Automatic verification of low-level code: C, assembly and binary

Université Grenoble Alpes

Jury

Antoine Miné	*Sorbonne Université*	Reviewer
Valérie Viet Triem Tong	*CentralSupelec*	Reviewer
Roland Groz	*Grenoble INP*	Examiner
Julia Lawall	*INRIA Centre de Paris*	Examiner
Marie-Laure Potet	*Grenoble INP*	Director
Richard Bonichon	*Nomadic Labs*	Supervisor
Sébastien Bardin	*CEA, List*	Supervisor

Frédéric Recoules thesis defense, September 30th 2021

Verimag

list cea tech

Today's challenge :

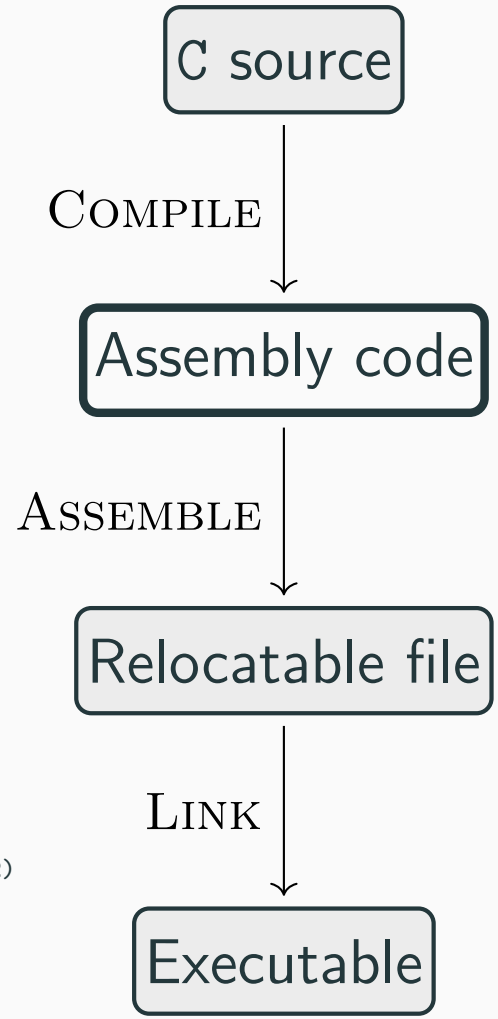
mixed C & inline assembly code

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

```
1563    # ifdef __PIC__
1565    __STRING_INLINE size_t
1566    __strcspn_g (const char *__s, const char *__reject)
1567    {
1568      register unsigned long int __d0, __d1, __d2;
1569      register const char *__res;
1570      __asm__ __volatile__
1571        ("pushl        %%ebx\n\t"
1572         "movl         %4,%%edi\n\t"
1573         "cld\n\t"
1574         "repne; scasb\n\t"
1575         "notl         %%ecx\n\t"
1576         "leal         -1(%%ecx),%%ebx\n"
1577         "1:\n\t"
1578         "lodsb\n\t"
1579         "testb        %%al,%%al\n\t"
1580         "je         2f\n\t"
1581         "movl         %4,%%edi\n\t"
1582         "movl         %%ebx,%%ecx\n\t"
1583         "repne; scasb\n\t"
1584         "jne        1b\n"
1585         "2:\n\t"
1586         "popl         %%ebx"
1587         : "=S" (__res), "=&a" (__d0), "=&c" (__d1), "=&D" (__d2)
1588         : "r" (__reject), "0" (__s), "1" (0), "2" (0xffffffff)
1589         : "memory", "cc");
1590      return (__res - 1) - __s;
1591    }
1618    # endif
```

