15-316: Software Foundations of Security and Privacy

Lecture Notes on Inlining Safety

Matt Fredrikson

Carnegie Mellon University

Lecture 7

1 Introduction & Recap

In the previous lecture we added memory to our language. We assume that the memory is just an array of values indexed by integers in the range $[0, U]$, and that it is undefined on any indices outside this range. Programs can read from memory by dereferencing it with the syntax $\texttt{Mem}(e)$, and update it with the syntax $\texttt{Mem}(e) := \tilde{e}$.

We introduced axioms for reasoning about memory updates, being careful about bounds on accesses as necessary.

$$([*]_=) \quad [\texttt{Mem}(e) := \tilde{e}]p(\texttt{Mem}) \leftrightarrow p(\texttt{Mem}\{e \mapsto \tilde{e}\}) \land 0 \le e < U$$

$$([*]_1) \quad \frac{\Gamma \vdash e = e' \quad \Gamma \vdash 0 \le e' < U}{\Gamma \vdash \texttt{Mem}\{e \mapsto \tilde{e}\}(e') = \tilde{e}}$$

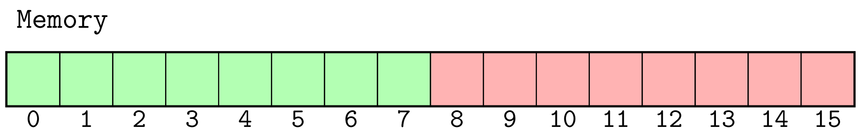
$$([*]_2) \quad \frac{\Gamma \vdash e \ne e' \quad \Gamma \vdash 0 \le e' < U}{\Gamma \vdash \texttt{Mem}\{e \mapsto \tilde{e}\}(e') = \texttt{Mem}(e')}$$

We then defined memory safety for our language as the set of traces for which any terminal error state Λ is not caused by a memory dereference or update. Using the axioms to prove safety covers most of memory safety as well, due to the bounds checks. But they don't cover dereferences that occur prior to updates, so if we want to ensure memory safety then we need to put assertions before each memory access that check to make sure its bounds are within $[0, U]$. Then proving any safety property for the resulting program will also be sufficient to demonstrate memory safety.

But what if we want to enforce a more granular type of memory safety policy to ensure that parts of our program don't read or write portions they aren't supposed to. This was motivated by our hypothetical career as an app developer who wants to monetize with advertising, and is thus compelled by Vladimir's discount ad shop to run untrusted rendering code within our program:

$$\texttt{if}(\textit{display ads})\,\alpha\,\texttt{else}\,\textit{continue without ads}$$

We discussed sandboxing policies where a region of memory is designated for the untrusted α to "play" in, such as the upper portion of memory at addresses 8-15 in the diagram below.



As long as we can enforce this policy, and we are careful about writing our program to save and restore variable state, then we can ensure that whatever the sandbox does will not affect the rest of our program's execution.

We can certainly enforce such a policy by inserting $\texttt{assert}(Q)$ commands before any memory read or write, to make sure that the indexed memory doesn't point outside the designated sandbox. But this approach has serious drawbacks. First, if α is simply buggy and makes accesses outside the sandbox, then the entire program will abort and our app will "crash" as far as the user is concerned. Second, Vladimir can actually force this outcome if he is maliciously inclined, and we certainly don't want to give such an attacker that kind of leeway.

Today we will discuss an approach called *software fault isolation* [2, 4] (SFI) for properly isolating the malicious or buggy effects of α from the rest of our program. SFI works by inlining enforcement directly into α, changing its behavior so that it can't violate the sandbox policy and if it attempts to do so then it still won't have any effect on the rest of our execution. SFI is a very practical technique, and has been used effectively in real applications to isolate untrusted code execution from browsers, operating systems, and other critical applications. In the next lab, you will implement a prototype SFI policy for your server.

