# 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
matthieu.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[1].

**Problem.** The problem with GNU inline assembly is twofold. First, it is hard to write correctly[2]: 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[3] 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 03e48c1). 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.

[1]Microsoft inline assembly is different and has no interface, see Sec. IX-C.
[2]From the *llvm-dev* mailing list [3]: "GCC-style inline assembly is notoriously hard to write correctly".
[3]Note that syntactically incorrect assembly instructions are caught during the translation from assembly to machine code.

**Goal and challenges.** *We address the challenge of helping developers write safer inline assembly code by designing and 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 glibc issue mentioned above. Moreover, they do not cover all subtleties 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 30cea1b dating back to early 2012. It was replaced 6 months later by commit 64d81cd 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).

```
179  AO_INLINE int
180  AO_compare_double_and_swap_double_full(volatile AO_double_t *addr,
181                                         AO_t old_val1, AO_t old_val2,
182                                         AO_t new_val1, AO_t new_val2)
183  {
184    char result;
     [...]
193    __asm__ __volatile__("xchg %%ebx,%6;" /* swap GOT ptr and new_val1 */
194                         "lock; cmpxchg8b %0; setz %1;"
195                         "xchg %%ebx,%6;" /* restore ebx and edi */
196                         : "=m"(*addr), "=a"(result)
197                         : "m"(*addr), "d" (old_val2), "a" (old_val1),
198                           "c" (new_val2), "D" (new_val1) : "memory");
     [...]
209    return (int) result;
210  }
```

Figure 1: `atomic_ops/sysdeps/gcc/x86.h`@30cea1b

**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_val2)` indicates that register `%edx` should be initialized with the value of `old_val2`, 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, "$\xleftarrow{c}$" a conditional assignment, "$e_{\{h..l\}}$" the bits extraction and "$\mathsf{zext}_n$" 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

```
"=m" (*addr)              &0          ⟵   addr
"d" (old_val2)            %edx        ⟵   old_val2
"a" (old_val1)            %eax        ⟵   old_val1
"c" (new_val2)            %ecx        ⟵   new_val2
"D" (new_val1)            %edi        ⟵   new_val1
xchg %ebx, %edi           %ebx        ⇆   %edi
lock                      z           ⟵   (%edx :: %eax) = *(&0)
cmpxchg8b %0              %edx :: %eax ⟵  *(&0)
                          *(&0)       ⟵z  %ecx :: %ebx
setz %al                  %eax        ⟵   %eax{31..8} :: (zext8 z)
xchg %ebx, %edi           %ebx        ⇆   %edi
"=a" (result)             result      ⟵   %eax{7..0}
```

Figure 2: Assembly statement semantics

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` (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_val2` after the assembly chunk is executed, while it is not the case. Appx. A presents a scenario with a very similar compliance issue from the same project leading to a `deadlock` (the compliance bug was patched by the developers in 2010, 4 years after being introduced).

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.

```
        @@ -193,5 +193,6 @@
193     -  __asm__ __volatile__("xchg %%ebx,%6;" /* swap GOT ptr and new_val1 */
    193 +  AO_t dummy;
    194 +  __asm__ __volatile__("xchg %%ebx,%7;" /* swap GOT ptr and new_val1 */
194 195                        "lock; cmpxchg8b %0; setz %1;"
195     -                      "xchg %%ebx,%6;" /* restore ebx and edi */
196     -                      : "=m"(*addr), "=a"(result)
197     -                      : "m"(*addr), "d" (old_val2), "a" (old_val1),
    196 +                      "xchg %%ebx,%7;" /* restore ebx and edi */
    197 +                      : "=m"(*addr), "=a"(result), "=d" (dummy)
    198 +                      : "m"(*addr), "2" (old_val2), "a" (old_val1),
```

Figure 3: Frame-write corrective patch

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 lvalues (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 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.

$$
\begin{aligned}
\langle \textit{statement} \rangle &::= \text{`}\mathbf{\texttt{asm}}\text{'}\ [\ \text{`}\mathbf{\texttt{volatile}}\text{'}\ ]\ \text{`}\mathbf{\texttt{(}}\text{'}\ \langle \textit{template}:\texttt{string} \rangle\ [\ \langle \textit{interface} \rangle\ ]\ \text{`}\mathbf{\texttt{)}}\text{'} \\
\langle \textit{interface} \rangle &::= \text{`}\texttt{:}\text{'}\ [\ \langle \textit{outputs} \rangle\ ]\ \text{`}\texttt{:}\text{'}\ [\ \langle \textit{inputs} \rangle\ ]\ \text{`}\texttt{:}\text{'}\ [\ \langle \textit{clobbers} \rangle\ ] \\
\langle \textit{outputs} \rangle &::= \langle \textit{output} \rangle\ [\ \text{`}\mathbf{\texttt{,}}\text{'}\ \langle \textit{outputs} \rangle\ ] \\
\langle \textit{inputs} \rangle &::= \langle \textit{input} \rangle\ [\ \text{`}\mathbf{\texttt{,}}\text{'}\ \langle \textit{inputs} \rangle\ ] \\
\langle \textit{clobbers} \rangle &::= \langle \textit{clobber}:\texttt{string} \rangle\ [\ \text{`}\mathbf{\texttt{,}}\text{'}\ \langle \textit{clobbers} \rangle\ ] \\
\langle \textit{output} \rangle &::= [\ \text{`}\mathbf{\texttt{[}}\text{'}\ \langle \textit{identifier} \rangle\ \text{`}\mathbf{\texttt{]}}\text{'}\ ]\ \langle \textit{constraint}:\texttt{string} \rangle\ \text{`}\mathbf{\texttt{(}}\text{'}\ \langle \textit{Clvalue} \rangle\ \text{`}\mathbf{\texttt{)}}\text{'} \\
\langle \textit{input} \rangle &::= [\ \text{`}\mathbf{\texttt{[}}\text{'}\ \langle \textit{identifier} \rangle\ \text{`}\mathbf{\texttt{]}}\text{'}\ ]\ \langle \textit{constraint}:\texttt{string} \rangle\ \text{`}\mathbf{\texttt{(}}\text{'}\ \langle \textit{Cexpression} \rangle\ \text{`}\mathbf{\texttt{)}}\text{'}
\end{aligned}
$$

