Interface Compliance of Inline Assembly: Automatically Check, Patch and Refine

Frédéric Recoules  
Univ. Paris-Saclay, CEA, List Saclay, France  
frederic.recoules@cea.fr

Sébastien Bardin  
Univ. Paris-Saclay, CEA, List Saclay, France  
sebastien.bardin@cea.fr

Richard Bonichon  
Tweag I/O Paris, France  
richard.bonichon@gmail.com

Matthieu Lemerre  
Univ. Paris-Saclay, CEA, List Saclay, France  
mattieu.lemerre@cea.fr

Laurent Mounier  
Univ. Grenoble Alpes, VERIMAG Grenoble, France  
laurent.mounier@univ-grenoble-alpes.fr

Marie-Laure Potet  
Univ. Grenoble Alpes, VERIMAG Grenoble, France  
marie-laure.potet@univ-grenoble-alpes.fr

Abstract—Inline assembly is still a common practice in low-level C programming, typically for efficiency reasons or for accessing specific hardware resources. Such embedded assembly codes in the GNU syntax (supported by major compilers such as GCC, Clang and ICC) have an interface specifying how the assembly codes interact with the C environment. For simplicity reasons, the compiler treats GNU inline assembly codes as blackboxes and relies only on their interface to correctly glue them into the compiled C code. Therefore, the adequacy between the assembly chunk and its interface (named compliance) is of primary importance, as such compliance issues can lead to subtle and hard-to-find bugs. We propose RUSTINA, the first automated technique for formally checking inline assembly compliance, with the extra ability to propose (proven) patches and (optimization) refinements in certain cases. RUSTINA is based on an original formalization of the inline assembly compliance problem together with novel dedicated algorithms. Our prototype has been evaluated on 202 Debian packages with inline assembly (2656 chunks), finding 2183 issues in 85 packages – 986 significant issues in 54 packages (including major projects such as ffmpeg or ALSA), and proposing patches for 92% of them. Currently, 38 patches have already been accepted (solving 156 significant issues), with positive feedback from development teams.

I. INTRODUCTION

Context. Inline assembly, i.e., embedding assembly code inside a higher-level host language, is still a common practice in low-level C/C++ programming, for efficiency reasons or for accessing specific hardware resources – it is typically widespread in resource-sensitive areas such as cryptography, multimedia, drivers, system, automated trading or video games [1], [2]. Recoules et al. [1] estimate that 11% of Debian packages written in C/C++ directly or indirectly depend on inline assembly, including major projects such as GMP or ffmpeg, while 28% of the top rated C projects on GitHub contain inline assembly according to Rigger et al. [2].

Thus, compilers supply a syntax to embed assembly instructions in the source program. The most widespread is the GNU inline assembly syntax, driven by GCC but also supported by Clang or ICC. The GNU syntax provides an interface specifying how the assembly code interacts with the C environment. The compiler then treats GNU inline assembly codes as blackboxes and relies only on this interface to correctly insert them into the compiled C code.

Problem. The problem with GNU inline assembly is twofold. First, it is hard to write correctly: inline assembly syntax [4] is not beginner-friendly, the language itself is neither standardized nor fully described, and some corner cases are defined by GCC implementation (with occasional changes from time to time). Second, assembly chunks are treated as blackboxes, so that the compiler does not do any sanity checks and assumes the embedded assembly code respects its declared interface.

Hence, in addition to usual functional bugs in the assembly instructions themselves, inline assembly is also prone to interface compliance bugs, i.e., mismatches between the declared interface and the real behavior of the assembly chunk which can lead to subtle and hard-to-find bugs – typically incorrect results or crashes due to either subsequent compiler optimizations or ill-chosen register allocation. In the end, compliance issues can lead to severe bugs (segfault, deadlocks, etc.) and, as they depend on low-level compiler choices, they are hard to identify and can hide for years before being triggered by a compiler update. For example, a 2005 compliance bug introduced in the libatomic_obs library of lock-free primitives for multithreading made deadlocks possible; it was identified and patched only in 2010 (commit 03e48e1). A similar bug was still lurking in another primitive in 2020 until we automatically found and patched it (commit 05812c2). We also found a 1997 interface compliance bug in glibc (leading to a segfault in a string primitive) that was patched in 1999 (commit 7c97add), then reintroduced in 2002 after refactoring.

Goal and challenges. We address the challenge of helping developers write safer inline assembly code by designing and implementing RUSTINA, a formal technique for automatically checking inline assembly compliance with positive feedback from development teams.

1Microsoft inline assembly is different and has no interface, see Sec. IX-C.
2From the llvm-dev mailing list [3]: “GCC-style inline assembly is notoriously hard to write correctly.”
3Note that syntactically incorrect assembly instructions are caught during the translation from assembly to machine code.
developing automated techniques helping to achieve interface compliance, i.e. ensuring that both the assembly template and its interface are consistent with each other. This is challenging for several reasons:

**Define** The method must be built on a (currently missing) proper formalization of interface compliance, both realistic and amenable to automated formal verification;

**Check, Patch & Refine** The method must be able to check whether an assembly chunk complies with its interface, but ideally it should also be able to automatically suggest patches for bugs or code refinements;

**Wide applicability** The method must be generic enough to encompass several architectures, at least x86 and ARM.

Fehnker et al. [5] published the only attempt we know of to inspect the interface written by the developer. Yet, their definition of interface compliance is syntactic and incomplete – for example they cannot detect the glbit issue mentioned above. Moreover, they do not cover all subtilties of GCC inline assembly (e.g., token constraints), consider only compliance checking (neither patching nor refinement) and the implementation is tightly bound to ARM (much simpler than x86).

Note that recent attempts for verifying codes mixing C and assembly [1], [6] simply assume interface compliance.

**Proposal and contributions.** We propose RUSTINA, the first sound technique for comprehensive automated interface compliance checking, automated patch synthesis and interface refinements. We claim the following contributions:

- a novel **semantic and comprehensive** formalization of the problem of interface compliance (Sec. IV), amenable to formal verification;
- a new **semantic** method (Sec. V) to automatically verify the compliance of inline assembly chunks, to generate a corrective patch for the majority of compliance issues and additionally to suggest interface refinements;
- thorough experiments (Sec. VII) of a prototype implementation (Sec. VI) on a large set of x86 real-world examples (all inline assembly found in a Debian Linux distribution) demonstrate that RUSTINA is able to automatically check and curate a large code base (202 packages, 2640 assembly chunks) in a few minutes, detecting 2036 issues and solving 95% of them;
- a study of current inline assembly coding practices (Sec. VIII); besides identifying the common **compliance** issues found in the wild (Sec. VII-A), we also exhibit 6 recurring patterns leading to the vast majority (97%) of compliance issues and show that 5 of them rely on **fragile** assumptions and can lead to serious bugs (Sec. VIII).