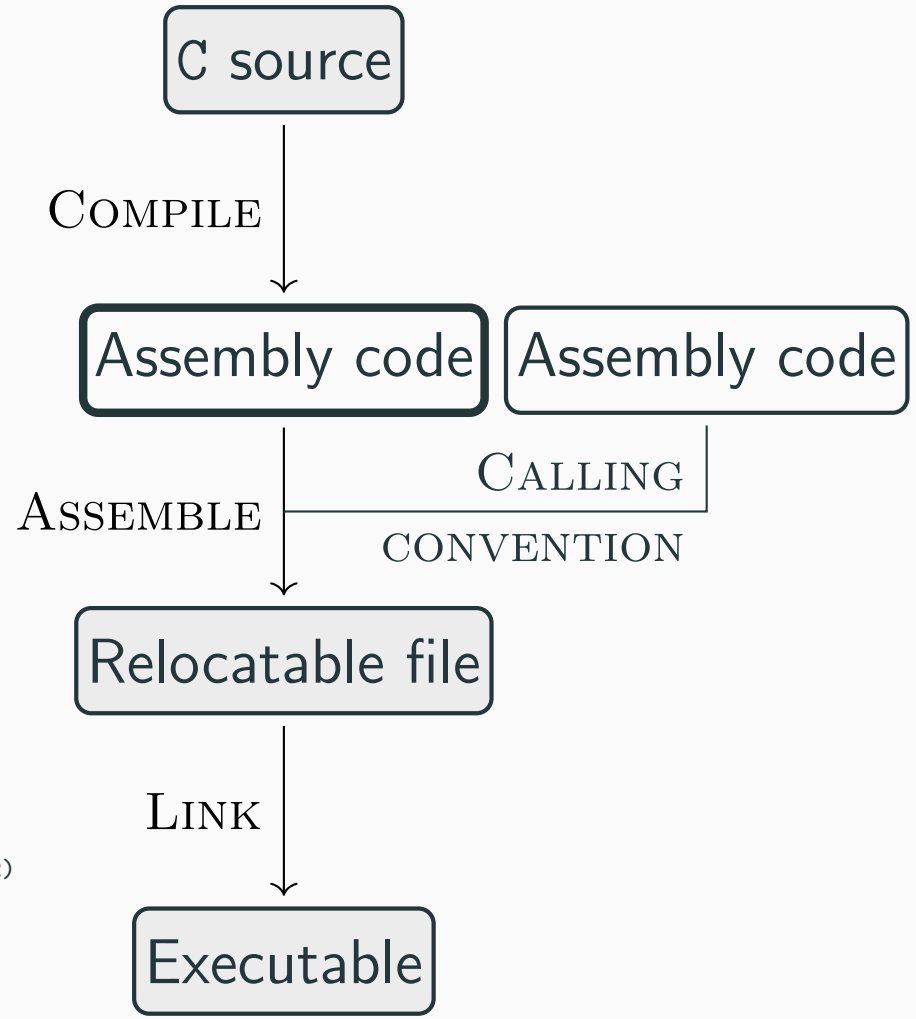
Inline assembly example (bits/strings.h@glibc_2.19)

```
1563    # ifdef __PIC__
1565    __STRING_INLINE size_t
1566    __strcspn_g (const char *__s, const char *__reject)
1567    {
1568      register unsigned long int __d0, __d1, __d2;
1569      register const char *__res;
1570      __asm__ __volatile__
1571        ("pushl        %%ebx\n\t"
1572         "movl         %4,%%edi\n\t"
1573         "cld\n\t"
1574         "repne; scasb\n\t"
1575         "notl         %%ecx\n\t"
1576         "leal         -1(%%ecx),%%ebx\n"
1577         "1:\n\t"
1578         "lodsb\n\t"
1579         "testb        %%al,%%al\n\t"
1580         "je         2f\n\t"
1581         "movl         %4,%%edi\n\t"
1582         "movl         %%ebx,%%ecx\n\t"
1583         "repne; scasb\n\t"
1584         "jne        1b\n"
1585         "2:\n\t"
1586         "popl         %%ebx"
1587         : "=S" (__res), "=&a" (__d0), "=&c" (__d1), "=&D" (__d2)
1588         : "r" (__reject), "0" (__s), "1" (0), "2" (0xffffffff)
1589         : "memory", "cc");
1590      return (__res - 1) - __s;
