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

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

Univ. Paris-Saclay, CEA, List
Univ. Paris-Saclay, CEA, List
Tweag I/O
Univ. Paris-Saclay, CEA, List
Univ. Grenoble Alpes, VERIMAG
Univ. Grenoble Alpes, VERIMAG

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

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

*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 A0_double_t *addr,
A0_t old_val1, A0_t old_val2,
A0_t new_val1, A0_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__ (*addr)
    : "m"(*addr), "a"(result); 
    [...] return (int) result;
}
AO_INLINE int 
AO_compare_double_and_swap_double_full(volatile AO_double_t *addr,
A0_t old_val1, A0_t old_val2,
A0_t new_val1, A0_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; " // eax
        "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
- mismatches code-interface can lead to bugs
A few known inline assembly bugs

- `strcspn` 
glibc – January 1999, commit 7c97add

- `compare_double_and_swap_double` 
libatomic_ops – Mars 2012, commit 30cea1b

- `compare_double_and_swap_double` 
libatomic_ops – September 2012, commit 64d81cd

- `bswap` 
libtomcrypt – November 2012, commit cefff85

Interface compliance does matter
Today’s challenge: Interface Compliance

Define – Check – Patch
## Goal & 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], [Intel], [x86], [arm] |
Our contributions (1/2)

<table>
<thead>
<tr>
<th>A novel semantics and comprehensive formalization</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 check, patch and refine the interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>- dataflow analysis + dedicated optimizations</td>
</tr>
<tr>
<td>- infer an <strong>over-approximation</strong> of the ideal interface</td>
</tr>
</tbody>
</table>
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


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 the missing formalism

Syntax GNU

```
__asm__ volatile (
   "
   lock;
   cmpxchg8b %3, %0;
   setz %1
   
   "
   : "=m" (*addr),
     "=q" (result)
   : "m" (*addr),
     "r" (new_val),
     "a" (old)
   : "memory"
)
```

\[ C^\circ : \text{asm}^\circ \]

```
domainx86
D(%0) = { *(%ebx), .. }  
D(%1) = { %eax, %ebx, %ecx, %edx }  
D(%3) = { %eax, %ebx, %ecx, %edx,  
                %esi, %edi, %ebp }  
D(%4) = { %eax }
```

\[ C^\bullet : \text{asm}^\bullet \]

\[ C^\circ = [C^\circ]_{x86} \]

```
extract

C^\bullet = [C^\circ]_{x86}
```

\[ \Gamma^\circ : \text{interface} \]

```
\begin{align*}
\mathcal{S}^b &= \{ (X_0, *addr, indirect), \\
&\quad (X_1, result, direct) \} \\
\mathcal{S}^i &= \{ (X_0, *addr, indirect), \\
&\quad (X_3, new\_val, direct), \\
&\quad (X_4, old, direct) \} \\
F &= false /* no memory separation */ \\
\mathcal{S}^c &= \{ \} /* no clobber registers */ \\
\mathcal{S}^g &= \{ \\
&\quad T_1 = \{ X_0 \mapsto *(X_{ebx}), X_1 \mapsto %eax, \\
&\quad \quad X_3 \mapsto %edx, X_4 \mapsto %eax \}, \\
&\quad T_2 = \{ X_0 \mapsto *(X_{ebx}), X_1 \mapsto %ecx, \\
&\quad \quad X_3 \mapsto %ebp, X_4 \mapsto %eax \}, \\
&\quad .. \}
\end{align*}
```

\[ \Gamma^\bullet : \text{interface} \]
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 of the compiler choices”
Interface compliance properties

**Frame-write**: $\forall l \not\in B^0 \cup S^C; \ S(l) = \text{exec}(S, C'<T>)(l)$

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

**Frame-read**: $\text{exec}(S_1, C'<T>) \overset{T}{\cong}_{B^0,F} \text{exec}(S_2, C'<T>)$

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

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

“The instruction behavior must not depend of the compiler choices”

(Unicity implies Frame-read)
Our prototype RUSTINA
Experimental evaluation

☐ How does perform RUSTINA at checking and patching?
☐ Why so many issues do not turn more often into bugs?
☐ What is the real impact of the reported issues?

(more research questions addressed in the paper)
Checking and patching statistics

<table>
<thead>
<tr>
<th>Category</th>
<th>Initial code</th>
<th>Patched code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Found issues</td>
<td>2183</td>
<td>183</td>
</tr>
<tr>
<td>significant issues</td>
<td>986</td>
<td>183</td>
</tr>
<tr>
<td>frame-write</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>frame-read</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>unicity</td>
<td>86</td>
<td>0</td>
</tr>
</tbody>
</table>

Over 2656 chunks
- 49% fully compliant
- 97% benign issues
- 3% serious issues

Over 202 packages
- 58% fully compliant
- 88% benign issues
- 12% serious issues

Total time: 2min – Average time per chunk: 40ms
Common issues (90%) do not break very often. Are they somehow under “implicit protections”? What if we stress out the compilation process? (“copy-paste”, -O3, -lto, etc.)
# 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>

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

792 80%
Real-life impact of RUSTIN

Submitted patches (applied or in review)

- 114 faulty chunks in 8 packages
- 538 severe issues (55%)

Libraries:
- libtomcrypt
- xfstt
- x264
- UDPCast
- ALSA
- FFMPEG
- libatomic_ops
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/