Interface Compliance of Inline Assembly: Automatically Check, Patch and Refine

Frédéric Recoules

Sébastien Bardin
Richard Bonichon
Matthieu Lemerre
Laurent Mounier
Marie-Laure Potet

Université Paris-Saclay, CEA, LIST

International Conference on Software Engineering, 2021
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;
}
Inline assembly is well spread

7k packages

Found 3107 x86 chunks in 202 packages

786
11%

1264 projets
355
28%

- full access to hardware
- hand-crafted optimization
- security / obfuscation

1 according to Rigger et al., 2018
“GCC-style inline assembly is notoriously hard to write correctly”

Oliver Stannard,
ARM Senior Software Engineer on LLVM threads, 2018
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;
}
```
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;
}
```
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;
}
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;
}
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" /* restore ebx and edi */
    "xchg %ebx,%6" /* swap GOT ptr and new_val1 */
    : "=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
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
Today’s challenge: Interface Compliance

Define – Check – Patch
# Challenges

## Define

must be built on a currently missing proper formalization

*indeed there is not even a complete documentation.*

## Check, Patch & Refine

must be able to check whether an assembly chunk is compliant

*ideally, should suggest a patch for the non compliant ones*

## Widely applicable

must be compiler & architecture agnostic

![GCC](logo_gcc.png)  ![Intel](logo_intel.png)  ![x86](logo_x86.png)  ![arm](logo_arm.png)
# Our contributions (1/2)

## A novel semantics and comprehensive formalization

- support GCC, Clang and mostly icc
- **Framing** condition & **Unicity** condition

## A method to check, patch and refine the interface

- dataflow analysis + dedicated optimizations
- infer an over-approximation of the ideal interface
## Our 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

**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)*
GNU documentation is informal & incomplete

- no standard, only based on GCC implementation
- non documented behaviors may change at any time
- Clang and icc follow “what they understood”
Looking for a formalism – reverse engineering

GNU Syntax

```c
__asm__ volatile (  
    "C\^: asm\^ 
      lock;  
      cmpxchgl %3, %0;  
      setz %1  
    
    : "=m" (*addr),  
       "=q" (result)  
    : "m" (*addr),  
       "r" (new_val),  
       "a" (old)  
    : "memory"
)
```

\( C^\diamond : \text{asm}^\diamond \)

\( C^\bigcirc = \left[ C^\diamond \right]^\text{x86} \)

\( I^\bigcirc : \text{interface} \)

\( B^\bigcirc = \{ (%0, *addr, indirect),  
                   (%1, result, direct) \} \)

\( B^\bigcirc = \{ (%0, *addr, indirect),  
                   (%3, new_val, direct),  
                   (%4, old, direct) \} \)

\( F = \text{false} /* no memory separation */ \)

\( S^\bigcirc = \{ \} /* no clobber registers */ \)

\( S^T = \{ \)

\( T_1 = \{ %0 \mapsto *(%ebx), %1 \mapsto %eax,  
         %3 \mapsto %edx, %4 \mapsto %eax \}, \)

\( T_2 = \{ %0 \mapsto *(%ebx), %1 \mapsto %ecx,  
         %3 \mapsto %ebp, %4 \mapsto %eax \}, \)

.. \)
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.** \( \forall l \notin B^0 \cup S^c; \ S(l) = \text{exec}(S, C^i<T>)(l) \)

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

**Frame-read.** \( \text{exec}(S_1, C^i<T>) \overset{\sim}{\cong}_{B^0,F} \text{exec}(S_2, C^i<T>) \)

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

**Unicity.** \( \text{exec}(S_1, C^i<T_1>) \overset{\sim}{\cong}_{B^0,F}^{T_1,T_2} \text{exec}(S_2, C^i<T_2>) \)

*The instruction behavior must not depend on the compiler choices (Unicity implies Frame-read)*
Checking the compliance

Dedicated *dataflow* analysis

**Frame-write.** Collect all the left hand side expressions.

**Frame-read.** *Liveness analysis* – collect all the living dependencies of right hand side expression.

**Unicity.** Check that no living location (tokens or registers) may be impacted by the side effect of another location write.

with precision enhancers: expression propagation + bit level liveness
Our prototype RUSTInA
Experimental evaluation of RUSTInA

- How does RUSTInA perform at checking and patching?
- Why do so many issues not turn more often into bugs?
- What is the real impact of the reported issues?
- What is the impact of the design choices?
## Checking and patching statistics

