15-316: Software Foundations of Security and Privacy

Lecture Notes on Memory Safety & Sandboxing

Matt Fredrikson

Carnegie Mellon University
Lecture 5

1 Introduction & Recap

In the previous lecture we looked into proving that programs satisfy safety properties given as formulas in the first-order dynamic logic. In particular, we can write contract properties with precondition P and postconditions Q for a program α as:

$$P \rightarrow [\alpha]Q \tag{1}$$

If this formula is valid, then it means that in every state ω, if $\omega \models P$ then after all terminating runs of α starting in ω the final state $\nu \models Q$.

So far the programs that we have studied are not too interesting when it comes to concerns about security—they can perform operations over integers, but they can't interact with memory or arrays. Today we will start by extending our language to support pointer operations over memory, and then discuss *memory safety* as well as a more flexible mechanism for policy enforcement called *software fault isolation*. Importantly, we will build on our understanding of sound axioms and proof rules to justify the correctness of this mechanism.

2 Memory Safety

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.

$$e, \tilde{e} \ ::= \ x \mid c \mid e + \tilde{e} \mid e \cdot \tilde{e} \mid *(e) \tag{2}$$

The term $*(e)$ 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.

$$\alpha, \beta \ ::= \ x := e \mid *(e) := \tilde{e} \mid \texttt{assert}(Q) \mid \texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta \mid \alpha; \beta \mid \texttt{while}(Q)\,\alpha \tag{3}$$

The command $*(e) := \tilde{e}$ 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 $\odot \in \{+, \times\}$
- $\omega[\![*(e)]\!] = \omega_M(\omega[\![e]\!])$ 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 $*(e)$ 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 $*(e) := \tilde{e}$. 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. $[\![x := e]\!] =$ $\{(\omega, \nu) \ : \ \omega[\![e]\!]$ is defined and $\nu = \omega$ except that $\nu_V(x) = \omega[\![e]\!]\} \cup$
   $\{(\omega, \Lambda) \ : \ \omega[\![e]\!]$ is not defined$\}$
2. $[\![*(e) := \tilde{e}]\!] =$ $\{(\omega, \nu) \ : \ 0 \leq \omega[\![e]\!] \leq U, \omega[\![\tilde{e}]\!]$ defined, $\nu_M = \omega_M\{\omega[\![e]\!] \mapsto \omega[\![\tilde{e}]\!]\}\} \cup$
   $\{(\omega, \Lambda) \ : \ \neg(0 \leq \omega[\![e]\!] \leq U)$ or $\omega[\![\tilde{e}]\!]$ not defined$\}$
3. $[\![\texttt{assert}(Q)]\!] =$ $\{(\omega, \omega) \ : \ \omega[\![e]\!]$ is defined and $\omega \models Q\} \cup$
   $\{(\omega, \Lambda) \ : \ \omega \models Q$ is not defined or $\omega \not\models Q\}$
4. $[\![\texttt{if}(Q)\, \alpha\, \texttt{else}\, \beta]\!] =$ $\{\sigma \in [\![\alpha]\!] \ : \ \sigma_0[\![e]\!]$ is defined and $\sigma_0 \models Q\} \cup$
   $\{\sigma \in [\![\beta]\!] \ : \ \sigma_0[\![e]\!]$ is defined and $\sigma_0 \not\models Q\} \cup$
   $\{(\omega, \Lambda) \ : \ \omega \models Q$ 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

