Get rid of inline assembly through verification-oriented lifting

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

November 13, 2019
Software is not always reliable

1996

European Space Agency
Agence spatiale européenne

500M$

2009

TOYOTA

400

2016

GM
Then came formal methods

With **industrial** success stories in **regulated domains**
A grand challenge

Many barriers to formal methods adoption:

- learnability
- scalability
- ...

- automatization
- feature set
  - mixed-language support
  - ...

...
Today’s challenge:
mixed C & inline assembly code

with reuse of existing tools
Inline assembly is well spread

7k packages

786
11%

1264 projets
355 28%\(^1\)

\(^1\)according to Rigger et al.
Inline assembly is a pain

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

done for function main

====== VALUES COMPUTED ======
Values at end of function mid_pred:
  i ∈ [---.--]   i ∈ [-5..5]
Values at end of function main:
  a ∈ {0; 1; 2; 3; 4; 5}
  b ∈ [-5..10]
  c ∈ [-10..0]
  i ∈ [---.--]   i ∈ [-5..5]

Incomplete
Imprecise
Common workarounds

```c
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;
}
```

Manual handling
- manpower intensive
- error prone

Dedicated analyzer
- substantial engineering effort
int mid_pred (int a, int b, int c) {
    int i = b;
    #ifndef DISABLE_ASM
    __asm__ (
        "cmp %2, %1
        cmovg %1, %0
        cmovg %2, %1
        cmp %3, %1
        cmovl %3, %1
        cmp %1, %0
        cmovg %1, %0
    : "r"(i), "r"(a) :
        "r"(b), "r"(c)
    );
    #else
    i = max(a, b);
    a = min(a, b);
    i = min(i, a);
    #endif
    return i;
}

Manual handling

dedicated analyzer

want to reuse existing analyses!

error prone

substantial engineering effort

manpower intensive

error prone

substantial engineering effort

manpower intensive
Our proposition

Automatically **lift** ASM to equivalent **C**

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

```
int mid_pred (int a, int b, int c)
{
    int i = b;
    {__tina_tmp3, __tina_tmp2;
    int __tina_tmp1, __tina_tmp4;
    __TINA_BEGIN_1__:
        if (a > b) __tina_tmp3 = a;
        else __tina_tmp3 = i;
        if (a > b) __tina_tmp2 = b;
        else __tina_tmp2 = a;
        if (__tina_tmp2 < c) __tina_tmp1 = c;
        else __tina_tmp1 = __tina_tmp2;
        if (__tina_tmp3 > __tina_tmp1)
            __tina_tmp4 = __tina_tmp1;
        else __tina_tmp4 = __tina_tmp3;
        i = __tina_tmp4;
        __TINA_END_1__:
        return i;
    }
```

Reuse **C** tools
Challenges

Widely applicable
architecture – assembly dialect – compiler agnostic

Verification friendly
decent enough analysis outputs

Trustable
usable in sound formal method context
Challenges & key enablers

**Widely applicable**
architecture – assembly dialect – compiler agnostic

leverage existing binary-to-IR lifters – x86/ARM, GCC/clang

**Verification friendly**
decent enough analysis outputs

novel high-level simplifications – improve KLEE & Frama-C

**Trustable**
usable in sound formal method context

novel dedicated equivalence checking – 100% in scope success
Challenges & key enablers

**Widely applicable**
architecture – assembly dialect – compiler agnostic
leverage existing binary-to-IR lifters – x86/ARM, GCC/clang

**Verification friendly**
decent enough analysis outputs
novel high-level simplifications – improve KLEE & Frama-C

**Trustable**
usable in sound formal method context
novel dedicated equivalence checking – 100% in scope success

Evaluated over 2000+ assembly chunks from Debian packages
## Panorama of existing works

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Goanna(^1)</th>
<th>Vx86(^2)</th>
<th>Inception(^3)</th>
<th>TINA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Semantic lifting</strong></td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Widely applicable</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Trust</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sanity check</strong></td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Validation</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Verifiability</strong></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  
\(^3\) Corteggiani et al. Inception: System-Wide Security Testing of Real-World Embedded Systems Software
**Lifting: the basic case**

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

__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__;
```

---

**Compilation + Debug**

**IR lifting**

**Insertion**

**C + ASM**

**Object code**

**IR**

---

**Equiv alence**

✓ / × ?
Lifting: verification threats

T1. low-level data & computation
T2. low-level packing & representation
T3. unusual & unstructured control flow

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

```
__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__;
```
Lifting: running example

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

**T1.** low-level data & computation

**T2.** low-level packing & representation

**T3.** unusual & unstructured control flow

```c
__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__;  
```
Lifting: high-level predicate (Djoudi et al.)

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

__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__;
```