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 could be accessed or modified.

$$
\begin{array}{lll}
\texttt{a} = \{\texttt{\%eax}\} & \texttt{b} = \{\texttt{\%ebx}\} & \texttt{c} = \{\texttt{\%ecx}\} \\
\texttt{d} = \{\texttt{\%edx}\} & \texttt{S} = \{\texttt{\%esi}\} & \texttt{D} = \{\texttt{\%edi}\}
\end{array}
$$

$$
\begin{array}{ll}
\texttt{U} = \texttt{a} \cup \texttt{c} \cup \texttt{d} & \texttt{q} = \texttt{Q} = \texttt{a} \cup \texttt{b} \cup \texttt{c} \cup \texttt{d} \\
\texttt{i} = \texttt{n} = \mathbb{Z} & \texttt{r} = \texttt{R} = \texttt{q} \cup \texttt{S} \cup \texttt{D} \cup \{\texttt{\%ebp}\}
\end{array}
$$

$$
\begin{aligned}
\texttt{p} = \{r_b + k \times r_i + c \text{ for } & r_b \in \texttt{r} \cup \{\texttt{\%esp}\} \cup \{0\} \\
& \text{and } r_i \in \texttt{r} \cup \{0\} \text{ and } k \in \{1,2,4,8\} \\
& \text{and } c \in \texttt{i}\}
\end{aligned}
$$

$$
\texttt{m} = \{\star p \text{ for } p \in \texttt{p}\} \qquad \texttt{g} = \texttt{i} \cup \texttt{r} \cup \texttt{m}
$$

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 `expression`s built over registers, memory accesses $\star$ 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.

$$
\begin{aligned}
\mathbf{\texttt{exec}} &:\ \texttt{asm} \times \texttt{mstate} \mapsto \texttt{mstate} \\
\mathbf{\texttt{eval}} &:\ \texttt{mstate} \times \texttt{expression} \mapsto \texttt{value} \\
\texttt{mstate} &:\ \texttt{location} \mapsto \texttt{value} \\
\texttt{expression as } e &::= \texttt{value} \mid \texttt{register} \mid \star e \mid e + e \mid e \times e \mid \ldots \\
\texttt{location} &::= \texttt{register} \mid \texttt{value} \\
\texttt{register} &::= \texttt{\%eax} \mid \texttt{\%ebx} \mid \texttt{\%ecx} \mid \texttt{\%edx} \mid \ldots \ \text{// case of x86} \\
\texttt{value} &:\ \texttt{int8} \mid \texttt{int16} \mid \texttt{int32} \mid \ldots
\end{aligned}
$$

Figure 6: Assembly types

**Assembly templates.** Inline assembly does not directly use assembly chunks, but rather *assembly templates*, denoted $\texttt{C}^\diamond$: $\texttt{asm}^\diamond$, which are assembly chunks where some operands are replaced by so-called *tokens*, i.e., placeholders for regular assembly `expression`s to be filled by the compiler (formally, they are identifiers `%0`, `%1`, etc.). Given a *token assignment* `T`: `token` ↦ `expression`, we can turn an assembly template $\texttt{C}^\diamond$:$\texttt{asm}^\diamond$ into a regular assembly chunk `C`:`asm` using standard syntactic substitution **`<>`**, denoted $\texttt{C}^\diamond$**`<`**`T`**`>`**:`asm`. The value of `token t` through assignment `T` is given by **`eval`**(`M`, `T`(`t`)).

**Formal interface.** We model an *interface* $\texttt{I} \triangleq (\texttt{B}^\texttt{O}, \texttt{B}^\texttt{I}, \texttt{S}^\texttt{T}, \texttt{S}^\texttt{C}, \texttt{F})$ as a tuple consisting of *output tokens* $\texttt{B}^\texttt{O}$:`token set`, *input tokens*[4] $\texttt{B}^\texttt{I}$:`token set` , a *memory separation* flag `F`: `bool`, *clobber registers* $\texttt{S}^\texttt{C}$: `register set` and *valid token assignments* $\texttt{S}^\texttt{T}$: `T set`.