As the time of writing, 38 patches have already been accepted by 7 projects, solving 156 significant issues (Sec. VII-C).

**Summary.** Inline assembly is a delicate practice. RUSTINA aids developers in achieving interface compliant inline assembly code. Compliant assembly chunks can still be buggy but RUSTINA automatically removes a whole class of problems.

Our technique has already helped several renowned projects fix code, with positive feedback from developers.

**Note:** supplementary material, including prototype and benchmark data, is available online [7].

## II. Context and motivation

The code in Fig. 1 is an extract from libatomic_obs, commit 30ce1ab dating back to early 2012. It was replaced 6 months later by commit 64d18cd because it led to a segmentation fault when compiled with Clang. By 2020, another latent bug was still lurking until automatically discovered and patched by our prototype RUSTINA (commit 05812c2).

**What the code is about.** This function uses inline assembly to implement the standard atomic primitive Compare And Swap – i.e. write new_val in +addr if this latter still equals to old_val (where 8-byte values old_val and new_val are split in 4-byte values old_val1, old_val2, etc.). The assembly statement (syntax discussed in Sec. III) comprises assembly instructions (e.g., "lock; cmpxchg8b %0;") building an assembly template where some operands have been replaced by **tokens** (e.g., %0) that will be latter assigned by the compiler. It also has a specification, the **interface**, binding together assembly registers, tokens and C expressions: line 196 declares the **outputs**, i.e. C values expected to be assigned by the chunk; lines 197 and 198 declare the **inputs**, i.e. C values the compiler should pass to the chunk. The string placed before a C expression is called a **constraint** and indicates the set of possible assembly operands this expression can be bound to by the compiler. For instance, "d" (old_val12) indicates that register %edx should be initialized with the value of old_val12, while "a" (result) indicates the value of result should be collected from %eax. Token %0 introduced by "m" (+addr) is an indirect memory access: its address, arbitrarily denoted %0 here, can be bound to several possibilities (cf. Fig. 5) – including %esi or %ebx.

Fig. 2 gives the functional meaning of this binding along with the semantics of the assembly instructions (where "..." is the concatenation, "%...%" a conditional assignment, "e\(\{0..n\}\)" the bits extraction and "zext\(n\)" the zero extension to size \(n\)).

This example allows us to introduce the concept of interface compliance issues and the associated miscompilation problems: A) (framing condition) incomplete interfaces, possibly
leading to miscompilations due to wrong data dependencies; B) (unicity) ambiguous interfaces, where the result depends on compiler choices for token allocation.

A) An incomplete frame definition. Here, register %edx is declared as read-only (by default, non-output locations are) whereas it is overwritten by instruction cmpxchg8b %0 (c.f. Fig. 2). %edx should be declared as output as well.

Impact: The compiler exclusively relies on the interface to know the framing-condition – i.e. which locations are read or written. When this information is incomplete, data dependencies are miscalculated, potentially leading to incorrect optimizations. Here, the compiler believes %edx still contains old_val12 after the assembly chunk is executed, while it is not the case.

Note that %ebx and %esi are not missing the output attribute: while overwritten by the xchg instructions, they are then restored to their initial value.

B) Ambiguous interface. Here, while most of the binding is fixed, the compiler still has to bind $0 according to constraint "m". Yet, if the compiler rightfully chooses %ebx, the data dependencies in the assembly itself differ from the expected one: pointer addr is exchanged with new_val1 just before being dereferenced, which is not the expected behaviour. The problem here is that the result cannot be predicted as it depends on token resolution from the compiler.

Impact: the function is likely to end up in a segmentation fault when compiled by Clang. Historically, GCC was not able to select %ebx and the bug did not manifest, but Clang did not had such restriction.

The problem. These compliance issues are really hard to find out either manually or syntactically. First, there is here clearly no hint from the assembly template itself ("cmpxchg8b %0") that register %edx is modified. Second, complex token binding and aliasing constraints must be taken into account. Third, subtle data flows must be taken into account – for example a read-only value modified then restored is not a compliance issue.

RUSTINA insights. To circumvent these problems, we have developed RUSTINA, an automated tool to check inline assembly compliance (i.e. formally verifying the absence of compliance errors) and to patch the identified issues.

RUSTINA builds upon an original formalization of the inline assembly interface compliance problem, encompassing both framing and unicity. From that, our method lifts binary-level Intermediate Representation (sketched in Fig. 2) and adapt the classical data-flow analysis framework (kill-gen [8]) in order to achieve sound interface compliance verification – especially RUSTINA reasons about token assignments. From the expected interface, it infers for each token an overapproximation of the set of valid locations and then computes the set of locations that shall not be altered before the token is used. Here, it deduces that writing in register %ebx may impact token $0. Also, it detects that a write occurs in the read-only register %edx, thus successfully reporting the two issues.

Moreover, RUSTINA automatically suggests patches for the two issues. For framing, Fig. 3 highlights the core differences between the two versions (%edx is now rightfully declared as output with "=d") – a similar patch now lives on the current version of the function (commit 05812c2). For unicity, it suggests to declare %ebx as clobber, yielding a working fix.

Yet, it also over-constrains the interface – the syntax does not allow a simple disequality between %0 and %ebx. Developers actually patched the issue in 2012 in a completely different way by rewriting the assembly template (commit 64d81cd) – such a solution is out of RUSTINA’s scope.

Generic and automatic, our approach is well suited to handle what expert developers failed to detect, while a simpler “bad” patterns detection approach would struggle against both the combinatorial complexity induced by the size of architecture instruction sets and the underlying reasoning complexity (dataflow, token assignments). Overall, RUSTINA found and patched many other significant issues in several well-known open source projects (Sec. VII).

III. GNU Inline Assembly Syntax

Overview. This feature allows the insertion of assembly instructions anywhere in the code without the need to call an externally defined function. Fig. 4 shows the concrete syntax of an inline assembly block, which can either be basic when it contains only the assembly template or extended when it is supplemented by an interface. This section concerns the latter only. The assembly statement consists of “a series of low-level instructions that convert input parameters to output parameters” [4]. The interface binds C values (i.e., expressions evaluating to C memory locations) and expressions to assembly operands specified as input or output, and declares a list of clobbered locations (i.e., registers or memory cells

Figure 2: Assembly statement semantics

Figure 3: Frame-write corrective patch
whose values could change). For the sake of completeness, the statement can also be tagged with volatile, inline or goto qualifiers, which are irrelevant for interface compliance, thus not discussed in this paper. The interface bindings described above are written by string specifications, which we will now explain.

