Get rid of inline assembly

through verification-oriented lifting

Frédéric Recoules (CEA LIST, France)

Sébastien Bardin (CEA LIST)
Richard Bonichon (CEA LIST)
Laurent Mounier (VERIMAG)
Marie-Laure Potet (VERIMAG)

November 13, 2019

Software is not always reliable

1996	2009	2016
European Space Agency Agence spatiale européenne	TOYOTA	GM
500M$	400	

Then came formal methods

frama C   Vcc   KeY

CODESONAR®   AbsInt

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



A grand challenge

Many barriers to formal methods adoption:

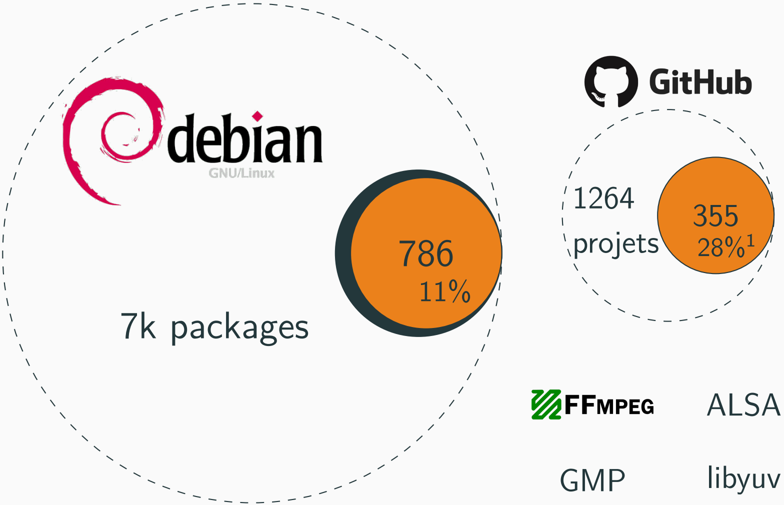
- learnability
- scalability
- ...
- automatization
- feature set
  - mixed-language support
  - ...

Today's challenge :

mixed C & inline assembly code

with reuse of existing tools

Inline assembly is well spread



[1] according to Rigger et al.

Inline assembly is a pain

```
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

```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

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

Lift

```
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

Analyze

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 packages

Panorama of existing works

		Manual	Goanna[1]	Vx86[2]	Inception[3]	TInA
	Semantic lifting	✓	✗	✓	✓	✓
	Widely applicable	✗	✗	✗	✓	✓
Trust	**Sanity check**	✓	✓	✗	✗	✓
Trust	**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 : 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 */
  );
```

```
__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__;
```

T1. low-level data & computation

T2. low-level packing & representation

T3. unusual & unstructured control flow

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

Lifting: high level simplifications


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

T1. low-level data & computation

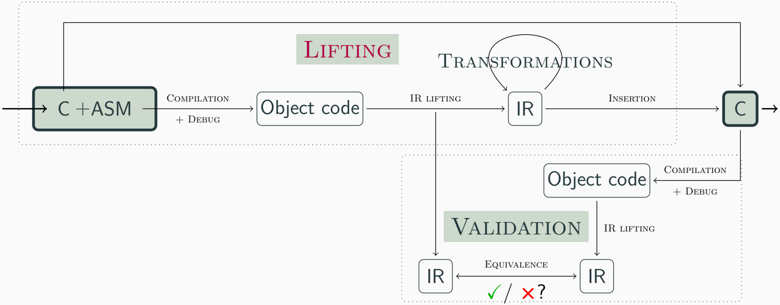
T2. low-level packing & representation

T3. unusual & unstructured control flow

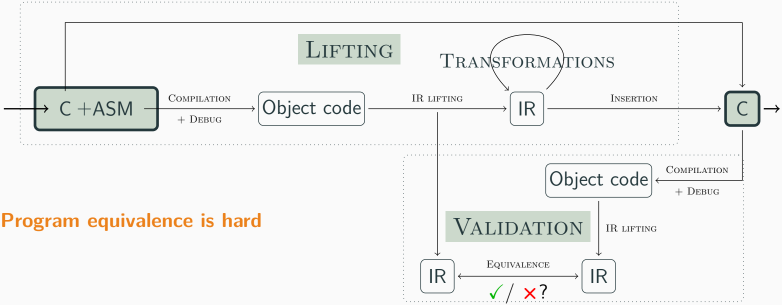
```
int __tmp__;
if (a > i)
  __tmp__ = a;
else
  __tmp__ = i;
i = __tmp__;
```

