Precise Static Analysis of Binaries by Extracting Relational Information

Alexander Sepp, Bogdan Mihaila and Axel Simon

Technical University Munich, Informatik 2, Germany
firstname.lastname@in.tum.de
October 20, 2011
Introduction

What do we want to achieve

- automatically find bugs/security exploits in executables
- soundness (no missed bugs)
- goal: no false positives
Introduction

What do we want to achieve

- automatically find bugs/security exploits in executables
- soundness (no missed bugs)
- goal: no false positives

What are we doing

- static analysis of executables constructed using abstract interpretation theory
- over-approximate reachability analysis by calculating fixpoints
Motivation

Why binaries

• no source code available
• compiler correctness
• security bugs can only be understood here
• reverse engineer, understand binary/legacy code
Motivation

Why binaries

- no source code available
- compiler correctness
- security bugs can only be understood here
- reverse engineer, understand binary/legacy code

Problems specific to binary analysis

- no variable boundaries/types
- many architectures
- large instruction sets
RREIL
(Relational Reverse Engineering Intermediate Language)

RREIL: yet another intermediate language

- translate one x86, ARM, AVR ... instruction into a handful of RREIL instructions \(\sim\) architecture independent analyses
- faithful down to the bit-level but platform independent
RREIL
(Relational Reverse Engineering Intermediate Language)

RREIL: yet another intermediate language

- translate one x86, ARM, AVR ... instruction into a handful of RREIL instructions $\sim$ architecture independent analyses
- faithful down to the bit-level but platform independent

What is special about RREIL

- concise: has 24 instructions vs. Intel x86 ca. 600 instructions
- arithmetic expressions only over operands of same bit-size
- special translation of relational tests such as $a < b$
- precise handling of e.g. x86 registers by using *fields*
Running example

```c
for (int i = -1; i > -100; i--) { /* body */ }
```

**x86-64 translation:**

```
01: mov eax, 0xffffffff
02: sub eax, 0x1
/* body */
03: cmp eax, 0xffffffff9c
04: jg 02
```

**RREIL translation:**

```
01.00: mov rax:32, -1:32
01.01: mov rax:32/32, 0:32
02.00: sub t0:32, rax:32, 1:32
02.01: cmpltu CF:1, rax:32, 1:32
02.02: cmples LE:1, rax:32, 1:32
02.03: cmplts LT:1, rax:32, 1:32
02.04: cmpltu CF:1, rax:32, 1:32
02.05: cmpeq ZF:1, rax:32, 1:32
02.06: cmplts SF:1, t0:32, 0:32
02.07: xor OF:1, LT:1, SF:1
02.08: mov rax:32, t0:32
02.09: mov rax:32/32, 0:32
/* body */
03.00: sub t0:32, rax:32, -100:32
03.01: cmpltu CF:1, rax:32, -100:32
03.02: cmpleu BE:1, rax:32, -100:32
03.03: cmplts LT:1, rax:32, -100:32
03.04: cmpltu CF:1, rax:32, -100:32
03.05: cmpeq ZF:1, rax:32, -100:32
03.06: cmplts SF:1, t0:32, 0:32
03.07: xor OF:1, LT:1, SF:1
04.00: xor t0:1, LE:1, 1:1
04.01: brc t0:1, 02.00
```
RREIL virtual flags

Consider the translation of x86 code:

01: \texttt{cmp} eax, ebx
02: jl sLess

RREIL code:

01.00: \texttt{sub} t0:32, eax:32, ebx:32
01.01: \texttt{cmpltu} CF:1, eax:32, ebx:32
01.02: \texttt{cmpeq} ZF:1, eax:32, ebx:32
01.03: \texttt{cmplts} SF:1, t0:32, 0:32
01.04: \texttt{cmpleu} CForZF:1, eax:32, ebx:32
01.05: \texttt{cmplts} SFxorOF:1, eax:32, ebx:32
01.06: \texttt{cmplies} SFxorOFForZF:1, eax:32, ebx:32
01.07: \texttt{xor} OF:1, SFxorOF:1, SF:1
02.00: \texttt{brc} SFxorOF:1, sLess:32

create \textit{virtual} flags SFxorOF that express numeric semantics
RREIL virtual flags with applied liveness