Then we will look at a related technique called *control flow integrity* [1], which ensures that the attacker cannot influence the control flow of a program to diverge from a predefined control flow policy. But in order for this defense to have any purpose, we need to introduce indirect control flow commands into our language, bringing it closer yet to the features that real platforms in need of rigorous security defenses have in practice.

2 SFI: isolating sandbox policy violations

Rather than checking whether memory accesses are safe and aborting if the check fails, perhaps we can force all untrusted accesses to be within the sandbox. In the diagram above, we use the specific sandbox policy $s_l = 8, s_h = 15$. Let us assume that our language operates over machine integers, so that the sandbox boundaries are the binary constants:

$$s_l = \texttt{0b1000}, s_h = \texttt{0b1111}$$

So the range of valid sandbox addresses is `0b1000`, `0b1001`, `0b1010`, ..., `0b1111`. Any valid address will have the fourth bit set, and all greater bits unset. Given an arbitrary term e, we can use bitwise operations to force it to a value in this range:

$$(e \,\&\, \texttt{0b1111}) \mid \texttt{0b1000} \tag{1}$$

From now on, we will use hexadecimal rather than binary when writing such constants, so Equation 2 becomes $(e \,\&\, \texttt{0xF}) \mid \texttt{0x8}$. If we assume that our sandbox regions always comprise integral boundaries (i.e., `0x0000`-`0x00FF`, `0x0100`-`0x01FF`, `0x0200`-`0x02FF`), then we can generalize this to:

$$(e \,\&\, s_h) \mid s_l \tag{2}$$

With this in mind, we change the way we instrument programs.

- Replace each command of the form $\texttt{Mem}(e) := \tilde{e}$ with a new composed command:

  $$\texttt{Mem}((e \,\&\, s_h) \mid s_l) := \tilde{e}$$

  This will ensure that α doesn't update any locations outside the sandbox.

- For any command β containing the term $\texttt{Mem}(e)$, replace $\texttt{Mem}(e)$ with $\texttt{Mem}((e \,\&\, s_h) \mid s_l)$. This will ensure that α doesn't read any locations outside the sandbox.

This is called *software fault isolation* (SFI). The benefit of this approach is that as long as the sandbox is configured correctly for the memory, so that

$$0 \le s_l \le s_h < U \tag{3}$$

Then after instrumenting the untrusted program α, we know that *(1)* it will not violate the sandbox safety policy, and *(2)* it will also be memory safe!

The semantics of the instrumented program will certainly differ from the original α, in particular if it made unsafe memory accesses, and this may lead to bugs in the instrumented code that cause it to behave otherwise than expected. But this need not concern us, as our program will be completely isolated from the effect of these bugs.

Correctness of write instrumentation. But how do we know that the instrumented program will actually satisfy the sandbox safety policy? Before when we used $\texttt{assert}(Q)$ commands, we might have gotten away with an informal argument because the correctness was totally obvious. But now our instrumentation does strange things with bitwise operators to force certain behaviors. We should really be more formal about this to make sure we didn't screw things up with a bad assumption.

The question becomes, how do we formalize the correctness of our sandbox policy as a safety property? Before we reasoned that the "bad thing" is a certain type of event, i.e. a read or write to memory locations outside the sandbox. We don't know how to prove things about these sorts of events, because all of the properties we have looked at so far define bad things directly in terms of state. Perhaps we can think in terms of the effect that violations will have on program state instead of the events that bring those effects into being.

The first type of instrumentation purports to cover all write events. If α violates the policy by writing outside the sandbox, then the bad thing in terms of state would be that the contents of non-sandbox memory after α terminates differ from their contents prior to running α. This sounds like something that we can formalize in dynamic logic using familiar properties, i.e. contracts.

$$\forall i. \neg(s_l \le i \le s_h) \land \texttt{Mem}(i) = v_i \to [\alpha]\texttt{Mem}(i) = v_i \tag{4}$$

But how can we prove this without knowing anything about what α is? We can reason inductively on the syntax of programs, which is what the proof of Theorem 1 does.

