Lecture Notes on Memory Safety

15-316: Software Foundations of Security & Privacy
Frank Pfenning

Lecture 6
September 12, 2024

1 Introduction

The classic *buffer overflow attack* (about which you learn in 15-213 *Computer Systems*) allows a program to take control of your machine while otherwise innocuous code is executed. It exploits that accessing memory that hasn't been explicitly allocated by a program is undefined behavior in C and therefore *unsafe*.

In this lecture we define a simplistic model of memory and introduce memory write and read operations into our language. We then define unsafe behavior and investigate how to prove safety of programs accessing memory.

What can we do if we cannot prove safety, but we'd still like to run a program? One option is to rewrite the program by checking that any memory access is in bounds before running it. We show how this can be done for our language, and prove safety for the resulting program. This is one of techniques used in *sandboxing* which refers to running untrusted programs in a manner that prevents them from doing damage ("*inside a sandbox*"). There are commercial tools such as Intel's Pin that can instrument binary code for Intel instruction set architectures.

2 Writing and Reading Memory

Mathematically, we model memory as a map from $\mathbb{Z}$ (the index domain) to $\mathbb{Z}$ (the value domain). It is total, but may be *indeterminate* on some indices. In the programming language we assume indices are limited to the range from 0 to a fixed U, and accessing memory outside these bounds in *unsafe*.

As explained in Lecture 5, we would like to keep expressions *safe*, but possibly *indeterminate*. Unsafe behavior is then exhibited only by commands and the programs constructed from them. Sticking to this approach, we add a new kind of variable, M, that stands for memory, and the new commands

- $M[e_1] := e_2$ write e_2 into memory M at address e_1. This is unsafe if the value of e_1 is out of bounds.
- $x := M[e]$ set x to the contents of memory at address e into x. This is unsafe if the value of e is out of bounds.

The design decision that memory access takes place via commands means that you have to rewrite hypothetical code such as

$$M[x] := (M[x-1] + M[x+1])/2$$

in the more verbose form

$$\begin{array}{l} t_1 := M[x-1]\ ; \\ t_2 := M[x+1]\ ; \\ t_3 := \mathbf{divide}\ (t_1 + t_2)\ 2\ ; \\ M[x] := t_3 \end{array}$$

Next, we need to rigorously define the semantics of the new commands so that (a) we can implement them, and (b) we can prove soundness of our axioms and rules to reason about them (including their safety).

The first issue is how to track the contents of memory. For that purpose, we change our definition of the state ω. So far the state has been a total function from variables to integers, $\omega : \mathsf{Var} \to \mathbb{Z}$ where Var is the (countably infinite) set of variables. Now variables can also map to memory, which is a total function from integers to integers.

$$\omega : \mathsf{Var} \to (\mathbb{Z} \cup (\mathbb{Z} \to \mathbb{Z}))$$

We assume that programs map lowercase variables to integers and uppercase variables to memory, so that there is never any confusion between the two forms. Mathematically, we use the letter $H : \mathbb{Z} \to \mathbb{Z}$ (suggesting a *heap*). All our previous semantic definitions remain unchanged since all variables in those definitions stand for integers.

We begin with safe and unsafe memory reads.

$$\omega[\![x := M[e]]\!]\nu \quad \text{iff} \quad \omega[\![e]\!] = i \text{ and } \omega[\![M]\!] = H \text{ and } \nu = \omega[x \mapsto H(i)] \quad \text{provided } 0 \le i < \mathsf{U}$$

$$\omega[\![x := M[e]]\!]\lightning \quad \text{iff} \quad \omega[\![e]\!] = i \text{ and } \textbf{not } 0 \le i < \mathsf{U}$$

It should be clear that unsafe programs continue to satisfy no postcondition.

