Chapter 12

Program Rewriting

With *program rewriting*, we ensure that an untrusted program S complies with a given security policy by modifying S before execution. The modifications might cause execution to halt when a security policy violation seems imminent, or they might involve other changes that prevent security policy violations. To avoid adding unnecessary checks, analyzers and optimizers can be used during program rewriting.

Program rewriting can be used to enforce security policies that prohibit certain program states, program actions, or sequences of states or actions. Moreover, with the entire program available for analysis and modification, these security policies can be formulated in terms of that program's abstractions. Analysis of the entire program also allows enforcement of policies where an execution is allowed only if certain other executions are possible.

Untrusted program S can be in a high-level language, assembly language, or machine language. We require only that the program rewriter be able to extract from S the information required for generating the necessary modifications. Program rewriting for machine language binaries is attractive because software is usually distributed in this form. However, information that a program rewriter needs for performing modifications can be difficult to extract from a machine language binary. With machine language binaries, abstractions—such as procedures, objects, and application-specific data structures—may hard to identify, and constants that are addresses (and might need to change when code is added) must be distinguished from constants that are integers (and should not be changed). So most program rewriting is formulated for assembly language or high-level language programs.

12.1 Expectations of Program Rewriters

A *program rewriter* for enforcing a security policy P implements a function $\mathcal{T}_P(\cdot)$ that transforms programs S into programs $\mathcal{T}_P(S)$ satisfying two requirements:

> *Soundness.* $\mathcal{T}_P(S)$ satisfies security policy P.
>
> *Transparency.* If S satisfies security policy P then the runtime environment would not distinguish executions of $\mathcal{T}_P(S)$ from executions of S.

The Soundness requirement should not be a surprise, but the Transparency requirement might be. The Transparency requirement is what enables $\mathcal{T}_P(S)$ to be used as a replacement for S. Typically, the Transparency requirement restricts the modifications that program rewriting can make to S. Limitations on modifications, however, can translate into limitations on what security policies can be enforced by program rewriting.

12.1.1 *Formalization of Soundness and Transparency

Formalizations of the Soundness and Transparency requirements depend on having formalizations of program execution and security policies. Figure 12.1 gives notation we use in those formalizations.

Program Execution. The possible executions of a program S define a set Σ_S of *execution traces*. Each execution trace $\sigma \in \Sigma_S$ is a finite or infinite sequence $s_0\, s_1\, s_2\, \ldots$ of states, where the sequence corresponds to a possible execution of S.

We require some form of program counter to be among the variables mapped by each state. The program counter value in a state s_i of an execution trace $s_0\, s_1\, s_2\, \ldots\, s_i\, s_{i+1}\, \ldots$ defines the instruction or statement that was executed in s_i to produce s_{i+1} in the execution trace.

- For machine language programs, variables mapped by each state give also values for registers and memory locations. An execution trace gives the state before and after each atomic action that executed.
- For high-level language programs, variables mapped by each state also give values for the high-level program's variables. An execution trace gives the state before and after each statement that executed.

In addition, a state might map variables that expose information managed by the runtime environment. For example, the state might map a variable that evaluates to the values being stored on an I/O device.

Security Policies. Security policies are formalized as predicates $P(\cdot)$ on sets of execution traces. Therefore, $P(\Sigma_S)$ holds if and only if S satisfies the security policy that $P(\cdot)$ specifies. The use of such predicates to specify security policies gives considerable expressive power about what a security policy can proscribe.

For s a state:

$s.x$:	the value that state s gives variable x.
$s.Expr$:	the value that state s gives expression $Expr$.
$s\|_V$:	the state that (i) gives the same values to the variables named in set V of variables as state s gives to them, and (ii) gives no values to other variables.
$s=_V s'$:	abbreviates $s\|_V = s'\|_V$

For $s_0\, s_1\, s_2\, \ldots$ an execution trace σ:

$\sigma[i]$:	s_i
$\sigma[..i]$:	$s_0\, s_1\, s_2\, \ldots s_i$
$\sigma[i..]$:	$s_i\, s_{i+1}\, \ldots$
$\sigma\|_V$:	is the execution trace derived from execution trace σ by replacing each state s in σ with state $s\|_V$ and then eliminating consecutive, identical states in that resulting execution trace.
$\sigma=_V \sigma'$:	abbreviates $\sigma\|_V = \sigma'\|_V$

Figure 12.1: Notation for States and Execution Traces

A predicate on sets of execution traces can stipulate that only certain states are allowed in an execution trace, that only certain orderings of states are allowed in an execution trace, and/or that a relation involving two or more execution traces be satisfied for any one of those execution traces to be allowed.

Formalization of Program Rewriter Requirements. The Soundness requirement for a program rewriter $\mathcal{T}_P(\cdot)$ asserts that some intended security policy will be enforced because the set of execution traces for $\mathcal{T}_P(S)$ satisfies the predicate $P(\cdot)$ specifying that intended security policy:

$$\textit{Soundness:}\ \ P(\Sigma_{\mathcal{T}_P(S)}) = \mathit{true} \quad \text{for all programs } S. \tag{12.1}$$

For formalizing the Transparency requirement, we posit that an execution trace $\tau \in \Sigma_{\mathcal{T}_P(S)}$ of $\mathcal{T}_P(S)$ will *resemble* an execution trace $\sigma \in \Sigma_S$ of S when both τ and σ give the same sequences of values to parts of the state that are visible to the runtime environment. Letting Ext be the set of variables containing those parts of the state that are visible to the runtime environment, we would have that τ resembles σ if and only if $\tau =_{Ext} \sigma$ holds. The Transparency requirement for program rewriter $\mathcal{T}_P(\cdot)$ is then formalized as follows.

$$\textit{Transparency:}\ \ P(\Sigma_S) \Rightarrow (\forall \tau \in \Sigma_{\mathcal{T}_P(S)}\colon\ (\exists \sigma \in \Sigma_S\colon\ \sigma =_{Ext} \tau)) \tag{12.2}$$

Notice, if $P(\Sigma_S)$ does not hold for a program S then (12.2) imposes no restrictions on set $\Sigma_{\mathcal{T}_P(S)}$ of possible executions by the modified program. So if

$P(\Sigma_S)$ does not hold, then the Transparency requirement allows $\mathcal{T}_P(S)$ to halt or undertake other enforcement actions—even though such executions of $\mathcal{T}_P(S)$ do not resemble executions of S.

Notice, too, that execution traces σ could not be found to make $\sigma =_{Ext} \tau$ in (12.2) hold if (i) the program rewriter produces $\mathcal{T}_P(S)$ by adding code to S and *Ext* includes the program counter or (ii) the program rewriter produces $\mathcal{T}_P(S)$ by adding variables to S and *Ext* includes those variables. The Transparency requirement thus implies that the behavior of the runtime environment may not be affected by the changes to the program counter or changes to the variables that the program rewriter has added.

12.1.2 Example: Confidentiality of Values in a Set

Elements that finite set *info* stores are assumed to be pairs $\langle v, t\rangle$, where v is some data, t is a tag, and tags have value C (confidential) or U (unrestricted). Tag values are ordered according to $\mathsf{U} \sqsubset \mathsf{C}$, with $\mathsf{U} \sqsubseteq \mathsf{C}$ an abbreviation for $\mathsf{U} \sqsubset \mathsf{C} \ \vee \ \mathsf{U} = \mathsf{C}$. The programs of concern are deterministic. They operate by reading from variable *info* and generating output by writing to a variable *out*. They also read variable *priv* to learn a tag giving the privileges of the user who is executing the program.

The security policy to be enforced is a form of confidentiality: information about whether a pair $\langle v, t\rangle$ is in *info* must not be allowed to affect updates to *out* unless $t \sqsubseteq priv$ holds. For example, *info* might be implementing a directory of files, where each element $\langle v, t\rangle$ of *info* is a directory entry and where tag t indicates whether that file is unrestricted or confidential. The security policy would then prevent a directory-listing program from revealing anything about the confidential files, including whether they are present in the directory.