Theorem 1. *Let α be a program whose memory update commands have been instrumented as prescribed by software fault isolation, and the sandbox low and high bounds are configured correctly, so that for all x:*

$$0 \le s_l \le (x \,\&\, s_h) \mid s_l \le b_h < U \tag{5}$$

Then all valid memory indices outside the sandbox retain the same value after executing α as they had prior to executing it. In other words, Equation 4 is valid.

Proof. We will proceed by induction on the structure of α. That is, we will show that for all of the simplest (base case) forms that α can take, the claim holds. Then we will use the inductive hypothesis for more complex forms of α, showing that the claim holds whenever we assume that it does for any subprograms inside of α. The inductive case thus covers all possible programs that can be constructed according to the syntax we introduced at the beginning of the lecture. This means that regardless of how α is implemented, the safety claim will hold.

The base cases of this proof correspond to programs that contain no other program constituents, i.e. $x := e$, $\texttt{Mem}(e) := \tilde{e}$, and $\texttt{assert}(Q)$. The inductive cases are programs that contain other programs, i.e. $\alpha; \beta$, $\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$, and $\texttt{while}(Q)\,\alpha$. We will complete the most challenging base case to outline the form of the proofs of the others, and leave the remaining ones as an exercise. We will do the same for one inductive case, leaving the rest as an exercise.

Base case $\texttt{Mem}(e) := \tilde{e}$: The instrumentation will replace this command with:

$$\texttt{Mem}((e \,\&\, s_h) \mid s_l) := \tilde{e}$$

Then the following sequent derivation demonstrates correctness. Note that we use ∀R, which is detailed in the aside at the end of these notes.

$$\dfrac{\dfrac{\dfrac{\neg(s_l \le i \le s_h), \texttt{Mem}(i) = v_i \vdash \texttt{Mem}\{(e \,\&\, s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i \qquad \ldots \vdash 0 \le (e \,\&\, s_h) \mid s_l < U}{\neg(s_l \le i \le s_h), \texttt{Mem}(i) = v_i \vdash [\texttt{Mem}((e \,\&\, s_h) \mid s_l) := \tilde{e}]\texttt{Mem}(i) = v_i}\,[*]{=}}{\vdash \neg(s_l \le i \le s_h) \land \texttt{Mem}(i) = v_i \to [\texttt{Mem}((e \,\&\, s_h) \mid s_l) := \tilde{e}]\texttt{Mem}(i) = v_i}\,{\to}\mathrm{R},{\land}\mathrm{L}}{\vdash \forall i. \neg(s_l \le i \le s_h) \land \texttt{Mem}(i) = v_i \to [\texttt{Mem}((e \,\&\, s_h) \mid s_l) := \tilde{e}]\texttt{Mem}(i) = v_i}\,\forall\mathrm{R}$$

The right branch is left open, but we can discharge it from our assumption (5) in the theorem statement. At this point we need to split into cases on the left branch, because it could either be that $i = (e \mathbin{\&} s_h) \mid s_l$ or $i \neq (e \mathbin{\&} s_h) \mid s_l$. Depending on which case it is, we use $[*]_1$ or $[*]_2$. We case split with the cut rule. In the following, let

$$P_1 \equiv i = (e \mathbin{\&} s_h) \mid s_l, P_2 \equiv i \neq (e \mathbin{\&} s_h) \mid s_l, P \equiv P_1 \lor P_2$$

Then we continue with the proof as follows:

$$\mathrm{cut}\dfrac{\mathbb{Z}_M\dfrac{*}{\vdash P} \quad \lor\mathrm{L}\dfrac{\textcircled{1} \quad \textcircled{2}}{\neg(s_l \le i \le s_h), \mathtt{Mem}(i) = v_i, P \vdash \mathtt{Mem}\{(e \mathbin{\&} s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i}}{\neg(s_l \le i \le s_h), \mathtt{Mem}(i) = v_i \vdash \mathtt{Mem}\{(e \mathbin{\&} s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i}$$