Memory write follows the same intuition, we just have to make sure to suitably update the map defining the state of memory.

$$\omega[\![M[e_1] := e_2]\!]\nu \quad \text{iff} \quad \omega[\![e_1]\!] = i \text{ and } \omega[\![e_2]\!] = a \text{ and } \omega[\![M]\!] = H \text{ and } H' = H[i \mapsto a] \text{ and } \nu = \omega[M \mapsto H'] \quad \text{provided } 0 \le i < \mathsf{U}$$

$$\omega[\![M[e_1] := e_2]\!]\lightning \quad \text{iff} \quad \omega[\![e_1]\!] = i \text{ and } \textbf{not } 0 \le i < \mathsf{U}$$

3 Reasoning about Memory[1]

In order to reason about memory we need to introduce expressions that capture what we know about the state of memory. Mathematically, this leads us to the *theory of arrays* [McCarthy, 1962] which we can view as being constructed on top of the theory of arithmetic we have assumed so far. Because of the importance of arrays in imperative programming, some efficient decision procedures have been devised (see, for example, Stump et al. [2001]) and implemented in provers such as Z3 or the CVC family.

In the theory of arrays we have two expressions $\mathbf{read}\ H\ i$ and $\mathbf{write}\ H\ i\ a$ where H denotes an array, i and index into an array, and a a value stored in the array. Note that the expression $\mathbf{write}\ H\ i\ a$ denotes a "new" array; semantically $H[i \mapsto a]$. For us, both the index domain and the values are integers.

There are two axioms, called *read over write*, that allow us to reason about these expressions.

$$i = k \rightarrow \mathbf{read}\ (\mathbf{write}\ H\ i\ a)\ k = a$$
$$i \neq k \rightarrow \mathbf{read}\ (\mathbf{write}\ H\ i\ a)\ k = \mathbf{read}\ H\ k$$

In addition we have an *axiom of extensionality* which states that two arrays are equal if they agree on all elements. In our use of the theory of arrays, the quantifier ranges over integers.

$$(\forall i.H(i) = H'(i)) \rightarrow H = H'$$

As usual, we treat the new expressions as always denoting either an array or an integer, although the value may sometimes be indeterminate.

The right rules of the sequent calculus for these new commands are now relatively straightforward.

$$\frac{\Gamma \vdash 0 \leq e < \mathsf{U}, \Delta \quad \Gamma, x' = \mathbf{read}\ M\ e \vdash Q(x'), \Delta}{\Gamma \vdash [x := M[e]]Q(x), \Delta}\ [\mathbf{read}]R^{x'}$$

As for assignment, the x' must be chosen fresh in $[\mathbf{read}]R^{x'}$. The same is true for M' in the following rule.

$$\frac{\Gamma \vdash 0 \leq e_1 < \mathsf{U}, \Delta \quad \Gamma, M' = \mathbf{write}\ M\ e_1\ e_2 \vdash Q(M'), \Delta}{\Gamma \vdash [M[e_1] := e_2]Q(M), \Delta}\ [\mathbf{write}]R^{M'}$$

Even if an program can only mention a single array M, while reasoning about a program we need to be able to relate arrays before and after an assignment. Therefore, the knowledge about M in the antecedents Γ (or succedents Δ) must not conflict with knowledge about the state of the array after the write operation and M' must be fresh.

The aspect of these rules critical for safety is the first premise that requires us to prove safety (independently of the postcondition).

[1] only part of the presented in lecture; the remainder promised and provided here for reference

4 A Small Example of Memory Safety

Consider the following program to initialize memory up to n:

$$i := 0 \,;\, \textbf{while}\ (i < n)\{\, M[i] := i \,;\, i := i + 1 \,\}$$

This is patently unsafe: just consider $n = \mathsf{U} + 1$. Then the last time around the loop we will have $i = \mathsf{U}$, leading to an unsafe memory access at $M[\mathsf{U}]$.

