Get rid of inline assembly through verification-oriented lifting

Frédéric Recoules*, Sébastien Bardin*, Richard Bonichon*, Laurent Mounier† and Marie-Laure Potet†
*CEA LIST, Paris-Saclay, France
firstname.lastname@cea.fr
†Univ. Grenoble Alpes. VERIMAG, Grenoble, France
firstname.lastname@univ-grenoble-alpes.fr

***Abstract*—Formal methods for software development have made great strides in the last two decades, to the point that their application in safety-critical embedded software is an undeniable success. Their extension to non-critical software is one of the notable forthcoming challenges. For example, C programmers regularly use inline assembly for low-level optimizations and system primitives. This usually results in rendering state-of-the-art formal analyzers developed for C ineffective. We thus propose TINA, the first automated, generic, *verification-friendly* and trustworthy lifting technique turning inline assembly into semantically equivalent C code amenable to verification, in order to take advantage of existing C analyzers. Extensive experiments on real-world code (including GMP and ffmpeg) show the feasibility and benefits of TINA.**

***Index Terms*—Inline assembly, software verification, lifting, formal methods.**

I. Introduction

Context. Formal methods for the development of high-safety software have made tremendous progress over the last two decades [1], [2], [3], [4], [5], [6], with notable success in regulated safety-critical industrial areas such as avionics, railway or energy. Yet, the application of formal methods to more usual (non-regulated) software, for safety or security, currently remains a scientific challenge. In particular, extending the applicability from a world with strict coding guidelines and disciplined mandatory validation processes to more liberal and diverse development and coding practices is a difficult task.

Problem. We consider here the issue of analyzing "mixed code", focusing on the use of inline assembly in C/C++ code. This feature allows to embed assembly instructions in C/C++ programs. It is supported by major C/C++ compilers like GCC, clang or Visual Studio, and used quite regularly — usually for optimization or to access system-level features hidden by the host language. For example, we estimate that **11**% of Debian packages written in C/C++ directly or indirectly depends on inline assembly, with chunks containing up to 500 instructions, while **28**% of the top rated C projects on GitHub contains inline assembly according to Rigger et al. [7]. As a matter of fact, *inline assembly is a common engineering practice in key areas such as cryptography, multimedia or drivers.* However, *it is not supported by current state-of-the-art C/C++ program analyzers*, like KLEE [4] or Frama-C [1], possibly leading to incorrect or incomplete results. *This is a clear applicability issue for advanced code analysis techniques.*

Given that developing dedicated analyzers from scratch is too costly, the usual way of dealing with assembly chunks is to write either equivalent host code (e.g, C/C++) or equivalent logical specification when available. But *this task is handled manually* in both cases, precluding regular analysis of large code bases: manual translation is indeed time-consuming and error-prone. The bigger the assembly chunks are, the bigger these problems loom.

Goal and challenges. *We address the challenge of designing and developing an automated and generic lifting technique turning inline assembly into semantically equivalent C code amenable to verification.* The method should be:

Verification-friendly The produced code should allow *good enough* analyses in practice (informally dubbed *verifiability*), independently of the underlying analysis techniques (e.g., symbolic execution [8], [9], deductive verification [10], [11] or abstract interpretation [12]);

Widely applicable It should not be tied to a particular architecture, assembly dialect or compiler chain, and yet handle a significant subset of assembly chunks found in the wild;

Trustworthy The translation process should be insertable in a formal verification context without endangering soundness: as such it should maintain exactly all behaviors of the mixed code, and provide a way to show this property.

Verifiability alone is already challenging: indeed, straightforward lifting from assembly to C (keeping the untyped byte-level view) does not ensure it as standard C analyzers are not well equipped to deal with such low-level C code.

Scarce previous attempts do not fulfill all the objectives above. Vx86 [13] is tied to both the x86 architecture and deductive verification, while the recent work by Corteggiani et al. [14] focuses on symbolic execution. None of them addresses verifiability or trust. At first sight, decompilation techniques [15], [16], [17] may seem to fit the bill. Yet, as they mostly aim at helping reverse engineers, correctness is not their main concern. Actually, *"existing decompilers frequently produce decompilation that fails to achieve full functional equivalence with the original program"* [18]. Some recent works partially target this issue: Schwartz et al. [19] do not *demonstrate* correctness (they instead measure a certain degree of it via testing), while Schulte et al. [18] use a correct-by-design but intractable (possibly non-terminating) search-based method. Again, none of them study verifiability.

Proposal. We propose TINA (Taming Inline Assembly), the first automated, generic, verification-friendly and trustworthy lifting technique for inline assembly. The main insight behind TINA is that by *focusing on inline assembly rather than arbitrary decompilation, we tackle a problem both more restricted (simple control-flow, smaller size) and better defined (interfaces with C code, no dynamic jumps), paving the way to powerful targeted methods*. TINA relies on the following key principles:

- Recent binary-code lifters [20], [21], [22] translating binary opcodes to *generic low-level intermediate representations* (IR) provide minimalist architecture-agnostic and well-tested IRs adapted to our goal;
- While direct byte-level lifting severely hinders current C analyzers, *verifiability is enhanced by dedicated transformations* refining the raw original IR with C-like abstractions such as explicit variables, arithmetic data manipulation, structured control-flow, etc.;
- Trust relies on *translation validation* [23] (validating each translation), a more tractable option than full translator validation, which reduces the trust base to a (usually simpler) *checker*. Here, this checker requires to prove program equivalence – a notoriously hard problem[1]. We propose a *dedicated equivalence checking algorithm* tailored to our processing chain.

Contributions. In summary, this paper makes the following contributions:

- A new cooperating toolchain allowing formal verification of programs mixing inline assembly and C, based on an original combination of novel and existing components (Sec. IV), addressing verifiability and trust issues;
- A new principled method lifting inline assembly to high-level C amenable to further formal analysis built upon 4 simplification steps (Sec. V) countering clearly identified threats to verifiability (Sec. II);
- The automated validation of said method to make the lifter trustworthy, via a new dedicated program equivalence checking algorithm taking advantage of our transformation process to achieve both efficiency and high success rate, with a limited trust base (Sec. VI) ;
- Thorough experiments (Sec. VII) of a prototype implementation on real-world examples to show its wide applicability (all Debian GNU/Linux 8.11 x86 assembly chunks, some ARM, GCC and clang) and its substantial impact on 3 different verification techniques on samples from GMP, ffmpeg, ALSA and libyuv.

Discussion. This work targets assembly chunks as found in real-world programs: we lift and validate 76% of all assembly chunks from Linux Debian 8.11 (Table I) and benefit a range of state-of-the-art verification tools and techniques (Sec. VII-B). Still, system and floating-point instructions are currently considered out-of-scope. Especially, floats are not tackled here since handling them well is a challenge in itself for the whole toolchain (lifter, solver, verifier) — see the extended discussion in Sec. VIII. Also, TINA's implementation targets C since this is the principal language used for low-level programs, but the method itself would work unchanged on similar imperative languages, like LLVM. Finally, though some prior work has addressed code lifting for verification, it is worth noting that *verifiability* has never been explicitly addressed so far.

[1]Recall that general software verification problems, including program equivalence, are undecidable. Yet, software verification tools do exist and have been proven useful in practice.

II. CONTEXT AND MOTIVATION

Consider the code snippet of Fig. 1a extracted from UDPCast sources. It consists of the x86 assembly code itself (here: `"cld; rep stosl"`) together with a *specification* linking C variables to registers and declaring inputs, outputs and clobbers (i.e., registers or memory cells possibly modified by the assembly chunk). The compiler, upon encounter of such an (extended) assembly chunk, may use this specification – for example during register allocation. However, it is fully blind to the rest of the information (e.g., mnemonics) and will forward the chunk *as is* until code emission.

Annotation. The code in Fig. 1a is suffixed by a specification, written in a concise constraint language (GCC/clang syntax), in zones separated by ':' (lines 16-23):

- It first describes allocation constraints for output variables:
  "0". `"=c" (__d0)` specifies that variable `__d0` should be assigned to register `ecx`;
  "1". `"=D" (__d1)` specifies that variable `__d1` should be assigned to register `edi`;
- Then, lines 20-22 detail inputs: `"a" (eax)` holds `0`, `sizeof(fd_set) / sizeof(__fd_mask)` is held in the register described in "0" (`ecx`) and the one described in "1" (`edi`) holds `&((read_set)->__fds_bits)[0]`;
- Finally, the whole memory (`"memory"`) can be assigned. This basically tells the compiler to flush its memory cache before entering the chunk.

Informal semantics. The code `"cld; rep stosl"` has the following informal semantics (Fig. 1b): put the direction flag `df` to `0`, then fill `ecx` double words from the `edi` pointer with the value from `eax`. Intel's manual [24] explains that `df` drives the sign of the increment: when `df` is `0`, the sign is positive. TINA produces the code in Fig. 1c: the loop from the informal semantics is there, but the lifter optimized away (see Sec. V) elements like `df`, `eax` or `edi`.

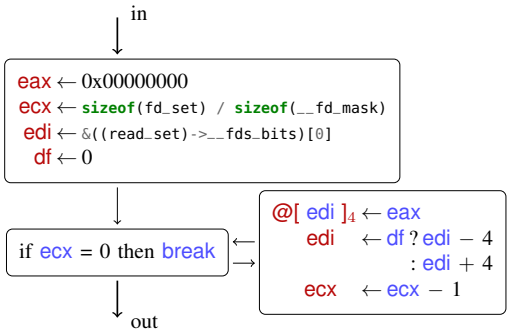
Running the analyzers. If we try to run industrial-strength C code analyzers on this code, we observe erratic behaviors: KLEE [4] stops with an error message; Frama-C, on the other hand, warns that 'Clobber list contains "memory" argument. Assuming no side-effect beyond those mentioned in output operands'. This message is clear but the behavior incorrect: the keyword `"memory"` stipulates that all memory may be assigned but Frama-C simply ignores it. This small example shows that a single line of assembly may throw off these tools. Of course, one may manually rewrite the chunks into semantically equivalent C code, then use C analyzers, but this is error-prone and not

```c
# 54 "/usr/include/i386-linux-gnu/sys/select.h"
typedef long int __fd_mask;

# 64 "/usr/include/i386-linux-gnu/sys/select.h"
typedef struct {
  __fd_mask __fds_bits[1024 / (8 * sizeof(__fd_mask))];
} fd_set;

# 1074 "socklib.c"
int udpc_prepareForSelect
(int *socks, int nr, fd_set *read_set)
{
  /* [...] */
  int maxFd;
  do {
    int __d0, __d1;
    __asm__ __volatile__
      ("cld; rep; " "stosl"
       : "=c" (__d0), "=D" (__d1)
       : "a" (0),
         "0" (sizeof(fd_set) / sizeof(__fd_mask)),
         "1" (&((read_set)->__fds_bits)[0])
       : "memory");
  } while (0);
  /* [...] */
  return maxFd;
}
```

(a) Original version



(b) Low-level semantics

```c
# 1074 "socklib.c"
int udpc_prepareForSelect
(int *socks, int nr, fd_set *read_set)
{
  /* [...] */
  int maxFd;
  {
    int __d0;
    int __d1;
    __fd_mask *__tina_4;
    unsigned int __tina_3;
    __tina_3 = sizeof(fd_set) / sizeof(__fd_mask);
    __tina_4 = & read_set->__fds_bits[0];
    {
      unsigned int __tina_ecx;
    __TINA_BEGIN_1__: ;
      __tina_ecx = __tina_3;
      while (0U != __tina_ecx) {
        *(__tina_4 + (__tina_3 - __tina_ecx)) = 0;
        __tina_ecx --;
      }
    __TINA_END_1__: ;
    }
  }
  /* [...] */
  return maxFd;
}
```

(c) TINA-generated version

Figure 1: Running example

scalable. **With TINA**, we are able to automatically generate the code of Fig. 1c, illustrative of our code transformations (see Sec. V), and automatically validate it *(trustworthy)*. We can then formally show with Frama-C [1], using abstract interpretation or deductive verification, that the code indeed verifies the informal semantics laid out before *(verification-friendly)*.

