SE547: Lecture 9

Overview

Certified software

Proof-carrying code

Machine model

Verification condition

Putting it all together

Issues in PCC

Certified software

Code comes from many sources: there are *code producers* and *code consumers*. Examples?

Code consumers may have a *safety policy* for their machine. What is this? Examples?

How can a consumer ensure the safety policy is maintained?

Certified software

What is a signed binary? Is a signed binary guaranteed to maintain the safety policy?

A complementary technology to signed binaries: bytecode verification. What is this? What guarantees does bytecode verification provide?

Problem: bytecode verification only applies to bytecode! Why is this a problem? What can we do about it?

Another possibility: model checking. Why not have the consumer model check the binary?

Proof-carrying code

Proof-carrying code consists of:

- a native executable, together with
- a proof of safety for the executable.

Obligation for the code producer: produce the proof of safety.

Obligation for the code consumer: check the proof of safety.

Where is most of the burden being placed?

Proof-carrying code

Claims for PCC:

1. Burden on the code producer, not consumer.

2. Consumer doesn't care where the proofs come from. Examples?

3. Code is taperproof. What happens if an attacker modifies the binary?

4. No reliance on trusted third parties. What is the trusted code base?

5. Static checking, not dynamic checking. Why is this good?

Proof-carrying code

Example PCC and related systems:

DEC Alpha Touchstone: certifies Alpha machine code.

Java Touchstone: certifies x86 machine code from Java source.

Foundational PCC: minimal trusted computing base.

Typed Assembly Language: typed x86 machine code.

Teams from Berkeley, Carnegie Mellon, Cornell, Ottowa, Princeton, Yale...

Machine model

A subset of Alpha assembly language (real thing does x86, jumps,...):

op ::= (Operand)
  n (Integer Constant)
  r_i (Register)

instr ::= (Instruction)
  ADDQ r_s, *op*, r_d (Put r_s + *op* in r_d)
  ...
  LDQ r_d $n(r_s)$ (Load from address $r_s + n$)
  STQ r_s $n(r_d)$ (Store to address $r_d + n$)
  RET (Return)

For example (what does this do? when is it safe?):

1. LDQ r2 0(r1)
2. ADDQ r2, 1, r3
3. STQ r3 0(r1)

Machine model

Model for Alpha assembly language is based on *stores* giving the contents of every register r_i.

We treat the memory as a special (very large!) register r_m.

For example:

$$(r_1 = 8, r_2 = 5, r_3 = 37, r_m = \{ 7, 2, 12 \})$$

Memory is word aligned, with 8-byte words so the memory has:

contents of 0 = 7, contents of 8 = 2, contents of 16 = 12

What will happen to this store after running the program:

1. LDQ r2 0(r1)
2. ADDQ r2, 1, r3
3. STQ r3 0(r1)

Machine model

Some operations on stores:

ρ [$r_i \leftarrow n$]
(Update store ρ so r_i is n)
sel(r_m, a)
(Select contents of memory address a)
upd(r_m, a, r_s)
(Update memory address a to contain contents of r_s)

For example, what is:

($r_1 = 8$, $r_2 = 5$, $r_3 = 37$, $r_m = \{ 7, 2, 12 \}$)
[$r_2 \leftarrow$ sel(r_m, r_1)]
[$r_3 \leftarrow r_2 + 1$]
[$r_m \leftarrow$ upd(r_m, r_1, r_3)]

Machine model

Machine model consists of steps:

$$(\rho_1, pc_1) \rightarrow (\rho_2, pc_2)$$

Given by rules (incomplete rules here!):

1. If instruction pc is ADDQ r_s, op, r_d then:

$$(\rho, pc) \rightarrow (\rho[\ r_d \leftarrow (r_s + op)\ ], pc + 1)$$

2. If instruction pc is LDQ r_d $n(r_s)$ then:

$$(\rho, pc) \rightarrow (\rho[\ r_d \leftarrow \mathrm{sel}(r_m, n + r_s)\ ], pc + 1)$$

3. If instruction pc is STQ r_s $n(r_d)$ then:

$$(\rho, pc) \rightarrow \text{??? what goes here ???}$$

All arithmetic is mod 2^{64}.

Machine model

Run the example:

1. LDQ r2 0(r1)
2. ADDQ r2, 1, r3
3. STQ r3 0(r1)

in initial state (ρ, 1) where:

$$\rho = (r_1 = 8, r_2 = 5, r_3 = 37, r_m = \{ 7, 2, 12 \})$$

Machine model

Assume predicates rd(a) and wr(a) saying when we have read-access and write-access to memory. Add side-conditions:

2. If instruction pc is LDQ r_d $n(r_s)$ then:

$$(\rho, pc) \rightarrow (\rho[\ r_d \leftarrow \mathrm{sel}(r_m, n + r_s)\ ], pc + 1)$$
$$\textit{as long as } \mathrm{rd}(n + r_s)$$

3. If instruction pc is STQ r_s $n(r_d)$ then:

$$(\rho, pc) \rightarrow \text{??? what goes here ???}$$

A program is *memory safe* as long as these side-conditions never fail!

1. LDQ r2 0(r1)
2. ADDQ r2, 1, r3
3. STQ r3 0(r1)

Verification condition

Each program has a *verification condition* VC_{pc} satisfying the property that:

If VC_{pc} is true in store ρ then (ρ, pc) is memory safe.

1. If instruction pc is ADDQ r_s, op, r_d then:

   $VC_{pc} = VC_{pc+1}\ [\ r_d \leftarrow (r_s + op)\ ]$

2. If instruction pc is LDQ r_d $n(r_s)$ then:

   $VC_{pc} = \mathrm{rd}(n + r_s) \wedge VC_{pc+1}[\ r_d \leftarrow \mathrm{sel}(r_m, n + r_s)\ ]$

3. If instruction pc is STQ r_s $n(r_d)$ then:

   (ρ, pc) $\rightarrow$??? what goes here ???

Verification condition

Assuming that VC_4 is **true**, what is VC_1?

1. LDQ r2 0(r1)
2. ADDQ r2, 1, r3
3. STQ r3 0(r1)

Verification condition

Every architecture comes with a *calling convention* which defines a *precondition* and a *postcondition* for function calls.

4. If instruction *pc* is RET then:

$$VC_{pc} = Post$$

We can check that a program is memory safe by verifying:

$$Pre \Rightarrow VC_0$$

We call this the *safety predicate* (*SP*).

Putting it all together

Code producer:

1. Generates native code.

2. Calculates SP.

3. Builds a proof of SP.

Code consumer:

1. Receives native code and proof from the producer.

2. Calculates SP.

3. Checks the proof of SP.

4. If the proof is valid, runs the code.

Issues in PCC

How to represent *SP* and it's proof: most PCC systems use an off-the-shelf implementation called *Edinburgh Logical Framework* (*LF*).

How to check proofs quickly: a dedicated C implementation of LF checker.

How to represent proofs compactly: lots of ad hoc compression techniques to make proofs smaller than the code!

How to minimize the trusted computing base (in particular the function which generates *SP*).

Try it out on-line.

Next week

Information flow.