To be concrete, the program

$$\textbf{if } \langle v, \mathsf{C}\rangle \in info \textbf{ then } out := 1 \textbf{ else } out := 2 \tag{12.3}$$

violates the security policy if executed when $priv = \mathsf{U}$ holds, because the update to *out* reveals whether *info* is storing a pair having tag C but $\mathsf{C} \sqsubseteq priv$ does not hold. The program

$$\begin{array}{l} oldSize := |info|; \ \ info := info \cup \{\langle v, \mathsf{C}\rangle\}; \\ \textbf{if } oldSize \neq |info| \textbf{ then } out := 1 \textbf{ else } out := 2 \end{array} \tag{12.4}$$

also violates the security policy. A set does not change its size as a result of adding an element if that element is already present in the set. Therefore, executing (12.4) in a state satisfying $priv = \mathsf{U}$ updates *out* in a way that reveals whether *info* is storing the pair $\langle v, \mathsf{C}\rangle$.

The predicate $Conf(\cdot)$ formalizing the security policy to be enforced implies that execution traces have the same updates to *out* if their initial states (i) agree on the value of *priv* and (ii) agree on all pairs $\langle v, t\rangle \in info$ satisfying $t \sqsubseteq priv$:

$$Conf(\Sigma_S): \quad (\forall \sigma, \sigma' \in \Sigma_S: \ init(\sigma[0], \sigma'[0]) \ \Rightarrow \ \sigma =_{\{out\}} \sigma') \tag{12.5}$$

where

$$init(s, s'): \quad s.priv = s'.priv \ \wedge \ s.F(info, priv) = s'.F(info, priv)$$

formalizes (i) and (ii) if function $F(set, lvl)$ evaluates to the subset of set containing pairs $\langle v, t\rangle$ satisfying $t \sqsubseteq lvl$:

$$F(set, lvl): \ \{\langle v, t\rangle \mid \langle v, t\rangle \in set \ \wedge \ t \sqsubseteq lvl\}$$

As a sanity test for formalization (12.5), we check whether $Conf(\Sigma_S)$ holds for programs (12.3) and (12.4). And we find that $Conf(\Sigma_S)$ is *false* for each. This is because Σ_S for (12.3) and (12.4) contains execution traces $\sigma, \sigma' \in \Sigma_S$ with initial states that agree on $priv = \mathsf{U}$, agree on subset $F(info, priv)$ of $info$, but disagree on other pairs in $info$. So different values would be written to out in σ and σ', which means that

$$init(\sigma[0], \sigma'[0]) \ \Rightarrow \ \sigma \not\approx_{\{out\}} \sigma'$$

holds. $Conf(\Sigma_S)$ is *false*, as it should be for these non-compliant programs.

Enforcement by Program Rewriting. Violations of $Conf(\cdot)$ can occur only if executions can start in a state satisfying $F(info, priv) \neq info$. We thus explore program rewriters to produce output programs $\mathcal{T}_{Conf}(S)$ that start executing S in states where $F(info, priv) = info$ holds.

The obvious approach would be for a program rewriter to embed S in an **if** statement that checks whether $F(info) = info$ holds.

$$\mathcal{T}_{Conf}(S): \quad \textbf{if } F(info, priv) = info \textbf{ then } S \textbf{ else halt} \tag{12.6}$$

This approach is flawed, though. The Soundness requirement is not satisfied by (12.6), because $Conf(\Sigma_{\mathcal{T}_{Conf}(S)})$ does not necessarily hold if (as we show below) there is an initial state satisfying $F(info, priv) = info$ where execution of input program S updates out in any way.

Let S be the program given in (12.3). An execution of (12.6) in an initial state satisfying $priv = \mathsf{U} \ \wedge \ F(info, priv) \neq info$ will halt without performing any updates to out, but an execution of (12.6) in an initial state satisfying $priv = \mathsf{U} \ \wedge \ F(info, priv) = info$ will store 2 in out. The two execution traces have different updates to out. So the final value of out reveals that $\langle v, \mathsf{C}\rangle \notin info$ holds even though $\mathsf{C} \not\sqsubseteq priv$ holds. Thus, executing the rewriter's output when S is (12.3) violates the requirement that a pair $\langle v, t\rangle$ stored by $info$ not affect updates to out unless $t \sqsubseteq priv$ holds—the program produced by the rewriter did not enforce the security policy. Formally, we have shown there exist execution traces $\sigma, \sigma' \in \Sigma_{\mathcal{T}_{Conf}(S)}$ where

$$init(\sigma[0], \sigma'[0]) \ \Rightarrow \ \sigma \not\approx_{\{out\}} \sigma'$$

holds, falsifying the universal quantification in $Conf(\Sigma_{\mathcal{T}_{Conf}(S)})$, which means the program rewriter does not satisfy formal defintion (12.1) of the Soundness requirement.

A different approach for ensuring that S is executed only in states that satisfy $F(info, priv) = info$ is to first remove from *info* all pairs having a tag t satisfying $t \notin priv$.

$$\mathcal{T}_{Conf}(S): \quad info := F(info, priv);\ S \tag{12.7}$$

To establish that program rewriter (12.7) satisfies the Soundness requirement, we must prove that $Conf(\Sigma_{\mathcal{T}_{Conf}(S)})$ holds for every program S. Definition (12.5) of $Conf(\cdot)$ implies that to prove $Conf(\Sigma_{\mathcal{T}_{Conf}(S)})$, we must prove $\sigma =_{\{out\}} \sigma'$ for any execution traces $\sigma, \sigma' \in \Sigma_{\mathcal{T}_{Conf}(S)}$ satisfying $init(\sigma[0], \sigma'[0])$. From $\mathcal{T}_{Conf}(S)$ definition (12.7), we have that σ is $s\tau$ and σ' is $s'\tau'$ for $\tau, \tau' \in \Sigma_S$. So to prove $\sigma =_{\{out\}} \sigma'$, it suffices to prove $\tau =_{\{out\}} \tau'$, because the first statement in $\mathcal{T}_{Conf}(S)$ is an assignment that does not update *out* and, therefore, $s\tau =_{\{out\}} s'\tau'$ follows from $\tau =_{\{out\}} \tau'$.

From $init(s, s')$, we conclude that (i) all variables except *info* agree on their values in s and s', and (ii) $s.F(info, priv) = s'.F(info, priv)$. The assignment statement at the beginning of $\mathcal{T}_{Conf}(S)$ thus causes $\tau[0].info = \tau'[0].info$ to hold. That means $\tau[0]$ and $\tau'[0]$ agree on all variables (including *info*). The assumption that S is deterministic then implies that the execution of S starting from states $\tau[0]$ and $\tau'[0]$ will be identical, so $\tau = \tau'$ holds and $\tau =_{\{out\}} \tau'$ does too.

We show that program rewriter (12.7) satisfies the Transparency requirement by proving that (12.2) holds. Assume antecedent $Conf(\Sigma_S)$ holds and let τ be any execution trace in $\Sigma_{\mathcal{T}_{Conf}(S)}$. We must show there exists some execution trace σ satisfying $\sigma \in \Sigma_S$ and $\sigma =_{\{out\}} \tau$. By construction, τ has the form $s_\tau \sigma_\tau$ where s_τ is a state and $\sigma_\tau \in \Sigma_S$ holds. The assignment statement at the beginning of $\mathcal{T}_{Conf}(S)$ does not update *out* and, therefore, $\sigma_\tau =_{\{out\}} s_\tau \sigma_\tau$ holds. Since $\tau = s_\tau \sigma_\tau$ we conclude $\sigma_\tau =_{\{out\}} \tau$ holds. So σ_τ is the witness needed for showing the existence of an execution trace σ satisfying $\sigma \in \Sigma_S$ and $\tau =_{\{out\}} \sigma$.

12.2 Software-Based Fault Isolation (SFI)

Sandboxing is a form of isolation. It prevents an untrusted program from corrupting memory outside of a specified *data region*. Addresses within the data region are *legal* for `store` instructions; addresses outside that region are *illegal*. Here is an example sandboxing policy.

