Get rid of inline assembly

through verification-oriented lifting

ASE'19

Frédéric Recoules Univ. Paris-Saclay, CEA, List

Sébastien Bardin Univ. Paris-Saclay, CEA, List
Richard Bonichon Univ. Paris-Saclay, CEA, List
Laurent Mounier Univ. Grenoble Alpes, VERIMAG
Marie-Laure Potet Univ. Grenoble Alpes, VERIMAG

2nd International **KLEE** Workshop on Symbolic Execution

A grand challenge

Many barriers to formal methods adoption:

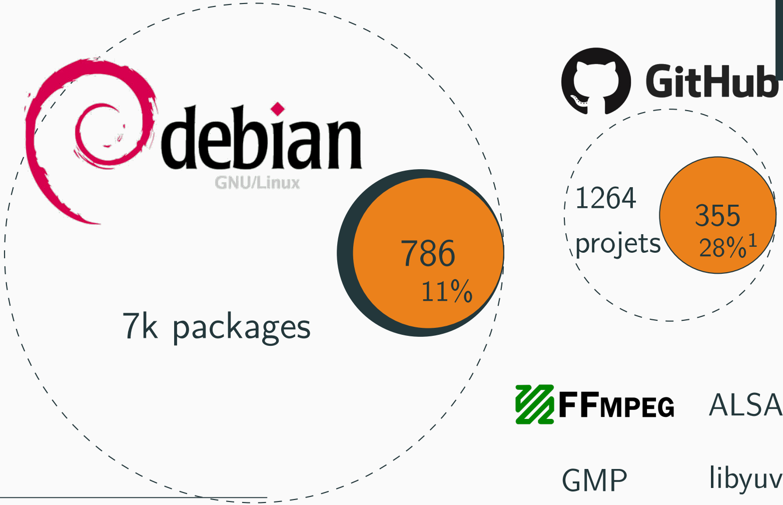
- learnability
- scalability
- ...

- automatization
- feature set
  - mixed-language support
  - ...

Today's challenge :
mixed C & **inline assembly** code

with **reuse** of existing tools

Inline assembly is well spread



debian GNU/Linux

7k packages

786

11%

GitHub

1264 projets

355

28%[1]

FFMPEG

ALSA

GMP

libyuv

[1] according to Rigger et al.

Inline assembly is a pain

[redacted]

```
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
```

```
done for function main
====== VALUES COMPUTED ======
Values at end of function mid_pred:
  i ∈ [--..--]     i ∈ [-5..5]
Values at end of function main:
  a ∈ {0; 1; 2; 3; 4; 5}
  b ∈ [-5..10]
  c ∈ [-10..0]
  i ∈ [--..--]     i ∈ [-5..5]
```

Incomplete

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

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

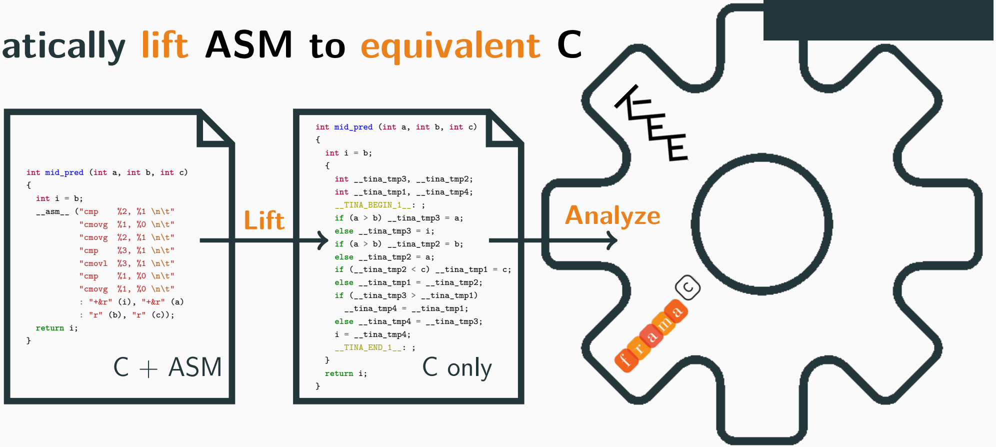
~~Dedicated analyzer~~

substantial engineering effort

Want to reuse existing analyses!

