Automatic verification of low-level code: C, assembly and binary

Frédéric Recoules  CEA, List
Marie-Laure Potet  Grenoble INP  Director
Richard Bonichon  Nomadic Labs  Supervisor
Sébastien Bardin  CEA, List  Supervisor

Journées nationales du GDR GPL, June 7th 2022
Today’s challenge: mixed C & inline assembly code
# Ifdef __PIC__
__STRING_INLINE size_t
__strcspn_g (const char *__s, const char *__reject)
{
    register unsigned long int __d0, __d1, __d2;
    register const char *__res;
    __asm__ __volatile__
    ("pushl  %%ebx
     "movl   %4,%%edi
     "cld
     "repne; scasb
     "notl
     "leal  -1(%%ecx),%%ebx
     "1:\n"
     "lodsb"
     "testb  %%al,%%al"
     "je    2f"
     "movl   %4,%%edi"
     "movl   %%ebx,%%ecx"
     "repne; scasb"
     "jne   1b"
     "2:\n"
     "popl   %%ebx"
    : "=S" (__res), "=a" (__d0), "=c" (__d1), "=D" (__d2)
    : "r" (__reject), "o" (__s), "1" (0), "2" (0xffffffff)
    : "memory", "cc";
    return (__res - 1) - __s;
}
Inline assembly is well spread

7k packages

Found 3107 x86 chunks in 202 packages

11% 786

1264 projets 355
28% 1

FFMPEG
ALSA
GMP
libyuv

1 according to Rigger et al.
Adapting formal methods to common software is challenging
Inline assembly makes C analyzers ineffective

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

Incomplete    Imprecise
“GCC-style inline assembly is notoriously hard to write correctly”

Oliver Stannard,
ARM Senior Software Engineer on llvm threads, 2018
A few known inline assembly bugs

- `strcspn`  
glibc – Mars 1998 .. January 1999

- `compare_double_and_swap_double`  
libatomic_ops – February 2008 .. Mars 2012

- `compare_double_and_swap_double`  
libatomic_ops – Mars 2012 .. September 2012

- `bswap`  
libtomcrypt – April 2005 .. November 2012

GNU-style interface is really error-prone
## Goals & challenges

<table>
<thead>
<tr>
<th><strong>Interface compliance</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>must ensure that no bug lies in the interface</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Enable formal verification</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>must allow to perform verification of mixed C &amp; inline assembly code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Widely applicable</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>must be as much architecture, compiler and analysis agnostic</td>
</tr>
</tbody>
</table>

![x86](image1.png) ![arm](image2.png) ![GCC](image3.png) ![KLEE](image4.png) ![frama](image5.png) etc.
## Prior work on inline assembly

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Goanna(^1)</th>
<th>Vx86(^2)</th>
<th>Inception(^3)</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interface compliance</strong></td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Enable formal verification</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>
</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*
### Contributions

**A novel operational semantics for inline assembly**
- an operational semantics between C & binary
- a method to automatically extract inline assembly semantics (*TINA-core*)

**A method to check, patch and refine the interface**
- comprehensive formalization of interface compliance (*Framing* conditions & *Unicity* condition)
- thorough experiments with RUSTINA over 2.6k+ real-world chunks (986 severe issues found, 803 patches, 7 package patch accepted)
- a study of current bad coding practices (6 recurrent patterns yield 90% of issues, including 5 fragile patterns)

**A trustworthy, verification-oriented lifting method**
- first verification friendly lifting
- tailored post-lifting validation pass
- experiments with TINA over KLEE and Frama-C

---

[ICSE 2021]  
[ASE 2019]
The **interface compliance** challenge
AO_INLINE int
AO_compare_double_and_swap_double_full(volatile AO_double_t *addr,
    AO_t old_val1, AO_t old_val2,
    AO_t new_val1, AO_t new_val2)
{
    char result;
    [...]__asm__ __volatile__(
        "xchg %%ebx,%6; /* swap GOT ptr and new_val1 */
        "lock; cmpxchg8b %0; setz %1;"
        "xchg %%ebx,%6; /* restore ebx and edi */"
        : "=m"(*addr), "a"(result)
        : "m"(*addr), "d" (old_val2), "a" (old_val1),
        "c" (new_val2), "D" (new_val1) : "memory";
    [...] return (int) result;
}
This code works fine prior to GCC 5.0, then suddenly crashes with a **Segmentation fault**

- compiler knowledge is limited to the interface
- register allocation and optimizations rely on it
- code-interface mismatches can lead to bugs
### Goals & challenges