Continuing with subtree $\textcircled{1}$:

$$\neg\mathrm{L}\dfrac{\mathtt{Mem}(i) = v_i, i = (e \mathbin{\&} s_h) \mid s_l \vdash s_l \le i \le s_h, \mathtt{Mem}\{(e \mathbin{\&} s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i}{\neg(s_l \le i \le s_h), \mathtt{Mem}(i) = v_i, i = (e \mathbin{\&} s_h) \mid s_l \vdash \mathtt{Mem}\{(e \mathbin{\&} s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i}$$

This part of the derivation asks us to prove that either $s_l \le i \le s_h$ or $\mathtt{Mem}\{(e \mathbin{\&} s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i$, from the assumptions that $\mathtt{Mem}(i) = v_i$ and $i = (e \mathbin{\&} s_h) \mid s_l$. Let's think about the cases a bit.

- We could try to prove that $\mathtt{Mem}\{(e \mathbin{\&} s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i$. At first glance this might seem promising, because of the assumption that $\mathtt{Mem}(i) = v_i$. But we also assume that $i = (e \mathbin{\&} s_h) \mid s_l$, and $[*]_1$ tells us then that $\mathtt{Mem}\{(e \mathbin{\&} s_h) \mid s_l \mapsto \tilde{e}\}(i) = \tilde{e}$. We don't have an assumption which says that $v_i = \tilde{e}$, which we would need to do the proof this way.

- We can alternately prove that $s_l \le i \le s_h$. We have in our context that $i = (e \mathbin{\&} s_h) \mid s_l$, and the theorem assumes (5) which gives us an even stronger property $0 \le s_l \le (x \mathbin{\&} s_h) \mid s_l \le b_h < U$. We can invoke $\mathbb{Z}_M$ to discharge the obligation:

  $$0 \le s_l \le (x \mathbin{\&} s_h) \mid s_l \le b_h < U \to s_l \le i \le s_h$$

  We complete the proof of subtree $\textcircled{1}$ this way, and can move on with the proof.

Now we complete this case of the proof by deriving $\textcircled{2}$.

$$\mathrm{G}\dfrac{\mathrm{cut}\dfrac{[*]_2\dfrac{\mathrm{id}\dfrac{*}{\mathtt{Mem}(i) = v_i, i \neq (e \mathbin{\&} s_h) \mid s_l \vdash i \neq (e \mathbin{\&} s_h) \mid s_l} \quad \mathtt{Mem}(i) = v_i, i \neq (e \mathbin{\&} s_h) \mid s_l \vdash 0 \le i < U}{\mathtt{Mem}(i) = v_i, i \neq (e \mathbin{\&} s_h) \mid s_l \vdash \mathtt{Mem}\{(e \mathbin{\&} s_h) \mid s_l \mapsto \tilde{e}\}(i) = \mathtt{Mem}(i)}}{\mathtt{Mem}(i) = v_i, i \neq (e \mathbin{\&} s_h) \mid s_l \vdash \mathtt{Mem}\{(e \mathbin{\&} s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i}}{\neg(s_l \le i \le s_h), \mathtt{Mem}(i) = v_i, i \neq (e \mathbin{\&} s_h) \mid s_l \vdash \mathtt{Mem}\{(e \mathbin{\&} s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i}$$

The unfinished portion of the proof assumes that $\mathtt{Mem}(i) = v_i$ and $i \neq (e \mathbin{\&} s_h) \mid s_l$ imply that $0 \le i < U$. Recall that memory dereferences are undefined whenever the index is out of bounds, and our assumption is that the memory at index i *is* in fact defined, and takes the value v_i. From this we conclude that i must be in bounds. This completes the base case for memory updates.

