Verified Abstract Interpretation Techniques for Disassembling Low-level Self-modifying Code

Sandrine Blazy[1] Vincent Laporte[1] David Pichardie[2]

[1]Université Rennes 1 – IRISA – Inria

[2]ENS Rennes – IRISA – Inria

July 16th 2014

Static analysis…

Prove program safety before running it

…of binary

Source not available

Compiler not trusted

Self-Modifying

Packed software

Obfuscation

JIT compiler

07	00	06	07	03	00	00	00	00	00	00	05
00	00	00	00	00	00	01	00	09	00	00	00
00	00	00	04	09	00	00	02	00	00	00	02
05	00	00	02	04	00	00	00	00	00	00	01
??	??	??	??	??	??	??	??	??	??	??	??
??	??	??	??	??	??	??	??	??	??	??	??

Register				
R_0:	??	??	??	??
R_1:	??	??	??	??
R_2:	??	??	??	??
R_3:	??	??	??	??
R_4:	??	??	??	??
R_5:	??	??	??	??
R_6:	04	a4	3b	09
R_7:	10	b9	e6	51
Flags:			EQ	

Next instruction: `cmp R6, R7`

07 00 06 07 03 00 00 00 00 00 00 05
00 00 00 00 00 00 01 00 09 00 00 00
00 00 00 04 09 00 00 02 00 00 00 02
05 00 00 02 04 00 00 00 00 00 00 01
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

R_0: ?? ?? ?? ??
R_1: ?? ?? ?? ??
R_2: ?? ?? ?? ??
R_3: ?? ?? ?? ??
R_4: ?? ?? ?? ??
R_5: ?? ?? ?? ??
R_6: 04 a4 3b 09
R_7: 10 b9 e6 51
Flags: LT LE

Next instruction: `gotoLE 5`

07 00 06 07 03 00 00 00 00 00 00 05
00 00 00 00 00 00 01 00 09 00 00 00
00 00 00 04 09 00 00 02 00 00 00 02
05 00 00 02 04 00 00 00 00 00 00 01
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

R_0: ?? ?? ?? ??
R_1: ?? ?? ?? ??
R_2: ?? ?? ?? ??
R_3: ?? ?? ?? ??
R_4: ?? ?? ?? ??
R_5: ?? ?? ?? ??
R_6: 04 a4 3b 09
R_7: 10 b9 e6 51
Flags: LT LE

Next instruction: `cst 4` $\rightarrow$ R_0

07	00	06	07	03	00	00	00	00	00	00	05
00	00	00	00	00	00	01	00	09	00	00	00
00	00	00	04	09	00	00	02	00	00	00	02
05	00	00	02	04	00	00	00	00	00	00	01
??	??	??	??	??	??	??	??	??	??	??	??
??	??	??	??	??	??	??	??	??	??	??	??

R_0: 00 00 00 04
R_1: ?? ?? ?? ??
R_2: ?? ?? ?? ??
R_3: ?? ?? ?? ??
R_4: ?? ?? ?? ??
R_5: ?? ?? ?? ??
R_6: 04 a4 3b 09
R_7: 10 b9 e6 51
Flags: LT LE

Next instruction: `cst 2` $\rightarrow$ R_2

07	00	06	07	03	00	00	00	00	00	00	05
00	00	00	00	00	00	01	00	09	00	00	00
00	00	00	04	09	00	00	02	00	00	00	02
05	00	00	02	04	00	00	00	00	00	00	01
??	??	??	??	??	??	??	??	??	??	??	??
??	??	??	??	??	??	??	??	??	??	??	??

R_0: 00 00 00 04
R_1: ?? ?? ?? ??
R_2: 00 00 00 02
R_3: ?? ?? ?? ??
R_4: ?? ?? ?? ??
R_5: ?? ?? ?? ??
R_6: 04 a4 3b 09
R_7: 10 b9 e6 51
Flags: LT LE

Next instruction: `store` $R_0 \rightarrow \star R_2$

07	00	06	07	03	00	00	00	00	00	00	04
00	00	00	00	00	00	01	00	09	00	00	00
00	00	00	04	09	00	00	02	00	00	00	02
05	00	00	02	04	00	00	00	00	00	00	01
??	??	??	??	??	??	??	??	??	??	??	??
??	??	??	??	??	??	??	??	??	??	??	??

R_0: 00 00 00 04
R_1: ?? ?? ?? ??
R_2: 00 00 00 02
R_3: ?? ?? ?? ??
R_4: ?? ?? ?? ??
R_5: ?? ?? ?? ??
R_6: 04 a4 3b 09
R_7: 10 b9 e6 51
Flags: LT LE

Next instruction: `goto 1`

07	00	06	07	03	00	00	00	00	00	00	04
00	00	00	00	00	00	01	00	09	00	00	00
00	00	00	04	09	00	00	02	00	00	00	02
05	00	00	02	04	00	00	00	00	00	00	01
??	??	??	??	??	??	??	??	??	??	??	??
??	??	??	??	??	??	??	??	??	??	??	??