<table>
<thead>
<tr>
<th>Define interface compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>must be built on a currently missing proper formalization</td>
</tr>
<tr>
<td><em>indeed there is not even a complete documentation...</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Check, Patch &amp; Refine</th>
</tr>
</thead>
<tbody>
<tr>
<td>must be able to check whether an assembly chunk is compliant</td>
</tr>
<tr>
<td><em>ideally, should suggest a patch for the non compliant ones</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Widely applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>must be as much compiler agnostic</td>
</tr>
<tr>
<td><img src="https://example.com/gcc.png" alt="GCC" /></td>
</tr>
<tr>
<td>C compiler</td>
</tr>
</tbody>
</table>
## Contributions (1/2)

<table>
<thead>
<tr>
<th>A <strong>formalization</strong> of interface of compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>- support GCC, Clang and mostly icc</td>
</tr>
<tr>
<td>- <strong>Framing</strong> condition &amp; <strong>Unicity</strong> condition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A method to <strong>check</strong>, <strong>patch</strong> and <strong>refine</strong> the interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>- dataflow analysis + dedicated optimizations</td>
</tr>
<tr>
<td>- infer an over-approximation of the ideal interface</td>
</tr>
</tbody>
</table>
Interface compliance properties

Frame-write

Only *clobber* registers and *output* location are allowed to be *modified* by the assembly template

Frame-read

All *read* values must be *initialized* – only *input* dependent values are allowed in output productions, memory addressing and branching condition

Unicity

*The instruction behavior must not depend on the compiler choices*
Interface compliance properties

**Frame-write.** \( \forall l \not\in B^0 \cup S^C; \ S(l) = \text{exec}(S, C^{\langle T \rangle})(l) \)

*Only clobber registers and output location are allowed to be modified by the assembly template.*

**Frame-read.** \( \text{exec}(S_1, C^{\langle T \rangle}) \cong_{B^0, F} \text{exec}(S_2, C^{\langle T \rangle}) \)

*All read values must be initialized – only input dependent values are allowed in output productions, memory addressing and branching condition.*

**Unicity.** \( \text{exec}(S_1, C^{\langle T_1 \rangle}) \cong_{T_1, T_2, B^0, F} \text{exec}(S_2, C^{\langle T_2 \rangle}) \)

*The instruction behavior must not depend on the compiler choices (Unicity implies Frame-read)*
## Contributions (2/2)

### Thorough experiments of our prototype

- **2.6k+** real-world assembly chunks (**Debian**)
- **2183** issues, including **986 severe** issues
- **2000** patches, including **803 severe** fixes
- **7** packages have already accepted the fixes

https://github.com/binsec/icse2021-artifact992

DOI: 10.5281/zenodo.4601172

### A study of current inline assembly bad coding practices

- **6** recurrent patterns yield **90%** of issues
- **5** patterns rely on **fragile** assumptions
  - (**80%** of severe issues)
### Checking and patching statistics

<table>
<thead>
<tr>
<th>Category</th>
<th>Initial code</th>
<th>Patched code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Found issues</strong></td>
<td>2183</td>
<td>183</td>
</tr>
<tr>
<td><strong>significant issues</strong></td>
<td>986</td>
<td>183</td>
</tr>
</tbody>
</table>

**frame-write**
- ❌ – flag register clobbered: 1197 (0)
- ❌ – read-only input clobbered: 17 (0)
- ❌ – unbound register clobbered: 436 (0)
- ❌ – unbound memory access: 68 (0)

**frame-read**
- ❌ – non written write-only output: 19 (0)
- ❌ – unbound register read: 183 (183)
- ❌ – unbound memory access: 177 (0)

**unicity**
- 86 (0)

---

**Total time:** 2min – **Average time per chunk:** 40ms
### Common bad coding practices

**6** recurrent patterns yield **90%** of issues  
**5** of them can lead to **bugs**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Omitted clobber</th>
<th>Implicit protection</th>
<th>Robust?</th>
<th># issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 –</td>
<td>&quot;cc&quot;</td>
<td>compiler choice</td>
<td>✔️</td>
<td>1197</td>
</tr>
<tr>
<td>P2 –</td>
<td>%ebx register</td>
<td>compiler choice</td>
<td>✗️ (GCC ≥ 5) + 🔄️</td>
<td>30</td>
</tr>
<tr>
<td>P3 –</td>
<td>%esp register</td>
<td>compiler choice</td>
<td>✗️ (GCC ≥ 4.6) + 🔄️</td>
<td>5</td>
</tr>
<tr>
<td>P4 –</td>
<td>&quot;memory&quot;</td>
<td>function embedding</td>
<td>✗️ (inlining, cloning) + 🔄️</td>
<td>285</td>
</tr>
<tr>
<td>P5 –</td>
<td>MMX register</td>
<td>ABI</td>
<td>✗️ (inlining, cloning)</td>
<td>363</td>
</tr>
<tr>
<td>P6 –</td>
<td>XMM register</td>
<td>compiler option</td>
<td>✗️ (cloning)</td>
<td>109</td>
</tr>
</tbody>
</table>