Our proposition

Automatically lift ASM to equivalent C



```
int mid_pred (int a, int b, int c)
{
  int i = b;
  __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));
  return i;
}
```

C + ASM

```
int mid_pred (int a, int b, int c)
{
  int i = b;
  {
    int __tina_tmp3, __tina_tmp2;
    int __tina_tmp1, __tina_tmp4;
    __TINA_BEGIN_1__: ;
    if (a > b) __tina_tmp3 = a;
    else __tina_tmp3 = i;
    if (a > b) __tina_tmp2 = b;
    else __tina_tmp2 = a;
    if (__tina_tmp2 < c) __tina_tmp1 = c;
    else __tina_tmp1 = __tina_tmp2;
    if (__tina_tmp3 > __tina_tmp1)
      __tina_tmp4 = __tina_tmp1;
    else __tina_tmp4 = __tina_tmp3;
    i = __tina_tmp4;
    __TINA_END_1__: ;
  }
  return i;
}
```

C only

Reuse C tools

Challenges

Widely applicable

architecture – assembly dialect – compiler agnostic

Verification friendly

decent enough analysis outputs

Trustable

usable in sound formal method context

Challenges & key enablers

Widely applicable

architecture – assembly dialect – compiler agnostic

leverage existing binary-to-IR lifters – x86/ARM, GCC/clang

Verification friendly

decent enough analysis outputs

novel high-level simplifications – improve KLEE & Frama-C

Trustable

usable in sound formal method context

novel dedicated equivalence checking – 100% in scope success

Challenges & key enablers

Widely applicable

architecture – assembly dialect – compiler agnostic

leverage existing binary-to-IR lifters – x86/ARM, GCC/clang

Verification friendly

decent enough analysis outputs

novel high-level simplifications – improve KLEE & Frama-C

Trustable

usable in sound formal method context

novel dedicated equivalence checking – 100% in scope success

Evaluated overs **2000**$^{+}$ assembly chunks from Debian *Jessie*

Panorama of existing works

[redacted]

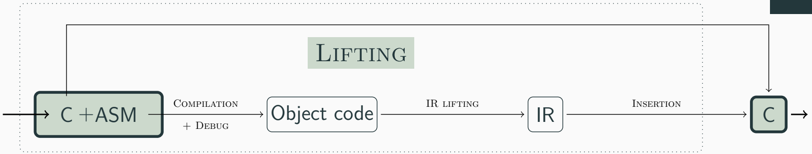
		Manual	Goanna[1]	Vx86[2]	Inception[3]	TINA
	Semantic lifting	✓	✗	✓	✓	✓
	Widely applicable	✗	✗	✗	✓	✓
Trust	Sanity check	✓	✓	✗	✗	✓
	Validation	✗	✗	✗	✓	✓
	Verifiability	✓	✗	✓	✓	✓

[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

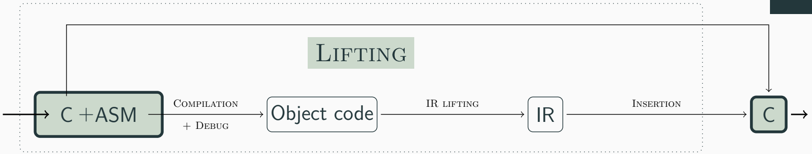
Lifting: the basic case



```
__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__;
```

Lifting: verification threats



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

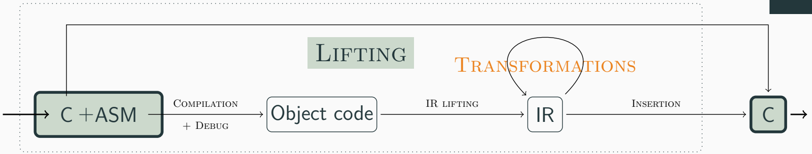
T1. low-level data & computation

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 simplifications



Lifting

Transformations

C +ASM

Compilation + Debug

Object code

IR lifting

IR

Insertion

C

```
__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__;
```

- types 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. FM'16)

```
__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. FM'16)