We can add a precondition $n \leq \mathsf{U}$ and then try to prove safety with

$$n \leq \mathsf{U} \rightarrow [i := 0 \,;\, \textbf{while}\ (i < n)\{\, M[i] := i \,;\, i := i + 1 \,\}]\top$$

We could try $i \leq n$ but that's insufficient. Here is what preservation would require:

$$i \leq n, i < n \vdash [M[i] := i \,;\, i := i + 1]i \leq n$$

We note two problems: (1) safety will fail because we cannot prove that $0 \leq i$ and (2) we have lost the precondition $n \leq \mathsf{U}$ so we also cannot conclude that $i < \mathsf{U}$.

Let's try a more complex invariant:

$$0 \leq i \leq n \leq \mathsf{U}$$

Now preservation requires

$$0 \leq i \leq n \leq \mathsf{U}, i < n \vdash [M[i] := i \,;\, i := i + 1]\,(0 \leq i \leq n \leq \mathsf{U})$$

This reduces in two steps to

$$0 \leq i \leq n \leq \mathsf{U}, i < n, M' = \textbf{write}\ M\ i\ i, i' = i + 1 \vdash 0 \leq i' \leq n \leq \mathsf{U}$$

Fortunately, this is valid (even with the useless assumption about M'). Because our postcondition is just $\top$, that is easily seen to be true, but the loop invariant does not hold initially because

$$n \leq \mathsf{U} \vdash 0 \leq 0 \leq n \leq \mathsf{U}$$

is not valid (counterexample: $n = -1$). So we need to strengthen our precondition to

$$0 \leq n \leq \mathsf{U}$$

5 Guards

Consider the scenario where you are given the program from the previous section but no loop invariant. We might be able to guess a loop invariant, but if not we are stuck. The program looks unsafe, even with the precondition $0 \leq n \leq \mathsf{U}$. If we still need to run it, what can we do? One option would be *dynamic monitoring*: we track

memory accesses as the program executes and abort it if it attempts to do something unsafe. Another one is to *instrument it with guards* before memory accesses. These guards abort the program if the access would be unsafe and let it go on if they are safe. Aborting programs is considered safe, because aborting is actually a well-defined operation that does no harm (except to the running program, but it is its own fault if it tries to execute an unsafe command). For this purpose we need a new command $\mathbf{test}\ P$. In the literature on dynamic logic this is called a *guard* and written as $?P$. It has the following specification.

$$\begin{array}{lll} \omega[\![\mathbf{test}\ P]\!]\nu & \text{iff} & \omega \models P \text{ and } \nu = \omega \\ \omega[\![\mathbf{test}\ P]\!]\lightning & \text{iff} & \text{false} \end{array}$$

The program $\mathbf{test}\ \bot$ will not have a poststate, but it is also safe because it aborts. As a result, based on the definition of $\omega \models [\alpha]Q$ it is the case that

$$\omega \models [\mathbf{test}\ \bot]Q$$

for any ω and Q. This in turn means that $[\mathbf{test}\ \bot]Q$ is logically valid and $\cdot \vdash [\mathbf{test}\ \bot]Q$ should be derivable.

Just to be sure, let's recall the definition of $\omega \models [\alpha]Q$ from Lecture 5, page L5.4.

$$\begin{array}{lll} \omega \models [\alpha]Q & \text{iff} & \text{for every } \nu \text{ with } \omega[\![\alpha]\!]\nu \text{ we have } \nu \models Q \\ & & \text{and } \mathbf{not}\ \omega[\![\alpha]\!]\lightning \end{array}$$

This is a *partial correctness* statement: if there is no poststate ν such that $\omega[\![\omega]\!]\nu$, then the first part of the condition is vacuously true.

What does this mean for the axiom for $[\mathbf{test}\ P]Q$? If P is true, then Q should also be true. But if the test succeeds then we know P, so we conjecture (somewhat rashly, perhaps)

$$[\mathbf{test}\ P]Q \leftrightarrow (P \rightarrow Q)$$

What if P is false? Then the program $\mathbf{test}\ P$ has no poststate, and yet it is safe. Consequently $[\mathbf{test}\ P]Q$ should be true, and by this axiom it will be becase $\bot \rightarrow Q$ is valid.