**T1.** low-level data & computation

**T2.** low-level packing & representation

**T3.** unusual & unstructured control flow
Lifting: high-level predicate (Djoudi et al.)

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

1. low-level data & computation
2. low-level packing & representation
3. unusual & unstructured control flow

```c
__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 ((int)__ebx__ > (int)__eax__) 
goto l1;
else goto l2;
l1: __tmp__ = __ebx__; goto l3;
l2: __tmp__ = __eax__; goto l3;
l3: __eax__ = __tmp__;
i = (int)__eax__;  
```
__asm__
(
    "cmp %0, %1 \n\t"
    "cmovg %1, %0 \n\t"
    /* [ ... ] */
    : "+&r" (i), "+&r" (a)
    : /* [ ... ] */
    : /* no clobbers */
);

__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 ((int)__ebx__ > (int)__eax__)
    goto l1;
else goto l2;
l1: __tmp__ = __ebx__; goto l3;
l2: __tmp__ = __eax__; goto l3;
l3: __eax__ = __tmp__; i = (int)__eax__;
Lifting: slicing

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

__eax__ = (unsigned int)i;
__ebx__ = (unsigned int)a;
if ((int)__ebx__ > (int)__eax__)
goto l1;
else goto l2;
l1: __tmp__ = __ebx__; goto l3;
l2: __tmp__ = __eax__; goto l3;
l3: __eax__ = __tmp__;
i = (int)__eax__;```

- **T1.** Low-level data & computation
- **T2.** Low-level packing & representation
- **T3.** Unusual & unstructured control flow
Lifting: structuring

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

if ((int)__ebx__ > (int)__eax__) __tmp__ = __ebx__; else __tmp__ = __eax__; __eax__ = __tmp__; i = __eax__;
Lifting: typing

```c
int __eax__ = i;
int __ebx__ = a;
int __tmp__;
if (__ebx__ > __eax__)
__tmp__ = __ebx__;
else
__tmp__ = __eax__;
__eax__ = __tmp__;
i = __eax__;
```

T1. low-level data & computation
T2. low-level packing & representation
T3. unusual & unstructured control flow
Lifting: expression propagation

```c
int __eax__ = i;
int __ebx__ = a;
int __tmp__;  // Initialize
if (__ebx__ > __eax__)  // Compare
    __tmp__ = __ebx__;  // Condition
else
    __tmp__ = __eax__;  // Condition
__eax__ = __tmp__;  // Store result
i = __eax__;  // Output
```

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

T1. low-level data & computation
T2. low-level packing & representation
T3. unusual & unstructured control flow
Lifting: expression propagation

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

int __eax__ = i;
int __ebx__ = a;
int __tmp__;
if (a > __eax__ i)
    __tmp__ = a;
else
    __tmp__ = __eax__ i;
__eax__ = __tmp__;
i = __eax__ __tmp__;
Lifting: expression propagation

```
__asm__

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

int __eax__ = i;
int __ebx__ = a;
int __tmp__;
if (a > i)
__tmp__ = a;
else
__tmp__ = i;
__eax__ = __tmp__;
i = __tmp__;
```

T1. low-level data & computation
T2. low-level packing & representation
T3. unusual & unstructured control flow
Lifting: high level simplifications

C + ASM → Object code → IR → C

Compilation + Debug → IR lifting → Insertion

Lifting

Transformations

__asm__

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

T1. low-level data & computation
T2. low-level packing & representation
T3. unusual & unstructured control flow

- types consistency
- high-level predicate
- unpacking

- structuring
- expression propagation
- loop normalization

int __tmp__; if (a > i) __tmp__ = a; else __tmp__ = i; i = __tmp__;
Validation: semantics equivalence

C + ASM → Object code → IR → C

LIFTING

Compilation + Debug

IR lifting

TRANSFORMATIONS

IR lifting

INSERTION

Object code → IR

VALIDATION

IR equivalence

✓ / ×?
Validation: tailored algorithm

Program equivalence is hard

C + ASM → Object code → IR → C

LIFTING

Compilation + Debug

Transformations

IR lifting

Insertion

Compilation + Debug

Object code

Validation

Equivalence

✓ / ×?
Validation: tailored algorithm

Program equivalence is hard

But all the process is controlled

Step 1: control flow graph isomorphism
labeled directed graph + debug information

Step 2: pairwise basic block equivalence check
SMT-based check
TInA: prototype

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

LIFTING

C + ASM → Object code → IR

TRANSFORMATIONS

IR lifting

VALIDATION

Equivalence

IR

Compilation + Debug

Object code

IR

IR lifting

Compilation + Debug

Object code

IR

IR
Experimental evaluation

- **Applicability & Trust**
  Debian, x86/ARM, GCC/clang

- **Verification friendly**
  KLEE and Frama-C
Widely Applicable: Debian 8.11 x86-32bit