1591    }
1618    # endif
```



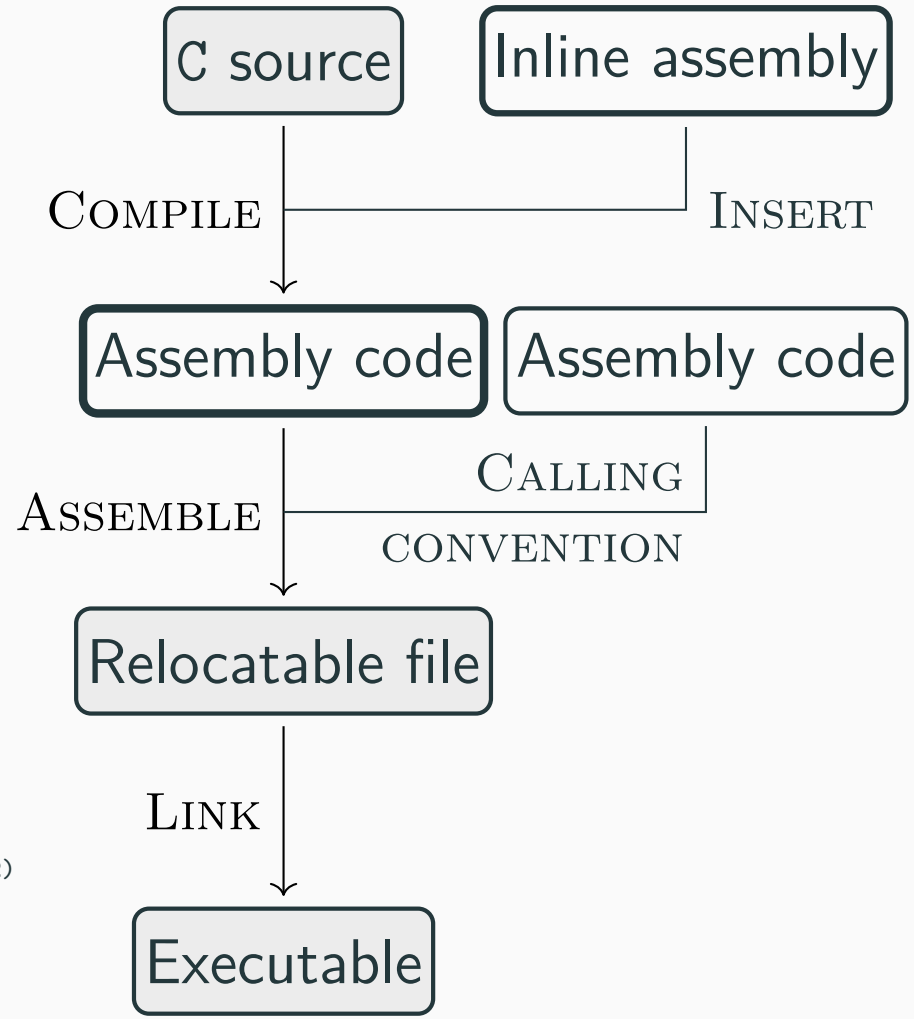
Inline assembly example (bits/strings.h@glibc_2.19)

```
1563  # ifdef __PIC__

1565  __STRING_INLINE size_t
1566  __strcspn_g (const char *__s, const char *__reject)
1567  {
1568    register unsigned long int __d0, __d1, __d2;
1569    register const char *__res;
1570    __asm__ __volatile__
1571      ("pushl        %%ebx\n\t"
1572       "movl         %4,%%edi\n\t"
1573       "cld\n\t"
1574       "repne; scasb\n\t"
1575       "notl         %%ecx\n\t"
1576       "leal         -1(%%ecx),%%ebx\n"
1577       "1:\n\t"
1578       "lodsb\n\t"
1579       "testb        %%al,%%al\n\t"
1580       "je         2f\n\t"
1581       "movl         %4,%%edi\n\t"
1582       "movl         %%ebx,%%ecx\n\t"
1583       "repne; scasb\n\t"
1584       "jne        1b\n"
1585       "2:\n\t"
1586       "popl         %%ebx"
1587       : "=S" (__res), "=&a" (__d0), "=&c" (__d1), "=&D" (__d2)
1588       : "r" (__reject), "0" (__s), "1" (0), "2" (0xffffffff)
1589       : "memory", "cc");
1590    return (__res - 1) - __s;
1591  }

1618  # endif
```