Identified threats to verifiability. Straightforward lifting from assembly to C (keeping the untyped byte-level view) does not ensure verifiability, as standard C analyzers are not well equipped to deal with such low-level C code. For example we cannot prove with Frama-C that a basic lifting of Fig. 1a meets its specification (cf. Appendix A). We identify 3 main threats to verifiability:

- **T1.** Low-level data: explicit flags – including overflows or carry, bitwise operations (masks), low-level comparisons, byte-level memory;
- **T2.** Implicit variables: variables in the untyped byte-level stack, packing of separate logical variables inside large-enough registers;
- **T3.** Implicit loop counters/index: structures indexed by loop counters at high-level are split into multiple low-level computations where the link between the different logical elements is lost.

Experiments in Sec. VII-B demonstrate that a straightforward encoding (BASIC) fails to get the best of any analysis – symbolic execution, abstract interpretation, or deductive verification.

Properties of inline assembly. TINA exploits the following properties, specific to inline assembly:

- **P1.** The control flow structure is limited: only a handful of conditionals and loops, hosting up to hundreds of instructions;
- **P2.** The interface of the chunk with the C code is usually given: programmers annotate chunks with the description of its inputs, outputs and clobbers with respect to its C context;
- **P3.** Furthermore, the chunk appears in a C context, where the types, and possibly more, are known: this kind of information is sought after in decompilers, using heuristics, whereas we only need to propagate it here.

All in all, the above points show that lifting assembly chunks is actually an interesting sub-problem of general decompilation, both simpler and richer in information and thus significantly more amenable to overall success.

III. BACKGROUND

A. Inline Assembly

We focus here on inline assembly in C/C++ code as supported by GCC and clang. MASM (Microsoft Macro Assembler) has a different syntax but works similarly.

Assembly chunks in the GAS syntax of GCC have two flavors: basic and (recommended) extended (see Fig. 2). *Basic assembly* (Fig. 2a) allows the insertion of assembly instructions anywhere in the code. They will be emitted *as is* during

```
__asm__ volatile
("movdqa b, %xmm0\n\t"
 "movdqa %xmm0, a\n\t");
```

(a) Basic version

```
__asm__
("movdqa %1, %%xmm0\n\t"
 "movdqa %%xmm0, %0\n\t"
 : "=m" (*a) : "m" (b)
 : "xmm0");
```

(b) Extended version

Figure 2: Assembly chunks: basic & extended versions

the production of the assembly file. In this case, compilers *assume* the chunk has no effect on its C scope, *preventing safe interactions between assembly and C code* – yet that does not stop developers from using it when the implicit context *looks* safe. In Fig. 2a, it is *implicitly* assumed that no optimization will occur on global variables `a` and `b` and that `xmm` registers are not used by default.

Extended assembly. *Extended assembly* allows in addition the description of the interactions with C through its inputs, outputs and clobbers (i.e., registers or memory cells whose value is rewritten by the chunk).

Such annotations work like a `printf` string format, as shown in Fig. 2b: some assembly operands may be replaced by placeholders referring to a list of C operands. The syntax requires binding C operands to location constraints, as in Fig. 1a and 2b. Constraints may also specify more than one location, and let the compiler choose the best way to place this operand. Common placement constraints include `r` to bind to a general register; `m` to bind to a memory address; `i` to an immediate value; and `g` which means "`r`, `m` or `i`". Operands may be read-only (for inputs) or write-only (for outputs) with the = modifier. A read-write operand is created either by linking an input to the same location as an output (`"0" (sizeof(fd_set) / sizeof(__fd_mask))` in Fig. 1a) or by using the + modifier instead of =. Without special modifiers, compilers assume read-only operands are consumed before write-only operands are produced, so that these may share the same locations. The clobber list may also contain keywords like `"memory"` (arbitrary memory cells may be read and/or written) and `"cc"` (conditional flags will be changed).

The specification of inputs, outputs and clobbers stands as a contract between the chunk and the compiler. Compilers are totally blind to what actually happens inside the chunk, relying on the contract, and will not warn about mistakes inside the chunk. Forgetting to list an input or a clobber is an easy mistake that can result in code which does not behave as expected.

Adoption. The use of inline assembly is pretty widespread – we estimate that 11% of Debian packages written in C/C++ directly or indirectly depends on inline assembly. It includes major projects like GMP and ffmpeg — Rigger et al. [7] actually reports that 28% of the top rated C projects on GitHub uses inline assembly. We further estimate that 75% of the chunks found in Debian Jessie 8.11 (used in Sec. VII-A) serve an optimizing purpose, with an average size of approximately 10 instructions, and up to 341. Inline assembly is often used in conjunction with C macros or inlineable functions to be specialized by the compiler at each location.

B. Binary-code lifters

Binary-code lifters are the cornerstones of modern binary-level analyzers. They are used to abstract the different binary Instruction Set Architectures (ISA) and formats into a single intermediate representation (IR) [20], [21], [22]. We rely on the IR of BINSEC [25], called DBA — other IRs are similar. Its syntax is shown in Fig. 3.

inst	:=	lv $\leftarrow$ e | goto e | if e then goto e else goto e
lv	:=	var | @[e]$_n$
e	:=	cst | lv | unop e | binop e e | e ? e : e
unop	:=	$\neg$ | $-$ | uext$_n$ | sext$_n$ | extract$_{i..j}$
binop	:=	arith | bitwise | cmp | concat
arith	:=	$+$ | $-$ | $\times$ | udiv | urem | sdiv | srem
bitwise	:=	$\wedge$ | $\vee$ | $\oplus$ | shl | shr | sar
cmp	:=	$=$ | $\neq$ | $>_u$ | $<_u$ | $>_s$ | $<_s$

Figure 3: Low-level IR for binary code

DBA is a minimalist language, comprising only two types of elements (bitvector values and memory) and three instructions: assignments, jumps and conditionals. Yet, this is enough to encode the functional semantics of major ISAs – including x86 and ARM.

Binary lifters provide specialized decoders for supported architectures, in the same spirit that a compiler has one code emitter per supported architecture. Lifters are then used in disassembly algorithms to (try to) recover the semantics of the binary program. We use them to disclose the semantics of compiled assembly chunks.

IV. TAMING INLINE ASSEMBLY: AN OVERVIEW

TINA lifts inline assembly to semantically equivalent C taking advantage of properties **P1**–**P3**. This original process consists mainly of two (new) phases: *verification-friendly lifting* and *validation*, detailed respectively in Sec. V and VI. First, let us discuss the overall approach, as schematized in Fig. 4.

Compilation. We compile the source code for the target architecture *with debug information*. Since we control code compilation, we also include all contextual data that can help to reconstruct C code, e.g., variable names and types.

Initial low-level IR lifting [genericity]. We now start the translation per se, by lifting the code back to the IR level. The use of binary code may seem gratuitous at first sight. This is however the best place to start working, since assembly chunks are totally instantiated and embedded in their context — register names and memory locations have been resolved by the compiler. Debug information here allows to locate the assembly chunk in the compiled code.

Transformation into high-level C [verifiability]. We then lift the IR back to C, through a combination of dedicated passes aiming at refining the low-level IR with high-level information (Sec. V). The end result is a C-only code where assembly chunks have been substituted by a lifted C code amenable to verification. *This step is original.*

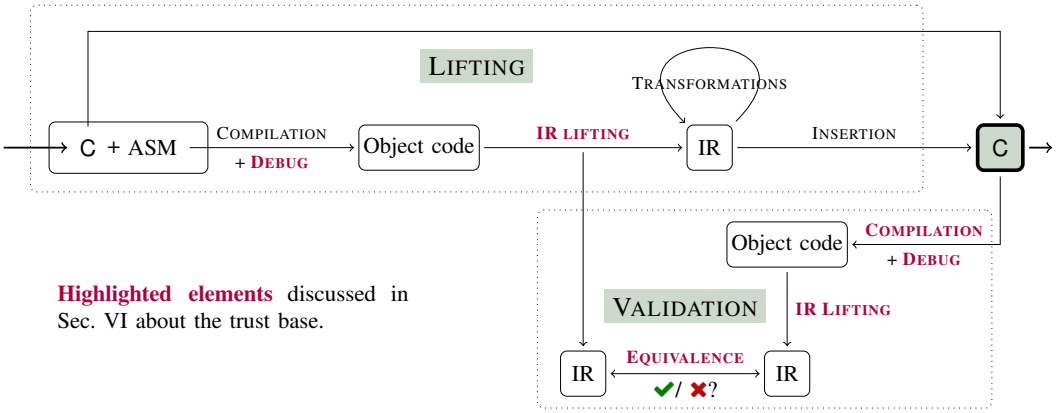


Figure 4: Overview of TINA

Validation [trust]. The validation phase starts by recompiling the pure C code, *without optimization* in order to preserve the code structure — our validation technique depends on it. We locate the binary code corresponding to the lifted code once more, and get back its IR representation. We now possess two distinct IR pieces: this one and the one from the first compilation. We will aim to prove their semantic equivalence in Sec. VI. *This step is original.*

We have ***implemented*** *a prototype of* TINA leveraging existing tools: Frama-C [1] for C source code manipulation (parsing, localization, C injection), BINSEC [25], [26] (IR lifting [21], SMT solvers integration [27]), and the DWARF [28] debug format to pass information to binaries with the compiler.

V. FROM LOW-LEVEL IR TO HIGH-LEVEL C

The goal of this lifting phase is to recover *verifiable* C code preserving the semantics of the original assembly chunk. The transformations at IR level mitigate the identified threats to verifiability (Sec. II), and reinforce each other (Sec. VII).

Type verification & propagation. To lift assembly code back to C, chunk operations on bitvectors and memory need to be mapped to C operations on integers (signed/unsigned) and pointers. To this end, we propagate types from the interface into the IR operations. IR types can either be addresses (typed pointers) or values (signed or unsigned, with an associated size). Type information is further synthesized using forward propagation and constraints imposed on operands by low-level operations. This step also guarantees that inputs' and outputs' types are respected. The lifter gives concrete C types using the type size information from DWARF.

High-level predicate recovery (threat T1). Low-level conditionals use flags — zero (`zf`), sign (`sf`), carry (`cf`) or overflow (`of`) — set by previous instructions. In most situations, they have little meaning on their own and the way they are computed hampers understanding the purpose of the condition. This pass applies Djoudi et al.'s recent technique [29] based on semantic equivalence proved by SMT solvers. It substitutes the low-level condition, built on flags, by a more readable arithmetic comparison. For example, this phase recovers `if (ecx + 1 > 1) goto label;` instead of `if (zf == 0 && sf == of) goto label;` from the assembly snippet `"decl ecx; jg label;"`.

Register unpacking (threat T2). Assembly chunks often contain optimizations exploiting data level parallelism in order to use the full capacity of the hardware by packing multiple value inside a bigger one fitting inside a machine register. For instance, loading 4 (byte) characters inside an integer is more efficient than doing four smaller loads. The concept has been exacerbated with *Single Instruction Multiple Data* extensions, providing vectorized registers up to 512-bits. The issue here is that such *packed code* has very low-level semantics (masks, shifts, etc.). Our novel *register unpacking* method uncovers the independent variables stored in a container, thus preventing packed arithmetic from destroying the abstractions of the analyzers. The method amounts to splitting registers into independent variables, whose size depends on the uncovered usage, rewrite the code accordingly and then clean up unused variables and code, and rebuild higher-level chunks through dedicated simplifications. The principle is the following: if a subpart of a variable is read in the code (e.g., $\mathsf{extract}_{0..15}$ `eax`), then this subpart is likely to correspond to a logical entity. So we generate a fresh variable for this entity, receiving the restricted value, and replace each such extraction by this new variable. To avoid the need for a fixpoint until every variable extraction is replaced, we perform the replacement eagerly, in 3 steps:

1) A forward pass where each assignment of 8×2^k bits is split into multiple fresh assignments of 8×2^i bits where $i \leq k$ (for instance, `eax` will be split into {{`al`, `ah`, `eax_16_23`, `eax_24_31`}, {`ax`, `eax_16_31`}, {`eax`}};
2) At the same time, each variable restriction $\mathsf{extract}_{i..j}$ `var` corresponding to one of the newly generated variables is replaced by this new variable;
3) A final pass of dead code elimination removes each unused freshly generated variable.