R_0: 00 00 00 04
R_1: ?? ?? ?? ??
R_2: 00 00 00 02
R_3: ?? ?? ?? ??
R_4: ?? ?? ?? ??
R_5: ?? ?? ?? ??
R_6: 04 a4 3b 09
R_7: 10 b9 e6 51
Flags: LT LE

Next instruction: `gotoLE 4`

07 00 06 07 03 00 00 00 00 00 00 04
00 00 00 00 00 00 01 00 09 00 00 00
00 00 00 04 09 00 00 02 00 00 00 02
05 00 00 02 04 00 00 00 00 00 00 01
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

R_0: 00 00 00 04
R_1: ?? ?? ?? ??
R_2: 00 00 00 02
R_3: ?? ?? ?? ??
R_4: ?? ?? ?? ??
R_5: ?? ?? ?? ??
R_6: 04 a4 3b 09
R_7: 10 b9 e6 51
Flags: LT LE

Next instruction: `halt` R_0

07	00	06	07	03	00	00	00	00	00	00	04
00	00	00	00	00	00	01	00	09	00	00	00
00	00	00	04	09	00	00	02	00	00	00	02
05	00	00	02	04	00	00	00	00	00	00	01
??	??	??	??	??	??	??	??	??	??	??	??
??	??	??	??	??	??	??	??	??	??	??	??

R_0: 00 00 00 04
R_1: ?? ?? ?? ??
R_2: 00 00 00 02
R_3: ?? ?? ?? ??
R_4: ?? ?? ?? ??
R_5: ?? ?? ?? ??
R_6: 04 a4 3b 09
R_7: 10 b9 e6 51
Flags: LT LE

Final value: 4

Given such a program, how to...

Disassemble it?

Where are the instructions?
What are they?

Compute its control-flow graph?

What are the targets of the computed jumps?

Automatically prove safety properties about it?

Can its execution be stuck?
May it access secret parts of the memory?

Trust the answers to above questions?

Is the analysis sound?

Inspiring related work

"Certified Self-Modifying Code"

(Cai, Shao, and Vaynberg PLDI'2007)

- Framework for manual verification of self-modifying programs

"WYSINWYX: What You See Is Not What You eXecute"

(Balakrishnan and Reps 2010)

- Static analysis of x86

Our approach

Formalize in Coq a static analysis

- that is flow sensitive
  - attach to each reachable program point an over-approximation of the state at that point
  - analyze the content of the memory and of the registers
- without a previous disassembling or CFG reconstruction

Architecture

Abstract interpreter

State abstraction

Numeric abstraction

Each layer is parameterized by the underlying one

Base Layer

Numeric abstract domains

Abstract sets of 32-bit machine integers

Finite sets e.g., $\{0; 1; 7\}$

Strided Intervals that combines interval and congruence information
e.g., $[1000; 2000].4$ represents $\{1000; 10004; 1008; \ldots; 2000\}$

Signature of abstract domains

Each abstract domain is equipped with a lattice structure

```
Class weak_lattice (A: Type) : Type :=
{ leb: A → A → bool    (* order *)
; top: A                (* maximal element *)
; join:  A → A → A      (* binary upper bound *)
; widen: A → A → A      (* extrapolation operator *)
}.
```

Concretization based specification

Each abstract domain comes with a concretization relation

```
Class gamma_op (A B: Type) : Type := γ : A → 𝒫(B).

Record adom (A B:Type)(_:weak_lattice A)(_:gamma_op A B): Prop :=
{ gamma_monotone: ∀ a1 a2, leb a1 a2 = true → γ a1 ⊆ γ a2
; gamma_top: ∀ x, x ∈ γ top
; join_sound: ∀ x y, γ x ∪ γ y ⊆ γ (join x y) }.
```

- No more properties required

Example (Strided intervals)

```
Instance si_gamma : gamma_op strided_interval int := λ x i,
    low_bound x ⩽ Int.signed i ⩽ up_bound x
  ∧ low_bound x ≡ Int.signed i [ stride x ].
```

Signature of the state abstraction

```
Record mem_dom (ab_mem: Type) (ab_num: Type) : Type :=
{ (* lattice *)
  as_wl: weak_lattice ab_mem
  (* queries *)
; var: ab_mem → register → ab_num
; load: ab_mem → addr → ab_num
  (* abstract transformers *)
; assign: ab_mem → register → ab_num → ab_mem
  (* more omitted ... *)
}.
```

Signature of the state abstraction

```
Record mem_dom (ab_mem: Type) (ab_num: Type) : Type :=
{ (* lattice *)
  as_wl: weak_lattice ab_mem
  (* queries *)
; var: ab_mem → register → ab_num
; load: ab_mem → addr → ab_num
  (* abstract transformers *)
; assign: ab_mem → register → ab_num → ab_mem
  (* more omitted ... *)
}.
```