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

Validation: semantics equivalence

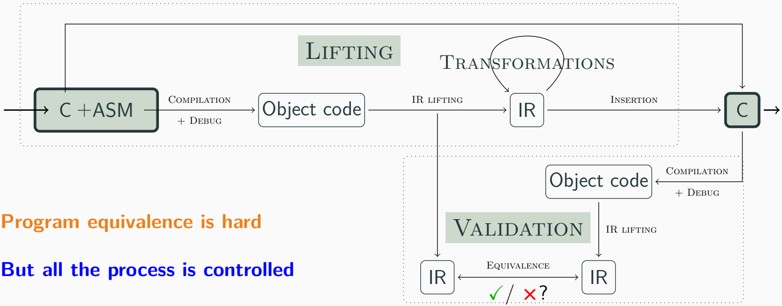


Validation: tailored algorithm



Program equivalence is hard

Validation: tailored algorithm



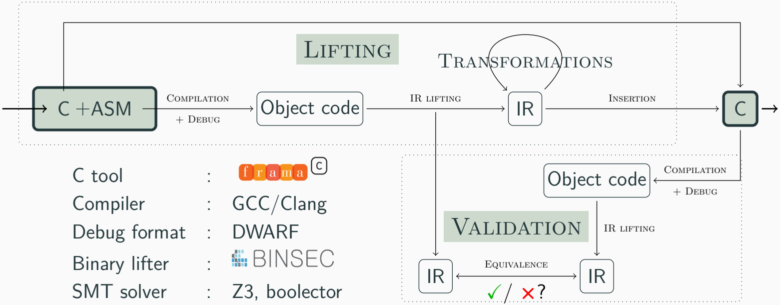
Program equivalence is hard

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

Experimental evaluation

- **Applicability & Trust**

  Debian, x86/ARM, GCC/clang

- **Verification friendly**

  KLEE and Frama-C

Widely Applicable : Debian 8.11 x86-32bit

	GCC v5.4		GCC v4.7		CLANG 3.8	
All chunks	**3039**		2955		2955	
Trivial	126		126		106	
Out-of-scope	449		366		404	
Rejected	138		137		412	
Relevant	2326	**76%**	2326	78%	2033	69%
Lifted	**2326**	**100%**	**2326**	**100%**	**2033**	**100%**
Validated	**2326**	**100%**	**2326**	**100%**	**2033**	**100%**
Average size	8		8		8	
Maximum size	**341**		341		341	
Translation time	121s		105s		89s	
Validation time	1527s		1528s		1336s	

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]	N/A	121s	106s
# explored paths for all functions	1 336k	1 459k	**6 402k**

58 functions from ALSA, ffmpeg, GMP & libyuv

[1] 10min timeout

Verifiability: Frama-C EVA (abstract interpretation)

	TInA	
Functions with returns (non void)	20	
Better **return precision**	**11**	**55%**
Functions with initial C alarm	27	
Alarm reduction in C	**23**	**82%**
New memory alarms ASM	17	26%
Positive impact	**45**	**77%**

58 functions from ALSA, ffmpeg, GMP & libyuv

Verifiability: Frama-C WP (deductive verification)

Function	# Instr	Lifting		
		None	Basic	TInA
saturated_sub	2	✗	✓	✓
saturated_add	2	✗	✗	✓
log2	1	✗	✗	✓
mid_pred	7	✗	✗	✓
strcmpeq	9	✗	✗	✓
strnlen	16	✗	✗	✓
memset	9	✗	✗	✓
count	8	✗	✗	✓
max_element	10	✗	✗	✓
cmp_array	10	✗	✗	✓
sum_array	20	✗	✗	✓
SumSquareError_SSE2	24	✗	✗	✓

Limits

Engineering

- floating point operations
- builtin crypto-operations

would challenge SMT & analyzers too

Genericity

- syscall
- hardware dependent

each analyzer has its own way to handle it

Conclusion

Inline ASM hinders the adoption of formal methods

TInA: Automated lifting

- **Widely applicable**
- **Verification-friendly**
- **Trustable**

Successful experimental evaluation over:

- **2000**$^{+}$ x86 **Debian chuncks** – ARM *experiments too*
- KLEE & Frama-C friendly – *principled approach*

Conclusion

Inline ASM hinders the adoption of formal methods

TInA: Automated lifting

- **Widely applicable**
- **Verification-friendly**
- **Trustable**

Post-analysis considerations:

- **567** compliance issues
- ffmpeg coding flaws

Successful experimental evaluation over:

- **2000**$^{+}$ x86 **Debian chuncks** – ARM *experiments too*
- KLEE & Frama-C friendly – *principled approach*

- Have a look @ the paper
- Meet us @ the conference

frederic.recoules@cea.fr

up to Thursday night

sebastien.bardin@cea.fr

up to Thursday noon

Any questions?