Note that subparts may overlap with each other (for instance, `al`, `ax` and `eax` share common parts) but we found that most of

the time, only one of them survives the final step. Thus, the size of the produced code does not increase much in the end.

Finally, we also rely on the fact that expression propagations together with concatenation-extraction simplification will automatically reconstruct bigger sized variables from concatenation of smaller sized ones (e.g., `ax` half-word from `al` and `ah` bytes).

In Fig. 5, the chunk loads two char in a register before adding them, using the `h` and `l` prefixes to access them. Without register unpacking, the lifter uses bitmasking (Fig. 5c), making the code more complex than its clear initial intent (Fig. 5b).

```
extern const
unsigned char src[2];
unsigned char sum;
__asm__
("movzxw   %1,  %k0\n\t"
 "addb     %h0, %b0\n\t"
 : "=&Q" (sum)
 : "m" (src));
```

(a) Source

```
unsigned char __tina_ah;
unsigned char __tina_al0;
unsigned char __tina_al1;
__TINA_BEGIN_0__: ;
__tina_al0 = *src;
__tina_ah = *(src + 1);
__tina_al1 =
  __tina_ah + __tina_al0;
sum = __tina_al1;
__TINA_END_0__: ;
```

(b) Lifting *with* unpacking

```
unsigned int __tina_eax0;
unsigned int __tina_eax1;
__TINA_BEGIN_0__: ;
__tina_eax0 = (*src) | (*(src + 1) << 8);
__tina_eax1 = (0xffffff00 & __tina_eax0) | (0xff &
  ((0xff & (__tina_eax0 >> 8)) + (0xff & __tina_eax0)));
sum = 0xff & __tina_eax1;
__TINA_END_0__: ;
```

(c) Lifting *without* unpacking

Figure 5: Register unpacking

Expression propagation (threats T1 and T2). We draw inspiration from compiler optimization techniques to devise a novel dedicated simplification mechanism geared toward our needs. In particular, we can afford very aggressive simplifications (small code size w.r.t. standard compilation setting) but we have to address particular kinds of low-level instructions (coming from IR translation). Our method originally combines *eager expression propagation* coupled with *dedicated (low-level) simplifications* and *a posteriori control* to revert fruitless propagations – when no simplification rule has been triggered.

Eager expression propagation relies on the idea that more expression propagation raises more opportunity for further simplifications by dedicated rules. Yet, systematic propagation can yield an exponential blowup of the code under analysis rather than the desired simplifications. To mitigate this problem we propose eager propagation coupled with a posteriori control to revert fruitless propagation. The algorithm works as follows:

- As a preliminary step, a data flow analysis collects all symbolic values (terms) associated to each pair (name, program point) used in the IR code;
- First, we *unconditionally propagate* symbolic values in a first pass but save a *reverse map* for each propagated expression (in case the propagation is not fruitful);
- Second, we expect simplification rules (described below) to *simplify* the whole expression;
- Third, we identify expressions not yet simplified (by syntactically comparing the terms before and after simplification) and revert back the propagation on such case thanks to the reverse map (*a posteriori control*);
- Finally, we *cleanup* the code by filtering out unused variables, dead branches and dead code.

Regarding *simplification rules*, we use a mixture of standard and dedicated simplification rules – standard for typical integer-level properties and dedicated for more low-level aspects. Here is a representative (incomplete) subset of these rules – see Appendix E for a complete set – where $|x|$ denotes the size of the expression x, $\diamond$ any binary operator, C a condition ($|C| = 1$), k is a constant.

- associativity-commutativity re-ordering:
$$x + 1 + a \hookrightarrow a + x + 1$$
- constant propagation (modular arithmetic):
$$10 + 5 \hookrightarrow 15, \quad 10 \times 2 \hookrightarrow 20$$
- standard algebraic simplifications (identity, neutral, absorbing and inverse elements, etc.):
$$x + 0 \hookrightarrow x, \quad x \times 1 \hookrightarrow x, \quad x \times 0 \hookrightarrow 0, \quad x - x \hookrightarrow 0$$
$$x \vee 0 \hookrightarrow x, \quad x \wedge 1 \hookrightarrow x, \quad x \wedge x \hookrightarrow x, \quad x \oplus x \hookrightarrow 0$$
- ternary expression simplification:
$$C \ ? \ x : x \hookrightarrow x, \qquad \neg C \ ? \ x : y \hookrightarrow C \ ? \ y : x$$
$$true \ ? \ x : y \hookrightarrow x, \qquad false \ ? \ x : y \hookrightarrow y$$
$$C \ ? \ true : false \hookrightarrow C, \quad C \ ? \ false : true \hookrightarrow \neg C$$
- ternary expression development:
$$x \diamond (C \ ? \ y : z) \hookrightarrow C \ ? \ x \diamond y : x \diamond z$$
$$(C \ ? \ w : x) \diamond (C \ ? \ y : z) \hookrightarrow C \ ? \ w \diamond y : x \diamond z$$
- two-complement arithmetic abstraction:
$$\neg x + 1 \hookrightarrow -x$$
$$\text{extract}_{|x|-1}(x) \hookrightarrow x <_s 0$$
$$\text{uext}_n(C) - 1 \hookrightarrow C \ ? \ -1_n : 0_n$$
$$\text{sext}_n(C) \hookrightarrow C \ ? \ -1_n : 0_n$$
- concatenation:
$$\text{uext}_{|x|+|y|}(x) \vee \text{concat}(y, 0_{|x|}) \hookrightarrow \text{concat}(y, x)$$
$$\text{uext}_{|x|+k}(x) \text{ shl } k \hookrightarrow \text{concat}(x, 0_k)$$
$$\text{concat}(0_k, x) \hookrightarrow \text{uext}_{k+|x|}(x)$$
- extraction-concatenation simplification:
$$\text{extract}_{0..|x|-1}(x) \hookrightarrow x$$
$$\text{concat}(\text{extract}_{i..j}(x), \text{extract}_{j+1..k}(x)) \hookrightarrow \text{extract}_{i..k}(x)$$
$$\text{extract}_{i..j}(\text{concat}(x, y)) \text{ when } j<|y| \hookrightarrow \text{extract}_{i..j}(y)$$
$$\text{extract}_{i..j}(\text{concat}(x, y)) \text{ when } |y|\leq i \hookrightarrow \text{extract}_{i-|y|..j-|y|}(x)$$

Fig. 6 shows how the addition of rewrite rules exposes the intended semantics of a branchless absolute value computation.

Loop normalization (threat T3). This pass aims at highlighting the relations between the current iteration of the loop and the variable values. We especially look for affine relations of the form $a \times x + b$ where x is the loop iteration counter. We indeed found out that tools much prefer to analyze `for (int i = 0; i < N; i++) T[i] = C;` instead of `for (char *t = T; t < T + N; t++) *t = C;`. Assembly code, though, is more likely to have the second form.

```
cltd              # sign extend eax in edx
xor %edx, %eax    # 1-complement eax if eax < 0
sub %edx, %eax    # add one to eax if eax < 0
```

(a) Branchless absolute value implementation

$\text{tmp}_{64} \leftarrow \text{sext}_{64}\ \text{eax}_0$
$\text{edx}_0 \leftarrow \text{extract}_{32..63}\ \text{tmp}_{64} \hookrightarrow \text{edx}_0 \leftarrow \text{eax}_0 <_s 0\ ?\ 0\text{xffffffff} : 0$
$\text{eax}_1 \leftarrow \text{eax}_0 \oplus \text{edx}_0 \hookrightarrow \text{eax}_1 \leftarrow \text{eax}_0 <_s 0\ ?\ \neg\text{eax}_0 : \text{eax}_0$
$\text{eax}_2 \leftarrow \text{eax}_1 - \text{edx}_0 \hookrightarrow \text{eax}_2 \leftarrow \text{eax}_0 <_s 0\ ?\ \text{-eax}_0 : \text{eax}_0$

(b) IR transformations

Figure 6: Expression propagation

We thus transform each self-incrementing (-decrementing) variables of the form `v = I; while (...) { ...; v = v + k; }` in order to get code more amenable to analysis. The transformation is done in (up to) 3 steps:

1) **rebasing** replaces the initial value `I` by 0 and each occurrence of the variable `v` by `I + v`;
2) **rescaling** replaces the increment `k` by 1 and each occurrence of the variable `v` by `k * v`;
3) **merging** unifies the transformed variables with the loop iteration counter.

For example, in Fig. 1c, the byte-level affine relation between the counter `ecx`, lifted as `__tina_ecx`, and the moving pointer `edi`, based at `__tina_4`, is `edi` $\equiv$ `__tina_4 + 4 * (__tina_3 - ecx)` — the code is lifted as `__tina_4 + (__tina_3 - __tina_ecx)` to take pointer arithmetic into account (`__tina_4` is an `int *`, pointing to 4 bytes long values in x86).

VI. Validation

For our translation to be trustworthy, we use a two-pronged approach: 1) We try to prove the semantic equivalence of the code prior to lifting with the lifted C code; 2) If this fails, we rely on intensive random testing (fuzzing) to increase the level of trust in the lifted C code.

Block-based semantic equivalence. The lifting process of Sec. V strives to preserve the isomorphism of the control-flow graphs based on basic blocks between the initial assembly chunk and its lifted C representation over their DBA IR representation. This property allows us to tackle the equivalence proof at basic block level. The proof of equivalence proceeds as follows:

S1. We *check the isomorphism of the control-flow graphs extracted from the two lifted programs*. Since we deal with deterministic labeled directed graphs, this check is immediate — and usually succeeds. TInA is actually very careful during simplifications and recompilation to preserve the control-flow structure (see details below). For the isomorphism check, we track the relation between the heads of IR basic blocks and the corresponding emitted C code thanks to C labels and debug line information. If the check succeeds, we go to **S2**, otherwise we **[fallback]** on fuzzing — in practice (Sec. VII-A), the latter has never happened.

S2. Once we know the two control-flow graphs are isomorphic, we *try to demonstrate the pairwise equivalence of corresponding vertices*. This allows to avoid directly dealing with loops. Each pairing of basic blocks is translated to logical formulas for which we ask SMT solvers: if inputs are identical, can outputs be different? If all queries are unsatisfiable then *equivalence is proven* **[success]**, otherwise we use our **[fallback]**.

Taming simplifications. In order to help the equivalence proof succeed, TInA passes were designed to *preserve the control-flow graph structure* and *to be traceable*. For the first goal, simplifications never modify jump instruction, except for trivial dead branch elimination and the lifter avoids inserting branches with lazy constructions such as `&&`, `||` or ternary operators. For the second goal, when a simplification changes the input-output relation of a basic block, it records the changes w.r.t the old ones and these properties will be added to the assumptions of **S2**. For instance, in Fig. 1, the expression propagation records that `eax` holds the value `0` for the entire chunk. It will then be used during **S2** to prove the equivalence of the loop body where the register no longer exists in the generated part (Fig. 1c).

What could go wrong? While TInA uses simplifications and lifting passes tailored to make the block-based semantic equivalence algorithm possible, the recompilation step is blind to this requirement and may therefore threaten it.

The **S1** check may fail if the compiler modifies the control flow graph, for example if some elements outside of the assembly chunk render a branch dead or a loop unrollable. In Fig. 1c, since `sizeof` is known at compile time, clang -O1 unrolls the loop, making the isomorphism check fail.

The **S2** query may fail if the compiler moves parts of the computation across basic blocks, changing the relation between inputs and outputs. It may happen during code motions, like loop-invariant code motions. In this case, the graph isomorphism still holds but the relation between basic blocks is lost. GCC -funroll-loops partially unrolls (8 times) the loop body in Fig. 1c leading to a failed equivalence query.

To avoid such problems, we recompile the code *without any optimization* (-O0).

Note that SMT checks never time out in our experiments (Sec. VII), probably due to the naturally small size of block-based queries. However, we can imagine that code showing hard-to-reverse behaviors, such as cryptographic hash functions, could make the **S2** query fail.

Trust base. Validation allows to increase the confidence in the lifting process, using 3 components as the *trust base*: the binary-code lifter, the compiler and the solver. All are well tested software and the last two are part of the trust base of (most) modern source-level verification tools anyway. Furthermore, while we trust the compiler debug information, we argue that the compilation process itself is not part of the trusted base: assembly chunks are untouched by it and validation will very likely catch errors during re-compilation. Besides, further mitigation includes systematic testing of assembly chunks vs. their IR representation, and using multiple compilers and/or solvers.

