Interface Compliance of Inline Assembly:

Automatically Check, Patch and Refine

Frédéric Recoules Univ. Paris-Saclay, CEA, List

Sébastien Bardin Univ. Paris-Saclay, CEA, List

Richard Bonichon Tweag I/O

Matthieu Lemerre Univ. Paris-Saclay, CEA, List

Laurent Mounier Univ. Grenoble Alpes, VERIMAG

Marie-Laure Potet Univ. Grenoble Alpes, VERIMAG

International **C**onference on **S**oftware **E**ngineering, 2021

Inline assembly example in C code

```c
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

debian GNU/Linux

786
11%

7k packages

Found **3107** *x86 chunks in 202 packages*

GitHub

1264 projets

355
28%[a]

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

[a] 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

Inline assembly example in C code

```c
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 example in C code

```c
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;
}
```

Assembly template

Inline assembly example in C code

```c
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;
}
```

Assembly template

Inline assembly example in C code

```
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;
}
```

Assembly template

Output list

Input list

Clobber list

Inline assembly example in C code

```c
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;
}
```

Assembly template

%eax

Output list

Input list

Clobber list

%ecx

%edi

%edx

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

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

`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)

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 (
  "       C◇ : asm◇
    lock;
    cmpxchgl %3, %0;

    setz %1
                   "
    : "=m" (*addr),
      "=q" (result)
    : "m" (*addr),
      "r" (new_val),
      "a" (old)
    : "memory"
);
```

extract

$C^{\blacklozenge} = [\![C^{\diamond}]\!]^{\diamond}_{x86}$

$C^{\blacklozenge}$: $\texttt{asm}^{\blacklozenge}$

$z \longleftarrow$ %eax = %0

%eax $\longleftarrow$ %0

%0 $\overset{z}{\longleftarrow}$ %3

%1 $\longleftarrow$ z

model

$I^{\diamond}$: interface

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

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

F = false /* no memory separation */

$S^{C} = \{ \}$ /* 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 $\}$,

.. $\}$

$\texttt{domain}_{x86}$

$\mathbb{D}$(%0) = { *(%ebx), .. }

$\mathbb{D}$(%1) = { %eax, %ebx, %ecx, %edx }

$\mathbb{D}$(%3) = { %eax, %ebx, %ecx, %edx,
%esi, %edi, %ebp }

$\mathbb{D}$(%4) = { %eax }

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 : $\forall \mathtt{l} \notin \mathtt{B}^{\mathtt{O}} \cup \mathtt{S}^{\mathtt{C}};\ \mathtt{S}(\mathtt{l}) \ = \ \mathtt{exec}(\mathtt{S},\ \mathtt{C}^{\iota}\mathtt{<T>})(\mathtt{l})$

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

Frame-read : $\mathtt{exec}(\mathtt{S}_1,\ \mathtt{C}^{\iota}\mathtt{<T>}) \ \overset{\blacklozenge}{\cong}{}^{\mathtt{T}}_{\mathtt{B}^{\mathtt{O}},\mathtt{F}} \ \mathtt{exec}(\mathtt{S}_2,\ \mathtt{C}^{\iota}\mathtt{<T>})$

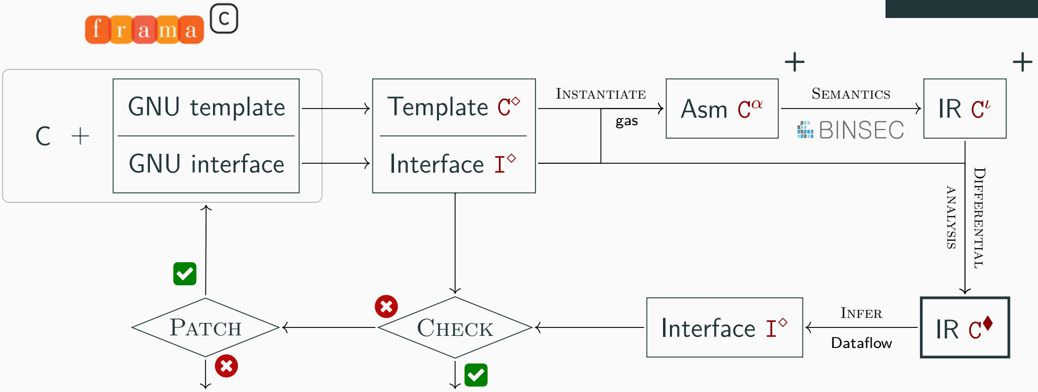
"All read values must be initialized – only input dependent values are allowed in output productions, memory addressing and branching condition"

Unicity : $\mathtt{exec}(\mathtt{S}_1,\ \mathtt{C}^{\iota}\mathtt{<T}_1\mathtt{>}) \ \overset{\blacklozenge}{\cong}{}^{\mathtt{T}_1,\mathtt{T}_2}_{\mathtt{B}^{\mathtt{O}},\mathtt{F}} \ \mathtt{exec}(\mathtt{S}_2,\ \mathtt{C}^{\iota}\mathtt{<T}_2\mathtt{>})$

"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?

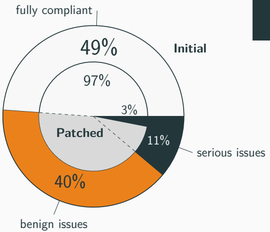
(more research questions addressed in the paper)

Checking and patching statistics

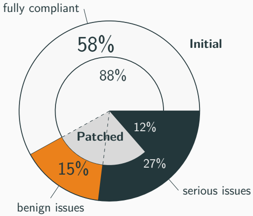
	Initial code	Patched code
Found issues	**2183**	183
significant issues	986	183
frame-write	**1718**	0
– flag register clobbered	1197	0
– read-only input clobbered	17	0
– unbound register clobbered	436	0
– unbound memory access	68	0
frame-read	**379**	183
– non written write-only output	19	0
– unbound register read	183	183
– unbound memory access	177	0
unicity	**86**	0

Total time: *2min* – Average time per chunk: *40ms*

Over 2656 chunks



Over 202 packages



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**

Pattern	Omitted clobber	Implicit protection	Robust?	# issues
P1 –	`"cc"`	compiler choice	✅	1197
P2 –	`%ebx` register	compiler choice	❌ (GCC $\geq$ 5) + 🐞	30
P3 –	`%esp` register	compiler choice	❌ (GCC $\geq$ 4.6) + 🐞	5
P4 –	`"memory"`	function embedding	❌ (inlining, cloning) + 🐞	285
P5 –	MMX register	ABI	❌ (inlining, cloning)	363
P6 –	XMM register	compiler option	❌ (cloning)	109
				792 80%

✅ : does not break – ❌ : has been broken – 🐞 : known bug

Real-life impact of RUSTInA

Submitted patches (applied or in review)

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

ALSA

libtomcrypt

FFmpeg

xfstt

haproxy

x264

libatomic_ops

UDPCast

- 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/