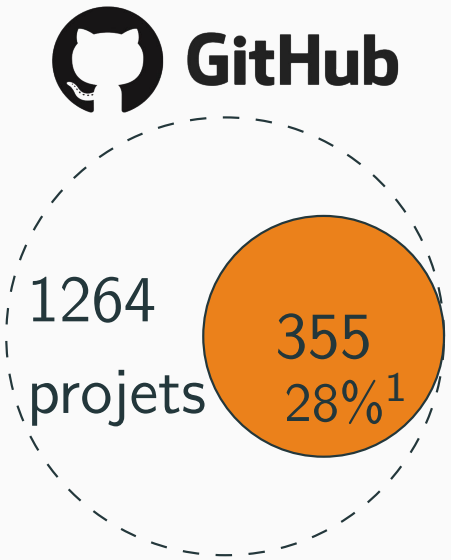


Inline assembly example (bits/strings.h@glibc_2.19)

```
1563    # ifdef __PIC__
1565    __STRING_INLINE size_t
1566    __strcspn_g (const char *__s, const char *__reject)
1567    {
1568      register unsigned long int __d0, __d1, __d2;
1569      register const char *__res;
1570      __asm__ __volatile__
1571        ("pushl        %%ebx\n\t"
1572         "movl         %4,%%edi\n\t"
1573         "cld\n\t"
1574         "repne; scasb\n\t"
1575         "notl         %%ecx\n\t"
1576         "leal         -1(%%ecx),%%ebx\n"
1577         "1:\n\t"
1578         "lodsb\n\t"
1579         "testb        %%al,%%al\n\t"
1580         "je         2f\n\t"
1581         "movl         %4,%%edi\n\t"
1582         "movl         %%ebx,%%ecx\n\t"
1583         "repne; scasb\n\t"
1584         "jne        1b\n"
1585         "2:\n\t"
1586         "popl         %%ebx"
1587         : "=S" (__res), "=&a" (__d0), "=&c" (__d1), "=&D" (__d2)
1588         : "r" (__reject), "0" (__s), "1" (0), "2" (0xffffffff)
1589         : "memory", "cc");
1590      return (__res - 1) - __s;
1591    }
1618    # endif
```



Inline assembly is well spread



FFmpeg ALSA

GMP libyuv

[1] according to Rigger et al.

Software verification is best-effort only

But formal methods work pretty well in practice

frama C Vcc KeY

CODESONAR® AbsInt

With **industrial** success stories in **regulated domains**

Still, adapting formal methods to common software is challenging

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

“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



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]

Outline

- A novel formalization
- The interface compliance challenge
- Verification-oriented lifting