Table I: Applicability on Debian 8.11 Jessie distribution (GCC 5.4)

	x86						ARM			
	TOTAL	BIG 100	ALSA	ffmpeg	GMP	libyuv	ALSA	ffmpeg	GMP	libyuv
Assembly chunks	3039	100	25	103	237	4	0	85	308	1
Trivial	126	0	0	6	13	0	–	1	28	0
Out-of-scope	449	40	0	17	0	3	–	2	0	0
Rejected	138	11	0	12	1	0	–	12	2	0
Relevant	2326 76%	49 49%	25 100%	68 66%	223 94%	1 25%	–	70 82%	278 90%	1 100%
Lifted	2326 **100%**	49 **100%**	25 **100%**	68 **100%**	223 **100%**	1 **100%**	–	70 **100%**	278 **100%**	1 **100%**
Validated	2326 **100%**	49 **100%**	25 **100%**	68 **100%**	223 **100%**	1 **100%**	–	70 **100%**	278 **100%**	1 **100%**
Average (Max) size	8 (341)	104 (341)	50 (70)	5 (10)	6 (31)	31 (31)	–	5 (16)	5 (10)	29 (29)
Lifting time (s)	121	98	2	63	2	< 1	–	< 1	4	< 1
Validation time (s)	1527	36	17	255	110	< 1	–	48	187	< 1

VII. EXPERIMENTAL EVALUATION

We evaluate our implementation of TINA on 3 research questions: **RQ1)** How applicable is it on assembly chunks found in the wild? **RQ2)** How do off-the-shelf program analyzers behave on lifted code? **RQ3)** What is the impact of each optimization?

A. *Wide applicability* (RQ1)

We run our prototype on *all* assembly chunks found in the Linux Debian 8.11 distribution (for x86), i.e. ≈ 3000 chunks distributed over 200 packages and 1000 functions. As chunk distribution is not smooth, we also fix 2 subsets of samples: one with the 100 biggest chunks, and another with all chunks from 4 key major projects exploiting low-level optimizations: GMP, ffmpeg, ALSA and libyuv. Table I sums up the results of lifting with TINA.

Table II: Applicability by compiler (x86)

	GCC 5.4	GCC 4.7	clang 3.8
Assembly chunks	3039	2955	2852
Relevant	2326 76%	2326 78%	1970 69%
Lifted	2326 **100%**	2326 **100%**	1970 **100%**
Validated	2326 **100%**	2326 **100%**	1970 **100%**

We exclude trivial (empty or unused), out-of-scope and rejected chunks. *Out-of-scope chunks* include those with floating point operations, OS-level hardware instructions or hardware-based crypto-primitives, like AES. *Rejected chunks* are those deemed unsafe because they do not respect their interface. Yet, we activate options in our tool to specifically regard accessing flags, `xmm` registers or memory as safe – allowing to consider 150 extra chunks as relevant, notably in ffmpeg. The statistics of Table I report on the tool's behavior with these settings.

On in-scope chunks, TINA *performs extremely well*, with **100**% chunks lifted *and* fully validated (no resort to testing) — this amounts to 76% of all chunks found — *for a negligible cost* (0.7s per chunk on average). The biggest 100 chunks are a little less successful as they have a fair amount of (unhandled) floating-point instructions. TINA works equally well on major projects for ARM or x86, and with GCC or clang on x86 (Table II), confirming its genericity.

B. *Adequacy to formal verification tools* (RQ2, RQ3)

We select 3 tools representing popular formal techniques *currently used in the industry*: KLEE [4] for symbolic execution [9] (bug finding), and Frama-C [1] with its EVA plug-in [30] for abstract interpretation [12], [31] (runtime error verification) and WP plug-in [32] for deductive verification [10], [11] (functional correctness).

Experiments on both symbolic execution and abstract interpretation use 58 functions (out of 366) from the 4 key projects in Sec. VII-A, selected due to the ease of automatically generating the initial contexts for both analyses. For all 3 tools, we also report the observed differences using a *basic* lifter and different optimization levels: *O1* (high-level predicate recovery), *O2* (O1 + register unpacking), *O3* (O2 + expression propagation) and *O4* (O3 + loop normalization). Note that ***O4* is TINA**.

Table III: Impact of TINA & lifting strategies on KLEE

	LIFTING					
	NONE	BASIC	O1	O2	O3	O4
Functions analyzed w/o blocking	3	58	58	58	58	58
Functions 100% covered	✖	25	25	25	25	25
Aggregate time	N/A	115s	115s	110s	103s	105s
# paths (all functions)	1.4M	1.5M	1.8M	4.6M	6.6M	6.6M

Symbolic execution. We perform our experiments with KLEE [4] which at present does not handle inline assembly chunks and stops upon meeting one – except for a very few simple cases such as assembly-level rotations. This fact can sometimes prevent the adoption of symbolic execution [33].

Table III summarizes our findings. (**RQ2**) First, KLEE alone can analyze only few functions (3/58) as (almost any block of) assembly stops the analysis, and none of them is fully path-covered. Adding lifting allows to *analyze all considered functions* (58/58), to completely path-cover 43% of them (25/58) and to explore significantly more paths within the same analysis budget ($\times 4.7$).

The lifting strategy (**RQ3**) does not impact the functions that KLEE can fully cover, but TINA optimizations considerably speed up code exploration, enabling to cover significantly more paths ($\times 4$) than basic lifting in the same amount of

time. This is explained by TINA-produced code being higher-level, with fewer instructions and local variables, thereby accelerating SMT-solving. Note that control-flow structure, and thus total number of paths, does not change. Moreover, each optimization step brings some degree of improvement. The major improvement gaps here are brought by register unpacking (O2) and expression propagation (O3). As expected, loop normalization (O4) has no impact as symbolic execution simply unrolls loops. Additional experiments (Supplementary material, Table VII) demonstrates that high-level recovery (O1) has also a substantial impact on the analysis (removing it leads to 5.4M explored paths, vs 6.6M in full TINA).

Abstract interpretation. We use the Frama-C EVA [30] plug-in. Frama-C has limited support for inline assembly based on interfaces, translating them into logical assigns annotations for modified variables – safely interpreted in EVA (and WP) as non-deterministic assignments.

Table IV sums up the results for **RQ2**. Lifting the assembly code with TINA almost always *reduces the number of alarms in the common C code* (23/27). This follows from the better precision of the analysis since modified variables in the lifted code are now accessible. In half the cases (11/20), we observe a *precision gain on function return values*. Most functions (31/34) with return values or initial C alarms show such improvements.

Table IV: Impact of TINA on EVA

Function with	ALSA	ffmpeg	GMP	libyuv	TOTAL
Returns (non `void`)	0	9	10	1	20
Better return values	–	9	1	1	11 55%
Initial C alarms	2	8	16	1	27
Alarm reduction in C	2	8	12	1	23 85%
New memory alarms	12	2	3	0	17 26%
Positive impact	14	17	13	1	45 77%

The lifted C code also contains *new alarms* (17/58) which we could not detect before and should be taken into account (usually out-of-bounds or other memory accesses). We also found some *possibly buggy behaviors* (Sec. VII-D).

For short, we observe positive impact from TINA w.r.t. non-lifted code on 77% (45/58) of the functions (more precision, reducing alarms from over-approximations of inline assembly, or new memory alarms in lifted code) .

Table V: Impact of lifting strategies on EVA

	LIFTING					
# Functions	NONE	BASIC	O1	O2	O3	O4
without any alarms	✗	12	12	14	14	**19**
with ASM memory alarms	N/A	29	29	29	21	17
errors	✗	1	1	1	2	2
emitted C alarms	231	184	184	177	177	177
emitted ASM alarms	N/A	316	244	199	165	128
total alarms	231	500	428	376	342	305

Table V additionally shows the impact of the lifting strategy (**RQ3**). Compared with basic lifting, each additional optimization increases the quality of the lifted code (fewer ASM and total alarms) and the precision of the analysis (more functions without alarms, fewer memory alarms, more errors) – including loop normalization which allows finer approximations of loop fixpoints (*widening*). TINA (O4) thus significantly improves all these aspects. Moreover, the produced alarms are more precise: possible buffer overflows (such as a ffmpeg -1 index access – see Appendix D) are now recognized as errors and not mere alarms. Additional experiments (Supplementary material, Table VII) demonstrates that removing any of the optimization steps leads us quite far from the whole chain result.

Table VI: Impact of TINA & lifting strategies on WP

	LIFTING					
FUNCTION	NONE	BASIC	O1	O2	O3	O4
saturated_sub	✗	✓	✓	✓	✓	✓
saturated_add	✗	✗	✓	✓	✓	✓
log2	✗	✗	✗	✗	✓	✓
mid_pred	✗	✗	✓	✓	✓	✓
strcmpeq	✗	✗	✗	✗	✓	✓
strnlen	✗	✗	✗	✗	✓	✓
memset	✗	✗	✗	✗	✓	✓
count	✗	✗	✗	✗	✓	✓
max_element	✗	✗	✓	✓	✓	✓
cmp_array	✗	✗	✗	✗	✓	✓
sum_array	✗	✗	✗	✗	✓	✓
SumSquareError	✗	✗	✗	✗	✓	✓

Weakest precondition calculus. We use the deductive verification Frama-C plug-in WP [34], [32]. We take 12 assembly-optimized functions (see details in Supplementary, Table IX): 6 excerpts from ffmpeg, GMP, libyuv, libgcrypt and UDPCast, 2 others adapted from optimized assembly snippets and 4 translated examples from ACSL by example [35]. Functional specifications and loop invariants are manually inserted before verification, as usual for WP-based methods – we do not insert any other annotation. Moreover, recall that without lifting, assembly chunks are correctly over-approximated by non-deterministic assignments to the modified C variables.

Table VI details our results. The unlifted code does not require invariants (no C-level loops), while lifted codes all require identical invariants as they share the same control-flow structure. A quick glance at Table VI shows that (**RQ2**) while WP without lifting *never* succeeds and basic lifting is far from enough (1/12), TINA does allow *to prove the functional correctness of all functions* (12/12). The simple over-approximations of assembly chunks provided by Frama-C without lifting are not sufficient to prove properties as strong as functional correctness.

Regarding optimization steps (**RQ3**), it turns out that loop normalization (O4) has no direct impact since the user must provide manual loop invariants. On the other hand, all other steps are complementary (Table VI) and crucial: removing only one of them yields at best a 6/12 success rate (Supplementary material, Table VII).

C. Conclusion

Experiments show that our code lifting method is *highly practical* (100% Debian 8.11 in-scope blocks are lifted and

validated), that it has a *positive and significant impact on all 3 formal verification tools considered* – allowing them to effectively handle code with inline assembly, and, finally, that full TINA (O4) is needed to facilitate further code analyses – as less refined lifting yields poorer analyses.

Interestingly, all analyses do not behave the same w.r.t the optimization chain: symbolic execution mostly takes advantage of register unpacking and expression simplifications, abstract interpretation is sensitive to the 4 optimization steps and weakest precondition calculus strongly requires all of them but loop normalization – which is already granted by user-supplied loop invariants.

D. Epilogue: post-analysis considerations

We found **567** compliance issues during our experiments. Most have no impact with *current* compilers but may induce bugs out of compiler changes, maintenance or code reuse.

While evaluating verifiability, we ran into *6* potential buffer overflows hidden in assembly chunks. For example, a ffmpeg function accesses index `-1` of its input buffer – this is actually reported in the comments (see details in Appendix D). All errors initially reported by Frama-C EVA were also reproduced with KLEE. After determining and adding relevant logical preconditions, we were able to show the absence of runtime errors in the reported "corrected" functions. Besides, we were able to prove (with Frama-C WP) the functional correctness of 6 functions from the Debian distribution code base, including `SumSquareError` (24 assembly instructions).

VIII. Discussion

A. Threats to validity

Benchmark representativeness. The considered code base is quantitatively and qualitatively representative of the use of inline assembly: it is extensive and comprises highly popular and respected projects. We mainly experiment on GCC and x86, but our experiments on ARM and clang show our results also hold in these settings. Still, we obviously miss closed-source software and code which relies on Microsoft's C compiler (different assembly syntax). Yet, there is no reason to believe it would behave differently.