```
__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 */
  );
```

T1. low-level data & computation

T2. low-level packing & representation

T3. unusual & unstructured control flow

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

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

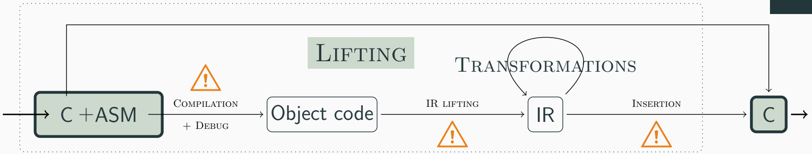
Lifting : result

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

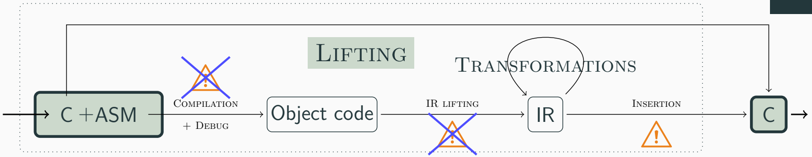
```
int __tmp__;
if (a > i)
  __tmp__ = a;
else
  __tmp__ = i;
i = __tmp__;
```

High-level C code **amenable** to verification

Trust: what are the threats?

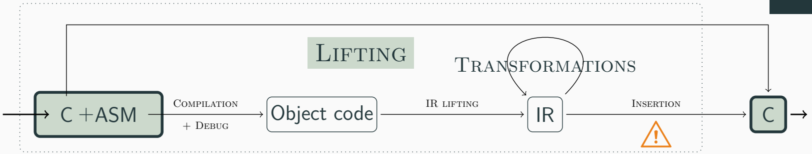


Trust: what are the threats?



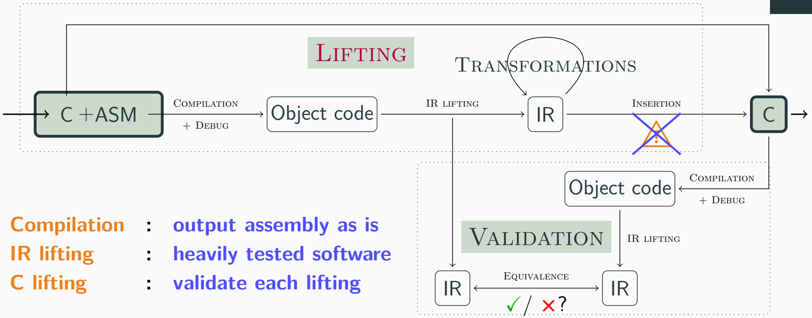
Compilation	:	output assembly as is
IR lifting	:	heavily tested software

Trust: what are the threats?



Compilation	:	output assembly as is
IR lifting	:	heavily tested software
C lifting	:	validation?

Trust: what are the threats?

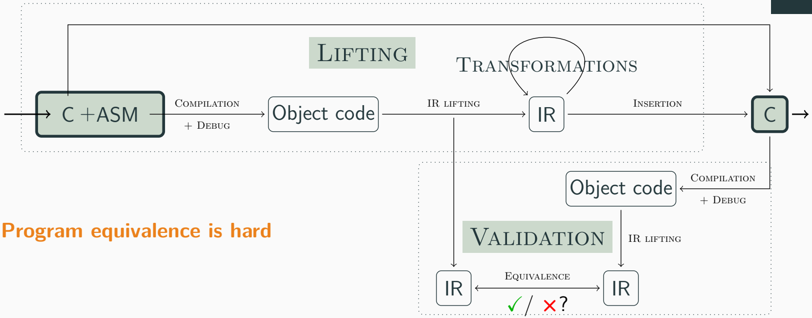


Compilation : output assembly as is

IR lifting : heavily tested software

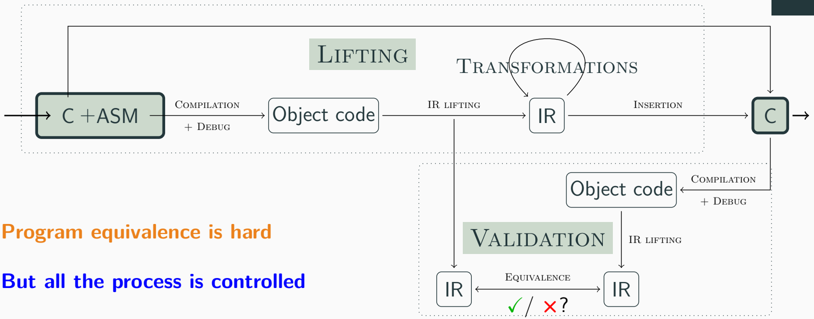
C lifting : validate each lifting

Translation validation: semantics equivalence



Program equivalence is hard

Translation validation: semantics equivalence



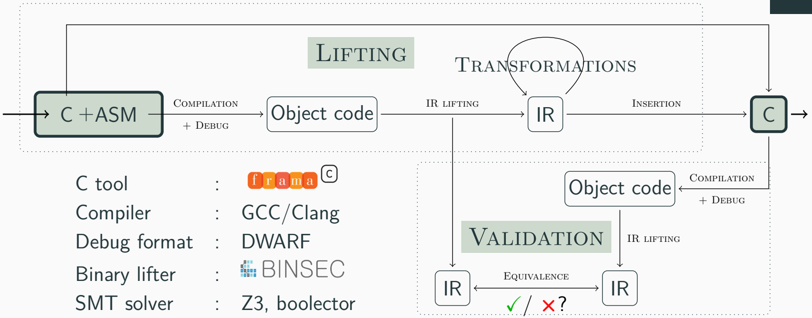
Program equivalence is hard

But all the process is controlled

Step 1: control flow graph isomorphism
labeled directed graph + debug information

Step 2: pairwise basic block equivalence check
SMT-based check

TInA: prototype



C tool	:	Frama-C
Compiler	:	GCC/Clang
Debug format	:	DWARF
Binary lifter	:	BINSEC
SMT solver	:	Z3, boolector

https://binsec.github.io/

Experimental evaluation

- **Applicability & Trust**

  Debian, x86/ARM

- **Verification friendly**

  KLEE, Frama-C EVA, Frama-C WP

