15-316: Software Foundations of Security and Privacy **Spring 2017**

Lecture 14: Proof-Carrying Code

Lecturer: Jean Yang *Based on material from: George Necula*

Prior to taking this class, you might have thought that dealing in assembly code was hopeless when it comes to security. Earlier this semester, we had talked about how we can use types in assembly code to prove safety properties about the code. Now that we've seen how to formalize and prove properties about high-level code, we can go back and formalize some of the ideas from proof-carrying code.

14.0.1 Basic Premises of Proof-Carrying Code

Recall that proof-carrying code is a general framework that allows us to quickly and easily check that code someone else gave to us has cerain safety properties. The key insight in PCC is the requirement that code comes with an *explanation* of why it is correct, and all we need to trust is the *verifier*, which verifies the code with respect to the explanation. In this lecture, we will look at a core PCC language for assembly instructions.

A PCC system is composed of two parts: an *agent* and a *host system*. The agent contains executable content, and checking support. The PCC infrastructure of the host system contains a *verification condition generator* (VCGen) and a *safety policy* that a *checker* uses to determine whether the agent code is safe or not. The role of the VCGen is to scan the executable content of the agent and check simple syntactic conditions, for instance that direct jumps are within the code boundary. Each time VCGen encounters a potentially dangerous instruction, it asks the checker to verify that the instruction executes safely in the current context.

In order to check agent code, two things happen.

- **Translation.** First, VCGen "compiles" programs to logical formulae that are relevant to proving the desired security policies. VCGen can usually be simple because it relies on the Checker to verify safety requirements. VCGen relies on *code annotations* that are part of the checking support and packaged with the agent, putting most of the burden on handling complex control-flow issues on the agent producer.
- **Checking.** Then the Checker module verifies for VCGen that all dangerous instructions are used in a safe context. The checker requires that VCGen formulates the safety *preconditions* of the dangerous instructions as formulas in a logic, called *verification conditions*. Again, the Checker expects that the agent provides the formal proofs needed to verify the verification conditions.

The *safety policy* in the VCGen infrastructure specifies the precise logic that VCGen uses to encode the verification conditions, along with trusted proof rules that can be used in the safety proofs supplied by the agent producer. The customizable elements of the safety policy are as follows:

- The language of symbolic expressions and formulas for expressing verification conditions.
- A set of function preconditions and postconditions for all functions in the interface between the host and agent.
- A set of proof rules for verification conditions.

```
type maybepair = Int of int | Pair of int * int
let rec sum(acc: int, x: maybepair list) =
  match x with
  | nil -> acc
  | (Int i) :: tail -> sum(acc + i, tail)
  | (Pair (l, r)) :: tail -> sum (acc + l + r, tail)
```

Figure 14.1: OCaml source code for example agent.

```
sum:                                ; r_x: maybepair list
Loop:
        if r_x ≠ 0 jump LCons       ; Is r_x empty?
        r_R := r_acc
        return
LCons:  r_t := Mem[r_x]             ; Load the first data
        if even(r_t) jump LPair
        r_t := r_t div 2
        r_acc := r_acc + r_t
        jump LTail
LPair:  r_s := Mem[r_t]             ; Get the first pair element
        r_acc := Mem[r_acc + r_s]
        r_t := Mem[r_t + 4]         ; and the second element
        r_acc := r_acc + r_t
LTail:  r_x := Mem[r_x + 4]
        jump Loop
```

Figure 14.2: Assembly code for our agent code.

14.1 A Simple Type-Checking Example

In this lecture (and for Assignment 3), you will be working with an example where we check a simple type-safety policy for an agent written in assembly language. Suppose we have the OCaml source shown in Figure 14.1. We will now write this agent in assembly, and show how to formulate a safety policy that can type-check the assembly according to the OCaml types.

Now let us decide how to represent lists and the `maybepair` type. For this example, we will represent a list as either the value 0 (for the empty list) or a pointer to a two-word memory area, where the first word contains a list element, and the second word contains a pointer to the tail of the list. To represent `maybepair` efficiently, we ensure that any element of kind `Pair`(x, y) is an even-valued pointer into a two-word memory area containing x and y, and we represent an element of kind `Int` x as the integer $2x + 1$, to ensure that it is odd.