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

$$
\begin{aligned}
6.\ [\![\texttt{while}(Q)\,\alpha]\!] = \ & \{\sigma^{(0)} \circ \cdots \circ \sigma^{(n)} \ : \ \text{for all } 0 \le i \le n\text{: } \sigma_0^{(i)} \models Q,\ \sigma^{(i)} \in [\![\alpha]\!], \text{ and} \\
& \ \sigma^{(n)} \text{ either doesn't terminate, or terminates with } \sigma_m^{(n)} \not\models Q\} \cup \\
& \{\sigma^{(0)} \circ \sigma^{(1)} \circ \sigma^{(2)} \circ \ldots \ : \ \text{for all } i \in \mathbb{N}\text{: } \sigma_0^{(i)} \models Q,\ \sigma^{(i)} \in [\![\alpha]\!]\} \cup \\
& \{(\omega) \ : \ \omega \not\models Q\} \cup \\
& \{\sigma^{(0)} \circ \cdots \circ \sigma^{(n)} \circ (\Lambda) \ : \ \text{for all } 0 \le i \le n\text{: } \sigma_0^{(i)} \models Q,\ \sigma^{(i)} \in [\![\alpha]\!], \text{ and} \\
& \ \sigma^{(n)} \text{ terminates with } \sigma_m^{(n)} \models Q \text{ not defined}\} \cup \\
& \{(\omega, \Lambda) \ : \ \omega \models Q \text{ not defined}\}
\end{aligned}
$$

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:

$$[x := 1; y := 1; *(x) := 0]*(y) \neq 0 \tag{4}$$

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

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

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:

$$[*(e) := \tilde{e}]p(*) \leftrightarrow p(*\{e \mapsto \tilde{e}\}) \tag{5}$$

Now when we repeat the derivation from before,

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

there is no way to close out the proof because $x = y$ and $*\{x \mapsto 0\}(y) = 0$. 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 5 with the in-bounds check.

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

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 the index like we would any other value mentioned in a program.

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 $*\{e \mapsto \tilde{e}\}$ and we perform an access $*\{e \mapsto \tilde{e}\}(e')$ 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 $[*]_1$ rule.

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

Note that because we must prove that $e = e'$ in this case, for the right premise it is fine to alternatively prove that $\Gamma \vdash 0 \leq eU$ for the right premise, as this will not affect soundness. In the case where $e \neq e'$, we use similar reasoning to conclude that $*\{e \mapsto \tilde{e}\}(e')$ takes whatever the value at index e' in $*$ was *before* the update, i.e. $*(e')$. This gives us the $[*]_2$ rules.

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

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

A syntactical sidenote. The axiom and rules that we introduced ask us to explicitly keep track of the updates that occur as we work through a dynamic logic formula using the $*\{e \mapsto \tilde{e}\}$ notation. As a result, we could end up with a verification condition that lists out a long sequence of upates, e.g.,

$$a < U \land a \geq c \rightarrow *\{a \mapsto b\}\{c \mapsto d\}\{a - c \mapsto 5\}(e) \geq 100$$

While the notation may seem verbose, we need to know the locations and values of potentially all past assignments, and the order in which they occurred. So, this is simply a way of accounting for all of the information needed to reason about the consequences of memory updates.

However, this notation is only useful when we are doing such reasoning. We need not (and should not) write programs that mention updated memories explicitly. For example, one might be tempted to write a program like the following:

$$*\{y \mapsto 0\}(x) := 1$$

Or perhaps,

$$x := *\{z \mapsto 1\}(y)$$

While these examples accomplish the task of expressing two computations with one command, they are not syntactically correct by our earlier definitions (defs. 2, 3), where we intend for the $*$ to be *literally* those characters, and nothing else. This is not just an arbitrary preference. Consider the following program, which has (to be generous) ambiguous semantics:

$$*(x) := 1; y := *\{x \mapsto 0\}(x)$$

To summarize, keep programs simple, and only use $*$ when writing memory operations. The only time that you should add explicit updates to the right of $*$ is when you are conducting a proof, and you use the $[*]_=$ axiom.

Example. Let's see a brief example of how to use the update-over-write and memory update rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{①}} \quad \text{id}\dfrac{}{0 \le w < U, x = a, y = b, w = z \vdash y = b}\,*}{0 \le w < U, x = a, y = b, w = z \vdash y = b \land *\{w \mapsto x\}(z) = a}\,\land\text{R}}{0 \le w < U, x = a, y = b, w = z \vdash [*(w) := x]y = b \land *(z) = a}\,[*]_=}{0 \le w < U, x = a, y = b, w = z \vdash [*(w) := x][x := y]x = b \land *(z) = a}\,[:=]}{0 \le w < U, x = a, y = b, w = z \vdash [*(w) := x][x := y][y := *(z)]x = b \land y = a}\,[:=]}{0 \le w < U, x = a, y = b, w = z \vdash [*(w) := x; x := y; y := *(z)]x = b \land y = a}\,[;],[;]$$