Inductive case $\alpha;\beta$: Suppose that the program is a composition of α and β. The inductive hypothesis lets us assume that:

$$\forall i.\neg(s_l \le i \le s_h) \land \texttt{Mem}(i) = v_i \to [\alpha]\texttt{Mem}(i) = v_i \tag{6}$$
$$\forall i.\neg(s_l \le i \le s_h) \land \texttt{Mem}(i) = v_i \to [\beta]\texttt{Mem}(i) = v_i \tag{7}$$

Then consider $(\omega, \dots, \mu) \in [\![\alpha]\!]$ and $(\mu, \dots, \nu) \in [\![\beta]\!]$. We have the following:

$$\forall i.\neg(s_l \le i \le s_h) \land \omega_M(i) = v_i \to \mu_M(i) = v_i \tag{8}$$
$$\forall i.\neg(s_l \le i \le s_h) \land \mu_M(i) = v_i \to \nu_M(i) = v_i \tag{9}$$

By the semantics of $\alpha;\beta$ and the transitive property of equality, it must be that for all $(\omega, \dots, \nu) \in [\![\alpha;\beta]\!]$ it is the case that:

$$\forall i.\neg(s_l \le i \le s_h) \land \omega_M(i) = v_i \to \nu_M(i) = v_i \tag{10}$$

Then (10) and the semantics of $[\alpha;\beta]$ tell us that

$$\forall i.\neg(s_l \le i \le s_h) \land \texttt{Mem}(i) = v_i \to [\alpha;\beta]\texttt{Mem}(i) = v_i \tag{11}$$

This completes the inductive case for composition.

Rest of the proof: The remaining cases are left as exercises. The remaining base cases are easy to complete, because they correspond to program forms that do not affect the memory state at all. The inductive cases follow the form outlined for $\alpha;\beta$ above, using the inductive hypothesis as well as the semantics of programs and dynamic logic to conclude that whenever subprograms satisfy the sandbox policy, the larger programs that contain them do as well. $\square$

So we have now concluded that software fault isolation prevents memory write operations from working outside the designated sandbox. What about read operations? The second form of instrumentation is applied to terms that read from the current memory state, and we expect that they will prevent programs from unauthorized reads for the same reasons that write operations are safe.

Correctness of read instrumentation. We based our proof of write operations on the fact that it can be formalized as a safety property over program state. We reasoned that if the instrumentation were not sufficient, then there would be evidence at the end of α's execution in the form of memory contents that were modified from their initial value. But can we say something similar about memory read operations? What evidence in the state will there be if the instrumentation is not correct, and α succeeds at reading a memory location outside the sandbox?

We might say that if there was a successful read outside the sandbox, then one of the program variables, or perhaps one of the sandbox memory cells, will contain a value that was initially in the memory outside the sandbox. But this need not be the case, because what if α makes an unauthorized read, and then performs an operation in the result before storing it in a variable or memory? On the other hand, suppose that in α's final state, one of the variables *did* take the same value as an unauthorized memory location. Are we certain that it took this value because of an unauthorized read, or could it be mere chance the α happened to compute a value that overlapped with outside memory?

This question drives to a fundamental difference between safety and information flow properties. We've learned that safety properties can be viewed as collections of traces, so all that we need to do to reason about whether a program satisfies such a property is make sure all of its traces are in the property. This is what SFI accomplishes when it forces memory accesses to a particular range, because the property says that all traces must only make accesses within that range. Likewise, this is what we prove when we use dynamic logic sequent calculus deductions to reason about safety: that all terminating traces are in the set described by the property.

But information flow properties are fundamentally different. They cannot be described as sets of traces, and in fact must consider what *might* have happened on a different trace if some variable or memory location had taken a different value. To reason convincingly about the correctness of the read operations we need to be able to refer to and prove things about information flow properties, i.e. that information outside the sandbox does not flow into any of the variables or memory locations within the sandbox. This will be a topic of future lectures, where we will take a completely different approach to policy enforcement.