> **64K Region Sandboxing Policy.** Do not execute `store` instructions that would update memory outside of the 64K data region that starts at address `0x`α`0000`[1] and ends at address `0x`α`FFFF`, where α is fixed and satisfies `0x0000` $\le \alpha \le$ `0xFFFF`. □

Such a policy would be useful, for example, when untrusted software is extending some system but, to keep overheads low, the extension is not run in

[1] The prefix "`0x`" indicates a hexadecimal value. A hexadecimal digit `0`, `1`, ..., `9`, `A`, `B`, ..., `F` conveys the value of a 4-bit binary number.

a different hardware memory segment. *Software-based fault isolation* (SFI) enforces instances of the above sandboxing policy by using a program rewriter to modify assembly language programs. The program rewriting adds code that substitutes for the checking that would have been done if a separate hardware segment was used for the extension.

Static analysis of an assembly language program S can enforce the above sandboxing policy for a `store` instruction that uses a symbolic label to identify the memory location that will be updated. A program rewriter can scan S to locate such `store` instructions. For each `store` instruction found, the program rewriter (i) locates the assembly language program's declaration for the symbolic label identifying the memory that the `store` instruction will update and (ii) checks that this memory location is being allocated within the data region.

But not all `store` instructions will use symbolic labels to identify the memory to be updated. A `store` instruction could, instead, use a register that contains the address of the memory location to update. The `store` instruction[2]

```
store  r1,[r2]          ; [r2]:=r1                    (12.8)
```

writes the value contained in register `r1` to the memory location at the address contained in register `r2`. To enforce a sandboxing policy for `store` instruction (12.8), we must ensure that the address in register `r2` is within the data region. Static analysis cannot be used for this, since determining the value `r2` (or any variable) is undecidable in general.

However, runtime checks inserted by a program rewriter can be used to ensure that the value of `r2` is a legal address. For the above 64K Region Sandboxing Policy, the straightforward approach is to insert checking code before the `store`:

```
cmp   r2, 0xα0000   ; compare r2 with data region lower bound
jl    error         ; if r2 < 0xα0000 then goto error
cmp   r2, 0xαFFFF   ; compare r2 with data region upper bound
jg    error         ; if 0xαFFFF < r2 then goto error
store r1, [r2]      ; [r2]:=r1
```

The modified program transfers control to `error` instead of violating the sandboxing policy.[3]

This program rewriting, however, depends on an assumption about control flow—that the `store` instruction is executed immediately after the inserted instructions have been executed. That assumption is not sound if an attacker can transfer control directly to the `store` instruction or into the middle of the inserted instructions. And an attacker could orchestrate such a control transfer by

[2] Assembly language instructions use the syntax "[r]" to indicate an operand stored at the address in register `r`.

[3] We are ignoring that processor status registers might be read after executing the `store`, having been set by the `store` or by some previous instruction. A program rewriter can use static analysis to detect such uses of processor status registers. So we assume that, when necessary, the program rewriter also generates the necessary code to save and restore those values.

using a `jmp` or `call` instruction where the destination is given in a register that the attacker can load; a `return` instruction also could be used if the attacker can overwrite the top of the runtime stack (where the return address is presumably stored). Although only `jmp`, `call`, and `return` instructions are mentioned here, the defense given below will be effective against attacks involving other control transfer instructions, too.

One way to defend against control-flow attacks for bypassing code that a program rewriter adds, is to ensure that a program's flow of control cannot diverge from predefined paths; a program rewriter for enforcing that security policy is the subject of §12.3. Here, we discuss an alternative. It replaces a `store` instruction (12.8) with code that satisfies:

SFI$_1$: If `r2` contains a legal address *addr* and if a normal control flow is followed to reach `store` instruction (12.8) then the memory at *addr* will be updated.

SFI$_2$: If `r2` does not contain a legal address or if a normal control flow is not followed to reach `store` instruction (12.8) then the memory location that is updated nevertheless will be located somewhere in the data region.

SFI$_2$ defends against updates to illegal addresses by `store` instructions in the original program. SFI$_2$ also defends against an attacker setting the value of `r2` and then transferring control directly to a `store` instruction or into the middle of the inserted code that precedes a `store` instruction. Also, rather than halt execution, SFI$_2$ redirects attempts to update illegal addresses, resulting in updates to unexpected locations in the data region. Because a program might continue executing from that corrupted state, SFI$_2$ can make program debugging quite difficult.

SFI$_1$ and SFI$_2$ each stipulate that an update be performed to some memory location that is legal and, thus, the update is to an address with α for its high-order bits. The following instruction sequence executes its `store` ι_4 only after constructing such addresses, using the low order bits of the address originally in register `r2`. The constructed address appears in a register `a0`, which (as discussed below) we assume is (i) available for use by code that the program rewriter inserts and (ii) is never updated by the original program code.[4]

```
ι1: load  a0, r2              ; a0 := r2
ι2: and   a0, 0x0000FFFF      ; extract low-order address bits
ι3: or    a0, 0xα0000         ; use α for high-order address bits
ι4: store r1, [a0]            ; [a0]:=r1
```

Instruction ι_2 zeros the high-order bits in register `a0`, leaving the low-order bits unchanged. Instruction ι_3 then sets the high-order bits in register `a0` to α,

[4]Instructions provided by some processors can be used to write shorter instruction sequences that have the same effect as the code we give. For example, x86 and MIPS provide a `load` instruction that changes only the high-order bits of a register. If `a0` contains a legal address, then such a load can replace ι_2 and ι_3.

without changing the low-order bits in that register. Therefore, if execution starts at ι_1 and register `r2` contains a legal address *addr* then register `a0` will contain *addr* when `store` instructions ι_4 is executed. So requirement SFI_1 is satisfied.

To establish that SFI_2 holds for instruction sequence ι_2 through ι_4, we must show (among other things) that the `store` instruction at ι_4 updates a legal address even when control transfers directly to instruction ι_2, ι_3, or ι_4. That possibility leads us to posit (returning to its enforcement, below):

> **Register `a0` Restriction.** Register `a0` is initialized with a legal address, and `a0` is never updated by the original program. □

We can use a case analysis to establish that SFI_2 holds under this restriction.

> *Case 1: Executions that start at ι_1 where `r2` contains an address that is not legal.* The subsequent execution of ι_3 changes the contents of register `a0` to have high-order bits α, which means that the `store` instruction at ι_4 performs a legal update, as required for SFI_2.
>
> *Case 2: Executions that start at instruction ι_2, ι_3, or ι_4.* Due to Register `a0` Restriction, execution that starts at instruction ι_4 is a legal update because `a0` contains a legal address. If, instead, execution starts at instruction ι_3 then the address ι_3 stores into `a0` remains legal (because only the low-order bits were changed by executing ι_3), and instruction ι_4 will be a legal update. Finally, executions that start at ι_2 cause `a0` to contain a legal address because ι_3 then produces a legal address by setting the high-order bits.

Register `a0` Restriction can be enforced by a compiler or by a program rewriter. Use of a program rewriter seems preferable.[5]

We have assumed that an instruction will be executed only if it appears in code that was output by the program rewriter. That assumption is not valid if control transfers are possible to addresses in the data region, since bit strings in the data region can be interpreted as instructions. However, the assumption we are making cannot be violated if certain restrictions are enforced:

R1: The program rewriter's output is stored as a separate *code region*.

R2: No writes to the code region are allowed during program execution.

R3: Only instructions in the code region are ever allowed to execute.

R1 can be subtle to enforce. Operating systems typically support allocation of memory regions having only certain sizes (usually, powers of 2), and values stored in uninitialized memory locations can be interpreted as instructions. To

[5] A compiler would enforce Register `a0` Restriction by generating code that does not use register `a0`. However, enforcement of the restriction with a complier precludes executing code that has not been generated by a compliant compiler, which is problematic because software is typically not distributed in source code form.

satisfy R1, a program rewriter's output must fill all of the memory serving as the code region. A program rewriter can fill any unoccupied memory by copying into each byte a bit pattern that causes the processor to halt. An attacker cannot benefit by performing a branch to one of those locations.

For enforcing R2, writes to the code region would be prevented if the data region and the code region have disjoint addresses, because then `store` instructions cannot update memory locations in the code region.

