overhead is 22% when checking both memory reads and writes on typical C benchmarks and the overhead is reduced to 4% when checking only memory writes. PittSFIeld (McCamant and Morrisett, 2006), an SFI system implemented on x86, reported 21% average

[3]Note that the relative efficiency of a shift operation can be dependent on CPU architectures and workload.

overhead on SPECint2000 benchmarks when checking both reads and writes and 13% overhead when checking only writes. Because of the significant performance benefit, many SFI systems sandbox only memory writes, including NaCl's implementation on x86-64 and ARM (Sehr *et al.*, 2010).

3.1.3 Address masking

Since SFI is about isolating faults, for data access it is sufficient to guarantee that the destination address of a memory operation always falls within the data region of the fault domain. Therefore, a memory operation using an out-of-data-region memory address would be safe if we can change the address to be inside the data region. With the data-region specialization discussed before (Sec. 3.1.1), forcing an address to be inside the data region can be achieved by changing the upper bits of the address to be the data region ID. This technique is called *address masking*, which is more efficient than address checking. As an example, let *DB* be `0x12340000` and let *DL* be `0x1234FFFF`, then instruction "mem(r1 + 12) := r2" can be instrumented as

$$
\begin{aligned}
&\text{r10} := \text{r1} + 12\\
&\text{r10} := \text{r10} \,\&\, \texttt{0x0000FFFF}\\
&\text{r10} := \text{r10} \mid \texttt{0x12340000}\\
&\text{mem(r10)} := \text{r2}
\end{aligned}
$$

The above sequence first uses a bitwise-and instruction to clear the upper 16 bits of the destination address in r10 and then uses a bitwise-or instruction to set the desired data region ID.

Address masking is more efficient than address checking since masking can be implemented by speedy bitwise operations and does not require conditional tests. PittSFIeld (McCamant and Morrisett, 2006) reported 12% performance gain of address masking compared to address checking on SPECint2000 benchmarks. Therefore, most SFI implementations adopt address masking for efficiency.

Before proceeding, it is worthwhile to mention two observations about address masking. First, it changes program semantics because it modifies an out-of-data-region address to an in-data-region one.

This is not viewed as a downside, however, because address masking would not change a "good" program's behavior since a "good" program should not access memory addresses outside the data region. Second, when there is a policy-violating memory operation, address masking does not stop and report the violating instruction. This is a downside of address masking because the information about which instruction caused the violation can be useful for both debugging and blame assignment.

Address masking enables more optimizations, which can further improve efficiency. One technique implemented in PittSFIeld is to fix the data region ID to have only one single bit on and make the zero-ID region unmapped in the virtual address space. By having the zero-ID region unmapped, a memory access is considered safe if it is either in the data region or in the zero-ID region, because an access to the zero-ID region generates a trap and is therefore considered safe. This set up can cut down the number of instructions for address masking. As an example, let *DB* be `0x20000000` and *DL* be `0x2000FFFF`; the data-region ID is `0x2000`, which has only the third most significant bit on. With this special data region, instruction "mem(r1 + 12) := r2" can be instrumented as

$$\begin{aligned} &\text{r10} := \text{r1} + 12 \\ &\text{r10} := \text{r10} \;\&\; \texttt{0x2000FFFF} \\ &\text{mem(r10)} := \text{r2} \end{aligned}$$

In the above sequence, if the value in r10 before masking has the third most significant bit on, the value in r10 after masking must have data region ID `0x2000`, meaning that it is an address within the data region. On the other hand, if the value in r10 before masking has the third most significant bit off, the value in r10 after masking must have the zero region ID and the following memory access via r10 generates a trap since the zero-ID region is unmapped; in this case, the memory access via r10 essentially serves as a check.

One downside of the one-instruction address-masking technique is that it limits the number of sandboxes a system can have; In a 32-bit system, if a sandbox's data region size is 2^n bytes, then we can have at most $32 - n$ sandboxes.

Henceforth, we use the term *data guards* to refer to the operations of either data-address checking or masking, when which one is used is irrelevant for the discussion. To hide how data guards are implemented, we introduce an instruction "$r' :=$ dGuard(r)", which takes an address in r and performs either checking or masking to result in a sandboxed address in r'. An implementation of data guards should have the following properties:

- When r holds an address in the data region, then "$r' :=$ dGuard(r)" succeeds and r' equals r.
- When r holds an address outside the data region, an error state is generated (for address checking) or r' gets an address within a safe range (for address masking). The safe range is typically the same as the address range of the data region, but it can be implementation specific. For PittSFIeld's one-instruction masking technique, it is the data region plus the zero-ID region.

3.1.4 Guard zones

One technique that was first described by Wahbe *et al.* (1993) is to place a *guard zone* of size *GSize* directly before and after the data region, as shown in Fig. 3.3. It is further assumed that guard zones are unmapped so that memory accesses to guard zones generate hardware traps. In this setup, a memory access is considered safe if the address falls within the range of $[DB - GSize, DL + GSize]$.

Zeng *et al.* (2011) implemented a series of optimizations that are enabled by guard zones and static analysis. The first one is called *in-place sandboxing*, which is applicable to memory operations that use a commonly used addressing mode for accessing memory: a base register plus a small constant displacement. For instance, this mode is used to access fields of a C-like struct; the base register holds the base address of the struct and the displacement holds the offset for a field. When a memory address of this mode is used, we can perform the optimization that sandboxes just the base register. For example,

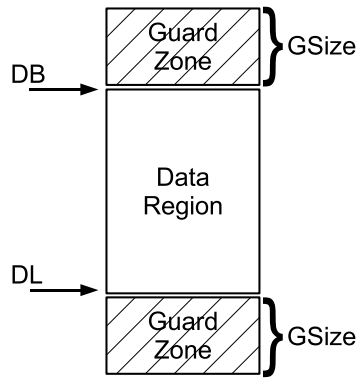

Figure 3.3: Data region surrounded by guard zones (from Zeng *et al.*, 2011).

"mem(r1 + 12) := r2" is instrumented as

$$\begin{aligned} &\text{r1} := \text{dGuard(r1)} \\ &\text{mem(r1} + 12) := \text{r2} \end{aligned}$$

To see why the above instrumentation is safe, let us assume that the implementation of "r1 := dGuard(r1)" constrains r1 to be within the data region.[4] Consequently, r1 + 12 must be within the data region plus the guard zones (assuming $GSize \geq 12$) and is therefore safe; the following memory operation through r1 + 12 either accesses the data region or generates a trap. The in-place sandboxing optimization sandboxes the base register directly and avoids the use of an extra scratch register; in contrast, the instrumentation sequences in Sec. 3.1.3 requires a scratch register to hold the value of r1 + 12. Additionally, it has the benefit of making it convenient to remove redundant guards, which will be discussed later.

A similar optimization is performed in NaCl-x86-64 (Sehr *et al.*, 2010), which takes advantage of the large address space of 64-bit machines and sets up large guard zones of size 40GB above and below its 4GB sandbox. As a result, an address constructed by the most

[4] If the one-instruction masking technique of PittSFIeld (see Sec. 3.1.3) is used, then r1 := dGuard(r1) puts r1 in the data region plus the zero-ID region; for security, guard zones should be added around the zero-ID region and should be left unmapped as well.

sophisticated x86 addressing mode, in the form of a base register plus a scaled indexed register and plus a displacement, can be guaranteed to stay in $[DB - GSize, DL + GSize]$ as long as the base register, the indexed register, and the displacement are carefully controlled.

Another optimization enabled by guard zones was described by Zeng *et al.* (2011). A natural strategy to remove unnecessary guard operations is to perform static analysis to determine if the possible range of an address used in a memory access is within the safe range $[DB - GSize, DL + GSize]$; if so, it is unnecessary to have a guard before the memory access. Zeng *et al.* (2011) proposed to implement *range analysis* to identify such optimization opportunities. At a program point, range analysis can determine the ranges of values in storage locations such as registers.

With the result of range analysis, we can perform redundant check elimination and loop check hoisting. We next briefly discuss these two optimizations using examples, but leave the details to Zeng *et al.*, 2011.

Redundant check elimination. In the following example, assume r1 holds the base address of a C struct. It loads two fields from the struct through two memory reads:

$$\begin{array}{l} \text{r2} := \text{mem}(\text{r1} + 4) \\ \ldots \text{// assume r1 is not changed in between} \\ \text{r3} := \text{mem}(\text{r1} + 8) \end{array}$$

For safety, one straightforward instrumentation adds a guard operation before each memory read, as shown next:

$$\begin{array}{ll} \text{r1} := \text{dGuard}(\text{r1}) & \text{// r1} \in [DB, DL] \\ \text{r2} := \text{mem}(\text{r1} + 4) & \text{// r1} \in [DB, DL] \\ \ldots & \text{// r1} \in [DB, DL] \\ \text{r1} := \text{dGuard}(\text{r1}) & \text{// r1} \in [DB, DL] \\ \text{r3} := \text{mem}(\text{r1} + 8) & \text{// r1} \in [DB, DL] \end{array}$$

The above program also shows the range of r1 after each instruction, determined by range analysis. It is easy to tell that the second guard can be removed because the range of r1 is already in the data region before the operation.