Let's prove that this axiom is valid, just as in Theorem 1 of Lecture 5 we proved that $[\mathbf{assert}\ P]Q \leftrightarrow (P \wedge Q)$.

Theorem 1 *The axiom*

$$[\mathbf{test}\ P]Q \leftrightarrow (P \rightarrow Q)$$

is valid.

Proof: From right to left we set up for an arbitrary ω

$\omega \models P \rightarrow Q$ (assumption)

...

$\omega \models [\mathbf{test}\ P]Q$ (to show)

The conclusion holds if for every ν such that $\omega[\![\mathbf{test}\ P]\!]\nu$ we have $\nu \models Q$ (and **not** $\omega \models [\![\mathbf{test}\ P]\!]\lightning$, which is true).

So we assume $\omega[\![\mathbf{test}\ P]\!]\nu$ and have to show that $\nu \models Q$. By definition, this second assumption give us $\omega \models P$ and $\nu = \omega$.

By the first assumption also $\omega \models Q$ and since $\nu = \omega$ we have $\nu \models Q$

For the left to right direction, we set up for an arbitrary ω

$\omega \models [\mathbf{test}\ P]Q$ (assumption)
...
$\omega \models P \to Q$ (to show)

So we assume $\omega \models P$ and it remains to show that $\omega \models Q$. Since $\omega \models P$, the first assumption gives us $\nu \models Q$ for any ν with $\omega[\![\mathbf{test}\ P]\!]\nu$. We can use this for $\nu = \omega$ (since $\omega \models P$) to obtain $\omega \models Q$. □

We can easily turn the two directions of the axiom into right and left rules of the sequent calculus.

$$\frac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash [\mathbf{test}\ P]Q, \Delta}\ [\mathbf{test}]R \qquad \frac{\Gamma \vdash P, \Delta \quad \Gamma, Q \vdash \Delta}{\Gamma, [\mathbf{test}\ P]Q \vdash \Delta}\ [\mathbf{test}]L$$

Here is a little table on the differences between $\mathbf{assert}\ P$ and $\mathbf{test}\ P$.

Poststate	$\omega[\![\mathbf{assert}\ P]\!]\nu$ iff $\omega \models P$ and $\nu = \omega$	$\omega[\![\mathbf{test}\ P]\!]\nu$ iff $\omega \models P$ and $\nu = \omega$
Safety	$\omega[\![\mathbf{assert}\ P]\!]\lightning$ iff $\omega \not\models P$	$\omega[\![\mathbf{test}\ P]\!]\lightning$ never
Axiom	$[\mathbf{assert}\ P]Q \leftrightarrow P \wedge Q$	$[\mathbf{test}\ P]Q \leftrightarrow (P \to Q)$

6 Sandboxing

Sandboxing unsafe behavior (including memory access through the read or write commands) proceeds as follows. We replace

- every memory read $x := M[e]$ with the program $\mathbf{test}\ 0 \le e < \mathsf{U}\ ;\ x := M[e]$
- every memory write $M[e_1] := e_2$ with the program $\mathbf{test}\ 0 \le e_1 < \mathsf{U}\ ;\ M[e_1] := e_2$
- every division $x := \mathbf{divides}\ e_1\ e_2$ with the program $\mathbf{test}\ e_2 \neq 0\ ;\ x := \mathbf{divides}\ e_1\ e_2$.

- every assertion **assert** P with the program **test** P

Now we can safely execute the program. Equally importantly, perhaps, we can prove the safety of the program transformed in this manner.

Theorem 2 (Safety of Sandboxed Programs) *Given a program α under precondition P, we obtain the sandboxed α' as defined in the preceding paragraph.*
Then $\cdot \vdash P \to [\alpha']\top$.