- Input and output tokens bind the assembly memory state and the C environment. Informally, the locations pointed to by tokens in $\texttt{B}^\texttt{I}$ are *input* initialized by the value of

[4] Actually, 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.

some C expressions while the values of the tokens in $\mathtt{B}^{\mathtt{O}}$ are *output* to some C lvalues. $\mathtt{B}^{\mathtt{O}} \cup \mathtt{B}^{\mathtt{I}}$ contains all token declarations and $\mathtt{B}^{\mathtt{O}} \cap \mathtt{B}^{\mathtt{I}}$ may be non-empty;

- If the flag $\mathtt{F}$ is set to false, then assembly instructions may have side-effects on the C environment – otherwise they operate on separate memory parts;
- $\mathtt{S}^{\mathtt{C}}$ and $\mathtt{S}^{\mathtt{T}}$ provide additional information about how the compiler should instantiate the assembly template to machine code: the clobber registers in $\mathtt{S}^{\mathtt{C}}$ can be used for temporary computations during the execution (their value is possibly modified by the chunk), while $\mathtt{S}^{\mathtt{T}}$ 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 $\mathtt{X} \triangleq (\mathtt{C}^{\diamondsuit}, \mathtt{I})$ is a pair made of an assembly template $\mathtt{C}^{\diamondsuit}$ and its related interface $\mathtt{I}$. The assembly template is the operational content of the chunk (modulo token assignment) while the interface 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 $\mathtt{I}$ is derived from concrete GNU syntax (Fig. 4). Tokens $\mathtt{B}^{\mathtt{O}}$ and $\mathtt{B}^{\mathtt{I}}$ come from the corresponding output and input lists except that: a) if an output entry is declared using the `'+'` modifier then it is added to both $\mathtt{B}^{\mathtt{O}}$ and $\mathtt{B}^{\mathtt{I}}$; 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 $\mathtt{S}^{\mathtt{C}}$. If the clobber list contains `"memory"`, the memory separation flag $\mathtt{F}$ is false, true otherwise. The set $\mathtt{S}^{\mathtt{T}}$ of valid token assignments $\mathtt{T}$ 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 `*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 `'&'` modifier).

Fig. 7 exemplifies the interface formalization of Fig. 1's chunk introduced in Sec. II. Tokens $\mathtt{B}^{\mathtt{O}}$ and $\mathtt{B}^{\mathtt{I}}$ simply enumerate the present entries respectively in output and input lists (L196-198). The 5$^{\text{th}}$ entry matches the same register `%eax` as the second, `%4` is unified with `%1`. 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 `false`.

```
B^O = { %0, %1 },
B^I = { %2, %3, %5, %6 }
S^T = { [ %1 ↦ %eax, %3 ↦ %edx, %5 ↦ %ecx, %6 ↦ %edi ] }              // fixed assignments
    × { [ %0 ↦ *%esi, %2 ↦ *%esi ], [ %0 ↦ *%ebp, %2 ↦ *%ebp ],
        [ %0 ↦ *%esi, %2 ↦ *%ebp ], [ %0 ↦ *%ebx, %2 ↦ *%ebx ], ... }   // possible variations
S^C = ∅
 F  = false
```

Figure 7: Formal interface $\mathtt{I}$

### C. Interface compliance

An extended assembly chunk $\mathtt{X} \triangleq (\mathtt{C}^{\diamondsuit}, \mathtt{I})$ 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 $\overset{\blacklozenge}{\cong}^{\mathtt{T}}_{B,\mathtt{F}}$ over memory states modulo a token assignment $\mathtt{T}$, a set of observed tokens $\mathtt{B}$ and a memory separation flag $\mathtt{F}$. We start by defining an equivalence relation $\overset{\lozenge}{\approx}^{\mathtt{T}}_{\mathtt{B}}$. We say that $\mathtt{M}_1 \overset{\lozenge}{\approx}^{\mathtt{T}}_{\mathtt{B}} \mathtt{M}_2$ if, for all `token t` in $\mathtt{B}$, $\mathbf{eval}(\mathtt{M}_1, \mathtt{T}(\mathtt{t})) = \mathbf{eval}(\mathtt{M}_2, \mathtt{T}(\mathtt{t}))$. We can generalize it to any pair of token assignments $\mathtt{T}_1$ and $\mathtt{T}_2$: $\mathtt{M}_1 \overset{\lozenge}{\approx}^{\mathtt{T}_1,\mathtt{T}_2}_{\mathtt{B}} \mathtt{M}_2$ if, for all `tokens t` in $\mathtt{B}$, $\mathbf{eval}(\mathtt{M}_1, \mathtt{T}_1(\mathtt{t})) = \mathbf{eval}(\mathtt{M}_2, \mathtt{T}_2(\mathtt{t}))$. Then, we define an equivalence relation $\overset{\bullet}{\sim}$ over memory states. We say that $\mathtt{M}_1 \overset{\bullet}{\sim} \mathtt{M}_2$ if for all (address) `location l` in $\mathtt{A}$, $\mathtt{M}_1(\mathtt{l}) = \mathtt{M}_2(\mathtt{l})$. The equivalence relation $\overset{\blacklozenge}{\cong}^{\mathtt{T}}_{B,\mathtt{F}}$ over memory states modulo a token assignment $\mathtt{T}$ (which can be generalized to a pair $\mathtt{T}_1$ and $\mathtt{T}_2$ as above), a set of tokens $\mathtt{B}$ and a memory separation flag $\mathtt{F}$ is finally defined as:
$\mathtt{M}_1 \overset{\blacklozenge}{\cong}^{\mathtt{T}}_{B,\mathtt{F}} \mathtt{M}_2$ if: $\mathtt{M}_1 \overset{\lozenge}{\approx}^{\mathtt{T}}_{\mathtt{B}} \mathtt{M}_2 \wedge (F = \mathtt{false}$ implies $\mathtt{M}_1 \overset{\bullet}{\sim} \mathtt{M}_2)$