(a) Before hoisting

```
    r3 := r1
    r4 := r2 * 4
    r4 := r1 + r4
    r5 := 0
loop:
    if (r3 ≥ r4) goto end
    r3 := dGuard(r3)
    r6 := mem(r3)
    r5 := r5 + r6
    r3 := r3 + 4
    jmp loop
end:
```

(b) After hoisting

```
    r3 := r1
    r4 := r2 * 4
    r4 := r1 + r4
    r5 := 0
    r3 := dGuard(r3)
loop:
    if (r3 ≥ r4) goto end
    r6 := mem(r3)
    r5 := r5 + r6
    r3 := r3 + 4
    jmp loop
end:
```

Figure 3.4: An example demonstrating loop check hoisting (adapted from Zeng *et al.*, 2011).

Loop check hoisting. In this optimization, a guard within a loop is hoisted outside of the loop so that the guard is executed only once per loop instead of once per loop iteration. The key observation is, if we know that a memory address is within the safe range $[DB - GSize, DL + GSize]$, then a successful (untrapped) memory access through the address narrows down the range of the address to $[DB, DL]$ since accesses to guard zones are trapped; therefore, a successful memory access effectively serves as a check.

Fig. 3.4 presents an example program showing how range analysis enables hoisting checks outside of loops. The program calculates the sum of an integer array. In particular, each integer occupies four bytes, r1 holds the initial address of the array, r2 holds the length of the array, and r3 holds a pointer value that iterates from the beginning of the array to the end. Without optimization, r3 needs to be guarded within the loop body. The guard operation (underlined) can actually be moved outside of the loop, avoiding the per-iteration execution. The optimized code is shown in Fig. 3.4 (b).

It is instructive to understand why the code in Fig. 3.4 (b) is safe even though it sandboxes only the beginning address of the array and there is no restriction on the array length. To validate its safety, it is sufficient to show that r3 $\in [DB, DL + 4]$ is a loop invariant since it implies the memory read via r3 is within the safe range (assuming $GSize \geq 4$). The condition is clearly true at the beginning of the loop since the guard instruction gives r3 $\in [DB, DL]$. 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 r3 $\in [DB, DL]$ holds after "r6 := mem(r3)"—a trap would be generated if r3 were in guard zones. With that result, the following add-by-four operation re-establishes the loop invariant.

3.1.5 Guarding changes instead of uses

Some registers are used often in memory operations but changed rarely. One example is the ebp register in x86-32 (or rbp in x86-64); it points to the base address of the current call frame. It is usually initialized in the prologue of a function to save the previous value of esp and not changed again in the body of the function. ebp is then used often in the function body to access arguments passed from the caller of the function. Therefore, we can insert a guard after a change to ebp to establish the invariant that ebp falls into the data region afterwards, instead of inserting a guard before each of its uses. ebp := esp in the function prologue becomes

$$\begin{aligned} &\text{ebp} := \text{esp} \\ &\text{ebp} := \text{dGuard(ebp)} \end{aligned}$$

This optimization is typically used together with guard zones. After ebp has been sandboxed, later memory accesses with address ebp + c, where c is a constant, is guaranteed to be safe as long as $GSize \geq c$.

The soundness of this optimization requires that a guard is inserted after every change to ebp. It may be necessary to insert a guard after a function call since the callee function may not maintain the invariant on ebp (for example, optimized code may use ebp as a general-purpose register).

3.1.6 Finding scratch registers

When performing SFI rewriting at the binary level (after disassembly), one issue is to find scratch registers for storing intermediate results. It is a common issue in binary-level IRM rewriting. For instance, r10 is used as a scratch register in many of our previous examples; those examples assume r10's old value is no longer needed. However, if that assumption does not hold, r10's old value must be saved (on the stack typically) before it is used as a scratch register.

Saving and restoring scratch registers is clearly costly. One simple approach relies on register liveness analysis (Zeng *et al.*, 2011). Liveness analysis is a classic compiler analysis. At each program point, it 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 registers are available, we can resort to the way of saving and restoring scratch registers.

When the SFI rewriter is implemented inside a compiler, the problem of finding scratch registers goes away. One simple approach is to reserve dedicated scratch registers inside the compiler. Dedicated scratch registers are used only in monitor code but not by the regular application code. As an example, the SFI toolchain of PittSFIeld (McCamant and Morrisett, 2006) reserves ebx as the scratch register by passing a special option to the GCC compiler. Since this approach reduces the number of registers available to applications, it has the downside of increasing the register pressure, especially on machines with few general-purpose registers. Another approach is to perform rewriting at the level of a compiler IR (e.g., the LLVM IR) that has an unlimited number of variables. Such an IR is before register allocation. The rewriting can introduce new variables for storing intermediate results and register allocation automatically maps variables to either registers or stack locations.

3.1.7 Architecture-specific optimizations

Since the SFI policy is expressed at the low level (in terms of memory reads/writes and control transfers), most implementations are tied

to specific architectures. Therefore, an implementation often takes advantage of specific hardware features provided by the underlying architecture for efficiency.

On x86-32, NaCl (Yee *et al.*, 2009) and VX32 (Ford and Cox, 2008) use the segmentation support for efficient sandboxing of memory operations. A data segment is set up and the segment's base gets the value of *DB* and the segment's limit gets *DL*. A memory access is automatically checked by hardware to be within the data segment. The segmentation-based SFI is extremely efficient. However, the 64-bit mode of x86-64 does not support the use of segmentation to limit memory accesses and neither does ARM.

If only one sandbox is needed, one very efficient way of performing address masking on x86-64 is to put the sandbox at the first 4GB of virtual memory and then prefix every memory-access instruction with an address-override prefix (Deng *et al.*, 2015). The execution of a memory-access instruction with the address-override prefix automatically clears the first 32 bits of its memory address and effectively masks an address to the first 4GB.

ARMlock (Zhou *et al.*, 2014) relies on the memory-domain feature available on certain ARM processors. Memory for a sandbox is associated with a memory domain and memory domain accesses are controlled by the Domain Access Control Register (DACR). By setting permission bits in DACR, the sandboxed code is restrained to its own memory. We note that memory domains are not universally supported by all ARM processors. Furthermore, ARMlock is implemented as an OS-kernel extension since changing DACR requires kernel privileges. In contrast, all other techniques introduced in this chapter can be implemented by user-space code.

3.1.8 Applicability in dynamic binary translation

Previous optimizations for enforcement of SFI's data-access policy were discussed with respect to the IRM approach. A dynamic binary translator based implementation such as libdetox (Payer and Gross, 2011) enforces SFI by dynamically translating one basic block at a time into an instrumented form and stores instrumented basic blocks into a

code cache to avoid redundant translation. The translation can adopt many of our previously discussed optimization techniques: the address range of the data region can be specialized to make address checking efficient; checking can be performed only on writes for integrity-only protection; address masking instead of checking can be performed to speed up enforcement; guard zones can be set up for efficient address checking/masking; static analysis can be performed on basic blocks to identify scratch registers. The DBT approach, however, cannot afford sophisticated static analysis for check optimization such as loop check hoisting (see Sec. 3.1.4) because it would slow down the translation process; as a result, DBT typically performs block-local analysis.[5]

3.2 Enforcing the control-flow policy

SFI's restriction on control flow requires that control transfers by the sandboxed code stay in the code region or target a safe external address. On the surface, it appears only slightly different from SFI's restriction on data access. However, when using the IRM approach for SFI enforcement, restriction on control flow has to be more stringent so that inlined guards cannot be bypassed. This can be illustrated through an example. In the following code, a data guard is inserted before a memory store:

$$\begin{aligned} l1: &\quad \text{r10} := \text{r1} + 12 \\ l2: &\quad \text{r10} := \text{dGuard(r10)} \\ l3: &\quad \text{mem(r10)} := \text{r2} \end{aligned}$$

Now imagine somewhere else in the code there is an indirect jump "jmp r" and the only restriction is that $r \in [CB, CL] \cup SE$. The danger is that a bug in the code may corrupt r's value. The corruption may cause r to have address $l3$, the address of the memory-store instruction; as a result, the execution of "jmp r" would bypass the guard and the unrestricted store could then access memory outside the data region, violating the data-access policy.

Wahbe *et al.* (1993) used dedicated registers to avoid the problem. One dedicated register r_d is used to hold the sandboxed data address and

[5] High-performance dynamic instrumentation frameworks such as DynamoRIO (Bruening *et al.*, 2012) heavily rely upon trace-based optimizations; a possible piece of future work is to perform trace-based SFI-check optimizations.

all memory operations access memory through this dedicated register. For example, a memory-read operation through r1 is transformed to "r_d := dGuard(r1)", followed by a memory-read operation through r_d. (Similarly, a dedicated register r_c is used to hold the sandboxed code address and all indirect branches use it as the target address.) The dedicated registers are modified only in monitor code, not in the regular application code. Furthermore, monitor code is designed to maintain the invariant that, before any branch instruction, r_d always holds an address within the data region. Thanks to the invariant maintained on r_d, even if some data corruption causes a branch to target a memory operation directly and bypass its guard, the memory operation will not access memory outside the data region. Note that, when a guard is bypassed, the address used in the following memory operation would not be the address intended by the programmer (even though it is guaranteed to stay in the data region).

A more direct approach to avoiding bypassing guards is to strengthen the control-flow policy so that no jumps can target the middle between a guard and the following dangerous instruction; that is, they should be executed as an "atomic" block. This approach does not need dedicated registers. Strengthening the control-flow policy can also be used to solve the problem posed by CISC machines' variable-sized instructions, which cannot be addressed by the approach of dedicated registers. Because the encoding of an instruction on CISC machines such as x86 can take multiple bytes, it is possible that a branch instruction can target an address in the middle of an instruction and start executing a different instruction stream. This behavior is ruled out by all SFI enforcement mechanisms, for ease of implementation and verification.

