15-316: Software Foundations of Security and Privacy

Lecture Notes on Software Fault Isolation

Matt Fredrikson

Carnegie Mellon University
Lecture 8

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.

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

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

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

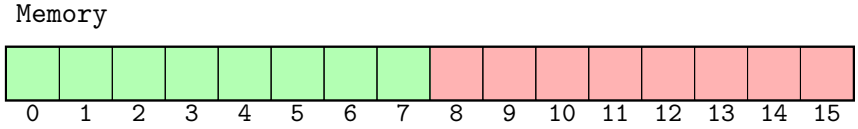
We then defined memory safety for our language as the set of traces for which any terminal error state Λ is not caused by a memory dereference or update. Using the axioms to prove safety covers most of memory safety as well, due to the bounds checks. But they don't cover dereferences that occur prior 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* [SMB+10, YSD+09] (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* [ABEL09], which ensures that the attacker cannot influence the control flow of a program to diverge from a pre-defined 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 to 1, and all greater bits set to 0. Given an arbitrary term e, we can use bitwise operations to force it to a value in this range:

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

What does this term accomplish? By first AND'ing the memory index e with `0b1111`, we ensure that none of the bits that are more significant than the fourth are set to 1. This forces the term to be no greater than `0b1111`, or 15 in decimal. By OR'ing this result with `0b1000`, we ensure that the bit in the fourth position is set to 1, which means that the result can be no less than $2^3 = 8$. Thus, this term over the original index e has the effect of forcing accesses within the sandbox,

$$s_l = 8 \leq (e \,\&\, \texttt{0b1111}) \mid \texttt{0b1000} \leq s_h = 15$$

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

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

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, at least when it comes to the state of the memory.

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.

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.

$$\forall i.\forall v.\neg(s_l \leq i \leq s_h) \land \texttt{Mem}(i) = v \rightarrow [\alpha]\texttt{Mem}(i) = v \tag{4}$$

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:*

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

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.

$$\forall\text{R},\forall\text{R}\;\frac{\;[*]_=\;\dfrac{\;\rightarrow\text{R},\wedge\text{L}\;\dfrac{\dfrac{\neg(s_l \le i \le s_h), \texttt{Mem}(i) = v \vdash \texttt{Mem}\{(e \,\&\, s_h) \mid s_l \mapsto \tilde{e}\}(i) = v \quad \ldots \vdash 0 \le (e \,\&\, s_h) \mid s_l < U}{\neg(s_l \le i \le s_h), \texttt{Mem}(i) = v \vdash [\texttt{Mem}((e \,\&\, s_h) \mid s_l) := \tilde{e}]\texttt{Mem}(i) = v}}{\vdash \neg(s_l \le i \le s_h) \wedge \texttt{Mem}(i) = v \rightarrow [\texttt{Mem}((e \,\&\, s_h) \mid s_l) := \tilde{e}]\texttt{Mem}(i) = v}}{\vdash \forall i.\forall v.\neg(s_l \le i \le s_h) \wedge \texttt{Mem}(i) = v \rightarrow [\texttt{Mem}((e \,\&\, s_h) \mid s_l) := \tilde{e}]\texttt{Mem}(i) = v}$$

Note that in the above, the elided (. . .) assumptions on the top-right branch are identical to those in the top-left branch. They are left out only to ensure that the tree fits in the margins.

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 $i = (e \,\&\, s_h) \mid s_l$ or $i \neq (e \,\&\, s_h) \mid s_l$. Depending on which case it is, we use $[*]_1$ or $[*]_2$. We case split with the cut rule. In the following, let

$$P_1 \equiv i = (e \,\&\, s_h) \mid s_l, P_2 \equiv i \neq (e \,\&\, s_h) \mid s_l, P \equiv P_1 \vee P_2$$

Then we continue with the proof as follows:

$$\text{cut}\;\frac{\mathbb{Z}_M\;\dfrac{*}{\vdash P} \quad \vee\text{L}\;\dfrac{\textcircled{1} \quad \textcircled{2}}{\neg(s_l \le i \le s_h), \texttt{Mem}(i) = v, P \vdash \texttt{Mem}\{(e \,\&\, s_h) \mid s_l \mapsto \tilde{e}\}(i) = v}}{\neg(s_l \le i \le s_h), \texttt{Mem}(i) = v \vdash \texttt{Mem}\{(e \,\&\, s_h) \mid s_l \mapsto \tilde{e}\}(i) = v}$$

The two remaining branches correspond to the cases where memory is dereferenced at the updated address $(e \,\&\, s_h) \mid s_l$ (①), or anywhere else (②). Continuing with subtree ①:

$$\neg\text{L}\;\frac{\texttt{Mem}(i) = v, i = (e \,\&\, s_h) \mid s_l \vdash s_l \le i \le s_h, \texttt{Mem}\{(e \,\&\, s_h) \mid s_l \mapsto \tilde{e}\}(i) = v}{\neg(s_l \le i \le s_h), \texttt{Mem}(i) = v, i = (e \,\&\, s_h) \mid s_l \vdash \texttt{Mem}\{(e \,\&\, s_h) \mid s_l \mapsto \tilde{e}\}(i) = v}$$

