Get rid of inline assembly through verification-oriented lifting

Frédéric Recoules (CEA LIST)
Sébastien Bardin (CEA LIST)
Richard Bonichon (CEA LIST)
Laurent Mounier (VERIMAG)
Marie-Laure Potet (VERIMAG)

May 16, 2019
Software is not always reliable

\[ \text{safety} \not\iff \text{security} \]

\[ \neg \text{safety} \not\iff \neg \text{security} \]
Formal methods for C code

With industrial success stories

frama c cc

CODESONAR® AbsInt
Today’s challenge:
mixed C & inline assembly code

- Applicability
- Verifiability
- Correctness
Example

# 76 "libavcodec/x86/mathops.h"
static inline int
mid_pred(int a, int b, int c)
{
    int i = b;
    __asm__ (
        "cmp %2, %1 \n"
        "cmovg %1, %0 \n"
        "cmovg %2, %1 \n"
        "cmp %3, %1 \n"
        "cmovl %3, %1 \n"
        "cmp %1, %0 \n"
        "cmovg %1, %0 \n"
        : "+r" (i), "+r" (a)
        : "r" (b), "r" (c)
        : /* no clobbers */
    );
    return i;
}
Statistics

7k packages

786
11%

1264
355
28%¹

¹ according to Rigger et al.
Symbolic execution & mixed code

```c
int mid_pred (int a, int b, int c) {
    int i = b;
    __asm__ (
        "cmp %2, %1 \n\t" 
        "cmovg %1, %0 \n\t" 
        "cmovg %2, %1 \n\t" 
        "cmp %3, %1 \n\t" 
        "cmovl %3, %1 \n\t" 
        "cmp %1, %0 \n\t" 
        "cmovg %1, %0 \n\t" 
        : "+&r" (i), "+&r" (a) 
        : "r" (b), "r" (c));

    return i;
}

int main (int argc, char *argv[]) {
    int a, b, c, i;
    klee_make_symbolic(&a, sizeof(int), "a");
    klee_make_symbolic(&b, sizeof(int), "b");
    klee_make_symbolic(&c, sizeof(int), "c");
    i = mid_pred(a, b, c);
    printf("mid(%d, %d, %d) = %d
", a, b, c, i);
    return 0;
}
```

**KLEE**

WARNING: function "main" has inline asm
ERROR: inline assembly is unsupported
NOTE: ignoring this error at this location

done: total instructions = 161
done: completed paths = 1
done: generated tests = 1

Incomplete
int mid_pred (int a, int b, int c) {
    int i = b;
    __asm__ ("cmp %2, %1 \n\t" \\
              "cmovg %1, %0 \n\t" \\
              "cmovg %2, %1 \n\t" \\
              "cmp %3, %1 \n\t" \\
              "cmovl %3, %1 \n\t" \\
              "cmp %1, %0 \n\t" \\
              "cmovg %1, %0 \n\t" \\
              : "+r" (i), "+r" (a) \\
              : "r" (b), "r" (c));
    return i;
}

int main (int argc, char *argv[]) {
    int a, b, c, i;
    a = Frama_C_interval(0, 5);
    b = Frama_C_interval(-5, 10);
    c = Frama_C_interval(-10, 0);
    i = mid_pred(a, b, c);
    printf("mid(%d, %d, %d) = %d\n", a, b, c, i);
    return 0;
}
int mid_pred (int a, int b, int c) {
    int i = b;
    #ifndef DISABLE_ASM
        __asm__
            ("cmp  %2, %1 \n\t" "cmovg %1, %0 \n\t" "cmovg %2, %1 \n\t" "cmp  %3, %1 \n\t" "cmovl %3, %1 \n\t" "cmp  %1, %0 \n\t" "cmovg %1, %0 \n\t"
              : "+&r" (i), "+&r" (a)
              : "r" (b), "r" (c));
    #else
        i = max(a, b);
        a = min(a, b);
        a = max(a, c);
        i = min(i, a);
    #endif
    return i;
}
**Our proposition**

**Automatically lift ASM to equivalent C**

```c
int mid_pred (int a, int b, int c)
{
    int i = b;
    _asm_ ("cmp %2, %1 \n\t"
            "cmovg %1, %0 \n\t"
            "cmovg %2, %1 \n\t"
            "cmp %3, %1 \n\t"
            "cmovl %3, %1 \n\t"
            "cmp %1, %0 \n\t"
            "cmovg %1, %0 \n\t"
            : "+&r" (i), "+&r" (a)
            : "r" (b), "r" (c));
    return i;
}
```

```c
int mid_pred (int a, int b, int c)
{
    int i = b;
    
    _tina_tmp3 = (a > b) ? a : i;
    _tina_tmp2 = (a > b) ? b : a;
    _tina_tmp1 = (_tina_tmp2 < c) ? c : _tina_tmp2;
    _tina_tmp4 = _tina_tmp3;
    i = _tina_tmp4;
    return i;
}
```