We define a *pseudoinstruction* to be either a non-dangerous instruction or a guard followed by a dangerous instruction. The strengthened control-flow policy is stated as follows:

Definition 3.1 (Strengthened control-flow policy)**.** All control-flow transfers must target the beginning of a pseudoinstruction in the code region or target a safe external address.

3.2.1 Indirect-jump control-flow enforcement

By definition, the strengthened policy does not allow a jump to target the middle of an instruction or the middle between a guard and a dangerous instruction. This policy can be checked statically for conditional jumps and direct jumps since their targets are statically known. For indirect jumps, there are several alternative ways of enforcing the strengthened control-flow policy: aligned-chunk enforcement, bitmap based enforcement, and enforcement by fine-grained control-flow integrity. An SFI system can adopt one of them for restricting indirect jumps.

Aligned-chunk enforcement. PittSFIeld (McCamant and Morrisett, 2006) proposed a chunk-based approach to enforcing the strengthened control-flow policy on indirect jumps; this idea was later adopted by NaCl (Yee *et al.*, 2009; Sehr *et al.*, 2010). In this approach,

- The code region is divided into chunks of fixed sizes such as 16 bytes or 32 bytes and the beginning addresses of chunks are aligned (a is aligned if $a \bmod sz = 0$, where sz is the chunk size).
- The SFI rewriter rewrites the code so that a guard and its following dangerous instruction stay within the same chunk. Moreover, all jump targets are at aligned addresses, the beginning of a chunk always starts an instruction, and no instructions cross the boundary of a chunk. The SFI rewriter achieves these requirements by inserting no-op instructions at appropriate places. Fig. 3.5 presents an illustration of the instruction alignment.
- Indirect jump targets are masked so that only aligned addresses in the code region can be targets.

In this scheme, since a guard and its guarded instruction are in one chunk and jumps target only the beginnings of chunks, the guard cannot be bypassed by jumps.

As an example about how indirect jumps can be masked, let us assume the code region is `[0x10000000,0x1000FFFF]`, the chunk size

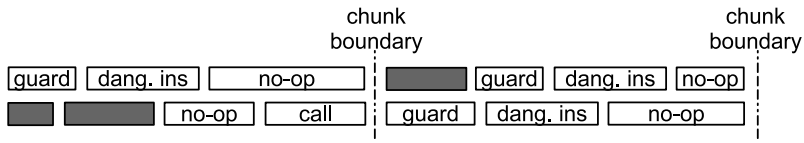

Figure 3.5: Illustration of the aligned-chunk enforcement. Black-filled rectangles represent regular (non-dangerous) instructions. Rectangles with "dang. ins" represent dangerous instructions, which are preceded by guards. For alignment, no-op instructions have to be inserted. Furthermore, call instructions are placed at the end of chunks since return addresses must be aligned.

is 16 bytes, and the zero-ID region (`[0x00000000,0x0000FFFF]`) is unmapped. In PittSFIeld, a single `and` instruction can ensure that the destination address in an indirect jump targets an aligned code address:

$$r := r \;\&\; \texttt{0x1000FFF0}$$
$$\text{jmp } r$$

The bitwise-and instruction clears the last four bits of r so that it becomes an aligned address. Furthermore, the result in r after masking has either ID `0x1000` or ID zero. Since the zero-ID region is unmapped, the following jump instruction succeeds only if r's ID is `0x1000`.

The instrumentation shows that aligned-chunk enforcement can be implemented efficiently. On the other hand, it requires extra no-op instructions be inserted so that jump targets are aligned at the beginnings of chunks. For instance, the address after a function call must be an aligned address since it is the target of a return instruction. If the address is not aligned, no-ops are inserted before the function call to make the return address aligned.

The extra no-ops slow down program execution because they are executed during runtime and they increase instruction cache pressure. PittSFIeld (McCamant and Morrisett, 2006) reports that inserted no-ops account for about half of its runtime overhead for enforcing data integrity. NaCl-JIT (Ansel *et al.*, 2011), a NaCl version that supports just-in-time compilation, reports no-ops account for half of the sandboxing cost (about 37% slowdown because of no-ops on x86-64). Also, inserted no-ops cause significant code-size increase. For instance, in NaCl-x86-64 (Sehr *et al.*, 2010), the code size after rewriting is roughly 60% larger than the size of the original code.

Bitmap based enforcement. MIP (Niu and Tan, 2013) generalizes the idea of fixed-size chunks and arranges instructions in *variable-size chunks*:

- Each chunk contains one instruction or a sequence of instructions and there are no jump targets in the middle of the chunk.
- Chunk beginnings are remembered through an immutable bitmap: if a code address is a chunk beginning, then its corresponding bit in the bitmap is one, and otherwise zero.
- Inter-chunk control flow with direct or indirect branches is restricted to target only chunk beginnings. Before an indirect branch, an inlined check consults with the bitmap and fails if the bit corresponding to the destination address is zero.

Compared to aligned-chunk enforcement, MIP is more flexible in that chunks have variable sizes; as a result, there is no need to insert no-ops for alignment. Its bitmap representation enables efficient checks before indirect branches. Niu and Tan (2013) presented two instruction sequences for checking. One sequence uses a bit-test instruction that is available on x86:

```
BT r, BMBase
JNC error
```

"BT r, `BMBase`" selects the bit in a bit string in memory and stores the value of the bit in the CF flag; the bit string starts at `BMBase` and the selected bit's position is the value in r. The following conditional jump instruction jumps to an error label if the CF flag is zero.

The bitmap based enforcement incurs small runtime and code-size overheads. SFI implemented on top of MIP incurs an overhead of 4% and 7%, on x86-32 and x86-64 respectively, which is comparable to other SFI systems. More importantly, it incurs a much smaller code-size increase than the aligned-chunk enforcement in PittSFIeld. On x86-32 and x86-64, MIP's average code-size increase is 13% and 16%, respectively.

Enforcement by fine-grained control-flow integrity. Control-Flow Integrity (CFI Abadi *et al.*, 2005) is a general solution for enforcing that a program's control flow follows a pre-determined control-flow graph. Several SFI systems enforce fine-grained CFI to restrict the control flow of the program. In fine-grained CFI, each indirect branch is associated with a set of control-flow targets according to the control-flow graph and the enforcement ensures that the destination of an indirect branch must be a member of the set of targets. For instance, MiSFIT (Small, 1997) builds a hash table of valid targets for each indirect call and performs a look up from the table before the call to check its security. Similarly, ARMor (Zhao *et al.*, 2011), an SFI implementation on ARM, uses the ID-based scheme introduced by Abadi *et al.* (2005) for restraining control flow.

Fine-grained CFI enforcement is sufficient to ensure the strengthened control-flow policy required by SFI. One benefit is that its fine-grained CFG enables better data-guard optimization results, for those optimizations that rely on the CFG for identifying optimization opportunities. For instance, the range-analysis based optimizations discussed in Sec. 3.1.4 compute value ranges of registers to identify where data guards can be removed or moved to new locations. A finer grained CFG enables the computation of more precise results in range analysis and therefore the range-analysis based optimizations benefit more from fine-grained CFI enforcement than aligned-chunk enforcement or bitmap based enforcement, as demonstrated by Zeng *et al.* (2011). On the other hand, many SFI systems such as PittSFIeld (McCamant and Morrisett, 2006) and NaCl (Yee *et al.*, 2009) use only data-guard optimizations that are agnostic to the CFG precision; for these systems, it is unnecessary to have one target set for each indirect branch, whose enforcement incurs additional overhead.

Abstracting control-flow enforcement. Similar to data guards, we introduce an abstract form of *control guards* to refer to either code-address checking or code-address masking, as a way of hiding implementation details. Instruction "$r' :=$ cGuard(VT, r) takes an address in r and a set of valid targets in VT, and performs either checking or masking to compute a valid code address in r'. Unlike data guards, control guards

take VT as a parameter because in some enforcement (e.g., the bitmap based enforcement) VT is program specific (i.e., computed from the input code).

A control guard should have the following properties:

- When r holds an address in VT, then "$r' :=$ cGuard(VT, r)" succeeds and r' equals r.
- When r holds an address outside VT, an error state is generated (for address checking) or r' gets an address within VT (for address masking).

Indirect branches are then instrumented via the help of control guards. For instance, an indirect branch through register r becomes "$r :=$ cGuard(VT, r)", followed by the indirect branch. An indirect branch through a memory operand is instrumented to (1) read the destination address from memory to some scratch register r, (2) perform control guarding on r, and (3) perform an indirect branch through r. A return instruction is instrumented to (1) pop the return address from the stack to some scratch register r, (2) perform control guarding on r, and (3) perform an indirect branch through r.

3.2.2 Interaction with the outside world

Security can be achieved by total isolation of untrusted code, but in practice total isolation is rarely the desired property. Untrusted, sandboxed code must be able to interact with other parts of the system to fulfill its promised functionality. In NaCl (Yee *et al.*, 2009), a browser plug-in module developed in native code can access NaCl's service runtime for memory-management operations and other system services, and can interact with other modules via an RPC (Remote Procedure Call) facility and via a subset of NPAPI (Netscape Plugin Application Programming Interface).

Interaction between sandboxed code and the outside world is modeled in this article by direct jumps to known, safe external addresses in SE. Safe external addresses are in the same process address space as sandboxed code, but host trusted code; we will call the trusted code at

external addresses a *trusted runtime*. The trusted runtime provides services that require higher privileges than what is given to sandboxed code. For instance, it can host system services including file operations and memory-management operations; it can also host services that enable communication between multiple sandboxes. The relationship between sandboxed code and the trusted runtime is similar to the relationship between a user application and the OS kernel, which communicate via the system-call interface.