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

- We could try to prove that $\texttt{Mem}\{(e \,\&\, s_h) \mid s_l \mapsto \tilde{e}\}(i) = v$. At first glance this might seem promising, because of the assumption that $\texttt{Mem}(i) = v$. But we also assume that $i = (e \,\&\, s_h) \mid s_l$, and $[*]_1$ tells us then that $\texttt{Mem}\{(e \,\&\, s_h) \mid s_l \mapsto \tilde{e}\}(i) = \tilde{e}$. We don't have an assumption which says that $v = \tilde{e}$, 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 $i = (e \,\&\, s_h) \mid s_l$, and the theorem assumes (5) which gives us an even stronger property $0 \le s_l \le (x \,\&\, s_h) \mid s_l \le b_h < U$. We can invoke $\mathbb{Z}_M$ to discharge the obligation:

  $$0 \le s_l \le (x \,\&\, s_h) \mid s_l \le b_h < U \rightarrow s_l \le i \le s_h$$

  We complete the proof of subtree ① this way, and can move on with the proof.

Now we complete this case of the proof by deriving ②. Note that we begin by applying the WL rule, which removes some unneeded assumptions from our context and makes the proof more concise.

$$\text{(WL)}\ \frac{\Gamma \vdash \Delta}{\Gamma, P \vdash \Delta}$$

This rule is perfectly sound, which you can verify yourself as a proof exercise.

$$\text{WL}\ \dfrac{\text{cut}\ \dfrac{[*]_2\ \dfrac{\text{id}\ \dfrac{*}{\texttt{Mem}(i) = v, i \neq (e\,\&\,s_h) \mid s_l \vdash i \neq (e\,\&\,s_h) \mid s_l} \quad \texttt{Mem}(i) = v, i \neq (e\,\&\,s_h) \mid s_l \vdash 0 \leq i < U}{\texttt{Mem}(i) = v, i \neq (e\,\&\,s_h) \mid s_l \vdash \texttt{Mem}\{(e\,\&\,s_h) \mid s_l \mapsto \tilde{e}\}(i) = \texttt{Mem}(i)}}{\texttt{Mem}(i) = v, i \neq (e\,\&\,s_h) \mid s_l \vdash \texttt{Mem}\{(e\,\&\,s_h) \mid s_l \mapsto \tilde{e}\}(i) = v}}{\neg(s_l \leq i \leq s_h), \texttt{Mem}(i) = v, i \neq (e\,\&\,s_h) \mid s_l \vdash \texttt{Mem}\{(e\,\&\,s_h) \mid s_l \mapsto \tilde{e}\}(i) = v}$$

The unfinished portion of the proof assumes that $\texttt{Mem}(i) = v$ and $i \neq (e\,\&\,s_h) \mid s_l$ imply that $0 \leq i < U$. 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. 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:

$$\forall i. \neg(s_l \leq i \leq s_h) \wedge \texttt{Mem}(i) = v \rightarrow [\alpha]\texttt{Mem}(i) = v \tag{6}$$
$$\forall i. \neg(s_l \leq i \leq s_h) \wedge \texttt{Mem}(i) = v \rightarrow [\beta]\texttt{Mem}(i) = v \tag{7}$$

Then consider $(\omega, \ldots, \mu) \in [\![\alpha]\!]$ and $(\mu, \ldots, \nu) \in [\![\beta]\!]$. We have the following which says that the memory outside the sandbox in the initial state remains unchanged until the final state, for both α and β:

$$\forall i. \neg(s_l \leq i \leq s_h) \wedge \omega_M(i) = v \rightarrow \mu_M(i) = v \tag{8}$$
$$\forall i. \neg(s_l \leq i \leq s_h) \wedge \mu_M(i) = v \rightarrow \nu_M(i) = v \tag{9}$$

This follows directly from the semantics of $\texttt{Mem}(i)$, which refer to the memory component of states ω, μ, and ν, and the box modalities in (6) and (7).

Notice that the right side of the implication in (8) matches up with the latter half of the conjunction in the left side of the implication in (9). From this and the semantics of $\alpha;\beta$, it must then be that for all $(\omega, \ldots, \nu) \in [\![\alpha;\beta]\!]$ we can say:

$$\forall i. \neg(s_l \leq i \leq s_h) \wedge \omega_M(i) = v \rightarrow \nu_M(i) = v \tag{10}$$

Then (10) and the semantics of the box modality with $\alpha;\beta$ tell us that

$$\forall i. \neg(s_l \leq i \leq s_h) \wedge \texttt{Mem}(i) = v \rightarrow [\alpha;\beta]\texttt{Mem}(i) = v \tag{11}$$

This completes the inductive case for composition.

Rest of the proof: The remaining cases are left as exercises. The remaining base cases should be relatively straightforward 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. □

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

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.

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

the sandbox. This will be a topic of future lectures, where we will take a completely different approach to policy enforcement.

References

[ABEL09] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Transactions on Information and Systems Security*, 13(1):4:1–4:40, November 2009.

[SMB+10] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX Conference on Security*, 2010.

[YSD+09] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.