To enforce restriction R3 when (as we have been assuming) the only control transfer instructions are `jmp`, `call`, and `return`, we engage in further program rewriting. First, we use a program rewriter to replace each `call` and `return` instruction with code that uses a `jmp` instruction for implementing control transfers.

- For a `call` instruction, the replacement code would (i) push a return address onto the runtime stack, (ii) load a register with the address of the procedure being invoked, and then (iii) execute a `jmp` instruction to effect the transfer of control.

- For a `return` instruction, the replacement code would (i) pop the return address from the runtime stack into a register and then (ii) execute a `jmp` instruction that transfers control to that return address.

That leaves `jmp` as the only remaining control transfer instruction. So we next turn attention to R3 for those control transfers.

Static analysis can be used to enforce restriction R3 for `jmp` instructions that use a symbolic label to name the destination—the program rewriter checks that the symbolic label is associated with an address in the code region. However, static analysis cannot enforce restriction R3 for a `jmp` instruction like

```
jmp [r2]
```
(12.9)

that transfers control to the instruction at the address contained in register `r2`.

Ensuring that the contents of `r2` in `jmp` instruction (12.9) is the address of an instruction in the code region is just like ensuring that the memory updated by `store` instruction (12.8) is a location in the data region. So we investigate inserting code before `jmp` instructions to ensure that restriction R3 holds. Analogous to SFI_1 and SFI_2 would be:

SFI_3: If `r2` contains address *dest* of an instruction in the code region and if normal control flow is followed to reach `jmp` (12.9) then control is transferred to execute the instruction at *dest*.

SFI_4: If `r2` does not contain the address of an instruction in the code region or if normal control flow is not followed to reach `jmp` (12.9) then control is transferred to execute some instruction in the code region.

To be concrete, we give replacement code for `jmp` instruction (12.9) in conjunction with a 64K byte code region starting at address `0x`β`0000` and ending

at address `0x`β`FFFF` for fixed β, with `0x0000` $\leq \beta \leq$ `0xFFFF` and $\alpha \neq \beta$.[6] To start, assume that all instructions are 4 bytes long and that each instruction starts at an address evenly divisible by 4. So a *legal instruction address* will have β for its high-order bits and 00 for its two lowest-order bits. We relax the assumption about fixed instruction lengths below.

The replacement code for `jmp` instruction (12.9) resembles the replacement code for `store` instruction (12.8), except that a differnt fresh register `a1` is used instead of register `a0`. The mask in ι_2^J for extracting the low-order address bits ensures that a legal instruction address will be constructed (because the address will be divisible by 4), and the mask in ι_3^J uses β for adding high-order address bits to locate the constructed address in the code region.

```
ι1^J: load  a1, r2              ; a1 := r2
ι2^J: and   a1, 0x0000FFFC      ; low-order address bits divisible by 4
ι3^J: or    a1, 0xβ0000         ; use β for high-order address bits
ι4^J: jmp   [a1]                ; load a1 into program counter
```

To guarantee that SFI_4 holds, it suffices that register `a1` contain the address of some instruction in the code region whenever replacement code for a `jmp` instruction is not executing.

> **Register `a1` Restriction.** Register `a1` is initialized with the address of some instruction in the code region, and `a1` is never updated by the original program. □

This restriction can be satisfied by using a compiler or a program rewriter in the same way that the analogous restriction is enforced for values in register `a0`.

Different Length Instructions. The assumption that all instructions are the same length does not hold for the x86 architecture or for most other CISC computers. On these computers, an instruction might start at any byte in the code region, but not every byte in the code region is the start of an instruction. In addition, since a byte in the middle of one instruction can be interpreted as starting a different instruction, a sequence of bytes in memory might be parsed as different sequences of instructions, depending on where you start. An attacker who can transfer control to an arbitrary byte in memory would be able to execute those "hidden" sequences of instructions. So to relax the assumption made earlier that all instructions are 4 bytes long, we must ensure that not only is the destination of a `jmp` instruction located in the code region but that destination is located at the start—and not in the middle—of an instruction that the program rewriter output.

If legal `jmp` destinations are not all divisible by 4 (as was assumed), then mask `0x0000FFFC` used above in ι_2^J no longer works for constructing legal destinations for control transfers. Note, however, that mask would work even on computers with varying length instructions if (i) the intended destinations for all control transfers happen to be divisible by 4, and (ii) every code region address

[6] By choosing a value for β that satisfies $\alpha \neq \beta$, we satisfy R2.

divisible by 4 happens to be the location for an instruction that was output by the program rewriter.

Generalizing, legal destinations for control transfers can be constructed if ι_2^J above uses a mask that ensures divisibility by 2^m provided that the following conditions hold for programs in the code region.

VLI$_1$: All intended destinations for control transfers are divisible by 2^m.

VLI$_2$: Any code region address that is divisible by 2^m is the start of an instruction that was output by the program rewriter.

Mask `0x0000FFFC` generates addresses divisible by 2^2 because the 2 low-order bits are set to 00, mask `0x0000FFF8` generates addresses divisible by 2^3 because the 3 low-order bits are set to 000, and so on.

There is no reason that VLI$_1$ and VLI$_2$ would hold for programs written in an assembly language where instructions have different lengths. But by inserting 1-byte `nop` instructions into such a program, a program rewriter can relocate instructions in that program to make VLI$_1$ and VLI$_2$ hold, with no change to the sequence of values that the modified program reads and writes to registers and memory. Thus, executions of the modified program are not materially different from executions of the original program.

In what follows, we use the term *chunk* to refer to a $2^m - 1$ byte sequence that begins at an address divisible by 2^m. VLI$_1$ is satisfied if each instruction that is the destination for a control transfer gets relocated to the start of some chunk. We assume that the list of destinations for control transfers is given in an assembly language program and, thus, the list is available to the program rewriter. This assumption does not preclude an assembly language program from performing calculations to generate the address of a destination—it merely requires the programmer to have anticipated and declared all possible outcomes of those calculations.[7] Note, also, that the assumption of having such a list of destinations is compatible with information that compilers typically output for use by debuggers.

Relocation to satisfy VLI$_1$ would then proceed as follows.

- Many assemblers include m-alignment directives for small integer values of m. Placed before an instruction ι, the m-alignment directive causes `nop` instructions to be inserted before ι so that ι will be located at a 2^m-byte boundary. To satisfy VLI$_1$, a program rewriter can simply insert instances of this assembler directive just before any instruction that is the destination for a control transfer.

- Execution of a `return` instruction (or its simulation in terms of `jmp`) tranfers control to the location following a `call` instruction (or its simulation in terms of `jmp`). For the destination of a `return` instruction to be at the start of a chunk, the corresponding `call` instruction (or its simulation in

[7] An attempt to transfer control to a location not on that list is considered a branch to an illegal instruction address and will branch to the start of some chunk.

terms of `jmp`) must be located at the end of the preceding chunk. A program rewriter can perform that relocation by inserting `nop` instructions to move each `call` (or its simulation) to the end of a chunk.

To satisfy VLI_2, the program rewriter can check for instructions that span two chunks. By inserting the assembler's alignment directive just before each instruction ι that is found, the program rewriter causes the assembler to do the necessary relocation of ι for VLI_2 to be satisfied.

12.3 Control-Flow Integrity (CFI)

Virtually all attacks change the control flow for the program that is being subverted. The changes enable the attack to execute new code that the attack injects, execute existing code in an unexpected context, avoid checks in existing code, or circumvent code that a program rewriter had added.[8] Enforcement of *control-flow integrity* (CFI) defends against such attacks by restricting transfers of control to specified locations. Code to enforce CFI can be added by a program rewriter.

Most modern processors offer a way to prevent attacks from altering the code that is being executed. Code is stored in a code region, data is stored in a disjoint data region, and memory is configured to enforce the following restrictions.

Non-writable Code. The code region cannot be updated at runtime.

Non-executable Data. The data region cannot be executed at runtime.

