15-316: Software Foundations of Security and Privacy

Lecture Notes on Memory Safety & Software Fault Isolation

Matt Fredrikson

Carnegie Mellon University
Lecture 6

1 Introduction & Recap

In the previous lecture we talked about *bounded model checking* and *symbolic execution*, two fully-automated techniques for checking safety properties on programs. Bounded model checking relies on the [unfold] axiom for while loops, applying it repeatedly on each loop in the program up to a bounded number of times. Once this bound is reached, other axioms are used to reduce the proof to a series of *verification conditions* (VCs), or formulas of arithmetic whose validity implies the safety of the corresponding path, and can be discharged by an automated decision procedure.

Bounded model checking is useful in that it checks safety on all paths of length up to the given bound. However, the fact that it is exhaustive in this way sometimes means that it is too expensive to apply to programs with many possible paths, even for a modest bound. An alternative approach is to use symbolic execution to select a subset of paths that are of particular interest according to some external heuristic. Symbolic execution then follows the same basic procedure as bounded model checking, using sound axioms to rewrite the safety property applied to a path until an arithmetic VC is obtained.

Both techniques are useful for finding safety violations in programs, but are of limited utility when it comes to proving the *absence* of violations on all traces. There is good reason for this limitation, as the problem of proving the latter claim is undecidable. Our focus in this class is on making sure that security violations do not occur, so we need not limit ourselves to static proofs of correctness with respect to the policy. Another alternative is to take steps to change the execution of a program so that it does not violate safety. We will examine this further today, first extending our language to support pointer operations over memory, and then discussing memory safety as well as a more flexible mechanism for policy enforcement called *software fault isolation* [1, 2].

2 Pointers & memory safety

So far the programs that we have studied are not too interesting. While it is possible to write some non-trivial programs in the simple imperative language like Euclidean division, lots of interesting functionality like searching and sorting would be tedious to implement without arrays or some other form of indexed storage. So let's add a new feature to address this.

While most imperative programming languages support convenient dynamic memory allocation and access with syntax like `malloc` and `a[i]`, at the end of the day this is nothing more than syntactic sugar for managing a large integer-indexed array of values. We can add basic support for this to our language by introducing pointers, and adding an integer-indexed memory to our program state. Now terms in our language will have the following syntax.

$$e, \tilde{e} \ ::= \ x \mid c \mid e + \tilde{e} \mid e \cdot \tilde{e} \mid \texttt{Mem}(e)$$

The term $\texttt{Mem}(e)$ denotes the value obtained by evaluating e in the current state, and accessing the memory at the corresponding index. This takes care of reading from the memory array, now we add support for updating memory by introducing a new type of program command.

$$\alpha, \beta \ ::= \ x := e \mid \texttt{Mem}(e) := \tilde{e} \mid \texttt{assert}(Q) \mid \texttt{if}(Q)\, \alpha\, \texttt{else}\, \beta \mid \alpha; \beta \mid \texttt{while}(Q)\, \alpha$$

The command $\texttt{Mem}(e) := \tilde{e}$ evaluates e and $\tilde{e}$ in the current state, and sets the value of memory indexed at the value of e to the value of $\tilde{e}$.

Now for the semantics. We will need to track the value of variables as we did before with a mapping from variables to values. But we will also need to track the state of the memory, which we will formalize as a partial mapping from non-negative integers to values. Real machines don't have unlimited memory, which is why the mapping is partial: we assume that the memory can hold at most U values, so the mapping is only defined on $0 \leq i \leq U$.

We will continue to denote states by ω, and write $\omega_V(x)$ to refer to the value of the variable mapping, and $\omega_M(x)$ to refer to the memory array. The semantics of terms can now be defined as follows.

