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

Inline assembly example (bits/strings.h@glibc_2.19)

```
# 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\n\t"
     "movl        %4,%%edi\n\t"
     "cld\n\t"
     "repne; scasb\n\t"
     "notl        %%ecx\n\t"
     "leal        -1(%%ecx),%%ebx\n"
     "1:\n\t"
     "lodsb\n\t"
     "testb        %%al,%%al\n\t"
     "je        2f\n\t"
     "movl        %4,%%edi\n\t"
     "movl        %%ebx,%%ecx\n\t"
     "repne; scasb\n\t"
     "jne        1b\n"
     "2:\n\t"
     "popl        %%ebx"
     : "=S" (__res), "=&a" (__d0), "=&c" (__d1), "=&D" (__d2)
     : "r" (__reject), "0" (__s), "1" (0), "2" (0xffffffff)
     : "memory", "cc");
  return (__res - 1) - __s;
}

# endif
```

Inline assembly is well spread

debian GNU/Linux

786
11%

7k packages

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

GitHub

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

KLEE

```
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
```

frama C

```
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

Interface compliance

must ensure that no bug lies in the interface

Enable formal verification

must allow to perform verification of mixed C & inline assembly code

Widely applicable

must be as much architecture, compiler and analysis agnostic

x86 arm GCC frama ⓒ etc.

Prior work on inline assembly

	Manual	Goanna[1]	Vx86[2]	Inception[3]	**Goal**
Interface compliance	✓	✓	N/A	✗	✓
Enable formal verification	✓	✗	✓	✓	✓
Widely applicable	✗	✗	✗	✓	✓

[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)

[ICSE 2021]

A **trustworthy**, **verification-oriented** lifting method

- first **verification friendly** lifting
- tailored post-lifting **validation pass**
- experiments with TINA over KLEE and Frama-C

[ASE 2019]

The interface compliance challenge

Inline assembly example (atomic_ops/sysdeps/gcc/x86.h)

```
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

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

Define interface compliance

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 as much compiler agnostic GCC C compiler

Contributions (1/2)

A formalization of interface of compliance

- 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

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. $\forall \texttt{l} \not\in \texttt{B}^{\texttt{O}} \cup \texttt{S}^{\texttt{C}};\ \texttt{S}(\texttt{l}) = \texttt{exec}(\texttt{S},\ \texttt{C}^{\iota}\texttt{<T>})(\texttt{l})$

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

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

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

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

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

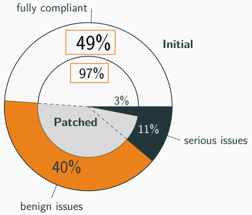
A study of current inline assembly bad coding practices

- 6 recurrent patterns yield **90%** of issues
- 5 patterns rely on **fragile** assumptions (**80%** of severe issues)

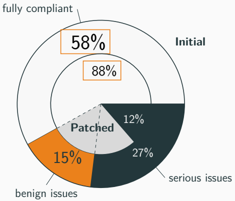
Checking and patching statistics

	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 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 ≥ 5) + 🐞	30
P3 –	`%esp` register	compiler choice	❌ (GCC ≥ 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

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
```

Incomplete

```
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
```

Imprecise

Common workarounds

```c
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;
}
```

Manual handling

manpower intensive

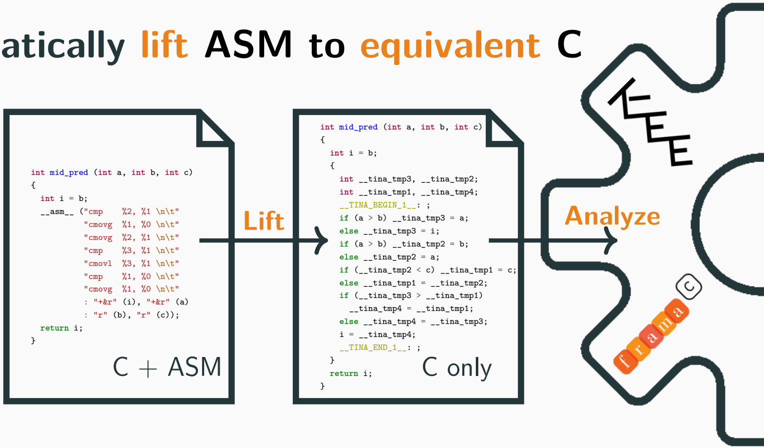
error prone

Dedicated analyzer

substantial engineering effort

Our proposition

Automatically lift ASM to equivalent C



Reuse C tools

Goals & challenges

Verification friendly

decent enough analysis outputs for verification process

Trustable

usable in sound formal method context

Widely applicable

must be generic and verification technique agnostic

KLEE   frama-c EVA   frama-c WP   etc.

Contributions

Dedicated high-level structure recovery mechanism

- identify 3 main threats to verifiability
- dedicated rexriting steps

Tailored validation pass

- preserve control flow graph isomorphism
- SMT based basic block equivalence checking

Thorough experiments of our prototype

- **100**% validation of lifted chunks
- positive impact of TInA for 3 standard verification tools
  (KLEE, Frama-C EVA, Frama-C WP)

Verification-oriented lifting

original

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

T1. low-level data & computation

T2. low-level packing & representation

T3. unusual & unstructured control flow

basic lifting

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

TINA lifting

```
int __tmp__;
if (a > i)
  __tmp__ = a;
else
  __tmp__ = i;
i = __tmp__;
```

- high-level predicate
- unpacking
- expression propagation
- loop normalization

Verifiability of lifted code

	Analysis	KLEE symbolic execution	Frama-C EVA abstract interpretation	Frama-C WP deductive verification
	Criterion	Number of explored paths in 10m timeout	Number of functions without alarms	Number of fully discharged proofs
Lifting	None	1 336k	0 / 58	0 / 12
	Basic	1 459k	12 / 58	1 / 12
	TInA	**6 402k**	**19** / 58	**12** / 12

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)

[ICSE 2021]

A trustworthy, verification-oriented lifting method

- first **verification friendly** lifting
- tailored post-lifting **validation pass**
- experiments with TINA over KLEE and Frama-C

[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

[ICSE 2021]

- 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

[ASE 2019]

- first **verification friendly** lifting
- tailored post-lifting **validation pass**
- experiments with TINA over KLEE and Frama-C