**Templates.** The assembly text is given in the form of a formatted string template that, like printf, may contain so-called tokens (i.e., place holders). These start with % followed by an optional modifier and a reference to an entry of the interface, either by name (an identifier between square brackets) or by a number denoting a positional argument. The compiler preprocesses the template, substituting tokens by assembly operands according to the entries and the modifiers (note that only a subset of x86 modifiers is fully documented [9]) and then emits it as is in the assembly output file.

\[
\text{⟨statement⟩} := \text{asm} [\text{volatile}] \{ \langle \text{template: string} \rangle \langle \text{interface} \rangle \}
\]

\[
\text{⟨interface⟩} := \{ \langle \text{outputs} \rangle : \{ \langle \text{inputs} \rangle : \{ \langle \text{clobbers} \rangle \} \} \}
\]

\[
\text{⟨outputs⟩} := \{ \langle \text{outputs} \rangle \} , \{ \langle \text{inputs} \rangle \}
\]

\[
\text{⟨clobbers⟩} := \{ \langle \text{clobber: string} \rangle \} \{ \langle \text{clobbers} \rangle \}
\]

\[
\text{⟨input⟩} := \{ \langle \text{input} \rangle \} \{ \langle \text{clobber: string} \rangle \} \{ \langle \text{Constraint} \rangle \}
\]

Figure 4: Concrete syntax of an extended assembly chunk

**Clobbers.** They are names of hard registers whose values may be modified by the execution of the statement, but not intended as output. Clobbers must not overlap with inputs and outputs. The "cc" keyword identifies, when it exists, the conditional flags register. The "memory" keyword instructs the compiler that arbitrary memory access could be modified or modified.

\[
a = \{%eax\}, \quad b = \{%ebx\}, \quad c = \{%ecx\}
\]

\[
d = \{%edx\}, \quad S = \{\text{esi}\}, \quad D = \{\text{edi}\}
\]

\[
\text{U} = a \cup \text{U} \quad \text{U} \quad \text{a} \quad \text{U} \quad \text{U} \quad \text{a} \quad \text{U} \quad \text{a}
\]

\[
i = n = Z \quad r = R = q \quad \text{US} \quad \text{US} \quad \text{US} \quad \text{US} \quad \{\text{ebp}\}
\]

\[
p = \{r_n + k \times r_i + c \times r_i \in r \cup \{\text{esp}\} \cup \{0\}
\]

\[
\text{and } r_i \in r \cup \{0\} \text{ and } k \in \{1, 2, 4, 8\}
\]

\[
\text{and } c \in i
\]

\[
m = \{p \text{ for } p \in p\}
\]

\[
g = i \cup \text{m}
\]

Figure 5: GCC i386 architecture constraints

**Constraints.** A third language describes the set of valid assembly operands for token assignment. The latter are of 3 kinds: an immediate value, a register or a memory location. Fig. 5 gives a view of common atomic constraints ("letters") used in x86. Constraint entries can have more that one atomic constraint (e.g., "rm"), in which case the compiler chooses among the union of operand choices. The language allows to organize constraints into multiple alternatives, separated by , . Additionally, matching constraint between an input token and an output token forces them to be equal; early clobber % informs the compiler that it must not attempt to use the same operand for this output and any non-matched input; commutative pair % makes an input and the next one exchangeable.

Finally, output constraints must start either with = for the write-only mode or with + for the read-write permission.

**IV. Formalizing Interface Compliance**

**A. Extended assembly**

**Assembly chunks.** We denote by C: asm a standard chunk of assembly code. Such a chunk operates over a memory state M: mstate, that is a map from location (registers of the underlying architecture or memory cells) to basic values (int8, int16, int32, etc.). We call A: value set the set of valid addresses for a given architecture. The value of an expression in a given memory state is given by function eval: mstate × expression ↦ value. The set of valid assembly expressions is architecture-dependent (Fig. 5 is for i386). We abstract it as a set of expressions built over registers, memory accesses + and operations. Finally, an assembly chunk C can be executed in a memory state M to yield a new memory state M′ with function exec: asm × mstate ↦ mstate. Fig. 6 recaps above functions and types.

\[
\begin{align*}
\text{exec} : & \text{asm} \times \text{mstate} \rightarrow \text{mstate} \\
\text{eval} : & \text{mstate} \times \text{expression} \rightarrow \text{value} \\
\text{mstate} : & \text{location} \rightarrow \text{value} \\
\text{expression as e} : & \text{value} \rightarrow \text{register} + \text{value} + \text{exec} \times \text{value} \\
\text{location as register} : & \text{register} \rightarrow \text{value} \\
\text{register as register} : & \text{register} \rightarrow \text{exec} \times \text{value} \\
\text{value as int8} : & \text{value} \rightarrow \text{int8} \\
\text{value as int16} : & \text{value} \rightarrow \text{int16} \\
\text{value as int32} : & \text{value} \rightarrow \text{int32} \\
\end{align*}
\]

Figure 6: Assembly types

**Assembly templates.** Inline assembly does not directly use assembly chunks, but rather assembly templates, denoted C^0: asm^0, which are assembly chunks where some operands are replaced by so-called tokens, i.e., placeholders for regular assembly expressions to be filled by the compiler (formally, they are identifiers %, %, etc.). Given a token assignment T: token ↦ expression, we can turn an assembly template C^0: asm^0 into a regular assembly chunk C:asm using standard syntactic substitution ◼, denoted C^0 ◼ T: asm. The value of token t through assignment T is given by eval(M, T(t)).

**Formal interface.** We model an interface I ◼ (B^0, B^1, S^0, S^C, F) as a tuple consisting of output tokens B^0:token set, input tokens B^1:token set, a memory separation flag F: bool, clobber registers S^C: register set and valid token assignments S^T: T set.

- Input and output tokens bind the assembly memory state and the C environment. Informally, the locations pointed to by tokens in B^1 are input initialized by the value of some C expressions while the values of the tokens in B^0 are output to some C lvalues. B^0 ∪ B^1 contains all token declarations and B^0 ∩ B^1 may be non-empty;

4Actually, a concrete interface also contains initializer and collector expressions in order to bind I/O assembly locations input and output to C. We skip them for clarity, as they do not impact compliance.
• If the flag \( F \) is set to false, then assembly instructions may have side-effects on the C environment – otherwise they operate on separate memory parts;

• \( S^C \) and \( S^T \) provide additional information about how the compiler should instantiate the assembly template to machine code: the clobber registers in \( S^C \) can be used for temporary computations during the execution (their value is possibly modified by the chunk), while \( S^T \) represents all possible token assignments the compiler is allowed to choose – the GNU syntax typically leads to equality, disequality and membership constraints between tokens and (sets of) registers.