**Framing condition.** The framing condition restricts what can be read and written by the assembly template. Given a token assignment $\mathtt{T}$, 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 $\mathtt{t}$, belongs to the clobber set $\mathtt{S}^{\mathtt{C}}$ or is a memory location $\mathtt{A}$ when there is no separation $\neg\mathtt{F}$), and *non-assignable* otherwise. We then have:

**frame-write** for all $\mathtt{M}$, for all $\mathtt{T}$ in $\mathtt{S}^{\mathtt{T}}$, for all non assignable `location l`: $\mathtt{M}(\mathtt{l}) = \mathbf{exec}(\mathtt{M}, \mathtt{C}^{\diamondsuit}\mathbf{<}\mathtt{T}\mathbf{>})(\mathtt{l})$.

**frame-read** for all $\mathtt{M}_1$, $\mathtt{M}_2$ and $\mathtt{T}$ in $\mathtt{S}^{\mathtt{T}}$ such that $\mathtt{M}_1 \overset{\blacklozenge}{\cong}^{\mathtt{T}}_{\mathtt{B}^{\mathtt{I}},\mathtt{F}} \mathtt{M}_2$: $\mathbf{exec}(\mathtt{M}_1, \mathtt{C}^{\diamondsuit}\mathbf{<}\mathtt{T}\mathbf{>}) \overset{\blacklozenge}{\cong}^{\mathtt{T}}_{\mathtt{B}^{\mathtt{O}},\mathtt{F}} \mathbf{exec}(\mathtt{M}_2, \mathtt{C}^{\diamondsuit}\mathbf{<}\mathtt{T}\mathbf{>})$,

**Unicity.** Informally, the unicity condition is respected when the evaluation of output tokens is independent from the chosen token assignment. More formally, for all $\mathtt{M}_1$, $\mathtt{M}_2$ , $\mathtt{T}_1$ and $\mathtt{T}_2$ in $\mathtt{S}^{\mathtt{T}}$ such that $\mathtt{M}_1 \overset{\blacklozenge}{\cong}^{\mathtt{T}_1,\mathtt{T}_2}_{\mathtt{B}^{\mathtt{I}},\mathtt{F}} \mathtt{M}_2$:

$$\textbf{exec}(M_1, C^{\diamond}\texttt{<}T_1\texttt{>}) \stackrel{\blacklozenge}{\simeq}^{T_1,T_2}_{B^O,F} \textbf{exec}(M_2, C^{\diamond}\texttt{<}T_2\texttt{>}).$$

Note that **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 $C^{\diamond}$, 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). Appx. C presents detailed algorithms.

**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. $B^O \cup S^C$ 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 $B^O$), 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. $B^I$ 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 **`may_impact`** over $\texttt{location}^{\circ}$ (incl. tokens) such that $l$ **`may_impact`** $l'$ is false if we can prove that (writing on) $l$ has no impact on (the evaluation of) $l'$ – whatever the token assignment. In our implementation, $l$ **`may_impact`** $l'$ returns false if there is no token assignment where $l$ is a sub-expression of $l'$. Then, using previous **frame-write** and **frame-read** analyses, we finally check at each assignment to a location $l$ that, for each live location $l'$, $l$ **`may_impact`** $l'$ returns `false`.

We now sketch the implementation of **`may_impact`**. The main challenge is to avoid enumerating all valid token assignments $S^T$ (c.f. Sec. IV-B) . We compute over a smaller set of abstract location facts $\texttt{location}^*$, 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 $\texttt{location}^*$, yielding $\mathbb{D}^* : \texttt{location}^{\circ} \mapsto \texttt{location}^*\ \texttt{set}$. We then define the relation $l^*$ $\textbf{impact}^*$ $l^{*\prime}$ over $\texttt{location}^*$ as:

$$l^*\ \textbf{impact}^*\ l^{*\prime} = \begin{cases} \texttt{Direct}\ r\ \textbf{impact}^*\ \texttt{Direct}\ r & : \texttt{true} \\ \texttt{Direct}\ r\ \textbf{impact}^*\ \texttt{Indirect}\ r & : \texttt{true} \\ \textit{others} & : \texttt{false} \end{cases}$$