Proof: We prove safety using the loop invariant $\top$ for every loop. Since any potentially unsafe command is immediately preceded by a guard, the safety condition incorporated into the rule will be provable since it is exactly the assumption enabled by the postcondition of the guard.

More formally, this proof would be carried out by an *induction over the structure of formulas and programs*. □

There are two optimizations that come to mind. We can introduce fresh temporaries in order to avoid recomputing the value of expressions. For example, instead of **test** $0 \leq e < \mathsf{U}$; $x := M[e]$ we would insert $t := e$; **test** $0 \leq t < \mathsf{U}$; $x := M[t]$ for a fresh temporary variable t.

The other optimization is a bit trickier. At first one might think that if we can *prove* P when we encounter **test** P during the verification of safety we can replace it by **skip** (or **assert** $\top$, which should be equivalent). However, in conditionals the postcondition is replicated:

$$[\textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta]Q \leftrightarrow (P \to [\alpha]Q) \wedge (\neg P \to [\beta]Q)$$

When the postcondition contains a program (which arises from sequential composition, for example), this program may be proved twice, once in each branch. A similar remark applies if we unfold loops because the program α is replicated.

So we can only replace **test** P with **skip** only if for all occurrences of $[\textbf{test } P]Q$ in a safety proof we can prove P.

Returning to our earlier example, we can sandbox

$$n \leq \mathsf{U} \to [i := 0 \,;\, \textbf{while } (i < n)\{\, M[i] := i \,;\, i := i + 1 \,\}]\top$$

as

$$n \leq \mathsf{U} \to [i := 0 \,;\, \textbf{while } (i < n)\{\, \textbf{test} 0 \leq i < \mathsf{U} \,;\, M[i] := i \,;\, i := i + 1 \,\}]\top$$

Without a loop invariant and stronger precondition we can't eliminate the guard. With the additional information from Section 4 it would be redundant and can be dropped.

7 Summary

Since it has been a while, we summarize the language so far. We restrict programs from containing certain expressions with indeterminate behavior to retain the property that for every given prestate ω, every program has three possible outcomes: a poststate ν, or no poststate in which case it may be safe or unsafe ($\lightning$).

Variables	x, y, z			
Memory	M			
Constants	c	$::=$	$\ldots, -1, 0, 1, \ldots$	
Expressions	e	$::=$	$c \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2$	determinate
		$\mid$	$e_1/e_2 \mid \textbf{read}\ M\ e \mid \textbf{write}\ M\ e_1\ e_2$	may be indeterminate
Programs	α, β	$::=$	$x := e \mid \alpha\ ;\ \beta \mid \textbf{skip}$	
		$\mid$	$\textbf{if}\ P\ \textbf{then}\ \alpha\ \textbf{else}\ \beta \mid \textbf{while}\ P\ \alpha$	
		$\mid$	$\textbf{test}\ P$	safe
		$\mid$	$\textbf{assert}\ P \mid x := \textbf{divide}\ e_1\ e_2$	may be unsafe
		$\mid$	$x := M[e] \mid M[e_1] := e_2$	may be unsafe
Formulas	P, Q	$::=$	$e_1 \leq e_2 \mid e_1 = e_2 \mid \top \mid \bot$	
		$\mid$	$P \wedge Q \mid P \vee Q \mid P \to Q \mid P \leftrightarrow Q \mid \neg P$	
		$\mid$	$\forall x.\, P(x) \mid \exists.\, P(x)$	
		$\mid$	$[\alpha]Q \mid \langle\alpha\rangle Q$	

References

John McCarthy. Towards a mathematical science of computation. In *2nd IFIP Congress on Information Processing*, pages 21–28, Munich, Germany, August 1962. North-Holland.

Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *Symposium on Logic in Computer Science (LICS 2001)*, pages 29–37, Boston, Massachusetts, June 2001. IEEE Computer Society.