Verification methods. We consider three of the most popular verification techniques (symbolic execution, abstract interpretation, deductive verification), representative of the major classes of analysis, both in terms of goal (bug finding, runtime error checking and proving functional correctness) and underlying core technologies (domain propagation, constraint solving & path exploration, first-order reasoning). Also, we rely on well-established verification tools, each applied in several successful industrial case studies. Thus, we reckon that our experiments support our claim regarding the general verifiability of the codes TINA produces.

B. Limitations

Our lifting has two main limitations: hardware-related instructions and floating-point operations.

Since we aim to lift assembly chunks back to C, the support of hardware related instructions cannot be achieved outside of modeling hardware in C as well — for example, neither DBA IR nor C can make direct reference to hardware interrupts. Here we probably cannot do better than having two (approximated) C models of hardware instructions, one for over- and one for under-approximations. While not necessarily that difficult for reasonable analysis precision, this is clearly a manpower-intensive task.

The float limitation is primarily due to the lack of support in BINSEC. Adding such support is also manpower-intensive, but not that hard. Yet, the real issue is that efficient reasoning over floats is still ongoing scientific work in both program analysis and automated solvers (e.g., theory support is new in SMT-LIB [36], only 2 solvers in the relevant category of SMT-COMP 2018). As such, it threatens our validation part, and most program analyzers would not be able to correctly handle these lifted floats anyway. *Despite these limits, we still lift and validate 76% of assembly chunks of a standard Linux distribution*.

Finally, our technique is amenable, to a certain extent, to standalone assembly code or even binary code decompilation. However, this case can quickly deteriorate to the usual difficult problem of lifting an arbitrary program. Especially, dynamic jumps or large-size complicated CFG would probably yield serious issues.

IX. Related Work

Though some prior work has addressed code lifting for verification, it is worth noting that *verifiability* has never been explicitly addressed so far. We hereafter review approaches (partly) related to our method.

Assembly code lifting and verification. Maus [13], [37] proposes a generic method simulating the behavior of assembly instructions in a virtual machine written in C. This work was used by the Verisoft project to verify the code of an hypervisor consisting of mixed low-level code. Maus' technique relies on VCC [38] to write and prove verification conditions regarding the state of its machine. While we strive to produced high-level code, Maus' virtual code contains all low-level code details, including flags.

Further work by Schmaltz and Shadrin [39] aims (only) at proving the ABI compliance of the assembly chunks. This method is however restricted to MASM and the Windows operating system. TINA, here applied to GCC inline assembly, is independent of the assembly dialect by leveraging binary level analyzers and is applicable to a wider range of architectures.

Fehnker et al. [40] tackle the analysis of inline assembly for ARM architecture, using a model-checking based syntactic analysis to integrate C/C++ analyses with inline assembly. This solution is however limited by its purely syntactic basis: first, it is restricted to one single inline assembly dialect; second it loses the soundness properties we target. Losing soundness may be an appropriate practical trade-off but not when targeting sound formal analyses.

Corteggiani et al. [14] also use code lifting within their framework. However, their end goal is to perform dynamic symbolic analyses on the produced lifted code. Sec. VII-B shows that such very targeted lifting may not be enough for other formal analyses. Moreover, correctness of the translation is not addressed.

Myreen et al. [41] targets verification of pure assembly code. The translation corresponds to our basic lifter, yet the approach proves the initial lifted IR is semantically equivalent to a very detailed ISA model. This paper then targets verification at the level of assembly code but requires code annotations and interactive proving. Our proposal targets the lifting of inline assembly within C for (general) verification purposes, is geared at ensuring the verifiability of the produced code, and its validation establishes the correctness of the IR transformations producing the final extracted C code.

Decompilation. Decompilation [15], [16], [17] tackles the challenge of recovering the original source code (or a similar one) from an executable. This goal is very difficult and requires hard work to find back the information lost during compilation [42]. Despite significant recent progress [19], decompilation remains an open challenge. Still, it is used to enhance program understanding, e.g., during reverse engineering. As such, correctness is not the main concern — for example it does not always need produce compilable source code.

Soundness is addressed by two recent works. Schulte et al. [18] use search-based techniques to generate source-code producing byte-equivalent binaries to the original executable. This technique, when it succeeds, ensures soundness by design but it is only applied to small examples, with limited success. Brumley et al. [19] on the other hand use testing to increase trust in their lifted code.

We do draw inspiration from some decompilation techniques for type reconstruction [43], [44]. Even though we do not construct types that are not derived from inputs, it helps in strengthening our type system.

Recovering the instructions and CFG of the code under analysis is a big challenge in decompilation [45], [46], especially for adversarial codes like malware. The regularity and patterns of managed codes allow a very good recovery in practice [43] by unsound methods, yet without any guarantee. Inline assembly chunks have more limited behaviors (clear control-flow, no dynamic jumps) and the fact that we control compilation makes it a non-issue for us.

Binary-level program analysis. For more than a decade now, the program analysis community has spent significant efforts on binary-level codes [47], either to analyze source-less programs (malware, COTS) or to check the code that is really running. The efforts have mainly been concerned with safe high-level abstraction recovery [48], [49], [50], [29], [51] and invariant computation.

Several generic binary lifters have been produced [20], [21], [22], reducing complex ISAs to a small set of semantically well-defined primitives. Though well tested [22], more trust could be achieved if lifters were automatically derived from something akin to ARM's formal specifications [52].

Mixed code problems. Morrisett et al. [53] have proposed Typed Assembly Language to ensure memory and control flow integrity in low-level assembly. Patterson et al. [54] have exploited the idea to mix low-level code with functional languages. We borrow some elements to propagate types between C and inline assembly.

Translation validation and code equivalence. In order to achieve safe lifting, we use translation validation [55], [23], [56], a technique also used in CompCert register allocation [57]. Our formal needs thus rely on well-established and tested tools (here SMT solvers), usable as blackboxes, instead of a full formal proof of the whole lifting chain.

Program equivalence checking is considered a challenging verification task. Dedicated approaches start to emerge, like relational weakest precondition calculus [58] (for proof) or relational symbolic execution [59] (for bug finding).

X. Conclusion

We have presented TINA, a method enabling the analysis of C/C++ code mixed with inline assembly, by lifting the assembly chunks to equivalent C code. This method is the first to generate well-structured C code amenable to formal analysis through a dedicated principled succession of transformations geared at *improving the verifiability of the produced code*. To boot, translation validation *builds trust* into the lifting process. Thorough experiments on real-world code show that TINA is widely applicable (100% of in-scope chunks from Linux Debian Jessie 8.11 are validated) and that its semantic transformations positively (and significantly) impact popular verification techniques.

References

[1] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Asp. Comput.*, vol. 27, no. 3, pp. 573–609, 2015.

[2] D. Delmas and J. Souyris, "Astrée: From research to industry," in *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, ser. Lecture Notes in Computer Science, H. R. Nielson and G. Filé, Eds., vol. 4634. Springer, 2007, pp. 437–451.

[3] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[4] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224.

[5] P. W. O'Hearn, "From categorical logic to facebook engineering," in *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. IEEE Computer Society, 2015, pp. 17–20.

[6] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg, "The static driver verifier research platform," in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. B. Jackson, Eds., vol. 6174. Springer, 2010, pp. 119–122.

[7] M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. Mössenböck, "An analysis of x86-64 inline assembly in c programs," in *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '18. New York, NY, USA: ACM, 2018, pp. 84–99.

[8] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[9] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[10] R. W. Floyd, "Assigning meanings to programs," in *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, J. T. Schwartz, Ed. Providence: American Mathematical Society, 1967, pp. 19–32.

[11] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[12] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 1977, pp. 238–252.

[13] S. Maus, M. Moskal, and W. Schulte, *Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 284–298.

[14] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 309–326.

[15] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to High-Level Language Translation," in *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*. IEEE Computer Society, 1998, pp. 228–237.

[16] C. Cifuentes, "Interprocedural data flow decompilation," *J. Prog. Lang.*, vol. 4, no. 2, pp. 77–99, 1996.

[17] C. Cifuentes and K. J. Gough, "Decompilation of Binary Programs," *Softw., Pract. Exper.*, vol. 25, no. 7, pp. 811–829, 1995.

[18] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Logino, "Evolving exact decompilation," in *BAR 2018, Workshop on Binary Analysis Research, San Diego, California, USA, February 18, 2018*, 2018.

[19] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, S. T. King, Ed. USENIX Association, 2013, pp. 353–368.

[20] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, *BAP: A Binary Analysis Platform*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469.

[21] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA framework for binary code analysis," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 165–170.

[22] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. IEEE Computer Society, 2017, pp. 353–364.

[23] G. C. Necula, "Translation validation for an optimizing compiler," in *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, M. S. Lam, Ed. ACM, 2000, pp. 83–94.

[24] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, September 2016.

[25] A. Djoudi and S. Bardin, *BINSEC: Binary Code Analysis with Low-Level Regions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 212–217.

[26] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion, "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 2016, pp. 653–656.

[27] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking.*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 305–343.

[28] DWARF Debugging Information Format Committee, *DWARF Debugging Information Format 5*, 2017.

[29] A. Djoudi, S. Bardin, and É. Goubault, *Recovering High-Level Conditions from Binary Programs*. Cham: Springer International Publishing, 2016, pp. 235–253.

[30] D. Bühler, "Structuring an abstract interpreter through value and state abstractions:eva, an evolved value analysis for frama-c," Ph.D. dissertation, University of Rennes 1, France, 2017.

[31] P. Cousot and R. Cousot, "Abstract interpretation: past, present and future," in *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, T. A. Henzinger and D. Miller, Eds. ACM, 2014, pp. 2:1–2:10.

[32] P. Baudin, F. Bobot, L. Correnson, and Z. Dargaye, *WP Manual*, Frama-C Argon-20181129 ed., 2018.

[33] M. Zmyslowski, "Feeding the Fuzzers with KLEE," 2018.

[34] N. Carvalho, C. da Silva Sousa, J. S. Pinto, and A. Tomb, "Formal verification of klibc with the WP frama-c plug-in," in *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, ser. Lecture Notes in Computer Science, J. M. Badger and K. Y. Rozier, Eds., vol. 8430. Springer, 2014, pp. 343–358.

[35] J. Burghardt, J. Gerlach, and T. Lapawczyk, *ACSL By Example 17.2.0*, Fraunhofer FOKUS.

[36] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[37] S. Maus, "Verification of hypervisor subroutines written in assembler = verifikation von hypervisorunterrutinen, geschrieben in assembler," Ph.D. dissertation, University of Freiburg, Germany, 2011.

[38] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, *VCC: A Practical System for Verifying Concurrent C*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–42.

[39] S. Schmaltz and A. Shadrin, *Integrated Semantics of Intermediate-Language C and Macro-Assembler for Pervasive Formal Verification of Operating Systems and Hypervisors from VerisoftXT*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 18–33.

[40] A. Fehnker, R. Huuck, F. Rauch, and S. Seefried, "Some assembly required - program analysis of embedded system code," in *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, Sept 2008, pp. 15–24.

[41] M. O. Myreen, M. J. C. Gordon, and K. Slind, "Machine-code verification for multiple architectures - an application of decompilation into logic," in *2008 Formal Methods in Computer-Aided Design*, Nov 2008, pp. 1–8.

[42] B.-Y. E. Chang, M. Harren, and G. C. Necula, *Analysis of Low-Level Code Using Cooperating Decompilers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 318–335.

[43] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," in *NDSS*, 2011.

[44] E. Robbins, A. King, and T. Schrijvers, "From minx to minc: Semantics-driven decompilation of recursive datatypes," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 191–203.

[45] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 583–600.

[46] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 24–35.

[47] G. Balakrishnan and T. W. Reps, "WYSINWYX: what you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, 2010.

[48] S. Bardin, P. Herrmann, and F. Védrine, "Refinement-Based CFG Reconstruction from Unstructured Programs," in *Verification, Model*

Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 54–69.

[49] J. Kinder and D. Kravchenko, "Alternating Control Flow Reconstruction," in *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, ser. Lecture Notes in Computer Science, V. Kuncak and A. Rybalchenko, Eds., vol. 7148. Springer, 2012, pp. 267–282.