Extended assembly chunk. An extended assembly chunk \( X \triangleq (C^0, I) \) is a pair made of an assembly template \( C^0 \) and its related interface \( I \). The assembly template is the operational content of the chunk (modulo token assignment) while the related interface \( I \) is a contract between the chunk, the C environment and its content of the chunk (modulo token assignment) while the relationship is a contract between the chunk, the C environment and low-level location management.

### B. (Detail) From GNU concrete syntax to formal interfaces

Let us see how the formal interface \( I \) is derived from concrete GNU syntax (Fig. 4). Tokens \( B^S \) and \( B^I \) come from the corresponding output and input lists except that: a) if an output entry is declared using the "\*e" modifier then it is added to both \( B^S \) and \( B^I \); and b) if an input token and an output token are necessarily mapped to the same register, they are unified. Each register in the clobber list belong to \( S^C \). If the clobber list contains "memory", the memory separation flag \( F \) is false, true otherwise. The set \( S^T \) of valid token assignments \( T \) is formally derived in 3 steps:

1) collection of string constraints, splitting constraints by alternative (i.e., ",", ":") (token \( \mapsto \) string) set;

2) architecture-dependent (e.g., Fig. 5) evaluation of string constraints: (token \( \mapsto \) expression set) set representing a disjunction of conjunctions of atomic membership constraints token \( \in \{ \exp, ..., \exp \} \);

3) flattening: (token \( \mapsto \) expression) set representing a disjunction of conjunctions of atomic equality constraints token \( = \) expression;

Still, token assignments must respect the following properties (and are filtered out otherwise):

• every output token maps to an assignable operand, either a register or a \( \ast e \) expression;

• every output token maps to distinct location;

• each token maps to a clobber-free expression where a clobber-free expression is an expression without any clobber register nor any early-clobber sub-expression (i.e. containing the mapping of an early-clobber token, introduced by the "\*6" modifier).

Fig. 7 exemplifies the interface formalization of Fig. 1’s chunk introduced in Sec. II. Tokens \( B^S \) and \( B^I \) simply enumerate the present entries respectively in output and input lists (L196-198). The 5th entry matches the same register \( \%eax \) as the second, \( \%eax \) is unified with \( \%e1 \). For the sake of brevity, we split the set of token assignments into two parts: one invariant w.r.t. compiler choices, and one that may vary (we only list 4 of them but there are other valid combination of memory references). Finally, it has no clobbered register and, because of keyword "memory", memory separation is \( F = \text{false} \).

\[
\begin{align*}
B^S & = \{ \%e0, \%e1 \} \\
B^I & = \{ \%e2, \%e3, \%e5, \%e6 \} \\
S^C & = \{ \%e1 \mapsto \%eax, \%e3 \mapsto \%edx, \%e5 \mapsto \%ebx, \%e6 \mapsto \%esi \} \\
S^T & = \{ \%	ext{movl} \mapsto \%esi, \%e3 \mapsto \%eax \} \\
& \quad \times \{ \%	ext{mov} \mapsto \%esi, \%e3 \mapsto \%eax \} \\
& \quad \times \{ \%	ext{movl} \mapsto \%esi, \%e3 \mapsto \%eax \} \\
& \quad \times \{ \%	ext{mov} \mapsto \%esi, \%e3 \mapsto \%eax \} \\
F & = \text{false}
\end{align*}
\]

Figure 7: Formal interface \( I \)

### C. Interface compliance

An extended assembly chunk \( X \triangleq (C^0, I) \) is said to be interface compliant if it respects both the framing and the unicity conditions that we define below.

Observational equivalences. As a first step, we define an equivalence relation \( \approx_{B,F} \) over memory states modulo a token assignment \( T \), a set of observed tokens \( B \) and a memory separation flag \( F \). We start by defining an equivalence relation \( \approx_{B,F} \). We say that \( M_1 \approx_{B,F} M_2 \) if, for all token \( t \) in \( B \), \( \text{eval}(M_1, T(t)) = \text{eval}(M_2, T(t)) \). We can generalize it to any pair of token assignments \( T_1 \) and \( T_2; M_1 \approx_{B_1,F_1} T_2 \) if, for all tokens \( t \) in \( B \), \( \text{eval}(M_1, T_1(t)) = \text{eval}(M_2, T_2(t)) \). Then, we define an equivalence relation \( \approx \) over memory states. We say that \( M_1 \approx M_2 \) if for all (address) location \( l \) in \( A \), \( M_1(l) = M_2(l) \). The equivalence relation \( \approx_{B,F} \) over memory states modulo a token assignment \( T \) (which can be generalized to a pair \( T_1 \) and \( T_2 \) as above), a set of tokens \( B \) and a memory separation flag \( F \) is finally defined as:

\[
M_1 \approx_{B,F} M_2 \text{ if } M_1 \approx M_2 \text{ and } (F = \text{false}) \Rightarrow M_1 \approx M_2
\]

\textbf{Framing condition.} The framing condition restricts what can be read and written by the assembly template. Given a token assignment \( T \), we define a location input (resp. location output) as a location pointed by a input (resp. output) token. Then the framing condition stipulates that: (frame-read) only initial values from input location can be read; (frame-write) only clobber registers and location output are allowed to be modified by the assembly template.

More formally, a location is assignable if it can be modified (i.e., if it is mapped to by an output token \( t \)), belongs to the clobber set \( S^C \) or is a memory location \( A \) when there is no separation \( \neg F \), and not-assignable otherwise. We then have:

\textbf{frame-write} for all \( M \), for all \( T \) in \( S^T \), for all non assignable location \( l ; M(l) = \text{exec}(M, C^0 \langle T \rangle)(l) \).

\textbf{frame-read} for all \( M_1, M_2 \) and \( T \) in \( S^T \) such that \( M_1 \approx_{B_1,F_1} M_2 \); \( \text{exec}(M_1, C^0 \langle T \rangle) \approx_{B_1,F_1} \text{exec}(M_2, C^0 \langle T \rangle) \).