衝: does not break – ✗️: has been broken – 🔄️: known bug
Submitted patches

- 114 faulty chunks in 8 packages (7 applied)
- 538 severe issues

- ALSA
  - libtomcrypt
  - xfstt
- FFMPEG
  - FFMPEG
  - x264
- haproxy
- UDPCast
- libatomic_ops
Verification-oriented lifting
Inline assembly makes C analyzers ineffective

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] expected

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

Incomplete

Imprecise
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 + ASM

Lift

C only

Reuse C tools

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));
    return i;
}

int mid_pred (int a, int b, int c)
{
    int i = b;
    { int __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;
}
## Goals & challenges

<table>
<thead>
<tr>
<th>Verification friendly</th>
</tr>
</thead>
<tbody>
<tr>
<td>decent enough analysis outputs for verification process</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trustable</th>
</tr>
</thead>
<tbody>
<tr>
<td>usable in sound formal method context</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Widely applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>must be generic and verification technique agnostic</td>
</tr>
</tbody>
</table>

- KEK
- frama
- EVA
- frama
- WP
- etc.
## Contributions

<table>
<thead>
<tr>
<th>Dedicated high-level structure recovery mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>• identify 3 main threats to verifiability</td>
</tr>
<tr>
<td>• dedicated rewriting steps</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tailored validation pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>• preserve control flow graph isomorphism</td>
</tr>
<tr>
<td>• SMT based basic block equivalence checking</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thorough experiments of our prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 100% validation of lifted chunks</td>
</tr>
<tr>
<td>• positive impact of TInA for 3 standard verification tools</td>
</tr>
<tr>
<td>(KLEE, Frama-C EVA, Frama-C WP)</td>
</tr>
</tbody>
</table>
Verification-oriented lifting

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

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

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

- high-level predicate
- unpacking
- expression propagation
- loop normalization
### Verifiability of lifted code

<table>
<thead>
<tr>
<th>Lifting</th>
<th>Analysis</th>
<th>KLEE symbolic execution</th>
<th>Frama-C EVA abstract interpretation</th>
<th>Frama-C WP deductive verification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Criterion</td>
<td>Number of explored paths in 10m timeout</td>
<td>Number of functions without alarms</td>
<td>Number of fully discharged proofs</td>
</tr>
<tr>
<td><strong>NONE</strong></td>
<td>1 336k</td>
<td>0 / 58</td>
<td>0 / 12</td>
<td></td>
</tr>
<tr>
<td><strong>BASIC</strong></td>
<td>1 459k</td>
<td>12 / 58</td>
<td>1 / 12</td>
<td></td>
</tr>
<tr>
<td><strong>TI&amp;A</strong></td>
<td>6 402k</td>
<td><strong>19</strong> / 58</td>
<td>12 / 12</td>
<td></td>
</tr>
</tbody>
</table>
**Summary**

### A novel operational semantics for inline assembly
- an operational semantics between C & binary
- a method to automatically extract inline assembly semantics (TINA-core)

### A method to check, patch and refine the interface
- comprehensive formalization of interface compliance
  - (Framing conditions & Unicity condition)
- thorough experiments with RUSTInA over 2.6k+ real-world chunks
  - (986 severe issues found, 803 patches, 7 package patch accepted)
- a study of current bad coding practices
  - (6 recurrent patterns yield 90% of issues, including 5 fragile patterns)

### A trustworthy, verification-oriented lifting method
- first verification friendly lifting
- tailored post-lifting validation pass
- experiments with TINA over KLEE and Frama-C

---

[ICSE 2021]

[ASE 2019]
Thank you for your attention
# Summary

**A novel operational semantics for inline assembly**
- an operational semantics between C & binary
- a method to automatically extract inline assembly semantics (TINA-core)

**A method to check, patch and refine the interface**
- comprehensive formalization of interface compliance
  - (Framing conditions & Unicity condition)
- thorough experiments with RUSTINA over 2.6k+ real-world chunks
  - (986 severe issues found, 803 patches, 7 package patch accepted)
- a study of current bad coding practices
  - (6 recurrent patterns yield 90% of issues, including 5 fragile patterns)

**A trustworthy, verification-oriented lifting method**
- first verification friendly lifting
- tailored post-lifting validation pass
- experiments with TINA over KLEE and Frama-C