Definition 1 (Semantics of terms)**.** The *semantics of a term* e in a state ω is its value $\omega[\![e]\!]$. It is defined inductively by distinguishing the shape of term e as follows:

- $\omega[\![x]\!] = \omega_V(x)$ for variable x
- $\omega[\![c]\!] = c$ for number literals c
- $\omega[\![e \odot \tilde{e}]\!] = \omega[\![e]\!] \odot \omega[\![\tilde{e}]\!]$, where $\odot \in \{+, \times\}$
- $\omega[\![\texttt{Mem}(e)]\!] = \omega_M(\omega[\![e]\!])$ if $0 \leq \omega[\![e]\!] < U$, else undefined

Adding pointers to our language has led to a complication: now terms can be undefined. Specifically, if e evaluates to a negative number, or a number larger than the maximum memory size U, then the term $\texttt{Mem}(e)$ is not defined.

This complication manifests in how we define the semantics of formulas. Because terms can now be undefined in certain states, we need to account for this in the semantics of formulas that might include terms. Whenever a term in a formula is undefined in a particular state, then the value of the formula is as well.

Definition 2 (Semantics of arithmetic formulas)**.** The DL formula P is true in state ω, written $\omega \models P$, as inductively defined by distinguishing the shape of formula P:

1. $\omega \not\models \bot$, i.e., $\bot$ is true in no states
2. $\omega \models \top$, i.e., $\top$ is true in all states
3. $\omega \models e = \tilde{e}$ iff $\omega[\![e]\!] = \omega[\![\tilde{e}]\!]$ and both terms are defined in ω.
4. $\omega \models e \leq \tilde{e}$ iff $\omega[\![e]\!] \leq \omega[\![\tilde{e}]\!]$ and both terms are defined in ω.
5. $\omega \models P \wedge Q$ iff $\omega \models P$ and $\omega \models Q$ if P and Q are defined in ω.
6. $\omega \models P \vee Q$ iff $\omega \models P$ or $\omega \models Q$ if P and Q are defined in ω.
7. $\omega \models \neg P$ iff $\omega \not\models P$ if P is defined in ω.
8. $\omega \models P \rightarrow Q$ iff $\omega \not\models P$ or $\omega \models Q$ and P and Q are defined in ω.
9. $\omega \models P \leftrightarrow Q$ iff both are true or both false and P and Q are defined in ω.

Finally, we get to the semantics of programs. Obviously we need to add a new definition for the memory update command $\texttt{Mem}(e) := \tilde{e}$. But programs may contain terms and formulas, which we now know can be undefined in some states. We define the semantics of a program with a term or formula that is undefined in a state as aborting in the next subsequent state.

First some notation. If ω_M is a memory in state ω, then we write $\omega_M\{e \mapsto \tilde{e}\}$ to denote the new memory obtained by copying ω_M, and changing its mapping at $\omega[\![e]\!]$ to map to $\omega[\![\tilde{e}]\!]$. So suppose that $\omega_M(0) = 1, \omega_M(1) = 2$. Then $\omega_M\{1 \mapsto 3\}(0) = 1$ and $(\omega_M\{1 \mapsto 3\})(1) = 3$. We can apply this update notation multiple times, so that:

$$\omega_M\{1 \mapsto 3\}\{0 \mapsto 4\}(0) = 4, \omega_M\{1 \mapsto 3\}\{0 \mapsto 4\}(1) = 3$$

We'll adopt the convention that the rightmost update to a particular index is the one that we use when looking up values. So for example,

$$\omega_M\{1 \mapsto 3\}\{1 \mapsto 4\}(1) = 4$$

Definition 3 (Trace semantics of programs)**.** The *trace semantics* $[\![\alpha]\!]$ *of a program* α is the set of all its possible traces and is defined inductively as follows:

1. $[\![x := e]\!] = \{(\omega, \nu) \ : \ \omega[\![e]\!] \text{ is defined and } \nu = \omega \text{ except that } \nu_V(x) = \omega[\![e]\!]\} \cup \{(\omega, \Lambda) \ : \ \omega[\![e]\!] \text{ is not defined}\}$

2. $[\![\texttt{Mem}(e) := \tilde{e}]\!] = \{(\omega, \nu) \ : \ 0 \leq \omega[\![e]\!] \leq U, \omega[\![\tilde{e}]\!] \text{ defined}, \nu_M = \omega_M\{\omega[\![e]\!] \mapsto \omega[\![\tilde{e}]\!]\}\} \cup \{(\omega, \Lambda) \ : \ \neg(0 \leq \omega[\![e]\!] \leq U) \text{ or } \omega[\![\tilde{e}]\!] \text{ not defined}\}$