<table>
<thead>
<tr>
<th></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>significant issues</td>
<td>986</td>
<td>183</td>
</tr>
<tr>
<td><strong>frame-write</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>– flag register clobbered</td>
<td>1197</td>
<td>0</td>
</tr>
<tr>
<td>– read-only input clobbered</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>– unbound register clobbered</td>
<td>436</td>
<td>0</td>
</tr>
<tr>
<td>– unbound memory access</td>
<td>68</td>
<td>0</td>
</tr>
<tr>
<td><strong>frame-read</strong></td>
<td>379</td>
<td>183</td>
</tr>
<tr>
<td>– non written write-only output</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>– unbound register read</td>
<td>183</td>
<td>183</td>
</tr>
<tr>
<td>– unbound memory access</td>
<td>177</td>
<td>0</td>
</tr>
<tr>
<td><strong>unicity</strong></td>
<td>86</td>
<td>0</td>
</tr>
</tbody>
</table>

**Total time:** 2min – **Average time per chunk:** 40ms
Common issues (90%) do not break very often. Why is that?

What if we stress out the compilation process?
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 – &quot;cc&quot;</td>
<td>compiler choice</td>
<td>✔️</td>
<td></td>
<td>1197</td>
</tr>
<tr>
<td>P2 – %ebx register</td>
<td>compiler choice</td>
<td>✗ (GCC ≥ 5) + 🚫</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>P3 – %esp register</td>
<td>compiler choice</td>
<td>✗ (GCC ≥ 4.6) + 🚫</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>P4 – &quot;memory&quot;</td>
<td>function embedding</td>
<td>✗ (inlining, cloning) + 🚫</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>P5 – MMX register</td>
<td>ABI</td>
<td>✗ (inlining, cloning)</td>
<td>363</td>
<td></td>
</tr>
<tr>
<td>P6 – XMM register</td>
<td>compiler option</td>
<td>✗ (cloning)</td>
<td>109</td>
<td></td>
</tr>
</tbody>
</table>

✔️: does not break – ✗: has been broken – 🚫: known bug

792 80%
Real-life impact of RUSTInA

Submitted patches

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

ALSA
- libtomcrypt
- xfstt

FFmpeg
- haproxy
- x264

UDPcast
- libatomic_ops
• Have a look @ the paper
• Have a look @ the artifact
• Have a look @ 🔄 BINSEC

Interface compliance is hard, it matters but it is no longer a problem thanks to RUSTInA

If you have any question, do not hesitate!

frederic.recoules@cea.fr    https://binsec.github.io/
## Panorama of existing works

<table>
<thead>
<tr>
<th></th>
<th>Binary lifter</th>
<th>Interface checker</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vx86&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Inception&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Frame check</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Unicity check</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Interface patch</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Widely applicable</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

<sup>1</sup>Schulte et al. Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving

<sup>2</sup>Corteggiani et al. Inception: System-Wide Security Testing of Real-World Embedded Systems Software

<sup>3</sup>Recoules et al. Get Rid of Inline Assembly through Verification-Oriented Lifting

<sup>4</sup>Fehnker et al. Some Assembly Required - Program Analysis of Embedded System Code
## Real-life impact (detailed)

<table>
<thead>
<tr>
<th>Project</th>
<th>About</th>
<th>Status</th>
<th>Patched chunks</th>
<th>Fixed issues</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALSA</td>
<td>Multimedia</td>
<td>Applied</td>
<td>20</td>
<td>64/64</td>
<td>01d8a6e, 0fd7f0c</td>
</tr>
<tr>
<td>haproxy</td>
<td>Network</td>
<td>Applied</td>
<td>1</td>
<td>1/1</td>
<td>09568fd</td>
</tr>
<tr>
<td>libatomic_ops</td>
<td>Multi-threading</td>
<td>Applied</td>
<td>1</td>
<td>1/1</td>
<td>05812c2</td>
</tr>
<tr>
<td>libtomcrypt</td>
<td>Cryptography</td>
<td>Applied</td>
<td>2</td>
<td>2/2</td>
<td>cefff85</td>
</tr>
<tr>
<td>UDPCast</td>
<td>Network</td>
<td>Applied</td>
<td>2</td>
<td>2/2</td>
<td>20200328</td>
</tr>
<tr>
<td>xfstt</td>
<td>X Server</td>
<td>Applied</td>
<td>1</td>
<td>3/3</td>
<td>91c358e</td>
</tr>
<tr>
<td>x264</td>
<td>Multimedia</td>
<td>Applied</td>
<td>11</td>
<td>83/83</td>
<td>69771</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>Multimedia</td>
<td>Review</td>
<td>76</td>
<td>382/382</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>114</strong></td>
<td><strong>538</strong></td>
<td>(55% of severe issues)</td>
</tr>
</tbody>
</table>