[50] T. Reinbacher and J. Brauer, "Precise control flow reconstruction using boolean logic," in *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, Eds. ACM, 2011, pp. 117–126.

[51] A. Sepp, B. Mihaila, and A. Simon, "Precise Static Analysis of Binaries by Extracting Relational Information," in *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, M. Pinzger, D. Poshyvanyk, and J. Buckley, Eds. IEEE Computer Society, 2011, pp. 357–366.

[52] A. Reid, "Trustworthy specifications of ARM® v8-A and v8-M system level architecture," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, pp. 161–168.

[53] J. G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," in *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, D. B. MacQueen and L. Cardelli, Eds. ACM, 1998, pp. 85–97.

[54] D. Patterson, J. Perconti, C. Dimoulas, and A. Ahmed, "Funtal: Reasonably mixing a functional language with assembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 495–509.

[55] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, H. Boehm and C. Flanagan, Eds. ACM, 2013, pp. 471–482.

[56] X. Rival, "Certification of compiled assembly code by invariant translation," *STTT*, vol. 6, no. 1, pp. 15–37, 2004.

[57] S. Blazy, B. Robillard, and A. W. Appel, "Formal verification of coalescing graph-coloring register allocation," in *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, ser. Lecture Notes in Computer Science, A. D. Gordon, Ed., vol. 6012. Springer, 2010, pp. 145–164.

[58] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, N. D. Jones and X. Leroy, Eds. ACM, 2004, pp. 14–25.

[59] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a doubt: testing for divergences between software versions," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 2016.

Get rid of inline assembly through verification-oriented lifting

Supplementary material

Frédéric Recoules*, Sébastien Bardin*, Richard Bonichon*, Laurent Mounier† and Marie-Laure Potet†
*CEA LIST, Paris-Saclay, France
firstname.lastname@cea.fr
†Univ. Grenoble Alpes. VERIMAG, Grenoble, France
firstname.lastname@univ-grenoble-alpes.fr

Appendix A
Motivating example

In Sec. II, Fig. 1c shows the code produced by TInA. Both Frama-C plugins EVA and WP were able to prove the absence of buffer overflow.

The Basic lifter on the code from Fig. 1a produces the code shown in Fig. 7. Even if the code remains small, the absence of simplifications adds a lot of unwanted complexity. Here, `__tina_df` is not propagated, thus the `__tina_edi` pointer is computed by a (extraneous) cast following bitwise operations. These operations actually destroys the analyzer's abstractions. On this very example, none of the Frama-C plugins was able to prove the absence of error.

Appendix B
Additional experiments: complementarity of optimization steps

Tables III, V and VI demonstrated the positive impact of stacking optimizations (levels $O1$ to $O4$) on common analyzers – recall that TInA is $O4$. What happens, now, if we remove any of the 3 initial optimizations from TInA? Here, we extend previous experiments with a new set of optimization levels:

- $\overline{O1}$: $O4$ - high-level predicate recovery;
- $\overline{O2}$: $O4$ - register unpacking;
- $\overline{O3}$: $O4$ - expression propagation.

Note that $\overline{O4}$ (i.e., $O4$ - loop normalization) is actually $O3$, hence this case is not discussed.

Symbolic execution. Table VII extends the results of Table III. We can observe that deactivating high-level predicate recovery or expression propagation leads to a drop in the number of paths explored equivalent to what their activation had gained.

Abstract interpretation. Table VIII extends the results of Table V. High-level predicate recovery is very important for abstract interpretation as common domains do not accurately handle flag computations. Any imprecision occurring at a loop exit point will generate a lot of alarms in the loop body and this is what we observe here. Register unpacking, in the other hand, has less impact in our experiments because imprecision

```
# 1074 "socklib.c"
int udpc_prepareForSelect
(int *socks, int nr, fd_set *read_set)
{
  /* [...] */
  int maxFd;
  {
    int __d0;
    int __d1;
    __fd_mask *__tina_4;
    unsigned int __tina_3;
    __tina_3 = sizeof(fd_set) / sizeof(__fd_mask);
    __tina_4 = & read_set->__fds_bits[0];
    {
      unsigned int __tina_ecx;
      unsigned int __tina_ebx;
      unsigned int __tina_edx_7;
      int __tina_df;
      __fd_mask *__tina_edi;
      __fd_mask *__tina_eax_12;
      __fd_mask *__tina_edx;
      unsigned int __tina_eax;
    __TINA_BEGIN_1__: ;
      __tina_ebx = 0;
      __tina_edx_7 = __tina_3;
      __tina_eax_12 = __tina_4;
      __tina_ecx = __tina_edx_7;
      __tina_edx = __tina_eax_12;
      __tina_edi = __tina_edx;
      __tina_eax = __tina_ebx;
      __tina_df = 0;
      while (! (0U == __tina_ecx)) {
        *__tina_edi = __tina_eax;
        __tina_edi = (__fd_mask *)
          ((!__tina_df - 1) & (unsigned int)(__tina_edi - 1)
           |
           (__tina_df - 1) & (unsigned int)(__tina_edi + 1));
        __tina_ecx --;
      }
    __TINA_END_1__: ;
    }
  }
  /* [...] */
  return maxFd;
}
```

Figure 7: C code generated by Basic lifter

Table VII: Impact of lifting strategies on KLEE (extended)

	Lifting								
	None	Basic	$O1$	$O2$	$O3$	$O4$	$\overline{O1}$	$\overline{O2}$	$\overline{O3}$
Functions 100% covered	✖	25	25	25	25	25	25	25	25
Aggregate time	N/A	115s	115s	110s	103s	105s	105s	105s	110s
# paths (all functions)	1.41M	1.50M +6%	1.83M +22%	4.59M +150%	6.64M +45%	6.62M ∼	5.42M -19%	4.57M -30%	4.59M -30%

on data do not affect the analysis as much as imprecision on the control flow.

Weakest precondition calculus. First, Table IX details the 12 functions under test for Frama-C in Sec. VII. Second, Table X extends the results of Table VI and shows that WP is sensitive to all optimizations outside of loop normalization. Removing any of the 3 other optimizations affects between 50% and 66% of the programs under study, rendering them non-provable.

Conclusion. Extended experiments show that the 4 optimizations are complementary and necessary to obtain the best of the analyzers. Register unpacking has the greatest impact on KLEE while high-level predicate recovery is necessary for the integer abstractions used in Frama-C EVA. Removing any optimization outside of loop normalization actually kills the effectiveness of Frama-C WP.

Table VIII: Impact of lifting strategies on EVA (extended)

	Lifting								
# Functions	None	Basic	$O1$	$O2$	$O3$	$O4$	$\overline{O1}$	$\overline{O2}$	$\overline{O3}$
without any alarms	✗	12	12	14	14	**19**	14	14	14
with ASM memory alarms	N/A	29	29	29	21	17	21	21	29
errors	✗	1	1	1	2	2	2	2	1
Aggregate of #									
emitted C alarms	231	184	184	177	177	177	177	184	177
emitted ASM alarms	N/A	316	244	199	165	128	253	171	199

Table IX: Functions under test for WP

Function	# Inst.	# Inv.	Description	Origin
saturated_sub	2	0	Maximum between 0 and big integer subtraction	ffmpeg, GMP
saturated_add	2	0	Minimum between MAX_UINT and big integer addition	ffmpeg, GMP
log2	1	5	Biggest power of 2 of an integer	libgcrypt
mid_pred	7	0	Median of 3 inputs	ffmpeg
strcmpeq	9	6	String equality testing	ASM snippet
strnlen	16	6	String length (or buffer length if no `'\0'`)	ASM snippet
memset	9	5	Set array contents to input	UDPCast
count	8	4	Count occurrences of inputs in array	ACSL by example [35]
max_element	10	7	First index of max element of the array	ACSL by example
cmp_array	10	6	Array equality testing (SIMD)	ACSL by example
sum_array	20	7	Sum of array elements (SIMD)	ACSL by example
SumSquareError	24	69	Sum of square differences between two arrays (SIMD)	libyuv

Table X: Impact of lifting strategies on WP (extended).

	Lifting								
Function	None	Basic	$O1$	$O2$	$O3$	$O4$	$\overline{O1}$	$\overline{O2}$	$\overline{O3}$
saturated_sub	✗	✓	✓	✓	✓	✓	✓	✓	✓
saturated_add	✗	✗	✓	✓	✓	✓	✓	✓	✓
log2	✗	✗	✗	✗	✓	✓	✓	✓	✗
mid_pred	✗	✗	✓	✓	✓	✓	✗	✓	✓
strcmpeq	✗	✗	✗	✗	✓	✓	✓	✗	✗
strnlen	✗	✗	✗	✗	✓	✓	✗	✗	✗
memset	✗	✗	✗	✗	✓	✓	✓	✗	✗
count	✗	✗	✗	✗	✓	✓	✗	✓	✗
max_element	✗	✗	✓	✓	✓	✓	✗	✓	✓
cmp_array	✗	✗	✗	✗	✓	✓	✗	✗	✗
sum_array	✗	✗	✗	✗	✓	✓	✗	✗	✗
SumSquareError	✗	✗	✗	✗	✓	✓	✗	✗	✗

APPENDIX C
ADDITIONAL EXPERIMENTS: SIZE OF PRODUCED CODE

The size of the generated code w.r.t the size of the assembly chunk is indicative of what one can expect of the code produced by TINA. Since the size of the C code is also correlated with the quality of subsequent analyses when produced by TINA, this information becomes more notable. Broadly speaking, the smaller the code, the easier it will be to analyze and the better the analysis will be.

Table XI: Ratio between C statements and assembly instructions w.r.t lifting strategies (all chunks, x86).

	LIFTING							
	BASIC	$O1$	$O2$	$O3$	$O4$	$\overline{O1}$	$\overline{O2}$	$\overline{O3}$
Min	0.36	0.36	0.55	0.23	0.23	0.23	0.23	0.55
Average	2.87	2.86	6.03	0.98	0.98	0.98	1.06	6.03
Max	11.54	11.54	55.58	19.35	19.35	19.35	11.08	55.58

Table XI shows the ratio between the number of generated C statements and the number of assembly instructions. As expected, two optimizations significantly change the ratio. First, register unpacking increases the number of generated statements as it splits single assignments into many smaller assignments of independent values. On the other hand, expression propagation greatly reduces the number of statements: on average, this number decreases by a factor of 6 when it comes from register unpacking and by a factor of 2.5 in other cases. In the end, TINA roughly generates one C statement per assembly instruction for the chunks of the Debian distribution.

APPENDIX D
ffmpeg BUG EXAMPLE

```
# 25 "libavcodec/lossless_videoencdsp.h"
typedef struct LLVidEncDSPContext {
/* [...] */

  /**
   * Subtract HuffYUV's variant of median prediction.
   * Note, this might read from src1[-1], src2[-1].
   */
  void (*sub_median_pred)(uint8_t *dst, const uint8_t *src1,
                          const uint8_t *src2, intptr_t w,
                          int *left, int *left_top);

/* [...] */
} LLVidEncDSPContext;
```

(a) Function API

```
1  # 44 "libavcodec/x86/lossless_videoencdsp_init.c"
2  void sub_median_pred_mmxext(uint8_t *dst, const uint8_t *src1,
3                              const uint8_t *src2, intptr_t w,
4                              int *left, int *left_top)
5  {
6      x86_reg i = 0;
7      uint8_t l, lt;
8
9      __asm__ volatile (
10         "movq  (%1, %0), %%mm0           \n\t" // LT
11         "psllq $8, %%mm0                 \n\t"
12         "1:                              \n\t"
13         "movq  (%1, %0), %%mm1           \n\t" // T
14         "movq  -1(%2, %0), %%mm2         \n\t" // L
15         "movq  (%2, %0), %%mm3           \n\t" // X
16         "movq %%mm2, %%mm4               \n\t" // L
17         "psubb %%mm0, %%mm2              \n\t"
18         "paddb %%mm1, %%mm2              \n\t" // L + T - LT
19         "movq %%mm4, %%mm5               \n\t" // L
20         "pmaxub %%mm1, %%mm4             \n\t" // max(T, L)
21         "pminub %%mm5, %%mm1             \n\t" // min(T, L)
22         "pminub %%mm2, %%mm4             \n\t"
23         "pmaxub %%mm1, %%mm4             \n\t"
24         "psubb %%mm4, %%mm3              \n\t" // dst - pred
25         "movq %%mm3, (%3, %0)            \n\t"
26         "add $8, %0                      \n\t"
27         "movq -1(%1, %0), %%mm0          \n\t" // LT
28         "cmp %4, %0                      \n\t"
29         " jb 1b                          \n\t"
30         : "+r" (i)
31         : "r" (src1), "r" (src2),
32           "r" (dst), "r" ((x86_reg) w));
33
34     l = *left;
35     lt = *left_top;
36
37     dst[0] = src2[0] -
38       mid_pred(l, src1[0], (l + src1[0] - lt) & 0xFF);
39
40     *left_top = src1[w - 1];
41     *left = src2[w - 1];
42 }
```

(b) Original version

Sec. VII-D refers to a ffmpeg function accessing index `-1` of its input buffer. It is part of the lossless video encryption provided by ffmpeg. More specifically, it is a field of `struct LLVidEncDSPContext` (Fig. 8a) which is initialized dynamically, depending on the hardware capabilities. Comments show that the developers know of this behavior. Fig. 8b and 8c show the original and generated version respectively of the function implementation using `mmxext` extensions. The latter has been sliced for readability reasons and the ellipsis stands for repetitive patterns. (each MMX instruction is translated to 8 independent C statements) The `-1` access occurs during the first iteration of the loop, when `%0` (`eax`) is `0`, at line 14 (Fig. 8b) in the assembly chunk. This bug is consequently found at line 27 in the lifted C version (Fig. 8c).