\textbf{Unicity.} Informally, the unicity condition is respected when the evaluation of output tokens is independent from the chosen token assignment. More formally, for all \( M_1, M_2 \), \( T_1 \) and \( T_2 \) in \( S^T \) such that \( M_1 \approx_{B_1,F_1} M_2 \)

\[
\text{exec}(M_1, C^0 \langle T_1 \rangle) \approx_{B_1,F_1} \text{exec}(M_2, C^0 \langle T_2 \rangle)
\]

Note that \textbf{frame-read} is a sub-case of unicity where \( T_1 = T_2 \).
V. CHECK, PATCH AND REFINE

Figure 8 presents an overview of RUSTINA. The tool takes as input a C file containing inline assembly templates in GNU syntax. From there, it parses the template code to produce an intermediate representation (IR) of the template \( C^0 \), and interprets the concrete interface to produce a formal interface \( I \). The tool then checks that the code complies with its interface using dedicated static dataflow analysis. If it succeeds, we have formally verified that the assembly template complies with its interface. If not, our tool examines the difference between the formal interface computed from the code and the one extracted from specification; it can then produce a patch (if some elements of the interface were forgotten) or refine the interface (if the declared interface is larger than needed). We cannot produce a patch in every situation, in that case the tool reports a compliance alarm – they can be false alarms, but it rarely happens on real code. Algorithms are fully detailed in the companion report [7].

A. Preliminary: code semantics extraction

Our analyses rely on Intermediate Representations (IR) for binary code, coming from lifters [10], [11] developed for the context of binary-level program analysis. We use the IR of the Binsec platform [12], [13] (Fig. 9), but all such IRs are similar. They encode every machine code instruction into a small well-defined side-effect free language, typically an imperative language over bitvector variables (registers) and arrays (memory), providing jumps and conditional branches. Still, code lifters do not operate directly on assembly templates but on machine code, requiring a little extra-work to recover the tokens. We replace each token in the assembly template by a distinct register, use an existing assembler (GAS) to transform the new assembly chunk into machine code and then lift it to IR. We perform the whole operation again where each token is mapped to another register, so as to distinguish tokens from hard-coded registers. Tokens are then replaced in IR by distinct new variable names.

B. Compliance Checking

This section discusses our static interface compliance checks. We rely on the dataflow analysis framework [8], intensively used in compilers and software verification. We collect sets of locations (token, register or the whole memory) as dataflow facts, then compare them against the sets expected from the interface. Checking frame-write requires a forward impact analysis, checking frame-read requires a backward liveness analysis, and finally unicity requires a combination of both. Our techniques are over-approximated in order to ensure soundness. Memory is considered as a whole – all memory accesses being squashed as memory, with a number of advantages: it closely follows the interface mechanisms for memory, helps termination (the set of dataflow facts is finite) and saves us the complications of memory-aware static analysis (heap or points-to). Finally, we propose two precision optimizations in order to reduce the risk of false positives (their impact is evaluated in Sec. VII-D).

Frame-write. Check must ensure that non-assignable locations have the exact same value before and after the execution. As first approximation, a location that is never written (i.e., never on the Left Hand Side LHS of an assignment) safely keeps its initial value – since IR expressions are side-effect free. Impact analysis iterates forward from the entry of the chunk, collecting the set of LHS locations (either a token, a register or the whole memory). We then check that each LHS location belongs to the set of declared assignable locations (i.e. \( B^0 \cup S^\circ \) together with memory if \( \neg F \)).

Frame-read. Check must ensure that no uninitialized location is read. This requires to compute (an overapproximation of) the set of live locations (i.e. holding a value that may be read before its next definition). Liveness analysis iterates backward from the exit of the chunk, where output locations are live (outputs tokens \( B^0 \)), computing dependencies of the Right Hand Side (RHS) expression of found definitions until the fix-point is reached. We then check at the entry point that each live location belongs to the set of declared inputs (i.e. \( B^1 \) together with memory if \( \neg F \)).

Unicity. Check must ensure that compiler choices have no impact on the chunk output. What may happen is that a location is impacted or not by a preceding write depending on the token assignment. To check that this does not happen, we first define a relation \( \text{may}_\text{impact} \) over location \( ^0 \) (incl. tokens) such that \( 1 \ \text{may}_\text{impact} \ 1' \) if we can prove that (writing on) \( 1 \) has no impact on (the evaluation of) \( 1' \) – whatever the token assignment. In our implementation, \( 1 \ \text{may}_\text{impact} \ 1' \) returns false if there is no token assignment where \( 1 \) is a sub-expression of \( 1' \). Then, using previous frame-write and frame-read analyses, we finally check at each assignment to a location \( 1 \) that, for each live location \( 1' \), \( 1 \ \text{may}_\text{impact} \ 1' \) returns false.

We now sketch the implementation of \( \text{may}_\text{impact} \). The main challenge is to avoid enumerating all valid token assignments \( S^\ast \) (c.f. Sec. IV-B). We compute over a smaller set of abstract location facts \( \text{location}_\ast \), indicating only whether a location is a constant value (Immediate), a register (Direct register) or is used to compute the address of a token (Indirect register). We abstract token assignments by reinterpreting the constraints over \( \text{location}_\ast \), yielding \( D' : \text{location}_\ast \mapsto \text{location}_\ast \) set. We then define the relation \( l^\ast \ \text{impact}_\ast \ l^\ast \ over \ \text{location}_\ast \ as:

\[
\begin{align*}
\text{Direct } r \ \text{impact}_\ast \ l^\ast & : \text{true} \\
\text{Indirect } r \ \text{impact}_\ast \ l^\ast & : \text{true} \\
\text{others} & : \text{false}
\end{align*}
\]

Finally, we build the relation \( 1 \ \text{may}_\text{impact} \ 1' \) such that it returns true (sound) except if one of the following holds:

- no \( l' \) in \( D(1) \times D(1') \) such that \( l^\ast \ \text{impact}_\ast \ l'^\ast \);
- \( 1 \) or \( 1' \) belongs to \( S^\circ \);
- \( 1 \) and \( 1' \) are tokens, \( 1 \) is early clobber ("&");
- \( 1 \) is equal to \( 1' \) (independent of compiler choice).
On the other hand, our technique could report compliance issues that do not exist (false positives). We propose below two precision improvements:

1. Expression propagation In Fig. 1, frame-write check, as is, would report a violation for `%ebx` and `%esi` because they are written. Yet, it is a false positive since both end up with their initial value. To avoid it, we perform a symbolic expression propagation for each written location, inlining the definition of written locations into their RHS expressions, and performing IR-level syntactic simplifications – such as $1 + x - 1 = x$ or $x \oplus x = 0$. Then, at fixpoint, frame-write checks before raising an alarm whether the original value has been restored (no alarm) or not (alarm);

2. Bit-level liveness dependency In Fig. 1, result takes only the lowest byte of `%eax`. However, our basic technique will count both $z$ and `%eax` as live while high bytes of `%eax` are actually not – such imprecisions may lead to false alarms (Sec. VII-D). We improve our liveness analysis to independently track the status of each location bit. For efficiency, we do not propagate location bits but locations equipped with a bitset representing the status of each of their bits. We modify propagation rules accordingly (especially bit manipulations like extraction or concatenation), with bitwise operations over the bitsets.

