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

7k packages

786
11%

Found 3107 x86 chunks in 202 packages

GitHub

1264 projets

355
28%[1]

- 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

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;
}
```

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

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

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

Output list

Input list

Clobber list

%eax

%ecx

%edx

%edi

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 intel x86 arm

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

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

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

extract

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

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

```
z     ⟵   %eax = %0
%eax  ⟵   %0
%0    ⟵z  %3
%1    ⟵   z
```

model

$I^{\lozenge}$: 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 = {
  T1 = { %0 ↦ *(%ebx), %1 ↦ %eax,
         %3 ↦ %edx, %4 ↦ %eax },
  T2 = { %0 ↦ *(%ebx), %1 ↦ %ecx,
         %3 ↦ %ebp, %4 ↦ %eax },
  .. }
```

domain_{x86}

```
D(%0) = { *(%ebx), .. }
D(%1) = { %eax, %ebx, %ecx, %edx }
D(%3) = { %eax, %ebx, %ecx, %edx,
          %esi, %edi, %ebp }
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 on 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 on the compiler choices

(**Unicity** implies **Frame-read**)

Checking the compliance

Dedicated dataflow analysis

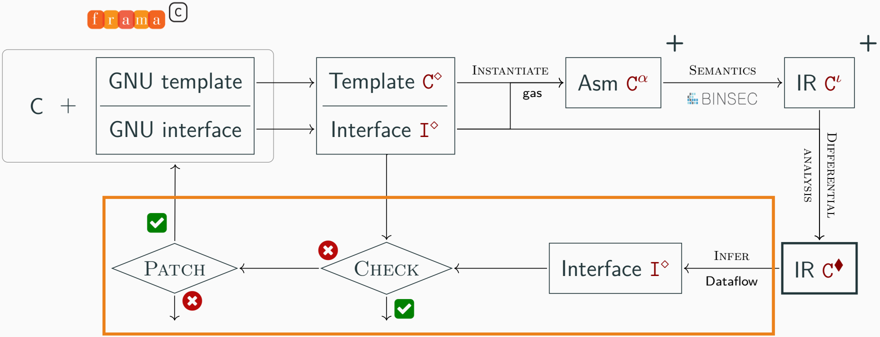
Frame-write. Collect all the left hand side expressions.

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



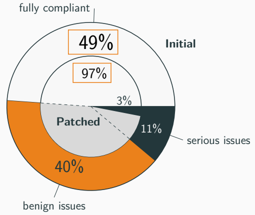
Experimental evaluation of RUSTInA

- ☐ 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?

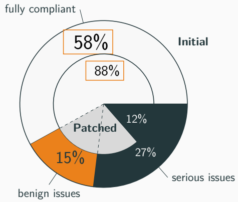
Checking and patching statistics

	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

Over 2656 chunks



Over 202 packages



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**

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

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

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/

Panorama of existing works

	Binary lifter			Interface checker	
	Vx86[1]	Inception[2]	TInA[3]	Goanna[4]	RUSTInA
Frame check	✗	✗	✓	✓	✓
Unicity check	✗	✗	✗	✗	✓
Interface patch	✗	✗	✗	✗	✓
Widely applicable	✗	✓	✓	✗	✓

[1] Schulte et al. Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving

[2] Corteggiani et al. Inception: System-Wide Security Testing of Real-World Embedded Systems Software

[3] Recoules et al. Get Rid of Inline Assembly through Verification-Oriented Lifting

[4] Fehnker et al. Some Assembly Required - Program Analysis of Embedded System Code

Real-life impact (detailed)

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