Consider the translation of x86 code:

01: \texttt{cmp eax, ebx}
02: \texttt{jl sLess}

RREIL code:

01.00: \texttt{nop}
01.01: \texttt{nop}
01.02: \texttt{nop}
01.03: \texttt{nop}
01.04: \texttt{nop}
01.05: \texttt{cmplts SFxorOF:1, eax:32, ebx:32}
01.06: \texttt{nop}
01.07: \texttt{nop}
02.00: \texttt{brc SFxorOF:1, sLess:32}

create \textit{virtual} flags SFxorOF that express numeric semantics
Abstract Domains Hierarchy

Overall Idea:

- RREIL instructions are *processed* by the domain hierarchy (down channel)
- indirect jumps are resolved by *querying* the hierarchy (up channel)
Numeric Domain

Numeric domains map variables $x \in \mathbb{X}$ to a subset of $\mathbb{Z}$.

- the **affine** tracks equalities $c_i x_i = \sum_j c_j x_j$ where $i < j$
- $\sim$ no need to store $x_i$ in child domains; some linear assignments need not be propagated to child
- the **interval** domain maps $x_k$ to $[l_k, u_k]$
Memory Domain

From operations on registers/memory $m \in \mathbb{M}$ to operations on numeric variables $x \in \mathbb{X}$:

Consider again x86-64: `mov eax,0xffffffff`

- 32-bit `mov` clears upper 32 bits of `rax`
- cannot store -1 in numeric domain since then $rax=0xffffffffffffffff$
- tracking `rax` as one numeric variable leads to precision loss
Memory Domain

From operations on registers/memory $m \in \mathbb{M}$ to operations on numeric variables $x \in \mathbb{X}$:

Consider again x86-64: \texttt{mov eax,0xffffffff}

- 32-bit \texttt{mov} clears upper 32 bits of \texttt{rax}
- cannot store -1 in numeric domain since then \texttt{rax=0xfffffffffffffffff}
- tracking \texttt{rax} as one numeric variable leads to precision loss

RREIL:

\texttt{mov rax:32/0, \texttt{-1:32}}
\texttt{mov rax:32/32, 0:32}

leads to:

<table>
<thead>
<tr>
<th>rax :32/32</th>
<th>rax :32/0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
Memory Domain

From operations on registers/memory $m \in \mathbb{M}$ to operations on numeric variables $x \in \mathbb{X}$:

Consider again x86-64: mov eax,0xffffffff

- 32-bit mov clears upper 32 bits of rax
- cannot store -1 in numeric domain since then rax=0xffffffffffffffff
- tracking rax as one numeric variable leads to precision loss

RREIL:

**mov** rax:32/0, -1:32

**mov** rax:32/32, 0:32

leads to:

<table>
<thead>
<tr>
<th>rax:32/32</th>
<th>rax:32/0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_2$</td>
</tr>
</tbody>
</table>
Dealing with finite integer arithmetic

We have a special view on numeric information:

- we store e.g. \( a_l \in [254, 257] \) (\( a_l \) is 8 bits big)

- this means:

  \[
  \begin{align*}
  254 & \equiv 11111110 \\
  255 & \equiv 11111111 \\
  256 & \equiv 00000000 \\
  257 & \equiv 00000001 \\
  \end{align*}
  \]

- which is also \([254, 255] \sqcup [0, 1] = [0, 255]\)

- we call this conversion \textit{wrapping}
Finite Domain

The finite domains associate a bit-size with each $x \in \mathbb{X}$.
Finite Domain

The finite domains associate a bit-size with each $x \in \mathbb{X}$.

Idea:

- wrap operand $i$ before each operation;
- wrapping is no-op if $i \in [-2^{31}, 2^{31} - 1]$.
Finite Domain

The finite domains associate a bit-size with each $x \in \mathbb{X}$.

Idea:

- wrap operand $i$ before each operation; wrapping is no-op if $i \in [-2^{31}, 2^{31} - 1]$
- problem: precision loss; add, sub carry no sign information (signedness-agnostic)
Finite Domain

The finite domains associate a bit-size with each $x \in \mathbb{X}$.