C. Interface Patching

When the compliance checking fails, RUSTINA tries to generate a patch to fix the issue. As our dataflow analysis infers an over-approximated interface for the chunk under analysis, we take advantage of it to strengthen the current interface.

Framing condition. We build a patch that makes the template $C^0$ compliant with its formal interface $I$ as follows:

frame-write Any hard-coded register (resp. token) written without belonging to $S^C$ (resp. $B^C$) is added;

frame-read Any token read without belonging to $B^I$ and without being initialized before, is added. Reading a register before assigning it prevents automatic patch generation.

In both cases, any direct access to a memory cell sets memory separation $F$ to false.

We then retrofit the changes of the formal interface in the concrete syntax to produce the patch. For instance, in Fig. 3, token `%3` (i.e. `%edx`) violates the frame-write condition. We add a new output token `%2` (dummy) bound to its old initializer : "2" (old_val2). Since we add a new token, we can already remove never-read inputs, never-written clobbers or undue "memory" keywords in absence of memory access.

Unicity. We give to the faulty register (resp. token) the (resp. early) clobber status preventing it to be mis-assigned to another token. Note however that, since we over-constrain the interface (the syntax does not allow to declare a pair of entries as distinct), the patch may fail if there is no more valid token assignment.

When a unicity patch is generated, the resulting chunk is ensured to meet the framing condition.

D. Bonus: Refining the interface

Even if overapproximated, the interface that is inferred by RUSTINA during the check may be smaller than the declared one, allowing to produce refinement patches removing unnecessary constraints in the interface – which can in turn give more room to the compiler to produce smaller or faster code.

We can already remove never-read inputs, never-written clobbers or undue "memory" keywords in absence of memory access. There is another case where a "memory" constraint can be removed. Indeed, as recommended in the documentation, single-level pointer accesses can be declared by common entries using the "m" placement constraint instead of the (much more expensive) "memory" keyword.

We design a dedicated "points-to" analysis to identify the candidates for this transformation. It is based on a dataflow analysis collecting, for each memory access, the precise location (on the form token or symbol + offset) and size of the access. If it succeeds, we can safely remove the "memory" keyword and instead add a new entry (input "m", output "=m" or both depending of the access pattern) for each of the identified base pointers.

\[^{1}\] If this is done on purpose, the chunk actually is out of this paper’s scope.

\[^{6}\] These refinements can be disabled for dummy constraints put on purpose.
VI. IMPLEMENTATION

We have implemented RUSTINA, a prototype for interface compliance analysis following the method described in Sec. V. RUSTINA is written in OCaml (~3 kLOC), it is based on Frama-C [14] for C manipulation (parsing, localization and patch generation), BINSEC [12], [13] for IR lifting (including basic syntactic simplifications), and GAS to translate assembly into machine code. Our tool can handle a large portion of the x86 and ARM instruction sets. Yet, float and system instructions are not supported (they are unsupported by BINSEC). Despite this, we handle 84% of assembly chunks found in a Debian distribution (Sec. VII).

VII. EXPERIMENTAL EVALUATION

Research questions. We consider 5 research questions: RQ1. Can RUSTINA automatically check interface compliance on assembly chunks found in the wild? RQ2. Incidentally, how many assembly chunks exhibit a compliance issue, and which ones are the most frequent? RQ3. Can RUSTINA automatically patch detected compliance issues? RQ4. What is the real impact of the compliance issues reported and of the generated patches? RQ5. What is the impact of RUSTINA design choices on the overall checking result?

Setup. All experiments are run on a regular Dell Precision 5510 laptop equipped with an Intel Xeon E3-1505M v5 processor and 32GB of RAM.

Benchmark. We run our prototype on all C-related x86 inline assembly chunks found in a Linux Debian 8.11 distribution, i.e., 3107 x86 chunks in 202 packages, including big inline assembly users like ALSA, GMP or ffmpeg. We remove 451 out-of-scope chunks (i.e., containing either float or system instructions), keeping 2656 chunks (85% of the initial dataset), with mean size of 8 assembly instructions (max. size: 341).

A. Checking (RQ1,RQ2)

Table I sums up compliance checking results before (“Initial”) and after patching (“Patched”) – we focus here on the Initial case.

Results. RUSTINA reports in less than 2 min (40 ms per chunk in average) that 1292 chunks out of 2656 are (fully) interface compliant (resp. 117 packages out of 202), while 1364 chunks (resp. 85 packages) have compliance issues.

Among the noncompliant ones, RUSTINA allows to pinpoint 294 chunks (resp. 54 packages) with serious compliance issues – according to our study in Sec. VIII we count an issue as benign only when it corresponds to missing the flag register as clobber (P1 in Sec. VIII).

Quality assessment. While chunks deemed compliant by RUSTINA are indeed supposed to be compliant (yet, it is still useful to test it), compliance issues could be false alarms.

We evaluate these two cases with 4 elements. (qa1) We run RUSTINA on known libatomic_obs and glibc compliance bugs and on their patched versions : every time, RUSTINA returns the expected result. (qa2) We consider 8 significant projects (Sec. VII-C), manually review all their faulty assembly chunks (covering roughly 50% of the serious issues reported in Table Ic) as well as randomly chosen compliant chunks, and crosscheck results with RUSTINA: they perfectly match. (qa3) For compliance proofs, we also run the checker after patching: RUSTINA deems all patched chunks compliant. (qa4) Several patches sent to developers have been accepted (Sec. VII-C).

We conclude that results returned by RUSTINA are good: as expected, a chunk deemed compliant is compliant, and reported compliance issues are most likely true alarms – we
do not find any false alarm in our dataset.

**ARM benchmark.** We also run RUSTINA on the ARM versions of ffmpeg, GMP and libyuv (from *Linux Debian 8.11*) for a total of 394 chunks (average size 5, max. size 29). We found very few issues (78), all in ffmpeg and related to the use of special flag (accumulated saturations). Manual review confirms them. Interestingly, the "cc" keywords are not forgotten in other cases. As flags are explicit in ARM mnemonics, coding practices are different than those for x86.

**RQ1:** RUSTINA is effective at compliance checking, in terms of speed and precision – yielding compliance proofs and identifying compliance bugs with near-zero false alarm rate. RUSTINA is widely applicable: it runs on the full Debian assembly chunk base and, without change, on 2 different architectures.