Objective 1

Better understanding and

novel formalization

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;
}
```

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

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

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

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

%eax

Output list

Input list

%ecx

%edi

%edx

Clobber list

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”

Goals & challenges

Give a formal definition of inline assembly

there is not even a complete documentation...

Extract suitable intermediate representation

enable automatized reasoning

Widely applicable

must be as much architecture agnostic

arm

Contributions

An operational semantics of inline assembly

- intermediate semantics between binary level semantics (BINSEC) and C ANSI memory model (CompCert)
- formally define GNU-syntax components (pattern, tokens, inputs, outputs, etc.)

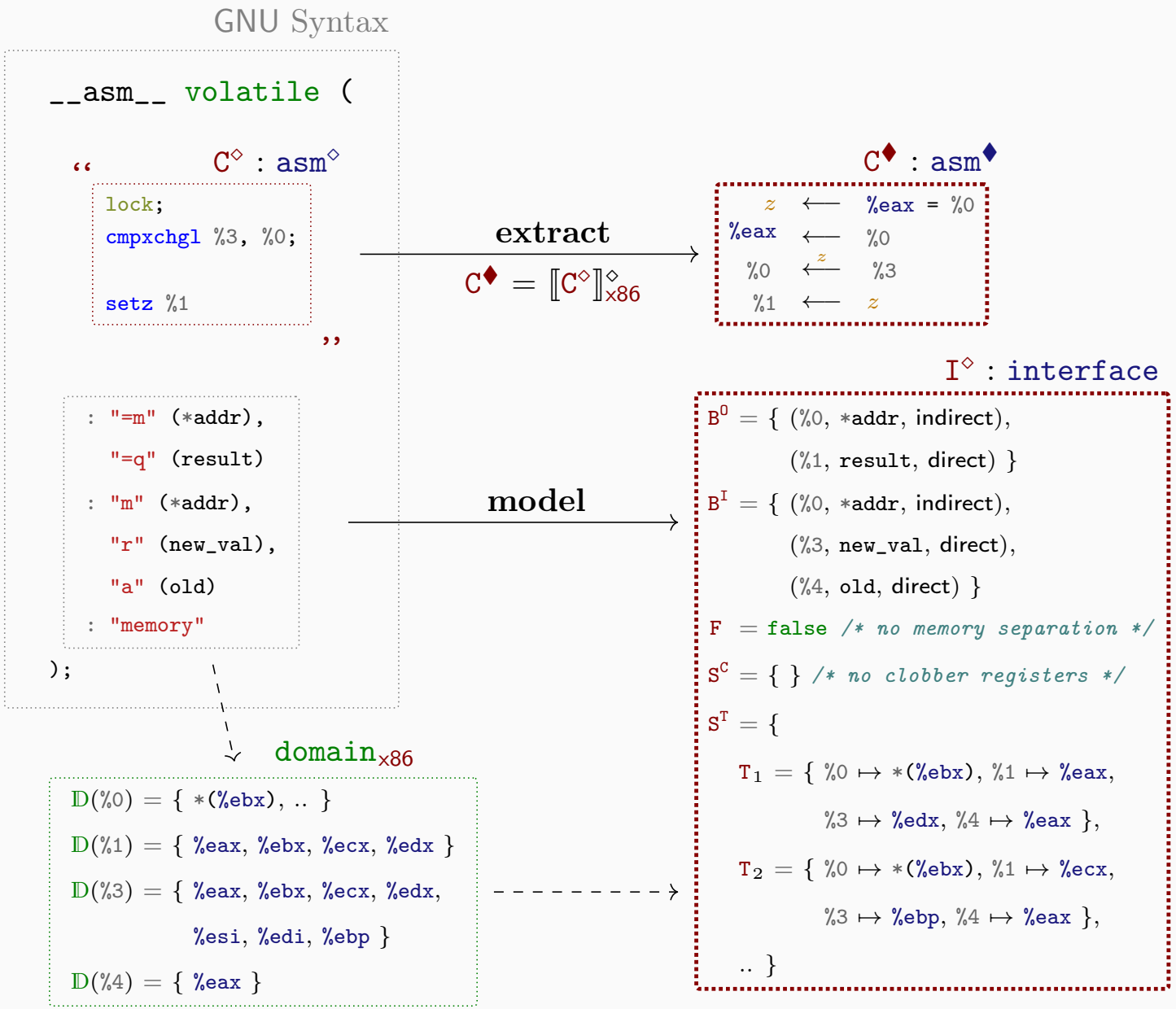
TInA-core, a method to extract inline assembly IR

- a combination of existing (Frama-C, gas, BINSEC) components
- and novel ones (constraint solver, token identifier)

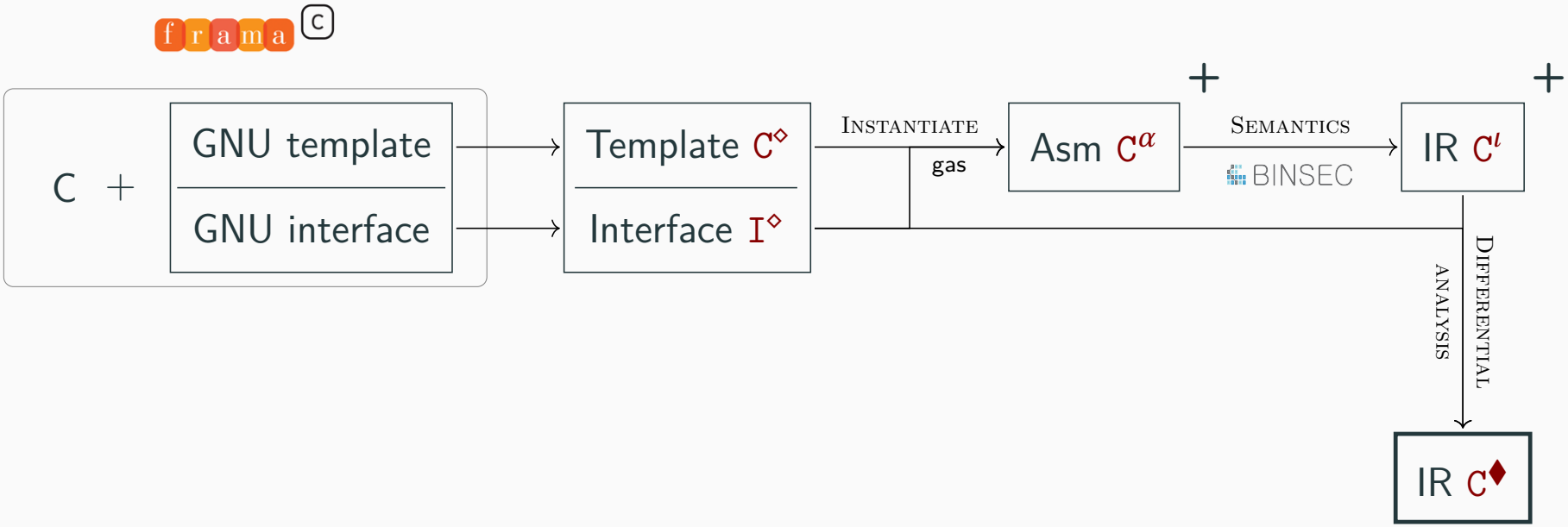
Thorough experiments of our prototype

- **2.6k**$^+$ of **3k** real-world x86 assembly chunks (**Debian**)
- 392 of 394 real-world ARM assembly chunks (ALSA, ffmpeg, GMP)

Looking for the missing formalism



Our prototype TInA-core



Experimental evaluation of TInA-core

- How many chunks can TInA-core extract the semantics from?
- What are the characteristics of the supported chunks?
- Does TInA-core work on different architectures?

Widely Applicable : Debian 8.11 – x86-32bit

	Total		ALSA		ffmpeg		GMP		libyuv	
All chunks	**3107**		25		103		237		4	
Supported chunks	**2656**	85%	25	100%	91	88%	237	100%	1	25%
Average (Max) size	3	(104)	69	(104)	12	(68)	1	(1)	40	(40)
System instructions	373	12%	0	0%	4	4%	0	0%	3	75%
Average (Max) size	4	(151)		–	10	(21)		–	6	(12)
Floating-point	40	1%	0	0%	5	5%	0	0%	0	0%
Average (Max) size	33	(506)		–	19	(38)		–		–