For this example, we will use the following simple generic assembly language:

$\mathtt{r}_x := e$	assign result of evaluating e to register $\mathtt{r}_x$
$\mathtt{r}_x := \mathtt{Mem}[e]$	load $\mathtt{r}_x$ from address e
$\mathtt{Mem}[e'] := e$	store the result of e to address e'
`jump` L	jump to a label L
`if` e `jump` L	branch to label L if e is true
`return`	return from the current function

We show the assembly version of our code in Figure 14.2. On entry, registers $\mathtt{r}_x$ nd $\mathtt{r}_{acc}$ contain the values of the arguments `x` and `acc`. The code also uses temporary registers $\mathtt{r}_r$ and $\mathtt{r}_s$, and the return value is in $\mathtt{r}_R$.

Informal safety policy. We now want to specify what it means for the assembly to adhere to a simple type safety property. Informally, the safety policy requires that all memory *reads* be from pointers that are either:

1. non-null lists, in which case we can read either the first or second field in a list cell, or
2. pointers to elements with constructor `Pair`.

For memory *writes*, the safety policy makes the following requirements:

1. In the first word of a list cell we can write either an odd value or an even value that is a pointer to an element of `Pair`.
2. In the second word of a list cell we can write either zero or a pointer to some list cell.

These requirements make sure the contents of accessible memory locations are consistent to the assigned types.

14.2 Formalizing the Safety Policy

We formalize the safety policy for PCC with respect to a first-order language of symbolic expressions and formulas, shown below:

$$\begin{array}{llll} \text{Formulas} & F & ::= & \mathtt{true} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Rightarrow F_2 \mid \forall x.F \mid \exists x.F \\ & & & \mid \mathtt{addr}\ E_a \mid E_1 = E_2 \mid E_1 \neq E_2 \mid F\ E_1 \ldots E_n \\ \text{Expressions} & E & ::= & x \mid \mathtt{sel}\ E_m\ E_a \mid \mathtt{upd}\ E_m\ E_a\ E_v \mid F\ E_1 \ldots E_n \end{array}$$

The VCGen produces formula ($\mathtt{addr}\ E_a$) as a verification condition for a memory read or memory write to address E_a, and holds whenever E_a denotes a valid address. The construct ($\mathtt{sel}\ E_m\ E_a$) denotes the contents of the memory address denoted E_a in memory state E_m. The construct ($\mathtt{upd}\ E_m\ E_a\ E_v$) denotes a new memory state that is obtained by storing E_v at address E_a in memory state E_m. As an example, we can express the contents of the memory address c obtained from memory state m after writing value 1 at address a, followed by value 2, as:

$$\mathtt{sel}\ (\mathtt{upd}\ (\mathtt{upd}\ m\ a\ 1)\ b\ 2)\ c \tag{14.1}$$

As we said before, we can think of PCC infrastructures as allowing us to "swap in" different safety policies. Here, we will define an example safety policy extends the syntax of the logic by defining new expression and formula constructors:

$$\begin{array}{llll} \text{Word types} & W & ::= & \mathtt{int} \mid \mathtt{ptr}\ \{S\} \mid \mathtt{list}\ W \mid \{x \mid F(x)\} \\ \text{Structure types} & S & ::= & W \mid W\mathtt{;}\ S \\ \text{Formulas} & F & ::= & \ldots \mid E\mathtt{:}\ W \mid \mathtt{listinv}\ E_m \end{array}$$

In addition to the expected types, we have a word type $\{x \mid F(x)\}$ containing all values for which the formula F is true, and the `listinv` formula over lists that says all elements of memory E_m satisfy the *representation invariant* for all lists of maybe-pairs. We can write the low-level version of the typing judgment x`:  maybepair list` as

$$x : \mathtt{list}\ \{y \mid (\mathtt{even}\ y) \Rightarrow y : \mathtt{ptr}\ \{\mathtt{int}, \mathtt{int}\}\} \tag{14.2}$$

$$\textsc{AndI}\ \frac{F_1 \quad F_2}{F_1 \wedge F_2} \qquad \textsc{AndEL}\ \frac{F_1 \wedge F_2}{F_1} \qquad \textsc{AndER}\ \frac{F_1 \wedge F_2}{F_2} \qquad \textsc{ImpI}_{F_1}\ \frac{\vdots\ F_2}{F_1 \Rightarrow F_2} \qquad \textsc{ImpE}\ \frac{F_1 \Rightarrow F_2 \quad F_1}{F_2}$$

$$\textsc{Mem0}\ \frac{A = A'}{\texttt{sel}\ (\texttt{upd}\ M\ A\ V)\ A' = V} \qquad \textsc{Mem1}\ \frac{A \neq A'}{\texttt{sel}\ (\texttt{upd}\ M\ A\ V)\ A' = \texttt{sel}\ M\ A'}$$

Figure 14.3: Built-in proof rules.