To invoke an external service, data required by the service must be passed from the sandbox. How data is passed is decided by the implementation and convention. Small data can be passed by registers; data can also be marshalled, as in RPC calls. Since external services are trusted and have access to the sandboxed memory, passing a pointer without marshalling the underlying buffer is more efficient than performing a deep copy (which copies also the buffer). When the sandboxed code is multithreaded, special care must be taken to avoid synchronization issues on data exchanged between the sandboxed code and the trusted runtime. When pointers are passed directly without marshalling the underlying buffers, there are potential race conditions when the sandboxed code and the runtime access the buffers concurrently. When deep copying is used to marshall pointers, the copied buffers in the runtime may be out of sync with the original buffers in the sandbox, which would cause correctness problems. In either implementation, proper synchronization code needs to added carefully for multithreaded code.

Interaction between sandboxed code and the trusted runtime should be carefully monitored to preserve security, as this is where attacks can happen. Additional security policies can also be enforced. For instance, sandboxed code should be prevented from invoking security-sensitive system calls and library calls directly; instead, the SFI rewriter should route those calls to external services that check arguments before performing actual calls. As an example, a direct call to the `open` system call should be disallowed; instead, a corresponding external service should check whether the underlying resource (a file, a network socket, etc.) can be accessed by sandboxed code according to some resource-access policy. As another example, some SFI implementations rely on memory page protection, which means that invocations of

`mprotect` must be checked to ensure that sandboxed code cannot modify permissions on memory pages in the sandbox arbitrarily.

Depending on what interface the trusted runtime provides to sandboxed code, the interaction can be extremely complex and additional support may be needed. Castro *et al.* (2009) summarized the difficulties when applying SFI to the interaction between OS kernel extensions (e.g., device drivers) and an OS kernel: (1) dynamic policy change: the interface may need to change the policy dynamically; for example, after an object is transferred from the OS kernel to an extension, the object is readable/writable; but after the extension is finished with the object, the read/write rights should be revoked; (2) granularity of protection: SFI's data-access policy allows access to a contiguous data region, while the kernel-extension interaction is at the level of byte granularity since an object exchanged between the kernel and an extension may reside in a data region that is not contiguous with the memory occupied by other exchanged objects; (3) policy language: SFI's policy is about low-level operations including reads, writes, and branches, while interaction with the kernel needs high-level access rights on objects; for example, the types of operations that can be performed on mutexes are different from the ones on kernel devices. To address these difficulties, BGI (Castro *et al.*, 2009) proposed to use a runtime table to record access rights of objects, at the granularity of bytes; these rights include not just read, write, indirect-call rights, but also type rights that allow specific operations on objects (it enables a coarse form of type safety). Further, the table for rights is adjusted dynamically by trusted primitives to accommodate dynamic policy changes. In a similar but more limited fashion, XFI (Erlingsson *et al.*, 2006) uses a runtime permission table and performs checks on memory writes based on the table in a slow path when fast-path checks fails (which checks if writes stay within the sandbox). LXFI (Mao *et al.*, 2011) generates checks automatically based on user-provided interface-function annotations about how access rights should be generated, transferred, checked, and revoked.

3.3 Portable enforcement

IRMs such as SFI are typically implemented through low-level rewriting, either by assembly-level rewriting, or by modifying a compilation tool chain's backend to emit code with inlined checks. The benefit of rewriting at the low level is that architecture-specific optimizations (see Sec. 3.1.7) can be easily applied to speed up policy enforcement.

However, an SFI implementation by low-level rewriting is tightly coupled with a specific architecture, resulting in poor reusability and retargetability. It is non-trivial to port a low-level SFI implementation from one architecture to another. 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 (Sehr *et al.*, 2010). One reason for the nontrivial effort is the differences among architectures, including the instruction set, the available hardware features, the number and size of registers, and many others. Another downside of low-level rewriting is that optimizations at the low level become more challenging because low-level code lacks high-level structural information such as types that can aid optimizations.

There have been several attempts at implementing SFI at the level of a compiler Intermediate Representation (IR), which provides the benefits of being architecture independent and carrying a wealth of structural information that helps analysis and optimizations. Google's NaCl team proposed Portable Native Client (PNaCl), described by a white paper (Donovan *et al.*, 2010). PNaCl requires code be transmitted in the LLVM IR format, with portability as the goal. After mobile IR code is downloaded into the Chrome browser, PNaCl compiles the IR code into SFI-compliant native code and reuses NaCl to constrain native code's behavior. This is analogous to Java's design; Java code is compiled to Java bytecode, which is then transmitted over the internet and interpreted (or compiled just-in-time) locally by a Java Virtual Machine.

Strato (Zeng *et al.*, 2013) also performs LLVM-IR level rewriting but with the emphasis on removing the compiler backend from the TCB. It includes a constraint language for encoding constraints required by IR-level rewriting and optimizations, and a process for checking those

constraints after backend lowering to ensure they are still obeyed in the low-level code. To further ensure the trustworthiness, it implements an independent verifier to validate the final low-level code, thus removing all the instrumentation, transformation, optimizations, and constraint checking from the TCB.

PSFI (Kroll *et al.*, 2014) performs SFI instrumentation at the level of Cminor, an intermediate language in a verified compiler (CompCert by Leroy (2006)). PSFI's focus is on obtaining formal assurance about its IR-level SFI rewriting; it relies on the correctness of CompCert's backend, which is equipped with a machine-checked formal proof.

4

SFI Verification and Formalization

SFI relies on a code rewriter to instrument memory and indirect branch instructions. Obviously, errors in the design and implementation of the SFI rewriter can result in code that may violate the SFI policy. For instance, when SFI rewriting is implemented as a rewriting pass inside a compiler, without extra care a later optimization pass that is unaware of the SFI policy may move or even remove the inserted SFI checks. Therefore, it is better to not trust the SFI rewriter.

One way is to show the correctness of the SFI rewriter by proving that it always produces code that obeys the SFI policy. The difficulty of performing the correctness proof depends on the complexity of the SFI rewriter. One attempt was by Kroll *et al.* (2014), who formalized an SFI rewriter on a C-like intermediate language called Cminor and proved in a theorem prover (Coq) that the rewriter is correct at the Cminor level. Their system, however, critically depends on CompCert (Leroy, 2006) for compiling Cminor code to assembly code. It is unclear how to compose CompCert's correctness proof with the correctness proof of the Cminor-level SFI rewriter to get an end-to-end correctness proof.

A more popular approach to removing SFI from the TCB is to construct a separate verifier (Wahbe *et al.*, 1993; McCamant and Morrisett,

2006; Yee *et al.*, 2009; Zeng *et al.*, 2011; Morrisett *et al.*, 2012; Zeng *et al.*, 2013; Niu and Tan, 2013), which verifies that checks are inserted at the right place in the code produced by the rewriter. This approach has a number of benefits:

- The verifier is much smaller than the rewriter and as a result this design reduces the size of the TCB. For instance, the first verifier in NaCl-x86 (Yee *et al.*, 2009) is about 600 lines of C code, while its rewriter is a modified compilation toolchain (including the `gcc` compiler, an assembler, and a linker). Morrisett *et al.* (2012) proposed a state-machine based approach, which constructs an even smaller NaCl verifier.

- The verifier can statically analyze input code and perform SFI enforcement by a combination of static and dynamic checks. For instance, a typical SFI verifier statically verifies the security of direct branches, while leaving the security of indirect branches to inlined dynamic checks.

- Finally, the verifier enables separation of duties between code producers and consumers. Code producers can send code to a code consumer (possibly on a different machine), which can use the verifier to check if the received code satisfies the SFI policy.

The implementation of a verifier, however, is tricky to get right. It operates at the machine-instruction level and needs to deal with machine specifics such as variable-sized instructions. There is a substantial risk that bugs may exist in the verifier's implementation. In a security contest run by Google, bugs were found in Google's NaCl verifier (*Native Client Security Contest* 2009).[1] Therefore, there is a need to remove the verifier from the TCB. This can be achieved by formally proving the correctness of the verifier, as performed in previous work (Morrisett *et al.*, 2012; McCamant, 2006; Zhao *et al.*, 2011).

To illustrate typical SFI verifiers and their correctness proofs, in this chapter we present the operational semantics of a machine with the

[1] For instance, during the contest Mark Dowd discovered that Google's verifier allowed dangerous prefixes before two-byte jump instructions (issue # 50), which would allow an attacker to execute arbitrary code.

$$
\begin{array}{lllll}
(State) & S & ::= & (M, R, pc) \\
(Mem) & M & \in & Word \rightarrow Word \\
(RegFile) & R & \in & Register \rightarrow Word
\end{array}
$$

Figure 4.1: Machine states.

instruction set in Fig. 3.2 of Ch. 3 and formally model an SFI verifier. We will then prove the correctness of the verifier based on the semantics. Informally, verifier correctness means that, if the verifier passes some code, the code's execution must follow the SFI policy.

4.1 Operational semantics

We next present the operational semantics of a machine with a variable-sized instruction set. The instruction set was presented in Fig. 3.2 on page 148. The machine operates by decoding the next instruction and changes its state according to the instruction's semantics. Fig. 4.1 presents the definition of machine states. We use *Word* for the domain that contains words of some fixed size and *Register* for the domain of registers; we use symbol w for a word and r for a register. A machine state S consists of a memory M, which is a map from addresses to values (both are words), and a register file R, which is a map from registers to values, and a program counter $pc \in Word$. We use notation $S.M$, $S.R$, and $S.pc$ for the memory, register file, and program counter of state S, respectively.