Parameterized by a numeric abstraction: `ab_num`

Signature of the state abstraction

```
Record mem_dom (ab_mem: Type) (ab_num: Type) : Type :=
{ (* lattice *)
  as_wl: weak_lattice ab_mem
  (* queries *)
; var: ab_mem → register → ab_num
; load: ab_mem → addr → ab_num
  (* abstract transformers *)
; assign: ab_mem → register → ab_num → ab_mem
  (* more omitted ... *)
}.
```

Equipped with a lattice structure: order, top element, join operator

Signature of the state abstraction

```
Record mem_dom (ab_mem: Type) (ab_num: Type) : Type :=
{ (* lattice *)
  as_wl: weak_lattice ab_mem
  (* queries *)
; var: ab_mem → register → ab_num
; load: ab_mem → addr → ab_num
  (* abstract transformers *)
; assign: ab_mem → register → ab_num → ab_mem
  (* more omitted ... *)
}.
```

Provides access to the abstraction of each memory unit

Signature of the state abstraction

```
Record mem_dom (ab_mem: Type) (ab_num: Type) : Type :=
{ (* lattice *)
  as_wl: weak_lattice ab_mem
  (* queries *)
; var: ab_mem → register → ab_num
; load: ab_mem → addr → ab_num
  (* abstract transformers *)
; assign: ab_mem → register → ab_num → ab_mem
  (* more omitted ... *)
}.
```

Abstract operators model concrete instructions

Signature of the state abstraction

```
Record mem_dom (ab_mem: Type) (ab_num: Type) : Type :=
{ (* lattice *)
  as_wl: weak_lattice ab_mem
  (* queries *)
; var: ab_mem → register → ab_num
; load: ab_mem → addr → ab_num
  (* abstract transformers *)
; assign: ab_mem → register → ab_num → ab_mem
  (* more omitted ... *)
}.
```

Example (Specification of the `load` query)

```
load_sound: ∀ ab, ∀ m,
  m ∈ γ(ab) → ∀ a:addr, m(a) ∈ γ(load ab a)
```

Abstract small-step

Specification

```
Lemma ab_step_correct : ∀ m m' ab,
  m ∈ γ(ab) → m ⇝ m' → m' ∈ γ(ab_step m.(pp) ab)
```

Algorithm

Given an abstract state `m` at an address `pp` (program point)

1. Decode all possible instructions from `pp`
2. For each of them predict
   - next program point
   - next abstract state
3. Propagate the results

Fix-point computation

Given a program P (partial initial memory), compute an abstract environment E such that

- $\texttt{init}(\texttt{P}) \sqsubseteq \texttt{E}[0]$, and
- $\forall p\ p'\ m', (p', m') \in \texttt{ab_step}(p)(\texttt{E}[p]) \rightarrow m' \sqsubseteq \texttt{E}[p']$

- Iterate from initial state
- Work-set of reachable program points that need further analysis
- Widening steps to ensure termination
  - The widening policy is a parameter of the analysis
- A posteriori validation

```
Lemma validate_correct : ∀ P E,
  validate_fixpoint P E = true → ⟦P⟧ ⊆ γ(E)
```

What can we do with this fix-point?

Given a (sound) fix-point:

Program safety can be proved (no run-time error)

```
Theorem analysis_sound : ∀ P dom fuel ab_num,
  analysis ab_num P dom fuel ≠ None → safe P
```

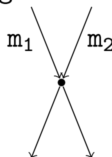
A CFG can be computed

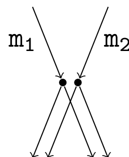
- Reachable program points
- Possible instructions when execution reaches a program point
- Possible targets of the jumps

Extension: Trace Partitioning

Some merges of abstract states incur a large precision loss:

- Do not merge abstract states that differ according to some criterion





- Generalizes the flow sensitivity
- Need to modify only the fix-point computation (and checker)

Experimental evaluation

Extract to OCaml and run on challenging examples[1]

- ✓ opcode modification overwrites code before execution
- ✓ multilevel RCG code that once executed produces code that once. . .
- × bootloader loads code from disk
- ✓ control flow modification overwrites jumps
- ✓ vector dot product specializes a program for a given argument
- ✓ runtime code checking validates code integrity
- ✓ Fibonacci sequence uses an instruction as accumulator
- × self-replication infinitely copies itself
- ✓ mutual modification two parts modify each other
- ✓ polymorphic code mutates itself to escape anti-viruses
- ✓ code obfuscation features fake instructions if naively disassembled
- ✓ code encryption decrypts a code, runs it, and crypts it back

[1] Cai, Shao, and Vaynberg PLDI'2007.

Conclusion

Summary

- Verified executable abstract interpreter for a binary language
- Handles seamlessly self-modifying programs
- Able to prove the safety of challenging examples
- Full development available online:
  `http://www.irisa.fr/celtique/ext/smc/`

Perspectives

- More realistic languages (x86)
- Generate the analyzer (and its proof) to cope with the huge number of instructions