Consider an alternative representation for `maybepair` where a value is a pointer to a tagged memory area containing a tag word where the tag 0 is followed by another word encoding an `Int`, and the tag 1 is following by two words encoding a `Pair`. The typing judgment for x: `maybepair list`, with memory m, would then be:

$$x : \texttt{list}\ \{y \mid \exists v.(v = \texttt{sel}\ m\ y) \wedge (v = 0 \Rightarrow y : \texttt{ptr}\ \{\texttt{int}, \texttt{int}\}) \wedge (v = 1 \Rightarrow y : \texttt{ptr}\ \{\texttt{int}, \texttt{int}, \texttt{int}\})\} \quad (14.3)$$

For the rest of this section we will use the abbreviation `mp_list` for `maybepair list`.

14.3 Preconditions and Postconditions

In the PCC infrastructure, the safety policy contains preconditions and postconditions of interface functions between the agent and host. At the assembly level, preconditions and postconditions are expressed in terms of argument and return registers, and thus also specify the calling convention. We model memory as pseudo-register $\texttt{r}_M$. The precondition and postcondition for our agent are:

$$\begin{array}{lcl} \texttt{Pre}_{\texttt{sum}} & = & \texttt{r}_x : \texttt{mp\_list} \wedge \texttt{listinv}\ \texttt{r}_M \\ \texttt{Post}_{\texttt{sum}} & = & \texttt{listinv}\ \texttt{r}_M \end{array}$$

14.3.1 Proof Rules

Besides the preconditions and postconditions, the safety policy contains a set of proof rules that can be used to reason about formulas, and in particular about verification conditions. We show the built-in rules in Figure 14.3.

Each safety policy needs to extend the built-in proof rules with new rules to accommodate its additional formula constructors. We show the rules specific to our new proof rules in Figure 14.4:

- The rules Nil and Cons describe typing for lists, and Set describes sets.
- The rules This and Next describe pointers to sequences (for pairs).
- The rules Sel and Upd talk about reading and writing pointers. The Sel rule says that the location referenced by a pointer to a word type in a well-typed memory state has the given word type. The Upd rule is used to prove that a well-typed write proves the well-typedness of memory.
- The PtrAddr rule says that addresses that can be proved to have pointer type are valid addresses.

$$\textsc{Nil}\ \frac{}{0 : \texttt{list}\ W} \qquad \textsc{Cons}\ \frac{E : \texttt{list}\ W \quad E \neq 0}{E : \texttt{ptr}\ \{W; \texttt{list}\ W\}}$$

$$\textsc{Set}\ \frac{E : \{y \mid F(y)\}}{F(E)}$$

$$\textsc{This}\ \frac{E : \texttt{ptr}\ \{W; S\}}{E : \texttt{ptr}\ \{W\}} \qquad \textsc{Next}\ \frac{E : \texttt{ptr}\{W; S\}}{E + 4 : \texttt{ptr}\ \{S\}}$$

$$\textsc{Sel}\ \frac{A : \texttt{ptr}\ \{W\} \quad \texttt{listinv}\ M}{(\texttt{sel}\ M\ A) : W} \qquad \textsc{Upd}\ \frac{\texttt{listinv}\ M \quad A : \texttt{ptr}\ \{W\} \quad V : W}{\texttt{listinv}\ (\texttt{upd}\ M\ A\ V)}$$

$$\textsc{PtrAddr}\ \frac{A : \texttt{ptr}\ \{W\}}{\texttt{addr}\ A}$$

Figure 14.4: Proof rules specific to our safety policy.

14.4 Verification-Condition Generation

Recall that the agent provides the source code and annotations, and the PCC infrastructure is responsible for checking these against the safety policy. How we do this is through *verification-condition generation* (VCGen), a process analogous to type checking. Just like type checking needs to know what typing rule to apply at each point, VCGen needs to know what proof rule to apply at each step. We will not go over in VCGen in this lecture. At a high level, VCGen generates and propagates constraints to ensure that we apply proof rules correctly.

Our PCC infrastructure separates the scanning of the code from the decision of what safety policy rules to apply. The roles are as follows:

- The VCGen scans the code, collecting typing constraints and identifying *what* must be checked for each instruction.
- The Checker module determines how to perform the checks, with help from the proof that accompanies the code.