3. $[\![\texttt{assert}(Q)]\!] = \{(\omega) \ : \ \omega[\![e]\!] \text{ is defined and } \omega \models Q\} \cup \{(\omega, \Lambda) \ : \ \omega \models Q \text{ is not defined or } \omega \not\models Q\}$

4. $[\![\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]\!] = \{\sigma \in [\![\alpha]\!] \ : \ \sigma_0[\![e]\!] \text{ is defined and } \sigma_0 \models Q\} \cup \{\sigma \in [\![\beta]\!] \ : \ \sigma_0[\![e]\!] \text{ is defined and } \sigma_0 \not\models Q\} \cup \{(\omega, \Lambda) \ : \ \omega \models Q \text{ is not defined}\}$

5. $[\![\alpha; \beta]\!] = \{\sigma \circ \varsigma \ : \ \sigma \in [\![\alpha]\!], \varsigma \in [\![\beta]\!]\}$;
the composition of $\sigma = (\sigma_0, \sigma_1, \sigma_2, \dots)$ and $\varsigma = (\varsigma_0, \varsigma_1, \varsigma_2, \dots)$ is

$$\sigma \circ \varsigma := \begin{cases} (\sigma_0, \dots, \sigma_n, \varsigma_1, \varsigma_2, \dots) & \text{if } \sigma \text{ terminates in } \sigma_n \text{ and } \sigma_n = \varsigma_0 \\ \sigma & \text{if } \sigma \text{ does not terminate} \end{cases}$$

6. $[\![\texttt{while}(Q)\,\alpha]\!] = \{\sigma^{(0)} \circ \dots \circ \sigma^{(n)} \ : \ \text{for all } 0 \leq i < n\text{: } \sigma_0^{(i)} \models Q, \sigma^{(i)} \in [\![\alpha]\!], \text{ and } \sigma^{(n)} \text{ either doesn't terminate terminates where } \sigma_m^{(n)} \not\models Q\} \cup \{\sigma^{(0)} \circ \sigma^{(1)} \circ \sigma^{(2)} \circ \dots \ : \ \text{for all } i \in \mathbb{N}\text{: } \sigma_0^{(i)} \models Q, \sigma^{(i)} \in [\![\alpha]\!]\} \cup \{(\omega) \ : \ \omega \not\models Q\} \cup \{(\omega, \Lambda) \ : \ \omega \models Q \text{ not defined}\}$

While it may be tedious to track the presence of undefined terms and formulas through the evaluation of a program, we will see that this is central to the very definition of what memory safety means for a particular programming language.

Axioms and Proof Rules. Now we have semantics for programs with pointers and indexed memory, the next logical thing to do is find some useful axioms to help us reason about them.

Just as we had an axiom for assignment to variables, we have a similar axiom for updates to a pointer. But in the assignment axiom, we performed a syntactic substitution of the target variable in the postcondition. In this case we can readily see that looking for mere occurrences of a pointer expression will not suffice. Consider the following:

$$[x := 1; y := 1; \texttt{Mem}(x) := 0]\texttt{Mem}(y) \neq 0 \tag{1}$$

After executing the first two assignments, $\texttt{Mem}(x)$ and $\texttt{Mem}(y)$ point to the same memory location. So if we tried to close out a proof like the following:

$$[:=]\frac{\texttt{Mem}(y) \neq 0, x = 1, y = 1 \vdash \texttt{Mem}(y) \neq 0}{\texttt{Mem}(y) \neq 0, x = 1, y = 1 \vdash [\texttt{Mem}(x) := 0]\texttt{Mem}(y) \neq 0}$$

then we would be misled to say the least. Rather, we need to make sure that the update is reflected in any subsequent memory read to the same address, regardless of the syntactic form of the index term. Perhaps something like the following:

$$[\texttt{Mem}(e) := \tilde{e}]p(\texttt{Mem}) \leftrightarrow p(\texttt{Mem}\{e \mapsto \tilde{e}\}) \tag{2}$$