However, if—as is prudent—we cannot rule out the presence of vulnerabilities in software, then we cannot rule out writes to the data region that change the destination for a control transfer instruction ι by changing a value used by ι as an operand. CFI can be violated if that new destination is not among those specified as being legal.

12.3.1 CFI Enforcement by Program Rewriting

For illustration, we (again) consider a machine language in which `jmp`, `call`, and `return` are the only control transfer instructions. A control transfer instruction is defined to be *vulnerable* if its destination can be changed by altering a value stored in a register or in the data region. In our hypothetical (but realistic) machine language, `jmp` or `call` instructions are vulnerable if the the branch destination is given in a register[9] and every `return` instruction is vulnerable

[8] SFI restrictions R2 and R3 (page 359) are not sufficient for defending against attacks that only execute existing code already present in the code region. So SFI does not defend against attacks that execute existing code in an unexpected context, avoid checks in that existing code, or circumvent code that a program rewriter had added.

[9] On modern processors, the branch destination for a `jmp` or `call` instruction is given either in a register or as a field in the instruction representation. We reason as follows to establish

because (we assume that) the stack holding return addresses is stored in the data region.

CFI enforcement for machine language programs requires having (i) the list of legal destinations for each of the vulnerable control transfer instructions, and (ii) a means for blocking other control transfers. To implement (ii), we essentially assign "colors" to the legal destinations as well as to the vulnerable control transfer instructions. And we halt execution of any control transfer instruction if its assigned color does not match the assigned color of the destination.

Such a scheme can be implemented if new instructions `CFIlab` and `CFIjmp` are the only instructions used to cause transfers of control. Each `CFIlab` and `CFIjmp` instruction has an *immediate operand*, which is a value that is incorporated into the instruction's representation and corresponds to the "color" discussed above. An immediate operand cannot be altered by an attack, due to the Non-writable Code restriction.

`CFIlab` *destid*	a form of `nop` instruction that is a destination for control transfers from `CFIjmp` instructions. Immediate operand *destid* is called the *destination identifier* for this destination.
`CFIjmp` *jmpid*, [*r*]	if a `CFIlab` instruction ι is stored at the memory address in register r, ι has destination identifier *destid*, and *destid* = *jmpid* holds for immediate operand *jmpid* then execution transfers control to ι; otherwise, execution halts.

The destination identifiers specify the legal destinations for control transfers, and the operation of `CFIjmp` and `CFIlab` instructions prevent branches to destinations that are not legal. So both elements necessary for CFI enforcement—a list of legal destinations and a means to block control transfers to other destinations—are present when machine language programs use `CFIjmp` and `CFIlab` instructions in place of vulnerable control transfer instructions.

A program rewriter can be configured to translate a machine language program that uses vulnerable control transfer instructions into a machine language program that instead uses `CFIjmp` and `CFIlab` instructions. As an example, con-

---

that a `jmp` or `call` instruction ι is not vulnerable if the branch destination is given as a field in the instruction representation. To define or modify a field in an instruction ι, an attack must write the memory containing ι. For that write to succeed, the Non-writable Code restriction requires that ι be stored in the data region, but the Non-executable Data restriction then would prevent ι from being executed. Because ι cannot be executed, it cannot transfer control to the modified branch destination, which is why ι is not a vulnerable control transfer instruction.


```
          ⋮
        lea     r3, L1             ; r3 := address(L1)
        push    r3                 ; return address onto runtime stack
        lea     r3, subr           ; r3 := address(subr)
        CFIjmp  0x12345678, [r3]   ; implement call [r3]
  L1 :  CFIlab  0x87654321         ; destination for return
          ⋮
 subr:  CFIlab  0x12345678         ; destination for call
          ⋮
        pop     r3                 ; pop return address into r3
        CFIjmp  0x87654321, [r3]   ; implement return
```

Figure 12.2: Translation of `call` and `return`

sider the following code excerpt, showing an invocation of a subroutine `subr`.

```
          ⋮
        lea   r3, subr    ; r3 := address(subr)
        call  [r3]        ; invoke subr
          ⋮                                          (12.10)
  subr :                  ; destination for call
          ⋮
        return
```

The `call` instruction in this excerpt is a vulnerable control transfer instruction because register `r3` holds the destination, and the `return` instruction is a vulnerable control transfer instruction because its destination comes from the runtime stack.

Figure 12.2 gives a translation for code excerpt (12.10) in terms of `CFIjmp` and `CFIlab` instructions. In this translation, executing the caller's `CFIjmp` instruction transfers control to the `CFIlab` instruction at the start of `subr`, because both instructions have the same destination identifier (`0x12345678`). And executing the `CFIjmp` instruction at the end of `subr` transfers control back to the caller by branching to the `CFIlab` instruction at `L1`, because the address of `L1` was saved on the runtime stack before `subr` started executing and because these `CFIjmp` and the `CFIlab` instructions both have the same destination identifier (`0x87654321`).

Notice, the `CFIjmp` instruction at the end of `subr` in Figure 12.2 forces the same destination identifier (`0x87654321`) to be used as the destination identifier in the `CFIlab` instruction that is part of the translation for every `call subr` instruction. There might be many. Control nevertheless does transfer back to the correct `CFIlab` instruction in the caller—the runtime stack is storing the return address for the invocation, and the `pop` instruction retrieves that address.

The translations for `call` and `return` given in Figure 12.2, however, can be subverted if an attack alters the return address on the runtime stack by writing

to the data region. The `CFIjmp` implementing `return` will not halt execution if the altered return address on the runtime stack is for a `CFIlab` instruction ι having destination identifier `0x87654321` but ι is part of a different call site. The `CFIjmp` then causes a control transfer to the end of the wrong call site. The problem arises because we are specifying legal control flows by using statically defined destination identifiers. Statically defined destination identifiers cannot specify that a control transfer is legal only when certain execution has preceded that control transfer. So the use of statically defined destination identifiers forces us to give an overly permissive specification of legal control flows, which attackers can exploit to cause control transfers that were not intended by a programmer.

Approaches do exist to avoid overly permissive specifications of legal control flows or to add checks that prevent their exploitation.

- *Have a separate copy of the subroutine for each* `call`. The copies allow unique destination identifiers to be used in each of the two control transfers that are implementing a `call` and the associated `return`. So each subroutine copy returns to one call site. This defense is equivalent to inlining the subroutine body, so it doesn't work for recursive subroutines and it expands the length of the code.

- *Implement the call stack in a memory region that attacks cannot update.* Either a separate SFI-protected memory region or a hardware-protected memory segment might be employed to store such a call stack. A program rewriter would insert code to ensure that updates to the call stack are performed only by the `push` and `pop` instructions that the program rewriter inserts for implementing `call` and `return`.

12.3.2 Translating `CFIlab` and `CFIjmp` Instructions

Modern processors do not offer `CFIlab` and `CFIjmp` instructions. But the effects of `CFIlab` and `CFIjmp` instructions can be achieved by using a progam rewriter to translate `CFIlab` and `CFIjmp` instructions into equivalent sequences of the available instructions.

The translations discussed below assume that the following assumption holds for the machine language program being translated.

> **Uniqueness of Destination Identifiers.** No destination identifier in a `CFIlab` or `CFIjmp` instruction appears as a bit string elsewhere in the machine language program being translated. □

When a machine language program does not satisfy this assumption, then a program rewriter can be used to substitute new destination identifiers into the `CFIlab` and `CFIjmp` instructions, preserving equalities. Such a replacement is always possible if destination identifiers are B-bit strings (typically $B = 32$) and the code region contains fewer than 2^B words, since all B-bit strings cannot be present in a code region that is smaller than 2^B words.


```
        ⋮
    lea     r3, L              ; r3 := address(L)
    CFIjmp  0x12345678, [r3]   ; transfer control to L;
        ⋮
    halt
L:  CFIlab  0x12345678         ; branch destination
    inst    x, y               ; some instruction
        ⋮
```

Figure 12.3: Code using `CFIlab` and `CFIjmp`

Translations for `CFIlab` and `CFIjmp` instructions must ensure that attacks cannot alter destination identifiers. The Non-writable Code restriction helps, if destination identifiers are stored in the code region. Therefore, in the translation, destination identifiers are stored either as data in the code region or as immediate operands in instructions.