Since the situation is acknowledged in the code documentation, it is arguably not a bug *sensu stricto*. However, the function is exported (not static), does not contain defensive programming to avoid the bad behavior, and the documentation is not directly on the function itself but in a record containing a possible function pointer to this function. Thus, we consider this a serious programming flaw that could lead to issues down the line in several situations: code maintenance and refactoring, code reuse (in other projects), compiler upgrades (taking advantages of potential undefined behaviors to trigger more aggressive code optimizations), etc.

Compliance issues. Moreover, the chunk interface (Fig. 8b, lines 30-32) misses information about clobbering the `mm` registers 0 to 5 and accessing memory from `src1`, `src2` and `dst`. It is actually not an issue *here* as caller functions assumes (according to the Application Binary Interface) that memory and `mm` registers are clobbered. However, it is not easy to know in advance how compilers will handle function inlining in terms of memory write barriers and register clobbering. The missing information at the interface of the chunk for clobbered entities could thus lead to serious issues when compiling in a different environment. Similar concerns arise in the cases of code reuse, especially in another project.

```c
void sub_median_pred_mmxext(uint8_t *dst, uint8_t const *src1,
                            uint8_t const *src2, intptr_t w, int *left,
                            int *left_top)
{
  uint8_t l;
  uint8_t lt;
  int tmp;
  x86_reg i = 0;
  {
    unsigned int __atoc_eax;
    uint8_t __atoc_mm0_0_7 /* , __atoc_mm0_8_15, ..., __atoc_mm0_56_63 */;
    uint8_t __atoc_mm1_0_7 /* , __atoc_mm1_8_15, ..., __atoc_mm1_56_63 */;
    uint8_t __atoc_mm2_0_7 /* , __atoc_mm2_8_15, ..., __atoc_mm2_56_63 */;
    /* uint8_t __atoc_mm3_56_63; */
    uint8_t __atoc_tmp44, __atoc_tmp52, __atoc_tmp60, __atoc_tmp68 /* ... */;
    __ATOC_BEGIN_2__: ;
    __atoc_eax = i;
    __atoc_mm0_0_7 = 0;
    /* __atoc_mm0_8_15 = *(src1 + i); */
    /* [ ... ] */
    /* __atoc_mm0_56_63 = *((src1 + i) + 6); */
    while (1) {
      __atoc_mm1_0_7 = *(src1 + __atoc_eax);
      /* __atoc_mm1_8_15 = *((src1 + __atoc_eax) + 1); */
      /* [ ... ] */
      /* __atoc_mm1_56_63 = *((src1 + __atoc_eax) + 7); */
      __atoc_mm2_0_7 = *((src2 + __atoc_eax) - 1);
      __atoc_mm2_8_15 = *(src2 + __atoc_eax);
      /* [ ... ] */
      /* __atoc_mm2_56_63 = *((src2 + __atoc_eax) + 6); */
      /* __atoc_mm3_56_63 = *((src2 + __atoc_eax) + 7); */
      /* if ((int)__atoc_mm2_56_63 > (int)__atoc_mm1_56_63) __atoc_tmp.. = __atoc_mm2_56_63; */
      /* else __atoc_tmp.. = __atoc_mm1_56_63; */
      /* [ ... ] */
      if ((int)__atoc_mm2_0_7 > (int)__atoc_mm1_0_7) __atoc_tmp52 = __atoc_mm2_0_7;
      else __atoc_tmp52 = __atoc_mm1_0_7;
      /* if ((int)__atoc_mm1_56_63 < (int)__atoc_mm2_56_63) __atoc_tmp.. = __atoc_mm1_56_63; */
      /* else __atoc_tmp.. = __atoc_mm2_56_63; */
      /* [ ... ] */
      if ((int)__atoc_mm1_0_7 < (int)__atoc_mm2_0_7) __atoc_tmp44 = __atoc_mm1_0_7;
      else __atoc_tmp44 = __atoc_mm2_0_7;
      /* if ((int).. < ((int)__atoc_mm2_56_63 - (int)__atoc_mm0_56_63) + (int)__atoc_mm1_56_63) */
      /*   __atoc_tmp.. = __atoc_tmp..; */
      /* else __atoc_tmp.. = ((int)__atoc_mm2_56_63 - (int)__atoc_mm0_56_63) + (int)__atoc_mm1_56_63; */
      /* [ ... ] */
      if ((int)__atoc_tmp52 < ((int)__atoc_mm2_0_7 - (int)__atoc_mm0_0_7) + (int)__atoc_mm1_0_7)
        __atoc_tmp68 = __atoc_tmp52;
      else __atoc_tmp68 = ((int)__atoc_mm2_0_7 - (int)__atoc_mm0_0_7) + (int)__atoc_mm1_0_7;
      /* if ((int)__atoc_tmp.. > (int)__atoc_tmp..) __atoc_tmp.. = __atoc_tmp..; */
      /* else __atoc_tmp.. = __atoc_tmp..; */
      /* [ ... ] */
      if ((int)__atoc_tmp68 > (int)__atoc_tmp44) __atoc_tmp60 = __atoc_tmp68;
      else __atoc_tmp60 = __atoc_tmp44;
      *(dst + __atoc_eax) = (int)__atoc_mm2_8_15 - (int)__atoc_tmp60;
      /* *((dst + __atoc_eax) + 1) = (int)__atoc_mm2_16_23 - (int)__atoc_tmp..; */
      /* [ ... ] */
      /* *((dst + __atoc_eax) + 6) = (int)__atoc_mm2_56_63 - (int)__atoc_tmp..; */
      /* *((dst + __atoc_eax) + 7) = (int)__atoc_mm3_56_63 - (int)__atoc_tmp.. */
      __atoc_eax += 8U;
      __atoc_mm0_0_7 = *((src1 + __atoc_eax) - 1);
      /* __atoc_mm0_8_15 = *(src1 + __atoc_eax); */
      /* [ ... ] */
      /* __atoc_mm0_56_63 = *((src1 + __atoc_eax) + 6); */
      if (__atoc_eax >= (unsigned int)w) break;
    }
    __ATOC_END_2__: ;
  }
  l = (unsigned char)*left;
  lt = (unsigned char)*left_top;
  tmp = mid_pred((int)l,(int)*(src1 + 0),
                 (((int)l + (int)*(src1 + 0)) - (int)lt) & 0xFF);
  *(dst + 0) = (unsigned char)((int)*(src2 + 0) - tmp);
  *left_top = (int)*(src1 + (w - 1));
  *left = (int)*(src2 + (w - 1));
  return;
}
```

(c) TINA-generated version

Figure 8: ffmpeg function accessing `src2[-1]`

APPENDIX E
TECHNICAL FOCUS ON SIMPLIFICATION RULES

We use a mixture of standard and dedicated simplification rules – standard for typical integer-level properties and dedicated for more low-level aspects. In the rules below, we use the following notations $|x|$ is the size of the expression x, $\diamond$ any binary operator, C a condition ($|C| = 1$), k is a constant. $\vec{1}_{|x|}$ denotes a bitvector of size $|x|$ with all bits set to 1.

- standard "lightweight" term normalization in order to ease further other simplifications, including common subexpression elimination (a.k.a. sharing), unused variables elimination, associativity-commutativity re-ordering.
- constant propagation (modular arithmetic)
- neutral elements:

$$\begin{aligned} x + 0 &\hookrightarrow x & x \wedge \vec{1}_{|x|} &\hookrightarrow x \\ x - 0 &\hookrightarrow x & x \vee 0 &\hookrightarrow x \\ x \times 1 &\hookrightarrow x & x \oplus 0 &\hookrightarrow x \\ x \text{ udiv } 1 &\hookrightarrow x & x \text{ shl } 0 &\hookrightarrow x \\ x \text{ sdiv } 1 &\hookrightarrow x & x \text{ shr } 0 &\hookrightarrow x \\ x \text{ urem } 2^{|x|} &\hookrightarrow x & x \text{ sar } 0 &\hookrightarrow x \end{aligned}$$

- idempotence:

$$\begin{aligned} x \wedge x &\hookrightarrow x & \text{sext}_{|x|}(x) &\hookrightarrow x \\ x \vee x &\hookrightarrow x & \text{extract}_{0..|x|-1}(x) &\hookrightarrow x \\ \text{uext}_{|x|}(x) &\hookrightarrow x \end{aligned}$$

- absorbing(-like) elements:

$$\begin{aligned} x \times 0 &\hookrightarrow 0 & x \text{ urem } 1 &\hookrightarrow 0 \\ x \wedge 0 &\hookrightarrow 0 & x \text{ srem } 1 &\hookrightarrow 0 \\ x \vee 1 &\hookrightarrow 1 \end{aligned}$$

- inverse elements:

$$\begin{aligned} x - x &\hookrightarrow 0 & x \oplus x &\hookrightarrow 0 \\ x \text{ udiv } x &\hookrightarrow 1 & x \text{ sdiv } x &\hookrightarrow 1 \\ x \text{ shl } k &\overset{|x|\leq k}{\hookrightarrow} 0 & x \text{ shr } k &\overset{|x|\leq k}{\hookrightarrow} 0 \end{aligned}$$

- involutivity:

$$\neg(\neg x) \hookrightarrow x \qquad -(-x) \hookrightarrow x$$

- double shift simplifications:

$$\begin{aligned} (x \text{ shl } y) \text{ shl } z &\hookrightarrow x \text{ shl } (y + z) \\ (x \text{ shr } y) \text{ shr } z &\hookrightarrow x \text{ shr } (y + z) \\ (x \text{ sar } y) \text{ sar } z &\hookrightarrow x \text{ sar } (y + z) \end{aligned}$$

- remainder / extension subsumption:

$$\begin{aligned} (x \text{ urem } k) \text{ urem } k' &\overset{k\leq k'}{\hookrightarrow} x \text{ urem } k \\ (x \text{ urem } k) \text{ urem } k' &\overset{k\geq k'}{\hookrightarrow} x \text{ urem } k' \\ \text{sext}_k(\text{uext}_{k'}(x)) &\overset{k'>|x|}{\hookrightarrow} \text{uext}_k(x) \\ \text{uext}_k(\text{uext}_{k'}(x)) &\hookrightarrow \text{uext}_k(x) \end{aligned}$$

- condition simplifications

$$\begin{aligned} C = 1 &\hookrightarrow C & x >_u x &\hookrightarrow 0 \\ C \neq 0 &\hookrightarrow C & x <_u x &\hookrightarrow 0 \\ C > 0 &\hookrightarrow C & x >_s x &\hookrightarrow 0 \\ x = x &\hookrightarrow 1 & x <_s x &\hookrightarrow 0 \\ x \neq x &\hookrightarrow 0 \end{aligned}$$