Finally, we build the relation $l$ **`may_impact`** $l'$ such that it returns `true` (sound) except if one of the following holds:

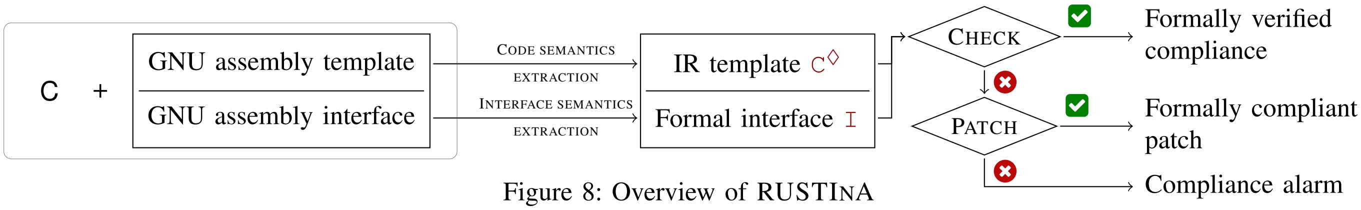


Figure 8: Overview of RUSTINA

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

Figure 9: The BINSEC intermediate representation

- no $l^*$, $l^{*\prime}$ in $\mathbb{D}$(`l`) $\times$ $\mathbb{D}$(`l`') such that $l^*$ **`impact`**$^*$ $l^{*\prime}$;
- `l` or `l`' belongs to `S`$^{\texttt{C}}$;
- `l` and `l`' are tokens, `l` is early clobber (`"&"`);
- `l` is equal to `l`' (independent of compiler choice).

Our checkers are *semantically sound* in the sense that they compute an *overapproximation* of the assembly template semantics. Hence, successfully checking an extended assembly chunk *ensures* it is *interface-compliant*.

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`$^\diamondsuit$ compliant with its formal interface `I` as follows:

**frame-write** Any hard-coded register (resp. token) written without belonging to `S`$^{\texttt{C}}$ (resp. `B`$^{\texttt{O}}$) is added;

**frame-read** Any token read without belonging to `B`$^{\texttt{I}}$ and without being initialized before, is added. Reading a register before assigning it prevents automatic patch generation[5].

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 : "=d" (dummy)` bound to its old initializer `: "2" (old_val2)`. Since we add a new token, we take care to keep template "numbering" consistent.

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

**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 be fully interface compliant if it still compiles.

### 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 accesses[6]. 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

[5] 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.

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.

Fig. 10 shows an example of refinement happening in libtomcrypt. In the original code, the `"memory"` constraint was forgotten. We can see that (patch) refinement produces a fix that does not add the missing keyword, but instead changes the way the content pointed by `key` is given to the chunk.

```
  asm __volatile__ (
-    "movl (%1), %0\n\t"
+    "movl %1, %0\n\t"
     "bswapl %0\n\t"
-    :"=r"(rk[0]): "r"(key));
+    :"=r"(rk[0]): "m"(*(uint32_t*)key));
```

Figure 10: Smart patch of a libtomcrypt chunk

## 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.

Table I: RUSTINA application on Debian 8.11 x86

(a) Overview at package level

| | | | |
|---|---|---|---|
| **Packages** considered | **202** | average chunks | 15 |
| | | max chunks | 384 |

| | Initial | | Patched | |
|---|---|---|---|---|
| ✅ – fully compliant | 117 | 58% | 178 | 88% |
| 🛡 – only benign issues | 31 | 15% | 0 | 0% |
| ❌ – serious issues | 54 | 27% | 24 | 12% |

(b) Overview at chunk level

| | | | |
|---|---|---|---|
| Assembly chunks | 3107 | | |
| out-of-scope (e.g. floats) | 451 | | |
| **Relevant** chunks | **2656** | average size | 8 |
| | | max size | 341 |

| | Initial | | Patched | |
|---|---|---|---|---|
| ✅ – fully compliant | 1292 | 49% | 2568 | 97% |
| 🛡 – only benign issues | 1070 | 40% | 0 | 0% |
| ❌ – serious issues | 294 | 11% | 88 | 3% |

(c) Overview of found issues

| | Initial | | Patched | |
|---|---|---|---|---|
| **Found** issues | **2183** | | 183 | |
| significant issues | 986 | | 183 | |
| **frame-write** | **1718** | | 0 | |
| 🛡 – flag register clobbered | 1197 | 55% | 0 | 0% |
| ❌ – read-only input clobbered | 17 | 1% | 0 | 0% |
| ❌ – unbound register clobbered | 436 | 20% | 0 | 0% |
| ❌ – unbound memory access | 68 | 3% | 0 | 0% |
| **frame-read** | **379** | | 183 | |
| ❌ – non written write-only output | 19 | 1% | 0 | 0% |
| ❌ – unbound register read | 183 | 8% | 183 | 100% |
| ❌ – unbound memory access | 177 | 8% | 0 | 0% |
| **unicity** | **86** | 4% | 0 | 0% |