For example, the machine language program excerpt in Figure 12.3 might be translated to the code in Figure 12.4. There, a `CFIlab` instruction having destination identifier *destid* is translated into a data value *destid* stored in the code region. That means `jmp` instruction ι_3 in the translation of the `CFIjmp` instruction must transfer control to the instruction positioned in memory after `L` (instead of at `L`), as if the memory location at address `L` stores a `nop` instruction. The semantics of `CFIjmp` imply that `jmp` instruction ι_3 should be reached—causing the transfer of control to occur—only if the data value stored at `L` matches immediate operand `0x12345678` in `cmp` instruction ι_1. The immediate operand in `cmp` instruction ι_1 is copied from the immediate operand in the `CFIjmp` instruction that ι_1 through ι_3 replace.

The translation in Figure 12.4 has a vulnerability, though. Destination iden-

```
        ⋮
    lea     r3, L              ; r3 := address(L)
; translate CFIjmp 0x12345678, [r3]
ι1: cmp     [r3], 0x12345678   ; compare CFIlab and CFIjmp dest idents
ι2: jne     error              ; wrong CFIlab dest ident found
ι3: jmp     [r3 + 4]           ; transfer control to correct destination
        ⋮
    halt
; translate L: CFIlab 0x12345678
 L: data    0x12345678         ; dest ident for CFIlab
    inst    x, y               ; some instruction
        ⋮
```

Figure 12.4: Translation of `CFIlab` and `CFIjmp`


```
        ⋮
    lea     r3, L               ; r3 := address(L)
; translate CFIjmp 0x12345678, [r3]
ι'1: load   r0, 0x12345677      ; used to compute dest ident for CFIjmp
ι'2: add    r0, 0x00000001      ; dest ident for CFIjmp
ι'3: cmp    r0, [r3]            ; compare CFIlab and CFIjmp dest idents
ι'4: jne    error               ; wrong CFIlab dest ident found
ι'5: jmp    [r3 + 4]            ; transfer control to correct destination
        ⋮
    halt
; translate L: CFIlab 0x12345678
 L: data   0x12345678           ; dest ident for CFIlab
    inst    x, y                ; some instruction
        ⋮
```

Figure 12.5: Alternative Translation of `CFIlab` and `CFIjmp`

tifiers for `CFIlab` instructions are stored in a way that is indistinguishable from values that are stored for the translations of `CFIjmp` instructions, yet only the former mark legal branch destinations. An attack might load `r3` with the memory address of the immediate operand in the representation of ι_1 and then transfer control to ι_1. Execution of `cmp` instruction ι_1 then would be comparing its immediate operand with its immediate operand. Those are the same values, so `jmp` instruction ι_3 will be reached and will transfer control to the instruction located in memory 4 bytes after the address contained in `r3`. Consequently, `jne` instruction ι_2 would be executed and, since the previous execution of a `cmp` instruction found equal values, the `jne` instruction will not transfer control to `error`. Therefore, ι_3 will be reached, and execution of ι_3 will transfer control to `jne` instruction ι_2 because ι_2 is the instruction in memory at address $[r3 + 4]$. Based on the status flags set by the last execution of `cmp` instruction ι_1, instruction ι_2 will not not transfer control to `error`, so ι_3 is again executed and again transfers control to ι_2. The result is a loop that never terminates, involving an illegal branch destination.

We prevent such attacks if the translation of `CFIlab` and `CFIjmp` distinguishes destination identifiers used in `CFIjmp` instructions from destination identifiers used in `CFIlab` instructions. One approach is for translations of `CFIjmp` instructions to calculate at runtime the destination identifier rather than storing that value in the code region. Figure 12.5 illustrates the improved translation for `CFIjmp`, assuming register `r0` is can be used by instructions ι'_1 through ι'_3. The attack to subvert the translation in Figure 12.4 no longer works. It fails because loading register `r3` with the address of the instruction ι'_1 immediate operand and then branching to `cmp` instruction ι'_3 always detects unequal destination identifiers, since immediate operand value `0x12345677` is not equal to the destination identifier stored at the address in register `r3`. So executing `jne` instruction ι'_4 transfers control to `error`, as desired.

The translations we have given for `CFIjmp` and `CFIlab` instructions often can be improved by exploiting unique features in a given processor's instruction set. The basic approach used for the translation is likely to be the same across all instruction sets, though, so we summarize it here.

- A `CFIjmp` instruction is translated into a sequence of instructions that checks if the correct destination identifier is present in the translation of a `CFIlab` instruction at the branch destination.
- The destination identifier for a `CFIlab` instruction is stored as a constant in the code region or as an immediate operand in an instruction. Either way, it is being stored in the code region and, therefore, attacks cannot update the value.
- The destination identifier for the `CFIjmp` instruction is stored in a way that it cannot be altered by attacks and it does not resemble the destination identifier for a `CFIlab` instruction.

12.4 Inlined Reference Monitors (IRM)

Modifications that a program rewriter makes to an untrusted program S can be designed to halt execution just before S is about to violate any security policy that a reference monitor can enforce.[10] The effect is the same as having a reference monitor present in the runtime environment—we have created an *inlined reference monitor* (IRM).

To implement an inlined reference monitor, a program rewriter is provided with a specification that gives the elements[11] of a reference monitor:

- monitor state,
- monitored accesses, and
- the monitor response associated with each monitored access.

Monitored accesses are actions by untrusted program S that pause execution of S and transfer control to the associated monitor response. A monitored access might immediately precede a reference by S to a specific operand or it might immediately precede execution by S of a specific operation. A monitor response is code that changes the monitor state and/or halts execution of S.

Specifying an Inlined Reference Monitor. A specification for an inlined reference monitor specification will give a set of declarations that define the monitor state and a set of *clauses* that define the monitored accesses along with associated monitor responses. We might use the syntax

$$\{\mathsf{pattern}\}\ \ \mathsf{code} \tag{12.11}$$

[10] See §11.3 for a characterization of security policies that reference monitors can enforce.
[11] See §11.1 for a detailed discussion of these elements.

var *inCalls* : **integer intial** 0
{**before:** `call`} **if** *inCalls* < 6 **then** *inCalls* := *inCalls* + 1
else halt
{**before:** `return`} **if** *inCalls* > 0 **then** *inCalls* := *inCalls* − 1
else halt

Figure 12.6: Inlined Reference Monitor for `call` and `return`

for a clause that asserts monitor response code should be executed when execution of the untrusted program S reaches a control point matching pattern. So pattern is defining a set of monitored accesses.

As an illustration, Figure 12.6 uses a hypothetical language to specify an inlined reference monitor that ensures (i) the call stack does not overflow because more than 6 procedure calls are in progress, and (ii) each `return` is executed in one-to-one correspondence with a previously executed `call`.[12] The specification begins with a declaration defining the monitor state to be an integer variable *inCalls*, initially 0. In the clauses that follow, pattern "**before:** T" creates monitored accesses for execution that reaches a control point appearing at the start of an instance of T. So the reference monitor being specified receives control immediately before execution of a `call` or a `return`. The program rewriter would (i) before each `call`, add code that checks *inCalls* < 6 and either increments *inCalls* or halts execution, and (ii) before each `return`, add code that checks *inCalls* > 0 and either decrements *inCalls* or halts execution.

The patterns available for defining monitored accesses will depend on the language of programs that the program rewriter is intended to modify. A program rewriter for incorporating reference monitors into assembly language programs would support patterns to match opcodes and operands, since those are the constituents of assembly language statements. For high-level language programs, a program rewriter would support patterns for matching facets of the various statements and expressions of that high-level language.

Any programming language can be used for writing the monitor responses. However, if code in (12.11) is not in the same programming language as the untrusted program S being modified then code must be translated to that programming language. Also, access to variables and control points defined in S is facilitated from within code when both are written in the same programming language.

