Security proof of critical embedded code using abstract interpretation

Marc Chevalier

June 7, 2018

What it's all about?

Abstract interpretation

Add assembly into the soup

Conclusion

The need of proofs – Cost of software failure

Bugs have various annoying consequences:

- Deaths (Patriot, Toyota)
- A lot of money: Ariane V, $60 billion/year in the US
- Privacy
- ...

The need of proofs – What we usually do

How developers think they can avoid bugs:

- High level/safe language
- Tests
- Strict code style

Still, Ariane V crashed. . . . "And here, poor fool[s], with all [their] lore, [they] stand no wiser than before".

What we prove

Usually, no runtime error:

- Signed integer overflow
- Out of bound access
- Invalid pointer dereference
- ...

Better:

- The result satisfies some property
- The execution path does not depend on some secret data

What I want to prove

Study case: the OS of an host platform in planes at the border between trusted (flight control) and untrusted (potentially hostile) world.

We want to prove some security properties: memory isolation, hosted applications don't get more privileges. . . .

Properties are not visible from C (check some CPU's registers, mainly): inline assembly $\Rightarrow$ analyze assembly.

Introduction

- Check an execution: test, limited.
- Check all executions at once: ok, but not computable.
- Check an over-approximation of all execution: sound, not complete.

An example

```
int f(int x)
{                            // x ∈ [−2^31, 2^31 − 1]
    int y = abs(x);          // y ∈ [0, 2^31 − 1] ∨ x = −2^31
    int z = y + 1;           // z ∈ [1, 2^31 − 1] ∨ y = 2^31 − 1
    return 1/z;              // 0 ∉ [1, 2^31 − 1] ⇒ OK!
}
```

Let's generalize

$(C, \subseteq)$ a concrete set too big (typically, set of memory environments).
Abstract domain:

- $(A, \sqsubseteq)$: abstract set with good properties (eg. $\overline{\mathbb{Z}}^2$)
- $\gamma : A \to C$: concretization (eg. $(a, b) \mapsto \{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}$)

Sound if for all program point, $c \subseteq \gamma(a)$: we don't lose anything by executing in the abstract.

And we want $\gamma(a) \subseteq \mathit{specification}$.

The incompleteness

```
/*@ requires -10 <= x <= 10; */
int g(int x)
{                           // x ∈ [−10, 10]
      int y = x;            // y ∈ [−10, 10]
      int z = x * y;
      /* z ∈ Interval({a × b | a ∈ [−10, 10], b ∈ [−10, 10]})
           z ∈ [−100, 100]
      */
      int t = z + 1;        // t ∈ [−99, 101]
      return 1/t;           // 0 ∈ [−99, 101] ⇒ Alarm!
}
```

But this program is clearly safe.
What happens? This abstract domain cannot understand the relation between x and y.

Other domains

- Numerical:
  Non relational:
  - Modulo: $x_i \equiv c_i[n_i]$
  - Bitwise: $x_i = 0?1??100010111????$
  - Sign: $x_i < 0$, $x_i > 0$, $x_i \leqslant 0 \dots$

  Relational:
  - Polytope: $\sum a_i x_i \leqslant c_i$
  - Octagon: $\pm x_i \pm x_j \leqslant c_i$

  And combination of domains.
- Memory: some value points to another, memory structures, separation logic. . . .
- Partitioning: $(x > 0 \Rightarrow ...) \wedge (x \leqslant 0 \Rightarrow ...)$

Back on security

OS ⇒ assembly (Intel x86).

Some properties:

- Memory isolation ⇒ register CR3 are correctly set and not modified (paging).
- "Sandboxing" ⇒ applications stay in ring 3.
- Static code ⇒ a writable segment never become executable.

Inline assembly

```
int a;
void f()
{
    // C code
    asm {
        ; assembly code
        mov a, 4
    }
}
```

Difficulties

Majority of C: need to analyze x86 in a analysis designed for C.

Why x86 is really different from C:

- Jumps across functions vs local goto and blocks,
- Computed jump destinations vs static CFG,
- Type-agnostic registers vs statically typed programs,
- Intensive usage of stack, register... vs independent from architecture and implementation.

Difficulties – Control Flow

Let's take a look at the control flow problem.

C: cfg, a lot of structured control flow (while, for, if...), gotos
x86: basically, only jumps. (For experts: only near/short jmp/call)

Problems (in increasing difficulty):

- Compute the destination.
- Compute jumps local to a C function.
- Compute jumps leading to anywhere else.

Computing the destination

```
mov EBX, 0
mov EAX, label
add EAX, 3
jmp EAX
label:
add EBX, 1   ; This instruction has 3 bytes: 83 C3 01
add EBX, 2
; Here EBX == 2
```

Computing the destination

Program point: (block number, statement number). An instruction. Useful in analysis

Code pointer: (label, offset) where the label and the offset can be imprecise. An address. How the assembly works.

We have to compute the byte length of each assembly instruction to reinterpret code pointer as program point.

A jump in the middle of an instruction is considered as an error.

Local jumps

```
int f()
{
      int x = 1; // x = 1
      goto l;    // ⊥,      l ↦ {x = 1}
      m:         // ...
      return x;  // ...
      l:         // x = 1
      goto m;    // ⊥,      m ↦ {x = 1}
}                // ⊥
```

Local jumps

```
int f()
{
    int x = 1; // x = 1, m ↦ {x = 1}
    goto l;    // ⊥,     l ↦ {x = 1}, m ↦ {x = 1}
    m:         // x = 1, l ↦ {x = 1}
    return x;  // return = 1, l ↦ {x = 1}
    l:         // x = 1
    goto m;    // ⊥,     m ↦ {x = 1}
}              // return = 1
```

Distant and return jumps

A bit of context:

- No recursion (call stack abstraction).
- Inlined analysis (functions always return where they were called).

2 kinds of jumps:

- To a new function (not in the stack): distant jump.
- To a function which is in the call stack: return jump.

Distant and return jumps

```
void f() {
    asm {      // P
        jmp pp ; ⊥
        ...
        pp2:
    }
}
void g() {     // ⊥, pp ↦ P
    asm {
        pp:       ; P
        ...
                  ; Q
        jmp pp2 ; ⊥, ret: pp2 ↦ Q
    }
}
```

Distant and return jumps

```
void f() {
    asm {        // P
        jmp pp  ; ⊥, pp2 ↦ Q
        ...
        pp2:    ; Q
    }
}
void g() {       // ⊥, pp ↦ P
    asm {
        pp:       ; P
        ...
                  ; Q
        jmp pp2  ; ⊥, ret: pp2 ↦ Q
    }
}
```

Distant and return jumps

Why so complicated?

C structure keep most of the control flow: essential for precision.

Conclusion

We need proofs for critical source code.

Some low level source code contains mixed C/assembly.

Pretty much everything works now.

Wow, really?

Yeah, close enough, except things related to the environment (interruptions, exceptions, task switch, channels. . .) $\Rightarrow$ stubs.

Analysis of the OS: in progress.

In our (my) dreams: analysis of the whole environment.