Widely Applicable : Key projects – ARM**v7**

	Total		ALSA	ffmpeg		GMP		libyuv	
All chunks	**394**		0	85		308		1	
Supported chunks	**392**	**99%**	–	83	**98%**	308	**100%**	1	100%
Average (Max) size	1	(27)	–	1	(15)	1	(1)	27	(27)
System instructions	2	1%	–	2	2%		–		–
Average (Max) size	4	(-)	–	4	(6)		–		–

Objective 1 – Conclusion

- ☑ Operational semantics enables formal reasoning
- ☑ Inline assembly semantics extraction is the keystone for **wide applicability**

  (**85**% of x86 `Debian` chunks, works for ARM too)

It opens the door to advanced
verification and transformation techniques

Objective 2

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 


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

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)

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

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

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

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

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

The instruction behavior must not depend on the compiler choices

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

Checking the compliance

Dedicated dataflow analysis

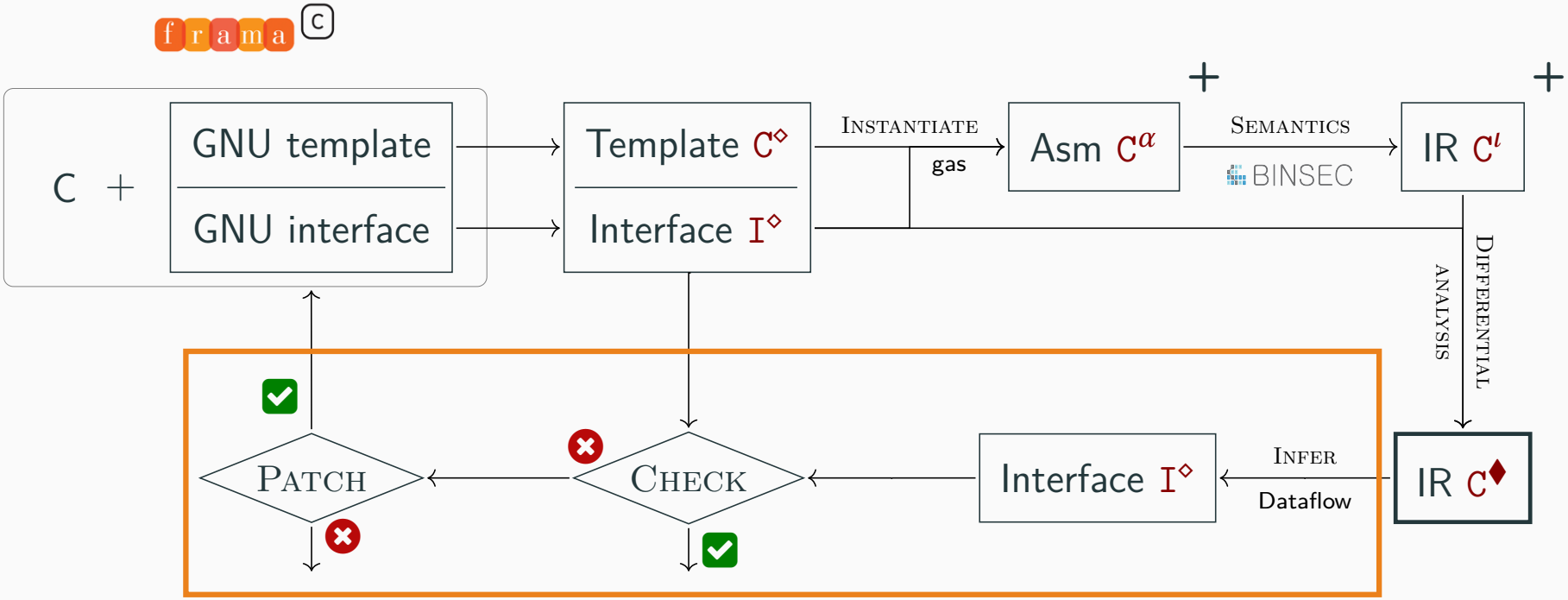
Frame-write. Collect all the left hand side expressions.

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



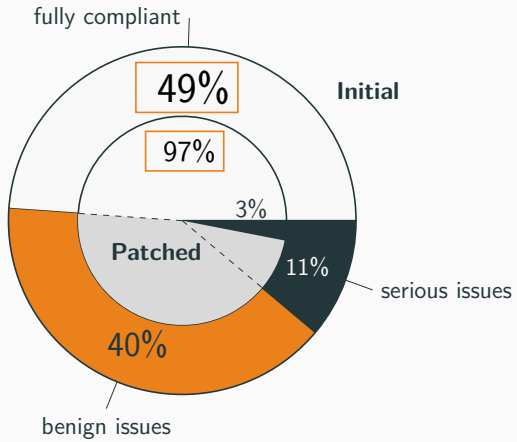
Experimental evaluation of RUSTInA

- 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?

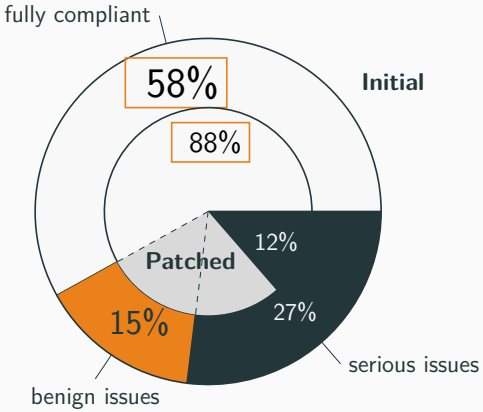
Checking and patching statistics

	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 ≥ 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

Objective 2 – Conclusion

- ☑ Interface compliance definition
  (**Framing** condition & **Unicity** condition)
- ☑ Dedicated dataflow analysis to **check**, **patch** and **refine**
- ☑ **Real impact** on the Debian code base

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

Objective 3

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