[12] This reference monitor can be used to defend against certain control-flow hijacking attacks. For example, with return-oriented programming (ROP) attacks, a branch to some desired address is implemented by pushing that address onto the call stack and then executing a `return` instruction. To cause execution of a sequence of code segments that each end with a `return`, an attack pushes a corresponding sequence of code addresses onto the call stack and executes a single `return`; the `return` that ends each code segment will transfer control to the next code segment in the sequence.

Protecting IRM Integrity. Different approaches are required for protecting the integrity of an inlined reference monitor that has been incorporated into an assembly language program than one that has been incorporated into a high-level language program. So we consider these cases separately.

For programs written in assembly language, there are two ways to subvert an inlined reference monitor. The untrusted program might (i) corrupt the monitor state or (ii) branch into or around the code segments that were added to detect monitored accesses or implement monitor responses. Both forms of subversion are prevented if the program rewriter used to install the inlined reference monitor also makes additional modifications.

- Use SFI to protect the integrity of the monitor state. The state of the original untrusted program would be stored in one data region and the monitor state would be stored in a disjoint data region.
- Use CFI to protect the integrity of the control flow.

For protecting the integrity of inlined reference monitors that have been incorporated into a high-level language program, it suffices for the programming language to have a suitable type system. The type system would be used to prevent subversion of the inlined reference monitor by preventing variables and code that a program rewriter adds from being accessed with statements of the original untrusted program S. To provide such a guarantee, the type system must reject a program T if (i) T can access objects or memory that has not been declared within T or (ii) T can transfer control to statements that do not have labels declared within T.

Comparison with Classical Reference Monitors. The code for monitor responses would be substantially the same for a reference monitor implemented as described in §11.2 versus one implemented by using a program rewriter. The assurance argument for the two implementation approaches would be quite different, though.

- With a classical implementation of a reference monitor, the trusted computing base includes hardware and runtime software that detects monitored accesses and invokes monitor responses.
- With an inlined reference monitor, the trusted computing base includes the inliner as well as the compilers, assemblers, and other tools involved in generating an executable from the inliner's output.

In both cases, parts of the operating system would be in the trusted computing base. Compilers, assemblers, and other tools used to generate executables for the operating system are also part of the trusted computing base in both cases. Thus, if untrusted systems are written in the same language as the operating system—allowing the same software to be used in generating the executables for both—then the program rewriter is the only significant difference in what must be trusted. And the program rewriter can be removed from the trusted computing base by using proof carrying code, as we discuss shortly, in §12.6.

12.5 Applicability of Program Rewriting

For program rewriting to be effective, all code that potentially could violate the policy to be enforced must first be processed by a program rewriter. This requirement implies that program rewriting cannot be used to enforce security policies in a single executable—such as an operating system or dynamically linked library—that is being shared by multiple clients having different security policies. It also precludes the use of program rewriting for adding enforcement to systems for which the code is not available and, thus, cannot be processed by a program rewriter.

Where the code is available, then program rewriting is attractive because enforcement code that has been added to an untrusted program can have lower overhead than enforcement code that is part of the runtime environment.

- Code added by program rewriting executes in the same address space as the untrusted program being secured, so a context switch is not required for control transfers to the enforcement code. In comparison, context switches are required if that functionality is being performed by code in the operating system or runtime environment. And context switches reduce processor performance by causing caches to be purged and pipelines to be drained.

- The tool chain to perform program rewriting can include optimizers for eliminating unnecessary checks in the code that a program rewriter adds. Those optimizations are not possible for mechanisms implemented in a common runtime environment, because the single body of code must serve all clients and, therefore, cannot be optimized for use by any one client.

However, it is hard to have assurance about a specification for code modifications if that specification is long and complicated. Such specifications would be required for defining the sequences of machine language instructions that constitute operations on high-level abstractions. So for enforcing application-specific security policies, program rewriting is better suited for use with high-level language programs than for use in assembly language or machine language programs. But program rewriting is well suited for enforcing memory safety and other lower-level policies in assembly language or machine language programs.

Program rewriting also is unlikely to be a suitable replacement for enforcement mechanisms that are widely used, such as those that a runtime environment provides for isolation and authorization. First, updates for effecting policy changes are easier to perform to the single copy of the runtime environment than to a large number of applications. Second, it is hard to test whether a prescription for program rewriting will break some piece of software when there are many different ones to be processed; it is (comparatively) easy to test changes made to a single runtime environment.

Finally, some of the security policies that a runtime environment enforces are expressly intended to protect that system and its clients against untrustworthy applications. That goal would is undermined if the enforcement of those policies

requires trusting a program rewriter that is external to the runtime environment, and the trusted computing base is expanded if the program rewriter is made internal to the runtime environment.

12.6 *Use of Proof-Carrying Code (PCC)

The use of program rewriting to enforce a security policy would seem to require having the program rewriter be part of the trusted computing base. But the program rewriter need not be in the trusted computing base, if modified program $\mathcal{T}_P(S)$ that the program rewriter produces from S is allowed to execute only after a program analyzer first establishes that $\mathcal{T}_P(S)$ complies with security policy P. The program analyzer would be ensuring that suitable checks were added for enforcing the security policy and that optimizations performed during or after program rewriting did not remove checks necessary for enforcing the security policy.

The undecidability of the halting problem means that a program analyzer cannot be built to check whether a program written in a general-purpose programming language will comply with certain security policies. One approach for circumventing this limitation is to be conservative. If the program analyzer on hand cannot certify that $\mathcal{T}_P(S)$ complies with security policy P then $\mathcal{T}_P(S)$ is not permitted to execute (even though $\mathcal{T}_P(S)$ might actually comply with P).

A second approach is to use a program rewriter to modify $\mathcal{T}_P(S)$ in ways that facilitate checking that $\mathcal{T}_P(S)$ complies with security policy P. These modifications typically involve adding runtime checks to validate assertions that certain program behaviors are not possible. As an example, a program analyzer might not be able to determine whether a subscript in an array reference is guaranteed to be within bounds. But that determination becomes straightforward if a program rewriter moves the array reference into an **if** statement that is checking whether the subscript is within bounds. So by modifying the program to have the array reference in an **if** statement, a program rewriter enables a program analyzer to sidestep undecidability.

Program analyzers allow program rewriters to be removed from the trusted computing base. However, program analyzers can be large and complicated, so it is better if they too aren't part of the trusted computing base. With *proof-carrying code* (PCC), they needn't be included in the trusted computing base.

> **Trust Relocation by using PCC.** To remove a program analyzer[13] from the trusted computing base:
>
> (i) modify the program analyzer to output a proof that justifies the results of its analysis,
>
> (ii) incorporate a *checker* into the trusted computing base, and

[13] In the PCC literature, the program analyzer would be considered a *code producer* and the checker would be considered a *proof validator*.

(iii) before executing a modified program, run a program analyzer to output a proof that the modified program complies with the intended security policy and then use the checker to validate that proof. □

If a checker that is in the trusted computing base validates the program analyzer's proof then conclusions justified by that proof can be trusted—even though the program analyzer is not in the trusted computing base.

PCC reduces the size of the trusted computing base if the checker is smaller and simpler than the program analyzer. Such a size reduction will be observed if the program analyzer is producing a proof in a formal logic. Recall, a formal logic defines a syntax for formulas and defines a finite set of inference rules. Each inference rule gives a mechanical way to derive a conclusion, which is a formula, from a finite set of hypotheses, which also are formulas.[14] Various styles of formal proof are used by logicians. A *Hilbert-style* formal proof is a sequence of lines of the form

$$L_i\text{: } F_i \textbf{ by } inf_i \textbf{ from } L_i^1, L_i^2, \ldots, L_i^m$$

where L_i is a unique label, F_i is a formula of the logic, inf_i is the name of an inference rule of the logic, and $L_i^1, L_i^2, \ldots, L_i^m$ is a list of labels on earlier lines in the proof. Such a line is sound if inference rule inf_i derives F_i as its conclusion using as hypotheses the formulas at labels $L_i^1, L_i^2, \ldots, L_i^m$. The formal proof is sound if every line is sound.