Now we finish subtree ①. It's clear that we want to use $[*]_1$, because we know that we are reading from the location that we've already updated. We need to be able to show that the index we're looking up is the same one that we in fact updated before, which should be easy since our assumptions already have that $w = z$. We also need to show that the index is in bounds, which follows from our assumptions after applying an equality on the left to substitute z for w.

$$\dfrac{\dfrac{\text{id}\dfrac{}{0 \le w \le U, x = a, y = b, w = z \vdash w = z}\,*}{0 \le w < U, x = a, y = b, w = z \vdash *\{w \mapsto x\}(z) = a}\,[*]_1 \qquad \text{id}\dfrac{}{0 \le z < U, x = a, y = b, w = z \vdash 0 \le z < U}\,* \quad =\text{L}\dfrac{}{0 \le w < U, x = a, y = b, w = z \vdash 0 \le z < U}}{}$$

2.1 Defining memory safety.

Moving on, we're in a good place to state what we mean by memory safety for our language with pointer operations. Coloquially, the term *memory safety* refers to a set of properties that depends on the particulars of the language that a program is written in. In C and C++, where programmers are given the freedom to perform arbitrary pointer arithmetic, but tasked with the responsibility of managing memory on their own, memory safety primarily refers to invalid accesses, such as buffer overflows and use after free errors, as well as uninitialized and null pointers. In "managed" languages like Java and C#, where array bounds are always checked by the platform and arbitrary pointer arithmetic is forbidden, the main issue is null pointer dereference.

Our language is quite a bit simpler than either C or Java, so the relevant memory safety issues are less numerous. Essentially, memory safety in our language means that any time a memory read or update is made, the term specifying the index evaluates to a value in $[0, U]$. In our simplified language with pointers, any "bad" use of memory immediately leads to an abort on the corresponding trace, so we can informally define memory safety as the set of traces that do not abort due to a pointer read or write. Definition 4 expands on this, and makes it less ambiguous.

Definition 4 (Memory safety)**.** A program α satisfies memory safety if and only if for all traces $\sigma \in [\![\alpha]\!]$, and any state σ_i in that trace wherein α executes a memory operation:

- If the operation is a memory read $*(e)$, then e is in bounds, i.e. $\sigma_i \vDash 0 \leq e < U$.
- If the operation is an update $*(e) := \tilde{e}$, then e is in bounds, $\sigma_i \vDash 0 \leq e < U$.

□

One question that is often asked about the second bullet of Definition 4 is, *why is there only a condition on e, and not also $\tilde{e}$?* This question is motivated by the fact that $\tilde{e}$ may itself contain a memory read that could violate the safe bounds. But if this were the case, then the program would already fail to satisfy the first bullet, so we would not fail to notice that it is unsafe!

Another 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 or writes to memory in term $*(e)$ with the following composed command:

$$\texttt{assert}(0 \leq e < U); \alpha \tag{6}$$

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 (6), 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. This is a good exercise to complete on your own. □

3 Sandboxing memory access

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.

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

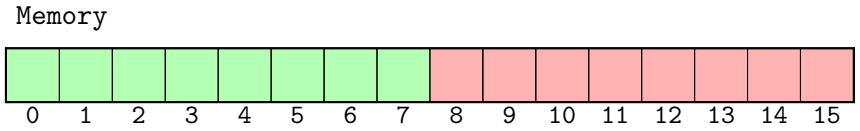
Suppose that α was provided by the ad network, then we may have good reason not to trust α. Perhaps the ad network hires dumb programmers, and we fully expect their α to accidentially trample on memory that it isn't supposed to. Or maybe we got a great deal from *Fancy Bear Ad Network*, and despite assurances 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 $*(e) := \tilde{e}$ with a new composed command:

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

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

- Replace any command β containing the term $*(e)$ 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. In the next lecture, we will see how to isolate the rest of our programs from these effects using software fault isolation.