**Compliance bugs in practice.** Our previous precision analysis allows to assume that a warning from the checker likely indicates a true compliance issue. Hence, according to Tables Ia and Ib, 1364/2656 chunks (resp. 85/202 packages) are not interface-compliant, and 294 chunks (resp. 54 packages) have significant issues. According to Table Ic, 53% of significant issues come from unexpected writes, 38% from unexpected reads while 9% are unicity problems.

**RQ2:** About half of inline x86 assembly chunks found in the wild is not interface-compliant, and a significant part (11%) even exhibits significant compliance issues – affecting 27% of the packages under analysis.

**B. Patching (RQ3)**

**Results.** Table I (column “Patched”) shows that RUSTINA performs well at patching compliance issues: in 2 min, it patches 92% of total issues (2000/2183), including 81% of significant issues (803/986). Overall, 1276 more chunks (61 more packages) become fully compliant, reaching 97% compliance on chunks (88%) on packages.

The remaining issues (unbound register reads) are out of the scope of patching. They often correspond to the case where some registers are used as global memory between assembly chunks while only C variables can be declared as input in inline assembly. This practice is however fragile (special case of pattern P6 in Sec. VIII).

**Quality assessment.** We assess the quality of the patches adapting qa1 and qa2 from Sec. VII-A as follows: (qa1) On known libatomic Obs and libglibc compliance bugs, comparing RUSTINA-generated patches to originals shows that they are functionally equivalent, with similar fixes. (qa2) We manually review all (114) generated patches on 8 significant projects (Sec. VII-C) and check that they do fix the reported compliance issues. Also, recall that patched chunks pass the compliance checks (qa3) and that several patches have been accepted by developers (qa4). Overall, in most cases our automatic patches are optimal and equivalent to the ones that would be written by a human. Still, the "memory" keyword may have a significant impact on performance and developers usually try to avoid it. We address this issue with refinement (Sec. V-D). Finally, some unicity issues we found were actually resolved by developers by (deeply) rewriting the assembly template, instead of simply patching the interface.

**RQ3:** RUSTINA effectively generates patches for compliance issues, in terms of speed and patch quality. RUSTINA can automatically curate a large code base, removing the vast majority of compliance issues – the remaining ones require rewriting the code beyond mere interface compliance.

**C. Real-life impact (RQ4)**

We have selected 8 significant projects from our benchmark (namely: ALSA, ffmpeg, haproxy, libatomic_obs, libtomcrypt, UDPCast, xfstt, x264) to submit patches generated by RUSTINA in order to get real-world feedback. Note that submitting patches is time-consuming: patches must adhere to the project policy and our generated patches cannot be directly applied when the code uses macros (a common practice in inline assembly) as RUSTINA works on preprocessed C files.

Table III presents our results. Overall, we submitted 114 patches fixing 538 issues in the 8 projects. Feedback has been very positive: 38 patches have already been integrated, fixing 156 issues in 7 projects (ALSA, haproxy, libatomic_obs, libtomcrypt, UDPCast, xfstt, x264) – developers clearly expressed their interest in using RUSTINA once released. The ffmpeg patches are still under review.

**RQ4:** RUSTINA helps efficiently deliver quality patches.

**D. Internal evaluation: precision optimizations (RQ5)**

The observed absence of false positives in Sec. VII-A already takes into account the two precision enhancers (bit-level liveness analysis and symbolic expression propagation) presented in Sec. V-B. We seek now to assess the impact of these two improvements over the false positive rate (fpr) of RUSTINA. We ran a basic version of RUSTINA (no expression propagation, no bit-level liveness, but still the basic IR simplifications done by BINSEC) on our whole benchmark. It turns out that this basic version reports 127 false alarms (6% fpr) – 40 frame-write (2% fpr) and 87 frame-read (23% fpr). All these alarms concern potentially significant issues. Restricting to significant issues, this amount to false positive rates of 13% (total), 23% (frame-read) and 8% (frame-write). It turns out that our two optimizations are complementary: bit-level liveness analysis removes the 87 false frame-read alarms while expression propagation removes the 40 false frame-write alarms.

The two precision optimizations (expression folding, bit-level liveness) upon RUSTINA base technique are essential in order to get a near-zero false alarm rate.
Including 27 non automatically patchable issues, manually fixed.

<table>
<thead>
<tr>
<th>Patched chunks</th>
<th>Fixed issues</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 &quot;cc&quot;</td>
<td>64/64</td>
<td>01d84af, 06d7f0c</td>
</tr>
<tr>
<td>P2 %ebx register</td>
<td>1/1</td>
<td>05812c2</td>
</tr>
<tr>
<td>P3 %esp register</td>
<td>2/2</td>
<td>cef2f85</td>
</tr>
<tr>
<td>P4 &quot;memory&quot;</td>
<td>2/2</td>
<td>20200328</td>
</tr>
<tr>
<td>P5 MMX register</td>
<td>11/1</td>
<td>91c358e</td>
</tr>
<tr>
<td>P6 XMM register</td>
<td>11/1</td>
<td>69771</td>
</tr>
</tbody>
</table>

(*) There are discussions on GCC mailing list to change that [18].

### VIII. BAD CODING PRACTICES FOR INLINE ASSEMBLY

In this section, we aim to: 1) seek some sort of regularity behind so many compliance issues, in order to understand while developers introduce them in the first place; 2) understand in the same time why so many compliance issues do not turn more often into observable bugs; 3) assess the risk of such bugs to occur in the future.

**Common error patterns for inline assembly.** We have identified 6 patterns (P1 to P6, see Table II) responsible for 91% of compliance issues (1986/2183) – 80% of significant compliance issues (789/986). In each case, some input or output declarations are missing, but surprisingly it almost always concerns the same registers (%ebx, %esp, "cc", MMX or XMM registers) or memory, with similar coding practices (e.g. no XMM declaration together with compiler options for deactivating XMM, or no declaration of %ebx together with surrounding push and pop). Hence, these patterns are deliberate rather than mere coding errors.

**Underlying implicit protections and their limits.** It turns out that each pattern builds on implicit protections (Table II). We identified three main categories: (1) (P1-P2-P3) compiler choices regarding inline assembly (e.g., “protected” registers, default clobbers); (2) (P4-P5) the apparent protection offered by putting a single assembly chunk inside a C function (relying mostly on the limited interprocedural analysis abilities of compilers); and (3) (P6) specific compiler options.

Yet, all these reasons are fragile: compiler choices may change, and actually do, compilers enjoy more and more powerful program analysis engines including very aggressive code inlining like Link-time optimization (LTO), and refactoring may affect the compilation context.