**Reuse C tools**
Requirements & challenges

Widely applicable

architecture – assembly dialect – compiler agnostic
Widely applicable
architecture – assembly dialect – compiler agnostic
leverage IR lifters
Requirements & challenges

Widely applicable
architecture – assembly dialect – compiler agnostic
leverage IR lifters

Verifiability
decent enough analysis outputs
Requirements & challenges

Widely applicable
architecture – assembly dialect – compiler agnostic
leverage IR lifters

Verifiability
decent enough analysis outputs
dedicated high-level simplifications
Requirements & challenges

Widely applicable
architecture – assembly dialect – compiler agnostic
leverage IR lifters

Verifiability
decent enough analysis outputs
dedicated high-level simplifications

Provably correct
usable in sound formal method context
Requirements & challenges

Widely applicable
architecture – assembly dialect – compiler agnostic
leverage IR lifters

Verifiability
decent enough analysis outputs
dedicated high-level simplifications

Provably correct
usable in sound formal method context
lightweight equivalence checking
## Panorama of existing works

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Interface compliance(^1)</th>
<th>Vx86(^2)</th>
<th>Inception(^3)</th>
<th>TINA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Widely applicable</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Trust</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sanity check</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Semantic lifting</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Validation</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Verifiability</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

---

1. Fehnker et al. *Some Assembly Required* - Program Analysis of Embedded System Code
2. Schulte et al. *Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving*
Process overview

C + ASM → Object code

C tool: framaC
Compiler: GCC/Clang
Debug format: DWARF
Binary lifter: BINSEC
SMT solver: Z3, boolector

Object code → IR

IR lifting

C

Transformations

EQUIVALENCE
✓ / ?

Validation

Compilation + Debug
# Binsec intermediate representation

\[
\begin{align*}
\text{inst} & : = \text{lv} \leftarrow e \mid \text{goto } e \mid \text{if } e \text{ then goto } e \\
\text{lv} & : = \text{var} \mid @[e]_n \\
\text{e} & : = \text{cst} \mid \text{lv} \mid \text{unop } e \mid \text{binop } e \ e \mid e \ ? \ e \ : \ e
\end{align*}
\]

\[
\begin{align*}
\text{unop} & : = \neg \mid \lnot \mid \text{uext}_n \mid \text{sext}_n \mid \text{extract}_{i..j} \\
\text{binop} & : = \text{arith} \mid \text{bitwise} \mid \text{cmp} \mid \text{concat} \\
\text{arith} & : = + \mid - \mid \times \mid \text{udiv} \mid \text{urem} \mid \text{sdiv} \mid \text{srem} \\
\text{bitwise} & : = \wedge \mid \lor \mid \oplus \mid \text{shl} \mid \text{shr} \mid \text{sar} \\
\text{cmp} & : = = \mid \neq \mid >_u \mid <_u \mid >_s \mid <_s
\end{align*}
\]

# Multi-architecture

\(\text{x86-32bit – ARMv7}\)
TInA : Out-of-Scope

**Binary Lifter / SMT solvers**
- floating point operations
- cryptography operations

**Binary Lifter / Method**
- syscall
- multi-thread
- hardware dependent
Lifting: running example

```
__asm__

("cmp   %0, %1 \n\t"
 "cmovg %1, %0 \n\t"
 /* [... ] */
 : "+&r" (i), "+&r" (a)
 : /* [... ] */
 : /* no clobbers */
);
```

```

in

eax ← i
ebx ← a
res32 ← ebx - eax
ZF ← res32 = 0
SF ← res32 < 0
OF ← ebx[31] ≠ eax[31]
∧ ebx[31] ≠ res32[31]
if ¬ZF ∧ SF = OF
then goto B_1 else goto B_2

B_0

B_1 tmp ← ebx

B_2 tmp ← eax

B_3
eax ← tmp
i ← eax

out

__asm__

("cmp %0, %1 
  "cmovg %1, %0 
 /* [... ] */
 : "+&r" (i), "+&r" (a)
 : /* [... ] */
 : /* no clobbers */
);
```
Lifting : comparison

Basic

__eax__ = (unsigned int)i;
__ebx__ = (unsigned int)a;
__res32__ = __ebx__ - __eax__;
__zf__ = __res32__ == 0u;
__sf__ = (int)__res32__ < 0;
__of__ = ((__ebx__ >> 31)
          != (__eax__ >> 31))
          & ((__ebx__ >> 31)
          != (__res32__ >> 31));
if (!__zf__ & __sf__ == __of__)
  goto l1;
else goto l2;
l1: __tmp__ = __ebx__; goto l3;
l2: __tmp__ = __eax__; goto l3;
l3: __eax__ = __tmp__;
i = (int)__eax__;

TInA

int __tmp__;
if (a > i)
  __tmp__ = a;
else
  __tmp__ = i;
i = __tmp__;

- types consistency
- high-level predicate
- unpacking
- structuring
- expression propagation
- loop normalization
Questions?