The operational semantics assumes an abstract decode function, which takes a list of words and returns the first instruction encoded in the list, the size of the instruction, and the remaining list of words that has not been decoded:

$$\text{decode} : Word\ List \rightharpoonup Instr \times Nat \times (Word\ List)$$

That is, when given a word list wl, it returns an instruction i, the instruction's size sz, and the remaining list of words wl', if the decoding is successful. The decode function is partial since the initial words in wl may not encode an instruction. Note we use $\rightharpoonup$ for a partial function.

$(M, R, pc) \to (M', R', pc'),$ if $\text{decodeM}(M, pc) = (i, sz)$	
and $i =$	then $(M', R', pc') =$
$r_d := r_s \ aop \ op$	$(M, R[r_d \mapsto \widehat{aop}(R(r_s), R(op))], pc + sz)$, if $(pc + sz) \in [CB, CL]$
$r_d := \text{mem}(r_s + w)$	$(M, R[r_d \mapsto M(R(r_s) + w)], pc + sz)$, if $(R(r_s) + w) \in [DB, DL]$ and $(pc + sz) \in [CB, CL]$
$\text{mem}(r_d + w) := r_s$	$(M[(R(r_d) + w) \mapsto R(r_s)], R, pc + sz)$, if $(R(r_d) + w) \in [DB, DL]$ and $(pc + sz) \in [CB, CL]$
if $(r_s \ cop \ op)$ goto w	(M, R, w), if $\widehat{cop}(R(r_s), R(op)) = \text{true}$ and $w \in [CB, CL]$
if $(r_s \ cop \ op)$ goto w	$(M, R, pc + sz)$, if $\widehat{cop}(R(r_s), R(op)) = \text{false}$ and $(pc + sz) \in [CB, CL]$
jmp r	$(M, R, R(r))$, if $R(r) \in [CB, CL]$
jmp w	(M, R, w), if $w \in [CB, CL] \cup SE$

where $\text{decodeM}(M, pc) = (i, sz)$, if
$\text{decode}([M(pc), \ldots, M(pc + \text{maxSz} - 1)]) = (i, sz, wl')$
and $R(op) = R(r)$ when op is r, and $R(op) = w$ when op is w.

Figure 4.2: Operational semantics.

Fig. 4.2 presents the operational semantics of the machine. A step in the machine is modeled as $(M, R, pc) \to (M', R', pc')$. It operates by decoding the instruction at the program counter in memory and then executes the instruction. The function $\text{decodeM}(M, pc)$, defined at the bottom of the figure, retrieves a list of words from memory and invokes the decode function to get the next instruction and its size; constant maxSz in the definition is the maximum number of words in an instruction's encoding on the machine.[2] We overload the notation $R(-)$ on operands so that $R(op)$ is defined to be $R(r)$ when op is r and

[2]For x86-32, the maximum size is 15.

is defined to be w when op is w.

The semantics of instructions in Fig. 4.2 is largely standard, except that it enforces the SFI policy. For instance, when the next instruction is "$r_d := r_s \; aop \; op$", the value of r_d in the register file is updated to be the result of applying aop on the values in r_s and op. We use $\widehat{aop}$ for aop's interpretation, which is a function that takes two parameters and returns the result of applying aop. Note that the semantics requires $(pc + sz) \in [CB, CL]$, meaning the machine gets stuck when the program counter in the next state falls outside the code range. As another example, the semantics of "$r_d := \text{mem}(r_s + w)$" updates r_d to hold the value at address $r_s + w$ in memory and it further requires $r_s + w$ must be in the data region and the follow-up address in the code region. In the semantics, a direct jump "jmp w" is the only instruction that can transfer the control outside the code region, as destination w can be either in the code region or one of the safe external addresses; recall that SE stands for the set of safe external addresses.

In summary, the operational semantics is defined so that the machine gets stuck (i.e., does not have a next state to step to) in the following cases: (1) when the decode function fails, (2) when performing memory operations outside of the data region; (3) when the control transfers outside of code region, except that a direct jump can also jump to a safe external address.

In order to abstract away from implementations about how data and control guards are performed, we introduce two additional instructions: "$r_d := \text{dGuard}(r_s)$" is a data guard for r_s and the result after masking or checking is put in r_d; "$r_d := \text{cGuard}(VT, r_s)$" is a control guard on r_s according to a set of valid code addresses in VT and the result is put in r_d.

Fig. 4.3 presents the semantics of the two guard instructions. There are three rules for "$r_d := \text{dGuard}(r_s)$". If r_s's value is already in the data region, then the first rule just copies the value to r_d. If r_s's value is outside the data region, then the machine can nondeterministically choose to perform address checking or address masking. The second rule models address checking by not changing the state, meaning that the machine can get into an infinite loop; this is one way of modeling a safe "trapped" state. The last rule models address masking by arbitrarily

$(M, R, pc) \to (M', R', pc')$, if $\text{decodeM}(M, pc) = (i, sz)$	
and $i =$	then $(M', R', pc') =$
$r_d := \text{dGuard}(r_s)$	$(M, R[r_d \mapsto R(r_s)], pc + sz)$, if $R(r_s) \in [DB, DL]$ and $(pc + sz) \in [CB, CL]$
$r_d := \text{dGuard}(r_s)$	(M, R, pc), if $R(r_s) \notin [DB, DL]$
$r_d := \text{dGuard}(r_s)$	$(M, R[r_d \mapsto d], pc + sz)$, if $R(r_s) \notin [DB, DL]$ and $d \in [DB, DL]$ and $(pc + sz) \in [CB, CL]$
$r_d := \text{cGuard}(VT, r_s)$	$(M, R[r_d \mapsto R(r_s)], pc + sz)$, if $R(r_s) \in VT$ and $(pc + sz) \in [CB, CL]$
$r_d := \text{cGuard}(VT, r_s)$	(M, R, pc), if $R(r_s) \notin VT$
$r_d := \text{cGuard}(VT, r_s)$	$(M, R[r_d \mapsto d], pc + sz)$, if $R(r_s) \notin VT$ and $d \in VT$ and $(pc + sz) \in [CB, CL]$

Figure 4.3: Semantics of data and control guards.

choosing an address d from the data region and sets it to r_d. Similarly, "$r_d := \text{dGuard}(r_s)$" has three rules and its semantics is nondeterministic. The nondeterminism allows the following proof development to be general about how address guards are implemented.

For simplicity, the data-address masking rule puts the masked address in the data region, while PittSFIeld (McCamant and Morrisett, 2006) puts the masked address in the data region or an unmapped memory region (the zero-ID region in PittSFIeld). This additional complexity could be modeled by changing the masking rules and by extending the semantics of memory operations so that an access to unmapped memory regions gets to a trapped state; the verifier and proof development would stay the same.

4.2 Modeling an SFI verifier

At a high level, an SFI verifier takes in a piece of code encoded as a list of words and decides whether the code satisfies the SFI policy. For ease of checking, a typical verifier enforces stronger constraints on code than what the SFI policy requires. The SFI verifier we will introduce enforces the following constraints:

(1) The input piece of code can be uniquely disassembled into a sequence of instructions by sequential disassembly from the beginning of the code. This constraint ensures that the verifier can inspect the uniquely disassembled instruction sequence for SFI security. It assumes that the code producer separates code and data so that there are no undefined instructions in between two valid instructions.

(2) Immediately before every memory read/write instruction, there is a data guard "$r := \text{dGuard}(r)$" and register r is the address used in the memory instruction.

(3) Immediately before every indirect jump "jmp r", there is a control guard "$r := \text{cGuard}(VT, r)$.

(4) No direct jumps or conditional jumps can target the middle of instructions or the middle between a guard and the following dangerous instruction. This control-flow target constraint and the previous constraints ensure that the disassembled instruction sequence is respected by control flow, and the guard before a dangerous instruction cannot be bypassed.

(5) Only direct jumps can target safe external addresses.

We next formalize a verifier that enforces the aforementioned constraints, and Sec. 4.3 will show that these constraints are strong enough for ensuring the SFI policy.

The verifier is formalized in three steps: (1) a sequential disassembly procedure converts code as a list of words to a sequence of instructions and their addresses and sizes; (2) a procedure builds a set of valid jump targets; (3) a procedure checks that all jump instructions have targets that are in the set built in step (2).

$$\text{seqDis}(code, l) = \begin{cases} [\,], & \text{if } code = [\,] \\ (l, sz, i) :: (\text{seqDis}(code', l + sz)), & \text{if decode}(code) = (i, sz, code') \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Figure 4.4: Sequential disassembly of code.

Sequential disassembly. Function seqDis in Fig. 4.4 takes *code*, which is a list of words, and l, which is the start address of *code* in the code region, and performs sequential disassembly to output a program P. The program is represented as a list of (l, sz, i), where l is an address, i an instruction, and sz the instruction's size. In the definition, we use $[\,]$ for the empty list, and :: for the infix list-cons operator that takes an element e and a list l, and builds a new list whose head is e and tail is l. Since the decode function may fail, seqDis($code, l$) may be undefined.

Identifying valid jump targets. Function collectVT in Fig.4.5 takes the program P after sequential disassembly and collects a set of Valid Targets (VT) for jump instructions. Essentially, a valid target is the beginning of a pseudoinstruction, and does not admit an address in the middle of an instruction or the middle between a guard and the following dangerous instruction. It is undefined, when a data guard is not immediately followed by a memory instruction or when a control guard is not immediately followed by an indirect jump.