- (extended) De Morgan simplifications

$$\begin{aligned} \neg(C \wedge C') &\hookrightarrow \neg C \vee \neg C' & \neg(x >_u y) &\hookrightarrow y \geq_u x \\ \neg(C \vee C') &\hookrightarrow \neg C \wedge \neg C' & \neg(x \geq_u y) &\hookrightarrow y >_u x \\ x \oplus \vec{1}_{|x|} &\hookrightarrow \neg x & \neg(x <_s y) &\hookrightarrow y \leq_s x \\ \neg(x = y) &\hookrightarrow x \neq y & \neg(x \leq_s y) &\hookrightarrow y <_s x \\ \neg(x \neq y) &\hookrightarrow x = y & \neg(x >_s y) &\hookrightarrow y \geq_s x \\ \neg(x <_u y) &\hookrightarrow y \leq_u x & \neg(x \geq_s y) &\hookrightarrow y >_s x \\ \neg(x \leq_u y) &\hookrightarrow y <_u x \end{aligned}$$

- ternary expression simplification:

$$\begin{aligned} C \ ?\ false : true &\hookrightarrow \neg C \\ \neg C \ ?\ x : y &\hookrightarrow C \ ?\ y : x \\ C \ ?\ true : false &\hookrightarrow C \\ C \ ?\ x : x &\hookrightarrow x \\ x \diamond (C \ ?\ y : z) &\hookrightarrow C \ ?\ x \diamond y : x \diamond z \\ (C \ ?\ w : x) \diamond (C \ ?\ y : z) &\hookrightarrow C \ ?\ w \diamond y : x \diamond z \end{aligned}$$

- split elements:

$$\begin{aligned} k = \text{concat}(x, y) &\hookrightarrow (\text{extract}_{|y|..|x|+|y|-1}(k) = x) \\ &\quad \wedge (\text{extract}_{0..|y|-1}(k) = y) \\ k \neq \text{concat}(x, y) &\hookrightarrow (\text{extract}_{|y|..|x|+|y|-1}(k) \neq x) \\ &\quad \vee (\text{extract}_{0..|y|-1}(k) \neq y) \end{aligned}$$

- concatenation abstraction:

$$\begin{aligned} \text{uext}_{|x|+|y|}(x) \vee \text{concat}(y, 0_{|x|}) &\hookrightarrow \text{concat}(y, x) \\ \text{uext}_{|x|+k}(x) \text{ shl } k &\hookrightarrow \text{concat}(x, 0_k) \\ \text{concat}(0_k, x) &\hookrightarrow \text{uext}_{k+|x|}(x) \end{aligned}$$

- extraction simplification:

$$\text{extract}_{i..j}(\text{extract}_{k..l}(x)) \hookrightarrow \text{extract}_{i+k..j+k}(x)$$
$$\text{concat}(\text{extract}_{i..j}(x), \text{extract}_{j+1..k}(x)) \hookrightarrow \text{extract}_{i..k}(x)$$
$$\text{extract}_{i..j}(\text{uext}_k(x)) \overset{|x|\leq i}{\hookrightarrow} 0$$
$$\text{extract}_{0..|x|-1}(x) \hookrightarrow x$$
$$\text{extract}_{i..j}(\text{concat}(x,y)) \overset{j<|y|}{\hookrightarrow} \text{extract}_{i..j}(y)$$
$$\text{extract}_{i..j}(\text{concat}(x,y)) \overset{|y|\leq i}{\hookrightarrow} \text{extract}_{i-|y|..j-|y|}(x)$$
$$\text{extract}_{0..j}(\text{uext}_k(x)) \overset{|x|\leq j}{\hookrightarrow} \text{uext}_j(x)$$
$$\text{extract}_{i..j}(\text{uext}_k(x)) \overset{j<|x|}{\hookrightarrow} \text{extract}_{i..j}(x)$$
$$\text{extract}_{0..j}(\text{sext}_k(x)) \overset{|x|\leq j}{\hookrightarrow} \text{sext}_j(x)$$
$$\text{extract}_{i..j}(\text{sext}_k(x)) \overset{j<|x|}{\hookrightarrow} \text{extract}_{i..j}(x)$$

- two-complement arithmetic abstraction:

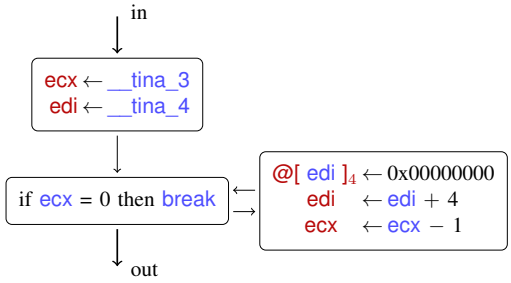
$$\neg x + 1 \hookrightarrow -x$$
$$(\text{uext}_k(x) \oplus 2^{|x|-1}) - 2^{|x|-1} \hookrightarrow \text{sext}_k(x)$$
$$\text{extract}_{|x|-1}(x) \hookrightarrow x <_s 0$$
$$\text{uext}_n(C) - 1 \hookrightarrow C \ ? \ 0 : \vec{1}_n$$
$$\text{sext}_n(C) \hookrightarrow C \ ? \ \vec{1}_n : 0$$

Appendix F
Loop Normalization Example

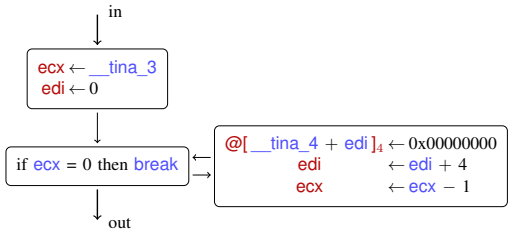
Fig. 9b to 9e illustrate the 3 steps on the motivating example. The `ecx` register stands as the loop counter, the `edi` register is rebased, rescaled and unified with `ecx`. Before merging `edi` with `ecx`, the following relation is recorded: `edi` $\equiv$ `__tina_4` + 4 × (`__tina_3` - `ecx`).

Appendix G
Validation Example

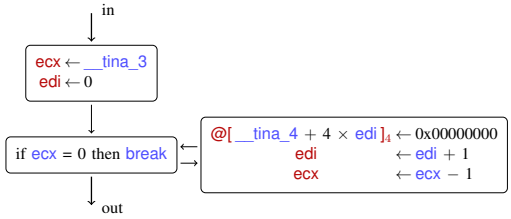
Fig. 10 shows the translation validation of the running example (Fig. 1). Codes before and after lifting have the same form (**S1**), as shown in Fig. 10a. Original blocks B_0, B_1 and B_2 have been paired respectively with their lifted counterparts B'_0, B'_1 and B'_2 for **S2**. Let us focus here on the equivalence check between B_2 and B'_2 (Fig. 10b). The two blocks are obviously syntactically different, due to simplification and recompilation. Compilation splits complex expressions using general registers as temporary variables while the ones from B_2 have been lifted to C variables with a close but different name. For instance `__tina_ecx` is equal to `ecx` whereas `eax` is used differently in B_2 and B'_2. Due to simplification passes, the two blocks no more have the same number of inputs or outputs, because lifting inferred some constraints and removed unused variables. Thus, `eax` and `df` are actually 0 upon entering B_2. As pointer `edi` is incremented by 4 while `ecx` is decremented by 1, the two variables are linked by the linear relation `edi` = `tina_4` + 4 × (`tina_3` − `ecx`). The formula is built to take into account these differences by adding logical assertions for each inferred constraint. In the end, the equivalence query is then discharged by a SMT solver. An (expected) unsat answer shows that there is no model such that the observable behaviors of the two basic blocks can differ.



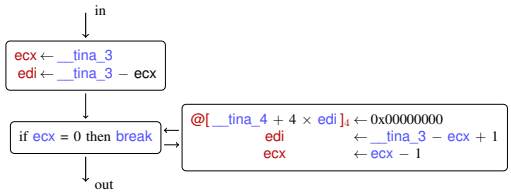
(a) Post $O3$



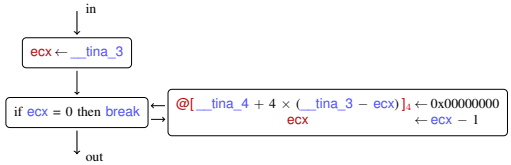
(b) Step 1: rebasing



(c) Step 2: rescaling

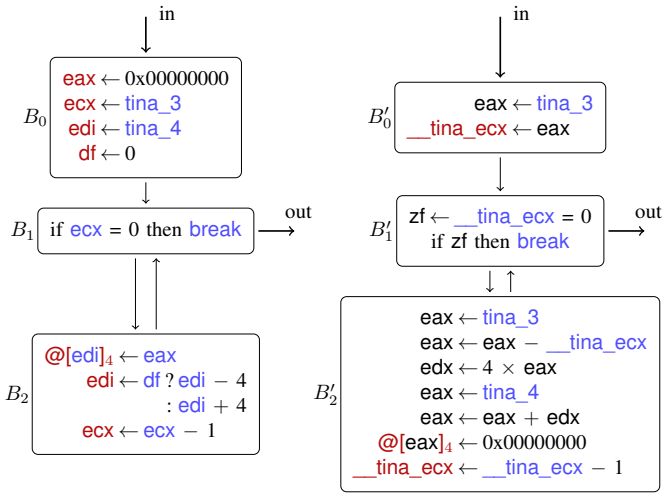


(d) Step 3: pre-merging



(e) Step 3: post-merging

Figure 9: Loop normalization



(a) Control flow graphs

; B_2 instructions sequence

$$
\begin{aligned}
\text{memory}_{\text{out}} &= \text{store}_4\ \text{memory}_{\text{in}}\ \text{edi}_{\text{in}}\ \text{eax}_{\text{in}} \\
\text{edi}_{\text{out}} &= \text{if } \text{df}_{\text{in}} \text{ then } \text{edi}_{\text{in}} - 4 \text{ else } \text{edi}_{\text{in}} + 4 \\
\text{ecx}_{\text{out}} &= \text{ecx}_{\text{in}} + 1
\end{aligned}
$$

; B'_2 instructions sequence

$$
\begin{aligned}
\text{eax'}_0 &= \text{tina\_3}_{\text{in}} \\
\text{eax'}_1 &= \text{eax'}_0 - \text{\_\_tina\_ecx}_{\text{in}} \\
\text{edx'}_0 &= 4 \times \text{eax'}_1 \\
\text{eax'}_2 &= \text{tina\_4}_{\text{in}} \\
\text{eax'}_3 &= \text{eax'}_2 + \text{edx'}_0 \\
\text{memory'}_{\text{out}} &= \text{store}_4\ \text{memory}_{\text{in}}\ \text{eax'}_3\ 0\text{x}00000000 \\
\text{\_\_tina\_ecx}_{\text{out}} &= \text{\_\_tina\_ecx}_{\text{in}} - 1
\end{aligned}
$$

; inputs restrictions

$$
\begin{aligned}
\text{eax}_{\text{in}} &= 0\text{x}00000000 \\
\text{df}_{\text{in}} &= \text{false} \\
\text{edi}_{\text{in}} &= \text{tina\_4}_{\text{in}} + 4 \times (\text{tina\_3}_{\text{in}} - \text{ecx}_{\text{in}}) \\
\text{ecx}_{\text{in}} &= \text{\_\_tina\_ecx}_{\text{in}}
\end{aligned}
$$

; outputs restrictions

$$
\begin{aligned}
\text{ecx'}_{\text{out}} &= \text{\_\_tina\_ecx}_{\text{out}} \\
\text{edi'}_{\text{out}} &= \text{tina\_4}_{\text{in}} + 4 \times (\text{tina\_3}_{\text{in}} - \text{ecx'}_{\text{out}})
\end{aligned}
$$

; outputs assertion

$$
\text{memory}_{\text{out}} \neq \text{memory'}_{\text{out}} \vee \text{ecx}_{\text{out}} \neq \text{ecx'}_{\text{out}} \vee \text{edi}_{\text{out}} \neq \text{edi'}_{\text{out}}
$$

(b) Logical formula for the equivalence between B_2 and B'_2

Figure 10: Basic block equivalence