Now when we repeat the derivation from before,

$$\frac{\texttt{Mem}(y) \neq 0, x = 1, y = 1 \vdash \texttt{Mem}\{x \mapsto 0\}(y) \neq 0}{\texttt{Mem}(y) \neq 0, x = 1, y = 1 \vdash [\texttt{Mem}(x) := 0]\texttt{Mem}(y) \neq 0}$$

there is no way to close out the proof because $x = y$ and $\texttt{Mem}\{x \mapsto 0\}(y) = 0$. But this proof rule isn't sound, because what if e evaluates to an out-of-bounds value? We need to add an assertion that the value of e is within the correct range. This leads to the $[*]_=$ axiom, which combines Equation 2 with the in-bounds check.

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

Axiom $[*]_=$ takes care of what to do when we update memory, but we also need a way to reason about reads from memory. If we only ever reason about programs that never update memory, then this is easy because anything we need to know about its value at particular indices is already in our assumptions. We can then work with it like we would any other set of value, essentially treatig each index like a separate constant.

But what about programs that update memory and then read from it afterwards? There are two cases to cover: reading from an index that was previously written to, and reading from one that was not. In the first case, we have some memory $\texttt{Mem}\{e \mapsto \tilde{e}\}$ and we perform an access $\texttt{Mem}\{e \mapsto \tilde{e}\}(e')$ where $e = e'$ in the current state. Then the value that is read from memory will be $\tilde{e}$. But of course we also need to make sure that we are reading from an index in the appropriate range. This is captured in the $[*]_1$ rules.

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

In the case where $e \neq e'$, we use similar reasoning to conclude that $\texttt{Mem}\{e \mapsto \tilde{e}\}(e')$ takes whatever the value at index e' in Mem was *before* the update, i.e. $\texttt{Mem}(e')$. This gives us the $[*]_2$ rules.

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

The axiom $[*]_=$ and rules $[*]_1$, $[*]_2$ are sufficient to prove safety properties about programs with pointer operations.

Memory safety. The term *memory safety* refers to a set of properties that depends on the particulars of the language and instruction set architecture on which the program ultimately executes. But the common intent shared by all such properties is that programs satisfying memory safety never use pointers in a way that causes undefined behavior or forces the program to abort.

In our simplified language with pointers, any "bad" use of memory immediately leads to an abort on the corresponding trace, so we can define memory safety as the set of traces that do not abort due to a pointer read or write.

Definition 4 (Memory safety)**.** A program α satisfies memory safety if and only if for all $\sigma \in [\![\alpha]\!]$, whenever σ is finite and $\sigma_n = \Lambda$ then the last command executed on σ was not a pointer read or write.

One thing to notice is that when we use these axioms to prove *any* property about a program that uses pointers, we are forced to prove memory safety as well. The only case that me might forget to prove memory safety for is when a read is performed on memory without first having updated it. We can help ourselves remember to do this by replacing each command α that reads from memory in term $\texttt{Mem}(e)$ with the following composed command:

$$\texttt{assert}(0 \leq e < U); \alpha \tag{3}$$

This is a theorem that we are able to prove, in fact.

Theorem 5. *For any formula P and program α that has been rewritten according to (3), if $\Gamma \vdash [\alpha]P$, i.e. $[\alpha]P$ is provable from assumptions Γ using $[*]_=$, $[*]_1$, and $[*]_2$ in addition to other axioms of dynamic logic, then α satisfies memory safety.*

Proof. The way to prove this is by induction on the structure of proofs, just as we did to prove the soundness of the propositional sequent calculus and will do to prove Theorem 6 later in this lecture. This is a good exercise to complete on your own. □

3 Software Fault Isolation

Memory safety is an important policy in that we would want any useful program to be memory safe. But there are other sorts of safety policies on memory that we might want to enforce more selectively on only certain programs. For example, consider the following pseudocode that checks a configuration variable to determine whether or not to display and advertisement. If so, then the program runs α to render an ad on the screen.