Checking jump targets. Function checkJmp in Fig. 4.6 returns true if jumps in program P have targets that are consistent with those in VT and SE. In particular, conditional jumps and indirect jumps can have targets in VT, and a direct jump can have targets in VT or SE:

By the definition, if checkJmp(VT, SE, P) returns true, the last instruction must be "jmp w". This indirect jump can either jump to a special external address for terminating the execution, or transfer to an address already in the code.

$$
\begin{array}{l}
\text{collectVT}(P) = \\
\quad \begin{cases}
\{\}, \quad \text{if } P = [\,] \\
\{l\} \cup \text{collectVT}(P') \\
\qquad \text{if } P = (l, sz, i) :: P' \text{ and } i = \text{“}r_d := r_s \; aop \; op\text{”}, \\
\qquad \text{ or “if } (r_s \; cop \; op) \text{ goto } w\text{” or “jmp } w\text{”} \\
\{l_1\} \cup \text{collectVT}(P') \\
\qquad \text{if } P = (l_1, sz_1, i_1) :: (l_1 + sz_1, sz_2, i_2) :: P' \\
\qquad \text{and } i_1 = \text{“}r := \text{dGuard}(r)\text{”} \\
\qquad \text{and } i_2 = \text{“}r' := \text{mem}(r)\text{” or “mem}(r) := r'\text{”} \\
\{l_1\} \cup \text{collectVT}(P') \\
\qquad \text{if } P = (l_1, sz_1, i_1) :: (l_1 + sz_1, sz_2, i_2) :: P' \\
\qquad \text{and } i_1 = \text{“}r := \text{cGuard}(VT, r)\text{”} \\
\qquad \text{and } i_2 = \text{“jmp } r\text{”} \\
\text{undefined}, \quad \text{otherwise}
\end{cases}
\end{array}
$$

Figure 4.5: Building a set of valid jump targets.

The verifier. The verifier definition is presented in Fig. 4.7. It passes a piece of code if the program that is the result of sequential disassembly at address *CB* can successfully pass the checkJmp procedure using the set of valid targets built from collectVT. By passing collectVT(P) to checkJmp, it ensures that the set of valid targets built by collectVT is respected by jumps in the code, enforcing that jumps can target only the beginning of a pseudoinstruction.

4.3 Verifier correctness

To state the correctness theorem, we first formalize a notion of *appropriate states*; a state S is appropriate with respect to *code* if the code is loaded into the region of memory with address range $[CB, CL]$:

Definition 4.1 (Appropriate state)**.**

$$
\begin{array}{ll}
\text{appropState}(S, code) = & \text{codeLoaded}(code, S.M, CB) \\
& \quad \wedge\, CB + |code| < CL \\
\text{codeLoaded}(code, M, l) = & \forall 0 \leq i < |code|.\; M(l + i) = code(i)
\end{array}
$$

$$
\begin{array}{l}
\text{checkJmp}(VT, SE, P) = \\
\quad \begin{cases}
\text{checkJmp}(VT, SE, P') \\
\qquad \text{if } P = (l, sz, r_d := r_s \ aop \ op) :: P' \\
w \in VT \ \wedge \ \text{checkJmp}(VT, SE, P') \\
\qquad \text{if } P = (l, sz, \text{if } (r_s \ cop \ op) \text{ goto } w) :: P' \\
\text{checkJmp}(VT, SE, P') \\
\qquad \text{if } P = (l_1, sz_1, i_1) :: (l_1 + sz_1, sz_2, i_2) :: P' \\
\qquad \text{and } i_1 = \text{“}r := \text{dGuard}(r)\text{”} \\
\qquad \text{and } i_2 = \text{“}r' := \text{mem}(r)\text{” or “mem}(r) := r'\text{”} \\
\text{checkJmp}(VT, SE, P') \\
\qquad \text{if } P = (l_1, sz_1, i_1) :: (l_1 + sz_1, sz_2, i_2) :: P' \\
\qquad \text{and } i_1 = \text{“}r := \text{cGuard}(VT, r)\text{”} \\
\qquad \text{and } i_2 = \text{“jmp } r\text{”} \\
w \in VT \cup SE \ \wedge \ \text{checkJmp}(VT, SE, P') \\
\qquad \text{if } P = (l, sz, \text{jmp } w) :: P' \\
w \in VT \cup SE, \quad \text{if } P = [(l, sz, \text{jmp } w)] \\
\text{false}, \quad \text{otherwise}
\end{cases}
\end{array}
$$

Figure 4.6: Checking that jumps use targets in VT and SE.

The verifier-correctness theorem then states, when a piece of code passes the verifier and is loaded into an initial state S_0 whose program counter is CB, the state can execute without getting stuck or can reach a safe external address.

Theorem 4.1 (Verifier correctness)**.**
If verifier($code$) = pass, appropState(S_0, $code$), $S_0.pc = CB$, and $S_0 \rightarrow^* S$, then either exists S' so that $S \rightarrow S'$ or $S.pc \in SE$.

Since we formulated the operational semantics so that the machine gets stuck when the SFI policy is violated, the theorem implies that the SFI policy is not violated during execution of code that passes the verifier.

We next present a proof sketch of the theorem. First, we formalize a notion of *safe states*: a safe state is an appropriate state whose program counter is at the beginning of a pseudoinstruction or in SE.

$$\text{verifier}(\mathit{code}) = \begin{cases} \text{pass} & \text{if seqDis}(\mathit{code}, \mathit{CB}) = P \\ & \text{and checkJmp}(\text{collectVT}(P), \mathit{SE}, P) = \text{true} \\ \text{fail} & \text{otherwise} \end{cases}$$

Figure 4.7: An SFI verifier.

Definition 4.2 (Safe state)**.**

$$\begin{aligned} &\text{safeState}(S, \mathit{code}) = \text{appropState}(S, \mathit{code}) \ \wedge \ \text{goodControl}(S, \mathit{code}) \\ &\text{goodControl}(S, \mathit{code}) = \\ &\quad \exists P.\ \text{seqDis}(\mathit{code}, \mathit{CB}) = P \ \wedge \ S.\mathit{pc} \in \text{collectVT}(P) \cup \mathit{SE} \end{aligned}$$

We can then show that the initial state is a safe state:

Lemma 4.2 (Safe initial state)**.**
If $\text{verifier}(\mathit{code}) = \text{pass}$, $\text{appropState}(S_0, \mathit{code})$, and $S_0.\mathit{pc} = \mathit{CB}$, then $\text{safeState}(S_0, \mathit{code})$.

However, after a finite number of steps from a safe state, the resulting state may not be a safe state. The reason is that a pseudoinstruction may consist of multiple instructions; consequently, states that are reached when the control is at the middle of pseudoinstructions are not safe states by definition. One observation we can make is that these intermediate states can reach safe states in finite numbers of steps. Therefore, we formalize a notion of safety within k steps:

Definition 4.3 (Safety in k steps)**.**

$$\begin{aligned} &\text{safeInK}(k, S, \mathit{code}) = \\ &\quad k > 0 \ \wedge \ \text{appropState}(S, \mathit{code}) \ \wedge \ (\exists S'.\ S \to S') \ \wedge \\ &\quad (\forall S'.\ S \to S' \ \Rightarrow \ (\text{safeState}(S', \mathit{code}) \ \vee \ \text{safeInK}(k-1, S', \mathit{code}))) \\ &\text{safeInSomeK}(S, \mathit{code}) = \exists k.\ \text{safeInK}(k, S, \mathit{code}) \end{aligned}$$

A critical lemma is to show that a safe state can always reach another safe state or reach a safe external address within a finite number of steps.

Lemma 4.3 (Safe state is k safe)**.**
If $\text{verifier}(\mathit{code}) = \text{pass}$ and $\text{safeState}(S, \mathit{code})$, then $\text{safeInSomeK}(S, \mathit{code})$ or $S.\mathit{pc} \in \mathit{SE}$.

Proof. Because of safeState(S, *code*), $S.pc$ must be before a pseudoinstruction or in SE. If it is in SE, then the proof is complete. If it is before a pseudoinstruction, the proof proceeds by case analysis over the pseudoinstruction. When it is one-instruction pseudoinstruction, we show it can reach a safe state within one step. This includes the cases for arithmetic-logic instructions, conditional-jump instructions, and direct-jump instructions. In the proof, we use "verifier(*code*) = pass" to show that the machine can always make one step and reach the next pseudoinstruction or reach a safe-external address (when the instruction is "jmp w" and $w \in SE$). When it is a two-instruction pseudoinstruction, we show S can reach a safe state within two steps. This includes the cases of a data guard followed by a memory operation and a control guard followed by an indirect jump. Again, we use "verifier(*code*) = pass" to show that the machine can make two steps without getting stuck and reach a safe state.

The following lemma about the preservation of safeInSomeK across the evaluation is a corollary of Lemma 4.3.

Lemma 4.4 (safeInSomeK preservation)**.**
If verifier(*code*) = pass, safeInSomeK(S, *code*), and $S \to S'$, then either safeInSomeK(S', *code*) or $S'.pc \in SE$.

The verifier-correctness theorem can then be proved as follows. By Lemma 4.2, we have safeState(S_0, *code*). We then get safeInSomeK(S_0, *code*) or $S_0.pc \in SE$ by Lemma 4.3. For the second case, we must have $S_0 = S$ and the proof is complete. For the first case, we use the preservation lemma (Lemma 4.4) to get either safeInSomeK(S, *code*) or $S.pc \in SE$. The second case means the proof is complete. For the first case, we get $\exists S'.\ S \to S'$ by the definition of safeInSomeK.