In regular type-checking, we do not need to separate code scanning from the construction of the typing derivation, but code written in assembly has very little structure, and so a checker needs to do more work. Some problems in analyzing assembly are as follows:

- Compiled code for a single construct may be spread over several non-contiguous instructions.
- Some high-level operations are split into several small operations.
- We cannot count on a variable having a single type throughout its scope.

The Pierce book used *symbolic execution* to analyze the unsafe program for all potentially unsafe operations, requiring that the program be annotated predicates to make the symbolic execution possible in finite time, without conservative approximations.

14.5 Soundness Proof

We now prove that "well-typed programs cannot go wrong." That is, if the global verification condition for a program is provable using the proof rules given by the safety policy, then the program is guaranteed to execute without *memory safety*. The proof of full soundness looks like soundness proofs for higher-level languages. Just as with information flow, the full proof of soundness will involve defining an operational semantics for the assembly language, along with what it means to "go wrong."

The notion of what it means to be a well-typed program is a bit trickier to formalize in a PCC system. Instead of there being a direct connection between typing derivations and the AST of the program, there is an indirect connection. We first use a verification condition generator and then exhibit a derivation of the global verification condition using the safety policy proof rules. To reflect this staging, our proof has two parts: a proof of soundness of the set of policy rules and a proof of soundness of the VCGen algorithm. In this class, we will cover the former only.

We now prove the soundness of the safety policy. In order to prove that the typing rules enforce memory safety, we need to define the semantics of the expression and formula constructors. The semantic domain for expressions is the set of integers, except for the memory expressions that we model using partial maps from integers to integers. We then define a type mapping $\mathcal{M}$ from a valid address to the word type of the value stored at that address, to provide context to the typing formulas involving pointer types and `listinv` formulas. Since we do not consider allocation or deallocation, our type system ensures that $\mathcal{M}$ remains constant throughout program execution.

We write $\models_{\mathcal{M}} F$ when formula F holds in the memory mapping $\mathcal{M}$. We show some of the more interesting cases below:

$$
\begin{array}{lcl}
\models_{\mathcal{M}} F_1 \wedge F_2 & \iff & \models_{\mathcal{M}} F_1 \text{ and } \models_{\mathcal{M}} F_2 \\
\models_{\mathcal{M}} F_1 \Rightarrow F_2 & \iff & \text{whenever } \models_{\mathcal{M}} F_1 \text{ then } \models_{\mathcal{M}} F_2 \\
\models_{\mathcal{M}} \forall x.F(x) & \iff & \forall e \in \mathbb{Z}.\ \models_{\mathcal{M}} F(e) \\
\models_{\mathcal{M}} a : \texttt{int} & \iff & a \in \mathbb{Z} \\
\models_{\mathcal{M}} a : \texttt{list}\ W & \iff & a = 0 \wedge (\mathcal{M}(a) = W \wedge \mathcal{M}(a+4) = \texttt{list}\ W) \\
\models_{\mathcal{M}} a : \texttt{ptr}\ \{S\} & \iff & \forall i. 0 \leq i < |S| \Rightarrow \mathcal{M}(a + 4 * i) = S_i \\
\models_{\mathcal{M}} a : \{y \mid F(y)\} & \iff & \models_{\mathcal{M}} F(a) \\
\models_{\mathcal{M}} \texttt{listinv}\ m & \iff & \forall a \in Dom(\mathcal{M}).a \in Dom(m) \text{ and } \models_{\mathcal{M}} m\ a : \mathcal{M}(a)
\end{array}
$$

We use the notation $|S|$ to denote the length of a sequence of word types S and S_i for the i^{th} element of the sequence.

We can now prove the soundness of the derivation rules. Given a rule with variable $x_1, \ldots, x_n$, premises $H_1, \ldots H_m$, and conclusion C, we must prove

$$\models_{\mathcal{M}} \forall x_1.\forall x_2 \ldots \forall x_n.(H_1 \wedge \ldots \wedge H_m) \Rightarrow C \tag{14.4}$$

For example, the soundness of rule Sel requires proving:

$$\models_{\mathcal{M}} \forall a.\forall W.\forall m.(a : \texttt{ptr}\ \{W\}) \wedge (\texttt{listinv}\ m) \Rightarrow (\texttt{sel}\ m\ a) : W \tag{14.5}$$

From the first assumption we derive that $\mathcal{M}(a) = W$. From the second assumption we derive that $\models_{\mathcal{M}} m\ a : W$ and since $\models_{\mathcal{M}} (\texttt{sel}\ m\ a) = m\ a$ we obtained the desired solution.