Widely Applicable : Debian *Jessie* x86

	Total		ALSA		ffmpeg		GMP		libyuv	
All chunks	**3107**		25		103		237		4	
Relevant chunks	**2656**	85%	25	100%	91	88%	237	100%	1	25%
Average size (Max)	3	(104)	69	(104)	12	(68)	1	(1)	40	(40)
Average # BB (Max)	1	(21)	12	(21)	2	(8)	1	(5)	3	(3)
Lifted chunks	**2656**	100%	25	100%	91	100%	237	100%	1	100%
Total translation time (average)	121s	(5ms)	2s	(8ms)	63s	(692ms)	2s	(1ms)	$<$ 1s	(10ms)
Validated lifting	**2656**	100%	25	100%	91	100%	237	100%	1	100%
Total validation time (average)	30min	(600ms)	17s	(680ms)	255s	(2.8s)	110s	(500ms)	$<$ 1s	(800ms)
Unsupported *OS*	373	12%	0	0%	4	4%	0	0%	3	75%
Unsupported `float`	40	1%	0	0%	5	5%	0	0%	0	0%
Unsupported *others*	38	1%	0	0%	3	3%	0	0%	0	0%

Verifiability: KLEE (symbolic execution)

	Lifting		
	None	Basic	TInA
# functions with 100% branch coverage[1]	×	**25** / 58	**25** / 58
Aggregate time for functions with 100% branch coverage[1]	×	121s	106s
# explored paths for all functions	1 336k	1 459k	**6 402k**

58 functions from ALSA, ffmpeg, GMP & libyuv

[1] 10min timeout

Open issues

Engineering

- floating point operations
- builtin crypto-operations

challenge for SMT & analyzers

Genericity

- syscall
- hardware dependent

very analyzer specific

Inline assembly hinders C verification tools

TInA lifts the assembly chunks in order to ease the verification

TInA is architecture agnostic (x86, ARM) and benefits different kinds of verification techniques

- Have a look @ the papers
- Have a look @ the artifacts
- Have a look @ BINSEC

– ASE'19 –

TInA: generic, verification-friendly & trustworthy lifting for inline assembly

– ICSE'21 –

ACM SIGSOFT Distinguished Paper Award

Automatically check & patch the Interface compliance with RUSTInA

If you have any question, do not hesitate!

frederic.recoules@cea.fr

`https://binsec.github.io/`