<table>
<thead>
<tr>
<th></th>
<th>GCC v5.4</th>
<th>GCC v4.7</th>
<th>CLANG 3.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>All chunks</td>
<td>3039</td>
<td>2955</td>
<td>2955</td>
</tr>
<tr>
<td>Trivial</td>
<td>126</td>
<td>126</td>
<td>106</td>
</tr>
<tr>
<td>Out-of-scope</td>
<td>449</td>
<td>366</td>
<td>404</td>
</tr>
<tr>
<td>Rejected</td>
<td>138</td>
<td>137</td>
<td>412</td>
</tr>
<tr>
<td>Relevant</td>
<td>2326</td>
<td>2326</td>
<td>2033</td>
</tr>
<tr>
<td>Lifted</td>
<td>2326</td>
<td>2326</td>
<td>2033</td>
</tr>
<tr>
<td>Validated</td>
<td>2326</td>
<td>2326</td>
<td>2033</td>
</tr>
<tr>
<td>Average size</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Maximum size</td>
<td>341</td>
<td>341</td>
<td>341</td>
</tr>
<tr>
<td>Translation time</td>
<td>121s</td>
<td>105s</td>
<td>89s</td>
</tr>
<tr>
<td>Validation time</td>
<td>1527s</td>
<td>1528s</td>
<td>1336s</td>
</tr>
</tbody>
</table>
**Verifiability:** KLEE (symbolic execution)

<table>
<thead>
<tr>
<th></th>
<th>LIFTING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NONE</td>
</tr>
<tr>
<td># functions</td>
<td></td>
</tr>
<tr>
<td>with 100% branch coverage</td>
<td>×</td>
</tr>
<tr>
<td>Aggregate time for functions</td>
<td></td>
</tr>
<tr>
<td>with 100% branch coverage</td>
<td>N/A</td>
</tr>
<tr>
<td># explored paths</td>
<td>1 336k</td>
</tr>
<tr>
<td>for all functions</td>
<td></td>
</tr>
</tbody>
</table>

58 functions from ALSA, ffmpeg, GMP & libyuv

10min timeout
Verifiability: Frama-C EVA (abstract interpretation)

<table>
<thead>
<tr>
<th>Feature</th>
<th>TINA</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions with returns (non void)</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Better <strong>return precision</strong></td>
<td><strong>11</strong></td>
<td><strong>55%</strong></td>
</tr>
<tr>
<td>Functions with initial C alarm</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td><strong>Alarm reduction</strong> in C</td>
<td>23</td>
<td><strong>82%</strong></td>
</tr>
<tr>
<td><strong>New memory alarms</strong> ASM</td>
<td>17</td>
<td><strong>26%</strong></td>
</tr>
<tr>
<td><strong>Positive impact</strong></td>
<td>45</td>
<td><strong>77%</strong></td>
</tr>
</tbody>
</table>

58 functions from ALSA, ffmpeg, GMP & libyuv
### Verifiability: Frama-C WP (deductive verification)

<table>
<thead>
<tr>
<th>Function</th>
<th># Instr</th>
<th>Lifting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>saturated_sub</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>saturated_add</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>log2</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>mid_pred</td>
<td>7</td>
<td>✓</td>
</tr>
<tr>
<td>strcmpeq</td>
<td>9</td>
<td>✓</td>
</tr>
<tr>
<td>strlen</td>
<td>16</td>
<td>✓</td>
</tr>
<tr>
<td>memset</td>
<td>9</td>
<td>✓</td>
</tr>
<tr>
<td>count</td>
<td>8</td>
<td>✓</td>
</tr>
<tr>
<td>max_element</td>
<td>10</td>
<td>✓</td>
</tr>
<tr>
<td>cmp_array</td>
<td>10</td>
<td>✓</td>
</tr>
<tr>
<td>sum_array</td>
<td>20</td>
<td>✓</td>
</tr>
<tr>
<td>SumSquareError_SSE2</td>
<td>24</td>
<td>✓</td>
</tr>
</tbody>
</table>
Limits

Engineering

- floating point operations
- builtin crypto-operations

would challenge SMT & analyzers too

Genericity

- syscall
- hardware dependent

each analyzer has its own way to handle it
Conclusion

Inline ASM hinders the adoption of formal methods

**TInA**: Automated lifting
- Widely applicable
- Verification-friendly
- Trustable

Successful experimental evaluation over:
- $2000^+$ x86 Debian chunks – ARM *experiments too*
- KLEE & Frama-C friendly – *principled approach*
Conclusion

Inline ASM hinders the adoption of formal methods

**TInA**: Automated lifting
- Widely applicable
- Verification-friendly
- Trustable

Post-analysis considerations:
- 567 compliance issues
- ffmpeg coding flaws

Successful experimental evaluation over:
- 2000+ x86 Debian chunks – ARM experiments too
- KLEE & Frama-C friendly – principled approach
- Have a look @ the paper
- Meet us @ the conference

frederic.recoules@cea.fr  
up to Thursday night

sebastien.bardin@cea.fr  
up to Thursday noon

Any questions?