**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. ($\mathbf{qa_1}$) We run RUSTINA on known libatomic_obs and glibc compliance bugs and on their patched versions : every time, RUSTINA returns the expected result. ($\mathbf{qa_2}$) 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. ($\mathbf{qa_3}$) For compliance proofs, we also run the checker after patching: RUSTINA deems all patched chunks compliant. ($\mathbf{qa_4}$) 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 `q` (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 $\mathbf{qa_1}$ and $\mathbf{qa_2}$ from Sec. VII-A as follows: ($\mathbf{qa'_1}$) On known libatomic_obs and glibc compliance bugs, comparing RUSTINA-generated patches to originals shows that they are functionally equivalent, with similar fixes. ($\mathbf{qa'_2}$) 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 ($\mathbf{qa_3}$) and that several patches have been accepted by developers ($\mathbf{qa_4}$). 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. More details in Appx. B.

Table II: Inline assembly recurrent (compliance) error patterns

| Pattern | Omitted clobber | Additional context | Implicit protection | Details | Robust? | # issues | Known bug |
|---|---|---|---|---|---|---|---|
| P1 | `"cc"` | – | compiler choice | `"cc"` clobbered by default | ✅ (*) | 1197 | – |
| P2 | `%ebx` register | – | compiler choice | `%ebx` protected in PIC mode | ❌ (GCC $\geq$ 5) | 30 | [15] |
| P3 | `%esp` register | `push`/`pop` | compiler choice | `%esp` protected | ❌ (GCC $\geq$ 4.6) | 5 | [16] |
| P4 | `"memory"` | single-chunk function | function embedding | functions treated separately | ❌ (inlining, cloning) | 285 | [17] |
| P5 | MMX register | single-chunk function | ABI | MMX are ABI caller-saved | ❌ (inlining, cloning) | 363 | – |
| P6 | XMM register | disable XMM | compiler option | no XMM generation | ❌ (cloning) | 109 | – |

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

Table III: Submitted patches

| Project | About | Status | Patched chunks | Fixed issues | Commit |
|---|---|---|---|---|---|
| ALSA | Multimedia | Applied | 20 | 64/64 | 01d8a6e, 0fd7f0c |
| haproxy | Network | Applied | 1 | 1/1 | 09568fd |
| libatomic_obs | Multi-threading | Applied | 1 | 1/1 | 05812c2 |
| libtomcrypt | Cryptography | Applied | 2 | 2/2 | cefff85 |
| UDPCast | Network | Applied | 2 | 2/2 | 20200328 |
| xfstt | X Server | Applied | 1 | 3/3 | 91c358e |
| x264 | Multimedia | Applied | 11 | 83/83 | 69771 |
| ffmpeg | Multimedia | Review | 76 | 382/382[1] | |

[1] Including 27 non automatically patchable issues, manually fixed.

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.

## 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 (`-O3`). 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


[1] F. Recoules, S. Bardin, R. Bonichon, L. Mounier, and M. Potet, "Get rid of inline assembly through verification-oriented lifting," in 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19). IEEE, 2019.

[2] 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 (VEE'18). ACM, 2018.

[3] O. Stannard, "[llvm-dev] [rfc] checking inline assembly for validity," November 2018. [Online]. Available: http://lists.llvm.org/pipermail/llvm-dev/2018-November/127968.html

[4] GCC, "Extended asm - assembler instructions with c expression operands," 2020. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

[5] A. Fehnker, R. Huuck, F. Rauch, and S. Seefried, "Some assembly required - program analysis of embedded system code," in Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'08), 2008.

[6] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in 27th USENIX Security Symposium. USENIX Association, 2018.

[7] F. Recoules, S. Bardin, R. Bonichon, M. Lemerre, L. Mounier, and M. Potet, "RUSTInA in a nutshell," 2021. [Online]. Available: https://binsec.github.io/new/publication/1970/01/01/nutshell-icse-21.html

[8] G. A. Kildall, "A unified approach to global program optimization," in Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL'73. ACM, 1973.

[9] D. McCall, "Use of input/output operands in __asm__ templates not fully documented," 2007. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=30527

[10] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in 23rd International Conference on Computer Aided Verification (CAV 2011). Springer, 2011.

[11] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA framework for binary code analysis," in 23rd International Conference on Computer Aided Verification (CAV'11). Springer, 2011.

[12] A. Djoudi and S. Bardin, "Binsec: Binary code analysis with low-level regions," in Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference (TACAS'15). Springer, 2015.

[13] 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'16). IEEE, 2016.

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

[15] I. Maidanski, "Fix compare_double_and_swap_double for clang/x86 in pic mode," September 2012. [Online]. Available: https://github.com/ivmai/libatomic_ops/commit/64d81cd475b07c8a01b91a3be25e20eeca2d27ec

[16] ——, "Fix ao_compare_double_and_swap_double_full for gcc/x86 (pic mode)," Mars 2012. [Online]. Available: https://github.com/ivmai/libatomic_ops/commit/30cea1b9ea06c4c25cc219e1197dfac8dfa52083

[17] P. Pelletier, "Add "memory" as a clobber for bswap inline assembly," September 2011. [Online]. Available: https://github.com/libtom/libtomcrypt/commit/cefff85550786ec869b39c0cb4a5904e88c84319

[18] S. Boessenkool, "Bug 68095 – comment 4," 2015. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=68095#c4

[19] C. Parnin and C. Treude, "Measuring api documentation on the web," in Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering. ACM, 2011.