We now provide a precise analysis of each error pattern:

**P1 omitted "cc" keyword.** x86 has been once a “cc0” architecture, i.e., any inline assembly statement implicitly clobbered "cc" so it was not necessary to declare it as written. As far as we know, compilers still unofficially maintain this special treatment for backward compatibility. However, some claim “that is ancient technology and one day it will be gone completely, hopefully” [18];

**P2 omitted %ebx register.** The Intel ABI states that %ebx should be treated separately as a special PIC (Position Independent Code) pointer register. Old version of GCC (prior to version < 5.0) totally dedicated %ebx to that role and refrained from binding it to an assembly chunk. Still, some chunks actually require to use %ebx (e.g. cmpxchg8b) and people used tricks to use it anyway without stating it. It becomes risky because current compilers can now spill and use %ebx as they need;

**P3 omitted %esp register.** %esp is here modified but restored by push and pop. Yet, compilers may decide to use %esp instead of %ebp to pass addresses of local variables. In fact, it became the default behavior since GCC version 4.6. Thus, code mixing local variable references and push and pop may read the wrong index of the stack, leading to unexpected issues;

**P4 omitted "memory".** Compilers’ analysis are often performed per function, with conservative assumptions on the memory impact of called functions, limiting the ability of the compiler to modify (optimize) the context of chunks. This is no longer true in case of inlining where assembly interface issues become more impactful;

**P5 omitted MMX register.** For the same reason as above, when a chunk is inside a function, it is also protected by the ABI in use. The Intel ABI specifies that MMX registers are caller-saved, hence the compiler must ensure that their value is restored when function exits. Yet, inlining may break this pattern since the ABI barrier is not there anymore once the function code is inlined;

**P6 omitted XMM register.** Using parts of the architecture out-of-reach of the compiler (the compiler cannot spill them, typically through adequate command-line options) is safe but fragile as it is sensitive to future refactoring (affecting the compiler options). Moreover, newer compiler options or hardware architecture updates can implicitly reuse registers otherwise deactivated, e.g. XMM registers reused as subpart of AVX registers.

**Breaking patterns.** We now seek to assess how fragile (or not) these patterns are. Replaying known issues [15], [16], [17] with current compilers shows that patterns P2 to P4 are (still) unsafe. In addition, we conducted experiments to show that current compilers do have the technical capacity to break the patterns. We consider two main threat scenarios:
Cloning developers copy the chunk as is to another project (bad but common development practice [19], [20]); Inlining projects import the code as a library and compile it statically with their code (link-time optimization).

We consider for each pattern 5 representative faulty chunks from the 8 projects. For each chunk, we craft a toy example aggressively tuned to call the (cloned or imported) chunk in an optimization-prone context. For instance, as P5 & P6 issues involve SIMD registers, the corresponding chunks are called within an inner loop while auto-vectorization is enabled \((-0.3)\). Results are reported in column “Robust?” of Table II. We actually break 5/6 patterns with code cloning (all but P1), and 4/6 with code inlining, demonstrating that these compliance issues should be considered plausible threats.

We identified a set of 6 recurring patterns leading to the majority of compliance issues. All of them build on fragile assumptions on the compiling chain. Especially, code cloning and compiler code inlining are serious threats.

IX. Discussion

A. Threats to validity

We avoid bias as much as possible in our benchmark: 1) the benchmark is comprehensive: all Debian packages with C-embedded inline assembly; 2) we mostly work on x86, but still consider 394 ARM chunks from 3 popular projects. Our prototype is based on tools already used in significant case studies [21], [22], [23], [24], including a well tested x86-to-IR decoder [25]. Also, results have been crosschecked in several ways and some of them manually reviewed. So, we feel confident in our main conclusions.

B. Limitations

Architecture. Our implementation supports the architectures of the Binsec platform, currently x86-32 and ARMv7. This is not a conceptual limitation, as our technique ultimately works on a generic IR. As soon as a new architecture is available in Binsec, we will support it for free.

Float. We do not yet support float instructions as Binsec IR does not. While adding support in the IR is feasible but time-consuming, our technique could also work solely with a partial instruction support reduced to I/O information about each instruction – at the price of some false positives.

System instructions. Our formalization considers assembly chunks as a deterministic way to convert well-identified inputs from the C environment to outputs. But system instructions often read or write locations hidden to the C context (system registers) and will thus appear to be non-deterministic – breaking either the framing or the unicity condition. Extending our formalization is feasible, but it is useful only if the GNU syntax is updated. Still, we consider that at most 13% of assembly chunks used such instructions.

C. Microsoft inline assembly

Microsoft inline assembly (inline MASM) proposed in Visual Studio [26] does not suffer from the same flaws as GNU’s. Indeed, each assembly instruction is known by the compiler such that no interface is required, and moreover developers can seamlessly write variables from C into the assembly mnemonics. Yet, this solution is actually restricted to a subset of the i386 instruction set, as the cost in term of compiler development is significantly more important.

X. Related work

Interface compliance. Fehnker et al. [5] tackle inline assembly compliance checking for ARM (patching and refinement are not addressed), but in a very limited way. This work restricts compliance to the framing case (no unicity condition) and is driven by assembly syntax rather than semantics, making it less precise than ours – for example, a saved-and-restored register will be counted as a framing-write issue. Moreover, it does not handle neither memory nor token constraints (tokens are assumed to be in registers and to be distinct from each other). Finally, their implementation is strongly tied to ARM with strong syntactic assumptions and their prototype is evaluated only on 12 files from a single project.

Assembly code lifting and mixed code verification. Two recent works [1], [6] lift GNU inline assembly to semantically equivalent C code in order to perform verification of mixed codes combining C and inline assembly. Their work is complementary to ours: their lifting assume interface compliance but in turn they can prove functional correctness of assembly chunks. Verifying code mixing C and assembly has also been active on Microsoft MASM assembly [27], [28], [29]. Yet, inline MASM does not rely on interface (Sec. IX-C).

Binary-level analysis. While binary-level semantic analysis is hard [30], [31], [32], [33], inline assembly chunks offer nice structural properties [1] allowing efficient and precise analysis. We also benefit from previous engineering efforts on generic binary lifters [10], [11], [25].

XI. Conclusion

Embedding GNU-like inline assembly into higher-level languages such as C/C++ allows higher performance, but at the price of potential errors due either to the assembly glue or to undue code optimizations as the compiler blindly trusts the assembly interface. We propose a novel technique to automatically reason about inline assembly interface compliance, based on a clean formalization of the problem. The technique is implemented in RUSTINA, the first sound tool providing comprehensive automated interface compliance checking as well as automated patch synthesis and interface refinements.
REFERENCES