5

Future Directions

As an isolation primitive, SFI is efficient, flexible, and can be performed entirely within user space. We next outline several future directions, with the goals of making SFI more accessible to software developers and making it more robust as a fault-isolation primitive.

Tool and programming support for program partitioning. Structuring a computer system into multiple protection domains can fundamentally improve its reliability and security. A general question is how to turn a monolithic system into components that are in separate protection domains and communicate via well-defined interfaces. This is in general referred to as *privilege separation* (Provos *et al.*, 2003). In particular, how to draw the boundary between code that should be put into an SFI sandbox and the rest of the system?

In some situations, the boundary is clear. For instance, plug-ins in extensible systems such as browsers and web servers form natural boundaries; similarly, a native library loaded by a Java Virtual Machine forms its own boundary. In many other situations, however, drawing boundaries can be a daunting task. For instance, it took Google a significant effort to implement the Chrome browser as a system of

cooperating processes (Barth *et al.*, 2008).

Therefore, tool and programming support that help programmers identify and enforce isolation boundaries can push the adoption of privilege separation in general and SFI in particular. Two strategies are possible. For legacy code, we can apply automatic program partitioning based on a small amount of user annotations (e.g., annotation about what data is sensitive). Previous systems (Brumley and Song, 2004; Bittau *et al.*, 2008; Yongzheng Wu and Dong, 2013; Liu *et al.*, 2017) have made good progress, but more work will be needed to make automatic program partitioning applicable to general software and design partitioning algorithms that balance performance and security. Another strategy is to design new programming abstractions about isolation and integrate them into existing programming languages; through the new programming abstractions, programmers can dynamically create/release sandboxes that contain code and data (as a form of "sandboxed memory") and be able to use the properties provided by sandboxes to reason about the high-level properties of their application. A compiler will analyze the application and decide how to implement each sandbox (e.g., through SFI, or a process, or a sandbox supported by Intel's SGX (*Intel Software Guard Extensions (Intel SGX)* 2016)).

Security enforcement on interface code. The SFI policy allows sandboxed code to call into the trusted runtime. It implicitly assumes that the runtime preserves confidentiality and integrity. This assumption, however, can be too strong in practice. For instance, the implementation of a service in the runtime can have a buffer overflow, which can allow sandboxed code to escape the sandbox and read/write arbitrary memory. Furthermore, data are exchanged for external service calls. Without care, exchanged data can break confidentiality and integrity, especially when the interface between sandboxed code and the trusted runtime is complex, involving features such as data ownership exchange and fine-grained access control on data.

Therefore, it would be beneficial to have a design in which a small amount of interface code is set up between SFI sandboxes and the runtime, and it acts as a security monitor that regulates the interaction. Experience has shown that bugs are plenty in interface code

between software components; for instance, hundreds of bugs were identified in the interface code between components developed in different languages (Furr and Foster, 2005; Tan and Croft, 2008; Kondoh and Onodera, 2008). We can apply program-analysis tools to identify security/reliability bugs in SFI interface code. Another way of enhancing interface security is to take a specification about interface security and inline checks into the interface code, as performed by LXFI (Mao *et al.*, 2011). Finally, since interface code is typically small, it might even be amenable to static verification (e.g., model checking).

Side channel control. Through SFI, we can put secret data outside an SFI sandbox and prevent untrusted code from directly learning the secret data by sandboxing memory reads and by carefully designing an interface between the sandbox and the trusted runtime. Nevertheless, information about the secret data can leak through side channels caused by shared resources between the sandbox and the trusted runtime.

It is well known that other isolation mechanisms can suffer from side channels. For instance, an attacker virtual machine can learn the secret (e.g., a cryptographic key) possessed by a victim virtual machine if the two virtual machines share a physical cache (Zhang *et al.*, 2012); similarly, an attacker process can learn information about a victim process via resources such as shared files. An SFI sandbox shares the same address space as the trusted runtime and other sandboxes. It is unclear whether SFI, as an isolation mechanism, enables more side channels. An interesting direction would be to compare SFI with other isolation mechanisms, in terms of what side channels they enable, measurement on leakage rates, and possible mitigation mechanisms.

Recovery mechanisms. Malicious code in an SFI sandbox cannot read/write out-of-sandbox memory or execute illegal instructions, but can try to disrupt or degrade system performance, by hoarding resources (e.g., CPU cycles, memory, network bandwidth) or by simply not cooperating with the rest of the system. Furthermore, since address masking changes an illegal address to a legal one, it may cause a benign but buggy sandboxed component to misbehave. Performing address checking can terminate a sandboxed component that violates the SFI

policy, except that termination may be insufficient for recovery; a proper recovery often requires releasing resources and returning the system to a clean state.

Seltzer *et al.* (1996) examined how to perform SFI recovery in the context of OS kernel extensions. The key idea is to wrap calls to sandboxed code in transactions. These transactions are aborted when misbehavior such as resource exhaustion or time outs is detected. Aborting a transaction would not only release the sandbox, but also all its acquired resources in the OS kernel (e.g., kernel locks). The transaction mechanism is heavyweight, as it requires the logging of all actions performed by a sandbox for possibly undoing those actions. One future direction is to study more lightweight SFI recovery mechanisms, for example, by having fine-grained sandboxes that can be individually rebooted (Candea *et al.*, 2004).

6

Going Beyond Fault Isolation

Fault isolation provides a strong line of defense of reliability and security of software systems. In many systems, it is often desirable to combine fault isolation with other strong properties. In this chapter, we will review other properties that provide strong integrity and confidentiality guarantees on software systems. Because of the high volume of related work, however, we can only briefly review these properties and for each property discuss one or two ways of realizing it, while omitting the discussion of many systems and optimizations.

Fine-grained memory access control. In SFI systems, the data region is a single contiguous memory region of a predetermined size and a memory read/write is allowed to access any byte in that memory region. This provides coarse-grained memory protection. It is sufficient for fault isolation, but it does not provide fine-grained memory access control.

In WIT (Akritidis *et al.*, 2008), for each memory write, pointer analysis is employed to compute the approximate set of objects that can be written by the memory write. At runtime, write sets are remembered by a color table and dynamic checks are used to prevent a memory write to change objects outside its write set. This enforces a stronger data

access policy than SFI, at the expense of more sophisticated analysis and more expensive runtime checks.

As briefly mentioned before, fine-grained memory access control is necessary when the interface between sandboxed code and the trusted runtime is complex, involving dynamic access-right changes on memory (Erlingsson *et al.*, 2006; Castro *et al.*, 2009; Mao *et al.*, 2011).

Control-flow integrity (CFI). SFI through inlined checks requires a coarse-grained form of control-flow integrity to ensure inlined checks cannot be bypassed. CFI (Abadi *et al.*, 2005), which enforces an arbitrary control-flow graph on the input program, is useful at thwarting many control-flow hijacking attacks, including return-to-libc attacks and return-oriented programming (Shacham, 2007). Many CFI systems have been implemented, with different tradeoffs between precision, efficiency, and support of features such as dynamic libraries.

Data-flow integrity (DFI). DFI (Castro *et al.*, 2006) enforces a Data-Flow Graph (DFG) computed by reaching-definition analysis and can prevent non-control data attacks (Chen *et al.*, 2005). A data-flow graph is a directed graph, in which nodes represent data definitions or data uses. A data-definition site is an instruction that writes to a memory location. A data-use site is an instruction that reads from a memory location. There is a directed edge that connects a data-definition node to a data-use node for the same memory location in the DFG. Given a DFG, DFI instruments the input program by inserting checks before memory reads and writes to enforce that the runtime data flow is compliant with the DFG.

Memory safety. A program is memory safe if it never accesses a buffer out of bound, called *spatial memory safety*, and it never accesses a deallocated buffer, called *temporal memory safety*. Since C and C++ are not memory-safe languages, attacks can exploit memory vulnerabilities in C/C++ code to gain illegal access. Many systems have been proposed to enforce spatial or temporal memory safety, or both (e.g., Nagarakatte *et al.*, 2009; Necula *et al.*, 2002; Criswell *et al.*, 2007; Dhurjati and

Adve, 2006). They track metadata including bounds information on buffers or pointers and insert checks before pointer operations to prevent out-of-bound buffer access and dangling pointer access.

7

Conclusions

In this article, we surveyed the area of software-based fault isolation, focusing on its policy, enforcement optimizations, and a formalization of an SFI verifier that captures its main constraints on untrusted code. Recent years have witnessed the increasing application of SFI and the substantial investment from Google on the Native Client technology in the Chrome browser. For greater adoption of SFI in diverse contexts, we believe there is a need for better integration of its concepts into tools (languages, compilers, runtime systems) that are readily accessible to programmers and for better support mechanisms (such as recovery mechanisms).

8

Acknowledgments

I thank Sun Hyoung Kim, Martin Abadi, John Sampson, and anonymous reviewers for their insightful comments. I also thank publisher James Finlay for providing constant guidance through the publication process. The author is supported by US NSF grants CNS-1624126, CCF-1624124, CCF-1723571, and US Office of Naval Research (ONR) under agreement number N00014-17-1-2539. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

Abadi, M., M. Budiu, Ú. Erlingsson, and J. Ligatti. 2005. "Control-flow integrity". In: *12th ACM Conference on Computer and Communications Security (CCS)*. 340–353.

Akritidis, P., C. Cadar, C. Raiciu, M. Costa, and M. Castro. 2008. "Preventing Memory Error Exploits with WIT". In: *IEEE Symposium on Security and Privacy (S&P)*. 263–277.