[20] D. Yang, A. Hussain, and C. V. Lopes, "From query to usable code: An analysis of stack overflow code snippets," in Proceedings of the 13th International Conference on Mining Software Repositories. ACM, 2016.

[21] S. Bardin, R. David, and J. Marion, "Backward-bounded DSE: targeting infeasibility questions on obfuscated codes," in International Symposium on Security & Privacy (S&P'17). IEEE, 2017.

[22] R. David, S. Bardin, J. Feist, L. Mounier, M. Potet, T. D. Ta, and J. Marion, "Specification of concretization and symbolization policies in symbolic execution," in Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16). ACM, 2016.

[23] L. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in International Symposium on Security and Privacy (SP'20). IEEE, 2020.

[24] J. Feist, L. Mounier, S. Bardin, R. David, and M. Potet, "Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free," in Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, SSPREW@ACSAC 2016. ACM, 2016.

[25] 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'17). IEEE, 2017.

[26] Microsoft, "Inline assembler," August 2018. [Online]. Available: https://docs.microsoft.com/en-us/cpp/assembler/inline/inline-assembler?view=vs-2019

[27] S. Maus, M. Moskal, and W. Schulte, "Vx86: x86 assembler simulated in c powered by automated theorem proving," in 12th International Conference on Algebraic Methodology and Software Technology (AMAST'08). Springer, 2008.

[28] S. Maus, "Verification of hypervisor subroutines written in assembler," Ph.D. dissertation, University of Freiburg, Germany, 2011.

[29] 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," in 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE'12). Springer, 2012.

[30] G. Balakrishnan and T. W. Reps, "WYSINWYX: what you see is not what you execute," ACM Trans. Program. Lang. Syst., vol. 32, no. 6, 2010.

[31] A. Djoudi, S. Bardin, and É. Goubault, "Recovering high-level conditions from binary programs," in FM 2016: Formal Methods - 21st International Symposium. Springer, 2016.

[32] S. Bardin, P. Herrmann, and F. Védrine, "Refinement-based CFG reconstruction from unstructured programs," in 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11). Springer, 2011.

[33] J. Kinder and D. Kravchenko, "Alternating Control Flow Reconstruction," in 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12). Springer, 2012.

## APPENDIX

This is supplementary material for paper "Interface Compliance of Inline Assembly: Automatically Check, Patch and Refine".

### A. Context and motivation: another example

*This section presents an interface compliance issue (frame condition) very similar to the one discussed in Sec. II (same project, similar function), leading to a deadlock. For the sake of clarity, it is written such that it is self-contained.*

Consider the code in Fig. 1 extracted from the libatomic_obs source code, commit 178cc98 (from 2005-10-11): it contains an inline assembly compliance bug responsible for a `deadlock`, patched 4 years later by commit 03e48c1 (2010-02-17).

```
122 /* Returns nonzero if the comparison succeeded. */
123 AO_INLINE int
124 AO_compare_and_swap_full(volatile AO_t *addr,
125                          AO_t old, AO_t new_val)
126 {
127   char result;
128   __asm__ __volatile__("lock; cmpxchgl %3, %0; setz %1"
129                        : "=m"(*addr), "=q"(result)
130                        : "m"(*addr), "r" (new_val), "a"(old) : "memory");
131   return (int) result;
132 }
```

(a) `atomic_ops/sysdeps/gcc/x86.h@178cc98`

Figure 11: Motivating example

**What the code is about.** This function uses inline assembly to implement the standard atomic primitive *Compare And Swap*.

The code comprises assembly instructions (`"lock; cmpxchgl %3, %0; setz %1"`), as an *assembly template* where some operands have been replaced by tokens (e.g., `%0`). The code also has a specification, the *interface*, binding together assembly registers, tokens and C entities. Line 129 declares the outputs, values expected to be assigned by the chunk; line 130 declares the inputs, 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. For instance, `"a"(old)` indicates that register `%eax` should be initialized with the value of `old` while `"=q"(result)` indicates the value of `result` should be collected from, at the choice of compiler, one of `%eax`, `%ebx`, `%ecx` or `%edx`. Note that inputs are read-only elements so that, if the compiler does not bind `%eax` to `result`, `%eax` is expected to hold `old` at the end template.

The crafted code in Fig. 12 calls this function to perform thread synchronization.

**An interface compliance bug, consequences and solution.** Using GCC (e.g., version 7.4), everything works well without optimization (`gcc -O0`), but compiling it with the first level of optimization (`gcc -O1`) is sufficient to create a *deadlock*.