A checker to validate Hilbert-style formal proofs would have a routine for each inference rule. This routine would determine whether the associated inference rule derives a given conclusion from given hypotheses and, therefore, that use of the inference rule produced a sound line in the proof. The code size for such a checker would be proportional to the number of inference rules of the formal logic, and the program's running time would be linear in the length of the proof that it is checking. In constrast, a program for generating proofs might involve routines that implement strategies to explore various possibilities for choosing sequences of inference rule to apply. Such a proof generator program might not be simple or small; and it might have exponential running time, or worse.

Since structural induction on the syntax of programs is easily packaged as the inference rules of a formal logic, Trust Relocation using PCC becomes applicable when we have a program analyzer that infers properties of code segments from properties it infers about the pieces constructing those code segments. The usual formalization of a type system, for example, gives an inference rule for each kind of statement and expression. A type checker uses these rules to add any run-time checking code needed for avoiding undecidability of the halting problem and then proving a theorem that asserts the modified program is type correct. Such type systems can be surprisingly powerful, rejecting programs that might exhibit various forms of misbehavior, including: operations on data of the wrong type, reading uninitialized memory, using out of bounds subscripts and pointers,

[14] A formula that is an axiom can be seen as an inference rule that has no hypotheses.

races and other unintended non-determinacy due to unsynchronized concurrent access, corrupting the integrity of a capability, and even making unauthorized accesses to variables. In short, many common security policies can be enforced in this way.

Notes and Reading

By the turn of the century, it had become clear that program analysis and program rewriting could enforce security policies and could reduce the size of a trusted computing base [14]—even for programs written in untyped languages or in assembly language. A new *language-based security* community formed around that agenda, applying programming language techniques to invent new defense mechanisms as well as to give rigorous characterizations for the classes of security policies that different defense mechanisms could be used to enforce.

Schneider [13] had proved that execution monitoring could be used only to enforce safety properties. Hamlen, a Cornell Ph.D. student, took the next steps with his advisors. They characterized computability classes for the security policies that could be enforced by using execution monitoring, program rewriting, and program analysis [8]. A discussion of those computability classes is out of scope for this book, but the formalizations for Soundness and Transparency[15] given in §12.1.1 are based on the formalizations in [8], and the confidentiality example given in §12.1.2 is inspired by the Secret File Policy in [8].

The first documented use of program rewriting to enforce a security policy seems to be the Spy monitoring system [4] in the Berkeley Timesharing System for the SDS 940. Spy ensured that an untrusted user's kernel extension was safe by checking that the extension would not introduce a loop, execute too long, or perform updates outside of a designated memory region that had been dedicated to collecting statistics.

Untrusted software extensions also were the reason for Wahbe et al. [16] to develop software-based fault isolation (SFI). The original design of SFI assumed that all instructions have the same fixed length. This assumption is satisfied by RISC architectures but not by CISC architectures. The use of "chunks" to implement SFI for CISC architectures was developed by McCamant and Morrisett [10] in the PittSFIeld[16] system for the Intel x86. PittSFIeld, in addition, showed that inserting a single instruction could suffice for generating safe addresses if the starting and ending addresses of the safe memory region have been carefully chosen. PittSFIeld also implemented efficient procedure returns by exploiting the shadow return stack found on modern processors like x86.

A considerable body of literature now exists about SFI implementations, including research prototypes as well as program rewriters (e.g., native client [17]) intended for deployment in production software. Issues that arise with support-

[15] Terms "soundness" and "transparency" had been introduced earlier by Ligatti et al. [9] for describing their edit automata extension to security automata.

[16] The name is almost an acronym: Prototype IA-32 Transformation Tool for Software-based Fault Isolation Enabling Load-time Determinations (of safety).

ing the MIPS, x86 (32 bit and 64 bit), and ARM instruction sets have been discussed. And SFI implementations have been developed to facilitate isolation of kernel extensions, application extensions, and browser extensions. A survey by Tan [15] gives citations for this work and also discusses the various techniques used in these SFI implementations.

Control-flow integrity (CFI) [1] uses program rewriting to prevent control-flow hijacking; SFI does not block this. So CFI defends against stack smashing, buffer overflows, as well as some jump to `libc` and return-oriented programming attacks. CFI also simplifies the task of building a program rewriter by providing a starting point that prevents attackers from circumventing code that the new program rewriter inserts or modifies. XFI [5], for example, incorporates CFI to extend SFI and support richer policies, isolating a broader range of extensions—all with improved performance. Hardware instructions to support CFI and XFI are proposed and their performance analyzed in a 2006 paper [3]; ARM and Intel have since introduced instructions that, like `CFILab`, mark valid branch destinations so that branches to other addresses can be blocked.[17]

Inlined reference monitors (IRM) were introduced in Erlingsson and Schneider [7]. First came SASI[18] [6], a program rewriter for incorporating a reference monitor into x86 or JVML object code. Experience with SASI drove the development of a successor, the PoET/PSLang[19] toolkit for modifying JVML object code to enforce security policies in Java programs. Prior work on the use of program rewriting to enforce security had employed program rewriters having a single input: the program to be modified. IRMs generalized the approach by adding a second input: a formulation of the security policy to be enforced. Besides their use for enforcing security policies, IRMs also are the basis for *runtime verification*. With this approach to software testing, an IRM is incorporated into the system under test as a way to validate that certain properties of the internal state are satisfied during executions. Bartocci et al. [2] is a good reference for learning about runtime verification.

Program rewriting modifies code to obtain a version that is guaranteed to satisfy given properties. But code that already satisfies those properties does not require modification—the program rewriting can be skipped, avoiding the runtime overhead of any added checks. Necula and Lee [12] developed proof-carrying code (PCC) to capitalize on that use case. The first implementation of PCC was intended for allowing code to be directly incorporated into a kernel without requiring that code to have been produced by a program rewriter or by a compiler for a language with strong typing. In subsequent work [11], PCC was used to extend ML programs with code written in assembly language.

The crux of PCC is for code to be accompanied by a proof that all executions satisfy some property. The proof is checked before the code is allowed to run. Checking proofs (in a formal logic) is straightforward, but creating them is not—

[17] To mark legal branch destinations, the arm8.5 instruction set in August 2019 added the `BTI` instruction, and 12th Generation Intel Core Processors in October 2021 added the `endbr32` and `endbr64` instructions.

[18] Security Automata SFI Implementation

[19] Policy Enforcement Toolkit/Policy Specification Language.

a limit on the feasibility of the approach. But strong typing was thought to be adequate for preventing an extension from subverting a system it would extend, and type-checkers mechanically construct such proofs. So PCC was feasible for the intended application. With the benefit of hindsight, though, the utility of PCC goes far beyond the enforcement of those security policies that can be specified as type systems. Rather, PCC is a general approach for relocating trust: by trusting a proof checker, we need not trust a proof generator. And any program rewriter can be regarded as being a proof generator.

Bibliography

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

[2] Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification — Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer International Publishing, 2018.

[3] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural support for software-based protection. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 42–51, New York, NY, USA, 2006. Association for Computing Machinery.

[4] Peter Deutsch and Charles A. Grant. A flexible measurement tool for software systems. In Charles V. Freiman, John E. Griffith, and Jack L. Rosenfeld, editors, *Information Processing, Proceedings of IFIP Congress 1971, Volume 1 - Foundations and Systems*, pages 320–326. North-Holland, August 1971.

[5] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 75–88. USENIX Association, November 2006.

[6] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms*, NSPW '99, page 87–95, New York, NY, USA, 1999. Association for Computing Machinery.

[7] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255, USA, May 2000. IEEE Computer Society.

[8] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, January 2006.

[9] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *Internatonal Journal of Information Security*, 4(1–2):2–16, February 2005.

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

[11] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[12] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 229–243, New York, NY, USA, 1996. Association for Computing Machinery.

[13] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information System Security*, 3(1):30–50, February 2000.

[14] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics — 10 Years Back. 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2001.

[15] Gang Tan. Principles and implementation techniques of software-based fault isolation. *Privacy and Security*, 1(3):137–198, 2017.

[16] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. Association for Computing Machinery.

[17] Bennet Yee, David Sehr, Gregory Dardyk, Bradley J. Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, January 2010.