Ansel, J., P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. Schuff, D. Sehr, C. Biffle, and B. Yee. 2011. "Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code". In: *ACM Conference on Programming Language Design and Implementation (PLDI)*. 355–366.

Barth, A., C. Jackson, C. Reis, and G. Chrome. 2008. "The security architecture of the Chromium browser". *Tech. rep.*

Bittau, A., P. Marchenko, M. Handley, and B. Karp. 2008. "Wedge: splitting applications into reduced-privilege compartments". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. 309–322.

Bruening, D., Q. Zhao, and S. Amarasinghe. 2012. "Transparent Dynamic Instrumentation". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. 133–144.

Brumley, D. and D. Song. 2004. “Privtrans: Automatically Partitioning Programs for Privilege Separation”. In: *13th Usenix Security Symposium*. 57–72.

Candea, G., S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. 2004. “Microreboot — A Technique for Cheap Recovery”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 31–44.

Castro, M., M. Costa, and T. Harris. 2006. “Securing Software by Enforcing Data-flow Integrity”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–160.

Castro, M., M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. 2009. “Fast byte-granularity software fault isolation”. In: *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 45–58.

Chen, S., J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. 2005. “Non-control-data attacks are realistic threats”. In: *14th Usenix Security Symposium*. 177–192.

Common Language Infrastructure (CLI). 2006. 4th. Standard ECMA-335. Ecma International.

Criswell, J., A. Lenharth, D. Dhurjati, and V. Adve. 2007. “Secure virtual architecture: a safe execution environment for commodity operating systems”. *SIGOPS Oper. Syst. Rev.* 41(6): 351–366.

Deng, L., Q. Zeng, and Y. Liu. 2015. “ISboxing: An Instruction Substitution Based Data Sandboxing for x86 Untrusted Libraries”. In: *30th Inernational Conference on ICT Systems Security and Privacy Protection*. 386–400.

Dhurjati, D. and V. S. Adve. 2006. “Backwards-compatible array bounds checking for C with very low overhead”. In: *International Conference on Software engineering (ICSE)*. 162–171.

Donovan, A., R. Muth, B. Chen, and D. Sehr. 2010. “PNaCl: Portable Native Client Executables (white paper)”. http://src.chromium.org/viewvc/native_client/data/site/pnacl.pdf.

Erlingsson, Ú., M. Abadi, M. Vrable, M. Budiu, and G. Necula. 2006. “XFI: Software Guards for System Address Spaces”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 75–88.

Ford, B. and R. Cox. 2008. "Vx32: Lightweight User-level Sandboxing on the x86". In: *USENIX Annual Technical Conference*. 293–306.

Furr, M. and J. Foster. 2005. "Checking type safety of foreign function calls." In: *ACM Conference on Programming Language Design and Implementation (PLDI)*. 62–72.

"Intel Software Guard Extensions (Intel SGX)". 2016. https://software.intel.com/en-us/sgx.

Jaleel, A. 2010. "Memory characterization of workloads using instrumentation-driven simulation". URL: http://www.jaleels.org/ajaleel/workload/SPECanalysis.pdf.

Kiriansky, V., D. Bruening, and S. Amarasinghe. 2002. "Secure Execution via Program Shepherding". In: *11th Usenix Security Symposium*. 191–206.

Kondoh, G. and T. Onodera. 2008. "Finding bugs in Java Native Interface programs". In: *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM. 109–118.

Krishnamurthy, A., A. Mettler, and D. Wagner. 2010. "Fine-grained privilege separation for web applications". In: *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. 551–560.

Kroll, J. A., G. Stewart, and A. W. Appel. 2014. "Portable Software Fault Isolation". In: *CSF*. 18–32.

Kuznetsov, V., L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. 2014. "Code-Pointer Integrity". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–163.

Lampson, B. W. 1974. "Protection". *SIGOPS Oper. Syst. Rev.* 8(1): 18–24.

Leroy, X. 2006. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant". In: *33rd ACM Symposium on Principles of Programming Languages (POPL)*. 42–54.

Liu, S., G. Tan, and T. Jaeger. 2017. "PtrSplit: Supporting General Pointers in Automatic Program Partitioning". In: *24th ACM Conference on Computer and Communications Security (CCS)*. To appear.

Mao, Y., H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. 2011. "Software fault isolation with API integrity and multi-principal modules". In: *SOSP*. 115–128.

McCamant, S. 2006. "A machine-checked safety-proof for a CISC-compatible SFI technique". *Tech. rep.* No. 2006-035. MIT Computer Science and Artificial Intelligence Laboratory.

McCamant, S. and G. Morrisett. 2006. "Evaluating SFI for a CISC Architecture". In: *15th Usenix Security Symposium*.

Mettler, A., D. Wagner, and T. Close. 2010. "Joe-E: A Security-Oriented Subset of Java". In: *Network and Distributed System Security Symposium (NDSS)*.

Miller, M. 2006. "Robust composition: towards a unified approach to access control and concurrency control". *PhD thesis*. Baltimore, MD: Johns Hopkins University.

Morrisett, G., G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. 2012. "RockSalt: Better, Faster, Stronger SFI for the x86". In: *ACM Conference on Programming Language Design and Implementation (PLDI)*. 395–404.

Morrisett, G., D. Walker, K. Crary, and N. Glew. 1999. "From System F to Typed Assembly Language". *ACM Transactions on Programming Languages and Systems*. 21(3): 527–568.

Nagarakatte, S., J. Zhao, M. M. K. Martin, and S. Zdancewic. 2009. "SoftBound: highly compatible and complete spatial memory safety for C". In: *ACM Conference on Programming Language Design and Implementation (PLDI)*. 245–258.

"Native Client Security Contest". 2009. https://developer.chrome.com/native-client/community/security-contest.

Necula, G., S. McPeak, and W. Weimer. 2002. "CCured: type-safe retrofitting of legacy code". In: *29th ACM Symposium on Principles of Programming Languages (POPL)*. Portland, Oregon. 128–139.

Niu, B. and G. Tan. 2013. "Monitor Integrity Protection with Space Efficiency and Separate Compilation". In: *20th ACM Conference on Computer and Communications Security (CCS)*.

Payer, M. and T. Gross. 2011. "Fine-grained user-space security through virtualization". In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual execution Environments (VEE '11)*. 157–168.

Provos, N., M. Friedl, and P. Honeyman. 2003. "Preventing privilege escalation". In: *12th Usenix Security Symposium*. 231–242.

Saltzer, J. H. 1974. "Protection and the Control of Information Sharing in Multics". *Communications of the ACM*. 17(7): 388–402.

Schroeder, M. D. and J. H. Saltzer. 1972. "A Hardware Architecture for Implementing Protection Rings". *Communications of the ACM*. 15(3): 157–170.

Scott, K. and J. Davidson. 2002. "Safe Virtual Execution Using Software Dynamic Translation". In: *Proceedings of the 18th Annual Computer Security Applications Conference. ACSAC '02*. 209–218.

Sehr, D., R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. 2010. "Adapting Software Fault Isolation to Contemporary CPU Architectures". In: *19th Usenix Security Symposium*. 1–12.

Seltzer, M. I., Y. Endo, C. Small, and K. A. Smith. 1996. "Dealing with Disaster: Surviving Misbehaved Kernel Extensions". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 213–227.

Shacham, H. 2007. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". In: *14th ACM Conference on Computer and Communications Security (CCS)*. 552–561.

Siefers, J., G. Tan, and G. Morrisett. 2010. "Robusta: Taming the Native Beast of the JVM". In: *17th ACM Conference on Computer and Communications Security (CCS)*. 201–211.

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

Sun, M. and G. Tan. 2012. "JVM-Portable Sandboxing of Java's Native Libraries". In: *17th European Symposium on Research in Computer Security (ESORICS)*. 842–858.

Tan, G. and J. Croft. 2008. “An empirical security study of the native code in the JDK”. In: *17th Usenix Security Symposium*. 365–377.

Wahbe, R., S. Lucco, T. Anderson, and S. Graham. 1993. “Efficient Software-Based Fault Isolation”. In: *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. New York: ACM Press. 203–216.

Yee, B., D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. 2009. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. In: *IEEE Symposium on Security and Privacy (S&P)*.

Yongzheng Wu Jun Sun, Y. L. and J. S. Dong. 2013. “Automatically partition software into least privilege components using dynamic data dependency analysis”. In: *International Conference on Automated Software Engineering (ASE)*. 323–333.

Zeng, B., G. Tan, and Ú. Erlingsson. 2013. “Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors”. In: *22nd Usenix Security Symposium*. 369–382.

Zeng, B., G. Tan, and G. Morrisett. 2011. “Combining control-flow integrity and static analysis for efficient and validated data sandboxing”. In: *18th ACM Conference on Computer and Communications Security (CCS)*. 29–40.

Zhang, Y., A. Juels, M. K. Reiter, and T. Ristenpart. 2012. “Cross-VM Side Channels and Their Use to Extract Private Keys”. In: *19th ACM Conference on Computer and Communications Security (CCS)*. 305–316.

Zhao, L., G. Li, B. D. Sutter, and J. Regehr. 2011. “ARMor: Fully Verified Software Fault Isolation”. In: *11th Intl. Conf. on Embedded Software*. ACM.

Zhou, Y., X. Wang, Y. Chen, and Z. Wang. 2014. “ARMlock: Hardware-based Fault Isolation for ARM”. In: *21st ACM Conference on Computer and Communications Security (CCS)*. 558–569.