3 Indirect control flow

So far the programs that we have considered have a particularly nice property. Namely, once the programmer decides which commands are in the program and how they are sequenced together with compositions, conditionals, and while loops, then all of the possible sequences of commands that the program will ever execute are fixed once and for all. There is no way for a user to provide data that could cause some of the commands to be skipped over or added, and as long as the program is executed faithfully to the semantics, the control flow will be as the programmer envisioned when the program was written.

Programs executed on "real" machines do not enjoy this property, thanks to indirect transfers of control flow. An indirect control flow transfer is commonly implemented using a function pointer in high-level languages, or a `jmp` instruction with a pointer operand. We'll add this functionality to our language by considering a program command of the form:

$$\texttt{if}(Q)\,\texttt{jump}\,e \tag{12}$$

The command in (12) first tests whether a formula Q is true in the current state. If it is, then control transfers to the instruction indexed by the term e. Otherwise, control proceeds to the next instruction.

But this doesn't make much since yet, because we haven't discussed indexing of instructions. Programs in the simple imperative language are themselves just commands, which can be built from other commands by connecting them with composition, conditional, and looping constructs. We will now change the language so that rather than having structured high-level commands like $\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$ and $\texttt{while}(Q)\,\alpha$, we will assume that programs are sequences of unstructured "atomic" commands. So the commands are defined by the syntax:

$$\alpha \ ::= \ x := e \mid \texttt{Mem}(e) := \tilde{e} \mid \texttt{assert}(Q) \mid \texttt{if}(Q)\,\texttt{jump}\,e \mid \texttt{halt}$$

Then a program Π is a finite sequence of commands,

$$\Pi = (\alpha_0, \alpha_1, \dots, \alpha_n) \tag{13}$$

We will write $\Pi(i)$, where $0 \le i \le n$, to refer to the command α_i in program Π. If i is negative, or $n < i$, then $\Pi(i)$ is undefined.

Think of this language as a simplified version of assembly language. Memory update commands $\texttt{Mem}(e) := \tilde{e}$ correspond to store instructions (i.e., `mov` into a memory cell), memory dereference terms $\texttt{Mem}(e)$ correspond to memory fetch instructions (i.e., `mov` from a memory cell to a register), and $\texttt{if}(Q)\,\texttt{jump}\,e$ to conditional `jmp` instructions. We don't have an explicit stack or notion of procedure to worry about, but if we did then `halt` commands would correspond to `ret` instructions.

Semantics Recall that program states ω are composed of a variable map ω_V and memory ω_M. Now that programs Π are composed of indexed commands, and can transfer control to any command in Π, states will also need to track a program counter ω_i that determines which command executes next. The program counter will range from $i \in 0$ to n, denoting that the command Π_i executes next. We will also use the special program counter value $\varnothing$ to denote the "empty" program counter that occurs when a program terminates.

Definition 2 (Small-step transition semantics of programs). The small-step transition relation $\rightsquigarrow_\Pi$ of program Π composed of commands $\alpha_0, \dots, \alpha_n$ in state $\omega = (\omega_i, \omega_V, \omega_M)$ is given by the following cases:

$$(\omega_i, \omega_V, \omega_M) \rightsquigarrow_\Pi \begin{cases} (\omega_i + 1, \omega_V\{x \mapsto \omega[\![e]\!]\}, \omega_M) & \textit{if} \ \ \Pi_i = x := e \text{ and } \omega[\![e]\!] \text{ is defined} \\ (\omega_i + 1, \omega_V, \omega_M\{\omega[\![e]\!] \mapsto \omega[\![\tilde{e}]\!]\}) & \textit{if} \ \ \Pi_i = \texttt{Mem}(e) := \tilde{e} \text{ and} \\ & \quad\ \ \omega[\![\tilde{e}]\!] \text{ is defined and } 0 \le e < U \\ (\omega_i + 1, \omega_V, \omega_M) & \textit{if} \ \ \Pi_i = \texttt{assert}(Q) \text{ and } \omega \models Q \\ (\omega[\![e]\!], \omega_V, \omega_M) & \textit{if} \ \ \Pi_i = \texttt{if}(Q)\,\texttt{jump}\,e \text{ and} \\ & \quad\ \ 0 \le \omega[\![e]\!] \le n \text{ and } \omega \models Q \\ (\omega_i + 1, \omega_V, \omega_M) & \textit{if} \ \ \Pi_i = \texttt{if}(Q)\,\texttt{jump}\,e \text{ and} \\ & \quad\ \ 0 \le \omega[\![e]\!] \le n \text{ and } \omega \models \neg Q \\ (\emptyset, \omega_V, \omega_M) & \textit{if} \ \ \Pi_i = \texttt{halt} \\ \Lambda & \textit{if} \ \ \text{otherwise} \end{cases}$$