```
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

Common workarounds

```
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

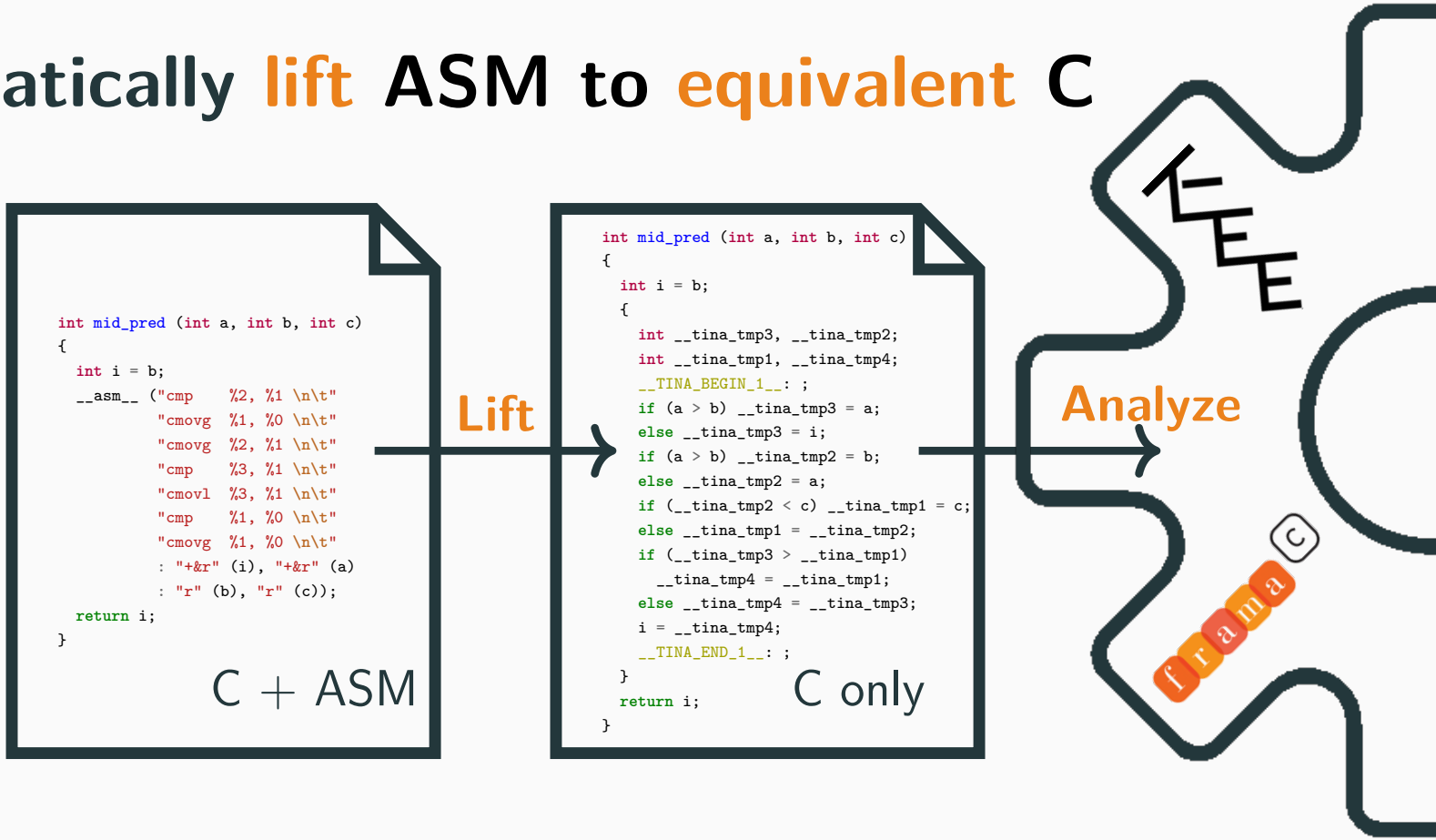
~~**Dedicated analyzer**~~

substantial engineering effort

Want to reuse existing analyses!

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


EVA


WP etc.

Contributions

Dedicated high-level structure recovery mechanism

- identify 3 main threats to verifiability
- dedicated rexriting steps

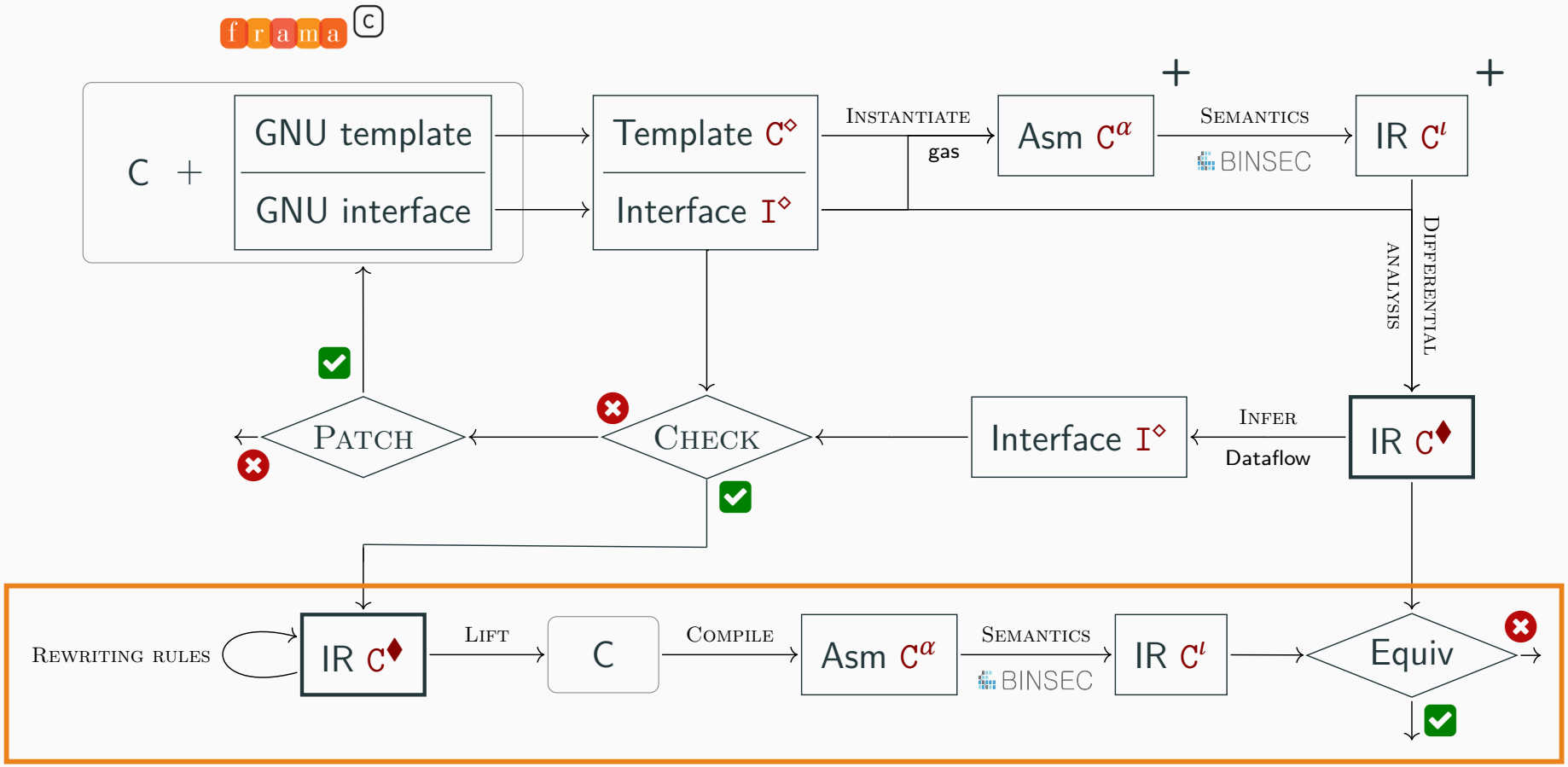
Tailored validation pass

- 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)

Our prototype TInA



Verification-oriented lifting

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