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

Suppose that α was provided by the ad network, then we may have good reason not to trust α. Perhaps the ad network hires dumb programmers, and we fully expect their α to accidentially trample on memory that it isn't supposed to. Or maybe we got a great deal from *Vladimir's Fancy Bear Ad Network*, and despite Valdimir's assurances that his rendering code is "Totally 100% safe!", there are lingering doubts.

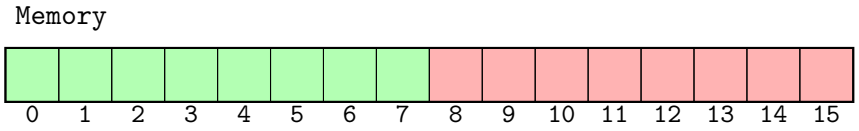
Luckily we know all about proving safety, so perhaps we can use logic and deduction to show that our program remains safe. What could α do to ruin our day? One thing is that because it executes in the state space of our original code, it can change the values of any variables at will to whatever it likes. Perhaps that's not so bad, because our program only uses a limited number of variables and we have a lot of memory to work

with. But α could also change memory arbitrarily, and this is certainly a bad thing. It could also read the contents of memory and render them to the screen, or worse yet, send them back to the ad network. This is also a bad thing that we want to prevent.

3.1 Sandboxing safety

One solution is to create a virtual sandbox for α to play in. We will give it free reign over a limited region of memory, and construct our program so that by the time α runs, our correctness doesn't depend on the contents of that region. We will also let it do whatever it wants with the variables, isolating the rest of our program from the effects of these operations by first saving all of our variables to a part of memory outside α's sandbox region, and restoring them after α finishes.

Supposing our machine only has a very limited 16-element memory, our segmentation would look something like the following with the parts shaded green comprising the safe memory set aside for our program, and the parts in red the sandbox for α.



Now that we have decided on a safety policy with which to execute α, we need to figure out how to actually enforce it in our program. Intuitively, our policy defines a "bad thing" that is any memory access outside of the sandbox region defined by upper and lower bounds s_l, s_h. So we might reasonably enforce the policy by first checking that the index i of any memory access operation in α satisfies $s_l \leq i \leq s_h$ before executing the operation. Luckily our language contains $\texttt{assert}(Q)$ commands, which come in handy when implementing such checks: if the check fails, the trace aborts rather than violating the policy.

So taking stock of our language, we propose to do the following *instrumentation* of α.

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

  $$\texttt{assert}(s_l \leq e \leq s_h); \texttt{Mem}(e) := \tilde{e}$$

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

- Replace any command β containing the term $\texttt{Mem}(e)$ with the command:

  $$\texttt{assert}(s_l \leq e \leq s_h); \beta$$

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

This seems pretty convincing. Our language is fairly simple, so we're pretty sure that all our bases our covered in terms of sandboxing α. The assertions themselves are a straightforward reflection of our sandboxing policy.

The downside to this type of enforcement is that any violation of the sandboxing policy, regardless of whether it is inadvertent or intentionally malicious, will cause our entire program to abort. This is less than ideal, as the malice or incompetence of α's developers still has a direct impact on the functioning of our code. Perhaps we can do better.

3.2 Isolating 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:

$$(e \,\&\, \texttt{0b1111}) \mid \texttt{0b1000} \tag{4}$$

From now on, we will use hexadecimal rather than binary when writing such constants, so Equation 5 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:

$$(e \,\&\, s_h) \mid s_l \tag{5}$$

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

$$0 \leq s_l \leq s_h < U \tag{6}$$

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.

$$\forall i. \neg(s_l \leq i \leq s_h) \land \texttt{Mem}(i) = v_i \rightarrow [\alpha]\texttt{Mem}(i) = v_i \tag{7}$$

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 6 does.