Having defined the small-step transition semantics, we can define the trace semantics as all sequences of states $\omega_1, \omega_2, \ldots$ that either terminate in a state $\omega_n = (\emptyset, \omega_V, \omega_M)$, diverge (i.e. terminate in no state and run forever), or abort by terminating in $\omega = \Lambda$.

Definition 3 (Trace semantics)**.** Given a program Π composed of commands $\alpha_0, \ldots, \alpha_n$, the trace semantics $[\![\Pi]\!]$ is the set of traces obtainable by repeated application of the small-step relation $\rightsquigarrow_\Pi$.

$$[\![\Pi]\!] = \{(\omega_0, \omega_1, \ldots) \; : \; \omega_i \rightsquigarrow_\Pi \omega_{i+1} \text{ for all indices } i \text{ of the trace}\}$$

The definitions of terminating, diverging, and aborting traces are the same as they were in previous definitions of the trace semantics.

4 Control Flow Integrity

Let's return to our problem with untrusted advertising code. Now that the language α is written in can make indirect jumps, what could go wrong? Assuming that we are using SFI to enforce a sandboxing policy, there is still no way for α to read or write memory outside the sandbox. Is this true? Consider the following situation, where we can assume that SFI has been applied to the untrusted α starting at command 20.

$$\begin{array}{rl} & \vdots \\ \texttt{10:} & z := \texttt{Mem}(x) \\ \texttt{11:} & \texttt{if}(i \geq 0)\,\texttt{jump}\,y \\ & \vdots \end{array}$$

$$\alpha \left\{ \begin{array}{rl} \texttt{20:} & i := 0 \\ \texttt{21:} & x := \textit{attacker's desired address} \\ \texttt{22:} & y := 24 \\ \texttt{23:} & \texttt{if}(0 = 0)\,\texttt{jump}\,10 \\ \texttt{24:} & \textit{copy memory contents from } z \\ & \vdots \end{array} \right.$$

Here, our original program (not the untrusted α) dereferences memory and makes use of indirect control flow transfer. More specifically,

1. At command 10, it dereferences memory on the variable x and stores the result into z. Because this command is not in the untrusted portion α, it was not rewritten with SFI and can readily access memory outside the sandbox.
2. At command 11, the program tests $i \geq 0$, and if the test holds then jumps to whatever command is currently in y.
3. The untrusted code sets up the program state: *(i)* the indirect jump at 11 will occur by setting $i := 0$ on line 20; *(ii)* the indirect jump at line 11 will return control to α by setting $y := 24$; *(iii)* setting $x := \ldots$ at 21 so that the address read at line 10 will be whatever the attacker wants, presumably outside the sandbox bounds.

4. The command at 23 then executes an indirect jump on a trivial test, targeting 10 so that the attacker's choice of memory is read and control returns to α after the indirect jump at 11.

To summarize, the attacker identifies a sequence of commands in the trusted portion of the program, and sets things up in a way so that unauthorized memory is copied into a variable that the attacker can later access once control is returned to the untrusted code. This should remind you of a *return-oriented programming* (ROP) attacks [3] that you learned about in 15-213. If we assume that the attacker knows the text of our program, then it is possible for them to identify "gadgets" in *code that we wrote* to do their bidding. But this crucially relies on the ability to change control flow using indirection so that commands are executed in the order needed by the attacker to carry out their goals.