```
__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__;
```

Verification-oriented lifting

```
__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

```
__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__;
```

Verification-oriented lifting

```
__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

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

- type consistency
- high-level predicate
- unpacking
- structuring
- expression propagation
- loop normalization

Lifting : running example

```
__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

```
__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__;
```

Lifting : high-level predicate (Djoudi et al.)

```
__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

```
__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__;
```

Lifting : high-level predicate (Djoudi et al.)

```
__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

```
__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 ((int)__ebx__ > (int)__eax__)
  goto l1;
else goto l2;
l1: __tmp__ = __ebx__; goto l3;
l2: __tmp__ = __eax__; goto l3;
l3: __eax__ = __tmp__;
i = (int)__eax__;
```

Lifting : slicing

```
__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

```
__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 ((int)__ebx__ > (int)__eax__)
  goto l1;
else goto l2;
l1: __tmp__ = __ebx__; goto l3;
l2: __tmp__ = __eax__; goto l3;
l3: __eax__ = __tmp__;
i = (int)__eax__;
```

Lifting : slicing

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

```
__eax__ = (unsigned int)i;
__ebx__ = (unsigned int)a;
if ((int)__ebx__ > (int)__eax__)
  goto l1;
else goto l2;
l1: __tmp__ = __ebx__; goto l3;
l2: __tmp__ = __eax__; goto l3;
l3: __eax__ = __tmp__;
i = (int)__eax__;
```

T1. low-level data & computation

T2. low-level packing & representation

T3. unusual & unstructured control flow

Lifting : structuring

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

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

T1. low-level data & computation

T2. low-level packing & representation

T3. unusual & unstructured control flow

Lifting : typing

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

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

T1. low-level data & computation

T2. low-level packing & representation

T3. unusual & unstructured control flow

Lifting : expression propagation

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

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

T1. low-level data & computation

T2. low-level packing & representation

T3. unusual & unstructured control flow

Lifting : expression propagation

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

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

T1. low-level data & computation

T2. low-level packing & representation

T3. unusual & unstructured control flow

Lifting : expression propagation

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

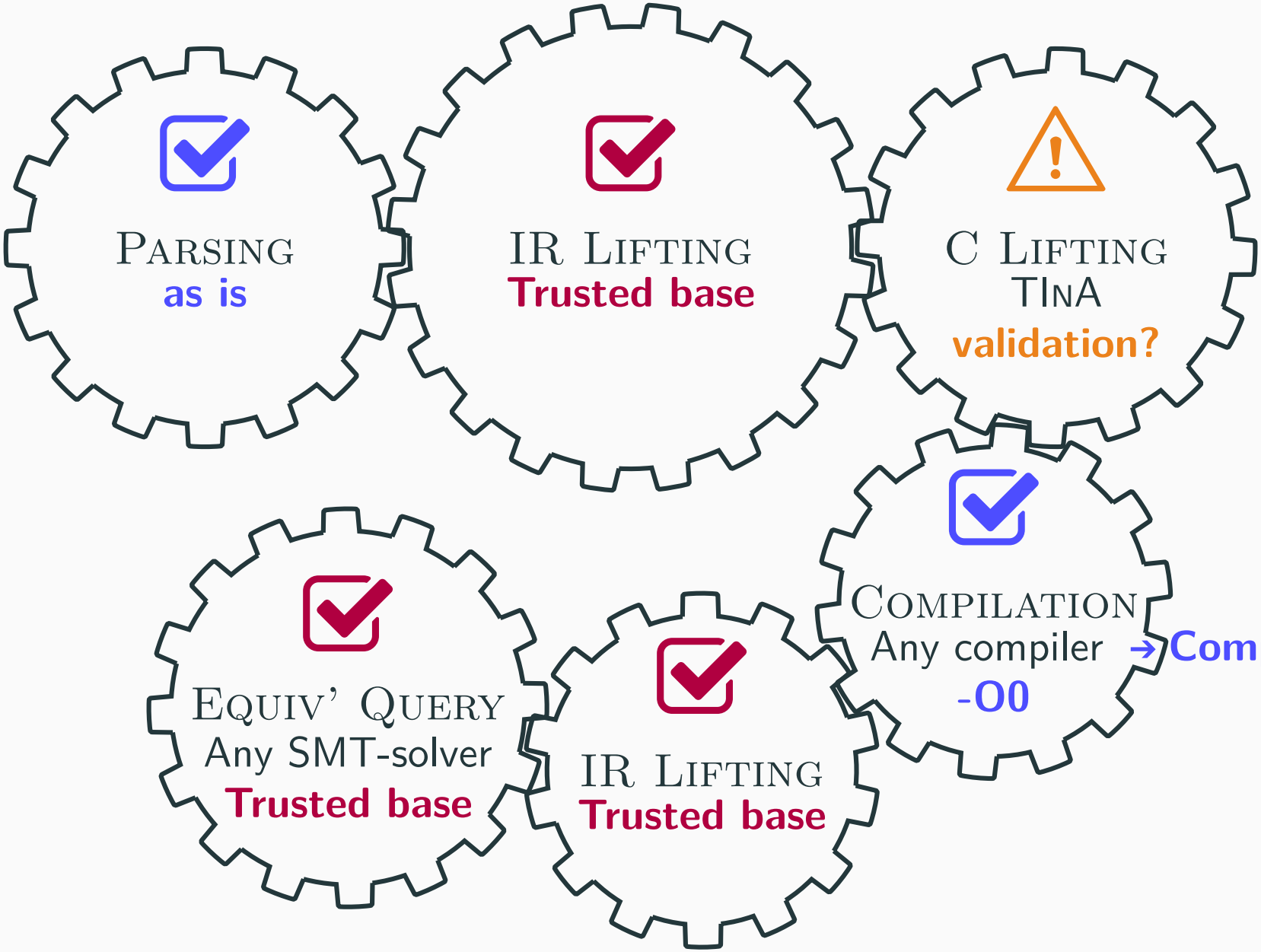
```
int __eax__ = i;
int __ebx__ = a;
int __tmp__;
if (a > i)
  __tmp__ = a;
else
  __tmp__ = i;
__eax__ = __tmp__;
i = __tmp__;
```

T1. low-level data & computation

T2. low-level packing & representation

T3. unusual & unstructured control flow

Validation trust & Trusted base



Experimental evaluation of TInA

- How many chunks can TInA lift to C?
- How many lifted chunks are automatically validated?
- How do off-the-shelf program analyzers behave on lifted code?
- What is the impact of each optimization?

Lifting & validation performance

	x86		ARM	
All chunks	**3107**		394	
Relevant	2568	82%	391	99%
Lifted	**2568**	100%	**391**	100%
Validated	**2568**	100%	**391**	100%
Translation time	155s		5s	
Validation time	1372s		48s	
Average time per chunk	600ms		135ms	

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

Objective 3 – Conclusion

- ☑ Verification-oriented lifting from inline assembly to C
- ☑ Tailored post-validation pass
  (**100**% success rate on benchmark)

TInA is a trustworthy,
verification-oriented lifting technique
enabling and enhancing existing verification tools

Conclusion

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]

Perspective and future work

Improve TInA

- add new architectures (x86-64bit and ARMv8 are coming)
- add support for floating-point and system instructions
- diversify the front- and back-end (Clang, llvm, etc.)

Toward certified decompilation

- small assembly functions may be good target too
- enable software verification of project linking with third party pre-compiled library

Design new (and safer) inline assembly syntax

- languages still add inline assembly feature (e.g. Rust)
- more meaningful and user-friendly syntax may improve reliability and efficiency

Thank you
for your attention

Conclusion

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]