Idea:

- wrap operand $i$ before each operation; wrapping is no-op if $i \in [-2^{31}, 2^{31} - 1]$
- **problem**: precision loss; **add**, **sub** carry no sign information (signedness-agnostic)
- **idea**: only wrap when unavoidable, e.g. before executing the test $i > -100$
Finite Domain

The finite domains associate a bit-size with each $x \in \mathbb{X}$.

Idea:

- wrap operand $i$ before each operation; wrapping is no-op if $i \in [-2^{31}, 2^{31} - 1]$
- **problem**: precision loss; **add**, **sub** carry no sign information (signedness-agnostic)
- **idea**: only wrap when unavoidable, e.g. before executing the test $i > -100$
- **problem**: $i \in [-1, -1], [-2, -1], [-3, -1], [-4, -1] \ldots$ is inferred during fixpoint computation
Finite Domain

The finite domains associate a bit-size with each $x \in \mathbb{X}$.

Idea:

- wrap operand $i$ before each operation; wrapping is no-op if $i \in [-2^{31}, 2^{31} - 1]$
- **problem**: precision loss; **add**, **sub** carry no sign information (signedness-agnostic)
- **idea**: only wrap when unavoidable, e.g. before executing the test $i > -100$
- **problem**: $i \in [-1, -1], [-2, -1], [-3, -1], [-4, -1] \ldots$ is inferred during fixpoint computation
- **widening** applied to $i \leadsto [-\infty, -1]$
Finite Domain

The finite domains associate a bit-size with each \( x \in \mathbb{X} \).

Idea:

- wrap operand \( i \) before each operation; wrapping is no-op if \( i \in [-2^{31}, 2^{31} - 1] \)
- **problem**: precision loss; **add**, **sub** carry no sign information (signedness-agnostic)
- **idea**: only wrap when unavoidable, e.g. before executing the test \( i > -100 \)
- **problem**: \( i \in [-1, -1], [-2, -1], [-3, -1], [-4, -1] \ldots \) is inferred during fixpoint computation
- **widening** applied to \( i \leadsto [\neg\infty, -1] \)
- \( \leadsto \) wrapping of widened value \([\neg\infty, -1]\) gives \([\neg2^{31}, 2^{31} - 1]\)
Finite Domain

The finite domains associate a bit-size with each \( x \in X \).

Idea:

- wrap operand \( i \) before each operation; wrapping is no-op if \( i \in [-2^{31}, 2^{31} - 1] \)
- **problem**: precision loss; \textbf{add}, \textbf{sub} carry no sign information (signedness-agnostic)
- **idea**: only wrap when unavoidable, e.g. before executing the test \( i > -100 \)
- **problem**: \( i \in [-1, -1], [-2, -1], [-3, -1], [-4, -1] \ldots \) is inferred during fixpoint computation
- **widening** applied to \( i \sim [\infty, -1] \)
- \( \sim \) wrapping of widened value \([-\infty, -1] \) gives \([-2^{31}, 2^{31} - 1] \)
- cannot infer that \( i \) is negative
Narrowing Domain

Avoiding precision loss incurred by widening: the *narrowing* domain

Fix wrapping + widening by:

- have a *narrowing* domain that tracks all tests that don’t affect the state (redundant)
- the test $i > -100$ is stored when analyzing the loop
- after widening $i$ to $[-\infty, -1]$ apply all stored tests
- $i \in [-99, -1]$ follows
- wrapping to positive values avoided
Summary

Framework for static analysis of binaries:

- disassemblers as simple stateless decoder frontends
- memory layout, finite integer arithmetic, fixpoints, etc.
  - accesses to part of memory regions, registers
  - combined widening and wrapping
  - associate flag with test after `cmp`
- extensible domain hierarchy
  - four well-defined interfaces (ILs)
  - can be extended with SAT solving, shape analysis etc.
- CFG reconstruction drops out for free
  - program execution model adjustable (fixpoint computation)
  - able to disassemble (some) obfuscated code
Future work

- Additional Disassemblers
  - ARM
- Additional Domains
  - array summaries domain
  - strings domain
  - heap memory models (malloc)
- Malware and obfuscated code