4.1 Coarse-grained safety

How can we prevent this? One idea is to use the same approach as we did for SFI. In that case, we designed a sandbox between memory address s_l and s_h, and then rewrote the indices in all memory operations to ensure that accesses stayed within those bounds. Perhaps we can do a very similar thing here, by assigning a "code sandbox" between commands at $\mathtt{pc}_l$ and $\mathtt{pc}_h$. Then we can rewrite indirect jump commands to ensure that their target always lies within these bounds.

$$\textit{Rewrite all } \mathtt{if}(Q)\,\mathtt{jump}\,e \textit{ commands as } \mathtt{if}(Q)\,\mathtt{jump}\,(e\,\&\,\mathtt{pc}_h)\mid\mathtt{pc}_l \tag{14}$$

This is a form of *control flow integrity* (CFI) [1], a technique for enforcing a broad class of safety properties that place limits on the allowed control flow paths in a program. As long as we choose $\mathtt{pc}_l$ and $\mathtt{pc}_h$ to satisfy similar conditions as those used in Theorem 1, i.e.

$$0 \le \mathtt{pc}_l \le (e\,\&\,\mathtt{pc}_h)\mid\mathtt{pc}_l \le \mathtt{pc}_h \le n \tag{15}$$

then we can ensure that the untrusted code will never jump out of its sandbox. This seems to address our concerns with the advertising scenario, when everything untrusted resides in a well-specified region of code known in advance.

4.2 Finer-grained control-flow safety

But what if this isn't the case? In the next lecture, we will look at a generalization of the course-grained CFI technique outlined here that gives us more control over the control-flow policy to enforce. We will then see how this is really just a special case of a powerful technique called *inlined reference monitoring*, which encompasses SFI, CFI, and pretty much any safety policy that can be enforced at runtime.

Aside: Rules for quantifiers

Our proof of SFI correctness used a rule that we have not seen before: $\forall R$. The rule allows us to remove the quantifier, replacing the bound variable with a new variable that does not appear anywhere else in the sequent. This is equivalent to saying that if we can prove that $F(y)$ holds on some y for which we make no prior assumptions, then we can conclude that it holds universally. The corresponding left rule ($\forall L$) says that if we can prove something assuming F holds for a particular term, say e, then we can prove it assuming that F holds universally. Intuitively, we've only made our assumptions stronger by assuming that F holds universally.

$$(\forall\text{L})\ \frac{\Gamma, F(e) \vdash \Delta}{\Gamma, \forall x.F(x) \vdash \Delta} \quad (\forall\text{R})\ \frac{\Gamma \vdash F(y), \Delta}{\Gamma \vdash \forall x.F(x), \Delta} \quad (y \text{ new})$$

The rules for existential quantifiers are similar, but in this case, it is the left rule in which we need to be careful about renaming. Similarly to the $\forall R$, if we can use the fact that $F(y)$ holds to prove Δ, and nothing in our assumptions or Δ mentions specific things about y, then we can conclude that the details of y don't matter for the conclusion, and the only important fact is that some value establishing $F(y)$ exists. The $\exists R$ simply says that if we can prove that F holds for term e, then we can conclude that it must hold for some value, even if we leave the value unspecified.

$$(\exists\text{L})\ \frac{\Gamma, F(y) \vdash \Delta}{\Gamma, \exists x.F(x) \vdash \Delta} \quad (y \text{ new}) \quad (\exists\text{R})\ \frac{\Gamma \vdash F(e), \Delta}{\Gamma \vdash \exists x.F(x), \Delta}$$

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Transactions on Information and Systems Security*, 13(1):4:1–4:40, Nov. 2009.

[2] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX Conference on Security*, 2010.

[3] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, Oct. 2007.

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