Theorem 6. *Let α be a program whose memory update commands have been instrumented as prescribed by software fault isolation. 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 7 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.

$$\forall R\,\frac{\to R,\wedge L\,\dfrac{[*]=\dfrac{\neg(s_l \le i \le s_h), \texttt{Mem}(i) = v_i \vdash \texttt{Mem}\{(e\,\&\,s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i}{\neg(s_l \le i \le s_h), \texttt{Mem}(i) = v_i \vdash [\texttt{Mem}((e\,\&\,s_h) \mid s_l) := \tilde{e}]\texttt{Mem}(i) = v_i}}{\vdash \neg(s_l \le i \le s_h) \wedge \texttt{Mem}(i) = v_i \to [\texttt{Mem}((e\,\&\,s_h) \mid s_l) := \tilde{e}]\texttt{Mem}(i) = v_i}}{\vdash \forall i. \neg(s_l \le i \le s_h) \wedge \texttt{Mem}(i) = v_i \to [\texttt{Mem}((e\,\&\,s_h) \mid s_l) := \tilde{e}]\texttt{Mem}(i) = v_i}$$

At this point we need to split into cases, 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:

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

Continuing with subtree ①:

$$\neg L\,\frac{\text{G}\,\dfrac{\mathbb{Z}_M\,\dfrac{*}{i = (e\,\&\,s_h) \mid s_l \vdash s_l \le i \le s_h}}{\texttt{Mem}(i) = v_i, i = (e\,\&\,s_h) \mid s_l \vdash s_l \le i \le s_h, \texttt{Mem}\{(e\,\&\,s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i}}{\neg(s_l \le i \le s_h), \texttt{Mem}(i) = v_i, i = (e\,\&\,s_h) \mid s_l \vdash \texttt{Mem}\{(e\,\&\,s_h) \mid s_l \mapsto \tilde{e}\}(i) = v_i}$$

We used the generalization rule to reduce the obligation to a claim about machine arithmetic, namely:

$$i = (e\,\&\,s_h) \mid s_l \to s_l \le i \le s_h \tag{8}$$

The generalization rule is given below.

$$\text{(G)}\ \frac{\Gamma_1 \vdash P}{\Gamma_1, \Gamma_2 \vdash P, \Delta}$$

Intuitively, all that G says is that if we want to prove that either P or Δ holds under assumptions Γ_1, Γ_2, then it suffices to prove that P holds only assuming Γ_1. The statement $\Gamma_1 \vdash P$ is more general than $\Gamma_1, \Gamma_2 \vdash P, \Delta$, and in fact encompasses the cases where Γ_2 is also assumed or Δ is also true.

Now we complete this case by deriving ②.

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

The unfinished portion of the proof is covered by the theorem's claim: that all *valid* memory indices remain the same. We chose not to formalize that indices are valid as an assumption, mainly to keep the proof less cluttered. But adding it and closing out this remaining branch is not difficult.

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

$$\forall i.\neg(s_l \leq i \leq s_h) \wedge \texttt{Mem}(i) = v_i \rightarrow [\alpha]\texttt{Mem}(i) = v_i \tag{9}$$
$$\forall i.\neg(s_l \leq i \leq s_h) \wedge \texttt{Mem}(i) = v_i \rightarrow [\beta]\texttt{Mem}(i) = v_i \tag{10}$$

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

$$\forall i.\neg(s_l \leq i \leq s_h) \wedge \omega_M(i) = v_i \rightarrow \mu_M(i) = v_i \tag{11}$$
$$\forall i.\neg(s_l \leq i \leq s_h) \wedge \mu_M(i) = v_i \rightarrow \nu_M(i) = v_i \tag{12}$$

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

$$\forall i.\neg(s_l \leq i \leq s_h) \wedge \omega_M(i) = v_i \rightarrow \nu_M(i) = v_i \tag{13}$$

Then (13) and the semantics of $[\alpha;\beta]$ tell us that

$$\forall i.\neg(s_l \leq i \leq s_h) \wedge \texttt{Mem}(i) = v_i \rightarrow [\alpha;\beta]\texttt{Mem}(i) = v_i \tag{14}$$

This completes the proof.

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. □

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.

References

[1] 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.

[2] 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.

Aside: Rules for quantifiers

Our proof of ④ 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.

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