The root cause is that the assembly interface fails to mention that the value in register `%eax` can be overwritten in the assembly template – admittedly, this is hard to see in the template itself, as it requires to know that the x86

```
#include <stdlib.h>
#include <pthread.h>
#include <atomic_ops.h>

volatile pthread_t pending = 0;

void *f (void *arg) {
  while (!AO_compare_and_swap_full(&pending, 0, pthread_self()));
  pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
  pthread_t t;
  int n = atoi(argv[1]);
  for (int i = n; i > 0; i -= 1) pthread_create(&t, NULL, &f, NULL);
  for (int i = n; i > 0; i -= 1) {
    while (t = pending, t == 0);
    pending = 0;
    pthread_join(t, NULL);
  }
  return 0;
}
```

Figure 12: A simple program impacted by the compliance bug

instruction `cmpxchgl` reads and modifies `%eax`. Thus, the compiler *wrongly* deduces that `%eax` is constant throughout the assembly chunk, and moves assignment `movl $0, %eax` out of the main loop, leading to the deadlock.

This problem was patched in 2010: register `%eax` is now used as an output of the statement (`"=a" (result)`), the compiler is aware that its value can change, no undue optimization is performed and the machine code is correct.

**RUSTINA.** RUSTINA automatically detects and patches the above bug (see Fig. 13a for the patch that we generate automatically, which is functionally equivalent to the one in Fig. 13b), and can formally verify compliance once either of the patches is applied.

```
@@ -125,8 +125,9 @@
                                AO_t old, AO_t new_val)
 {
   char result;
-  __asm__ __volatile__("lock; cmpxchgl %3, %0; setz %1"
-                       : "=m"(*addr), "=q"(result)
+  AO_t dummy;
+  __asm__ __volatile__("lock; cmpxchgl %4, %0; setz %1"
+                       : "=m"(*addr), "=q"(result), "=a"(dummy)
                    : "m"(*addr), "r" (new_val), "a"(old) : "memory");
   return (int) result;
 }
```

(a) RUSTINA generated patch

```
@@ -122,32 +122,49 @@
 /* Returns nonzero if the comparison succeeded. */
 AO_INLINE int
 AO_compare_and_swap_full(volatile AO_t *addr,
-                               AO_t old, AO_t new_val)
+                             AO_t old, AO_t new_val)
 {
   char result;
   __asm__ __volatile__("lock; cmpxchgl %3, %0; setz %1"
-                       : "=m"(*addr), "=q"(result)
-                   : "m"(*addr), "r" (new_val), "a"(old) : "memory");
+                   : "=m"(*addr), "=a"(result)
+                   : "m"(*addr), "r" (new_val), "a"(old) : "memory");
   return (int) result;
 }
```

(b) Developer's patch (+03e48c1-178cc98)

Figure 13: Patching the issue

### B. Internal evaluation: precision optimizations (**RQ5**)

*This section complements Sec. VII-D.*

We provide a detailed view of the results in Table IV.

Table IV: Benefits of precision enhancement

| Compliance issues | real issues | Basic fp | Basic fpr | RUSTInA fp | RUSTInA fpr |
|---|---|---|---|---|---|
| Total | 2183 | +127 | 6% | 0 | 0% |
| significant | 986 | +127 | 13% | 0 | 0% |
| **frame-read** | 379 | +87 | 23% | 0 | 0% |
| significant | 379 | +87 | 23% | 0 | 0% |
| **frame-write** | 1718 | +40 | 2% | 0 | 0% |
| significant | 521 | +40 | 8% | 0 | 0% |
| **unicity** | 86 | 0 | 0% | 0 | 0% |
| significant | 86 | 0 | 0% | 0 | 0% |

. Basic represents a version of RUSTInA with no enhancement at all
. fp: number of false positives
. fpr: false positive rate computed as fp/(real + fp)
. Expression folding removes all **frame-write** alarms
. Bit-level liveness removes all **frame-read** alarms

### *C. Check, Patch and refine: more details*

*This section complements Sec. V.*

*1) Data-flow technical details:* Data-flow analysis is a well known framework [8] to compute a fix-point over a program control-flow graph (CFG). Alg. 1 shows a standard implementation. The technique is generic over graph implementation and type of data that are computed.

**Graph** A graph `cfg` is made of `node` and `edge`. Each transition (`edge`) is a tuple containing the initial node (`src`), the destination node (`dst`) and the DBA instruction (`instr`);

**Direction** The analysis starts from the node given by `StartNode : cfg` $\rightarrow$ `node` (typically, entry node or exit node). Function `SuccEdges : node` $\rightarrow$ `edge set` returns the transitions to follow from the current node (typically, forward successors or backward predecessors);

**Data** The algorithm computes for each node the corresponding data-flow fact (`data`) which is generally a set. Function `Init : node` $\rightarrow$ `data` returns the initial value of a given `node` (usually an empty set). Functions `Transfer : edge` $\times$ `data` $\rightarrow$ `data` and `Join : data` $\times$ `data` $\rightarrow$ `data` define how a transition transmits dataflow facts and how the information flowing from different edges to the same node are merged together.

The algorithm works by fixpoint iteration. As `Join` can only grows the `data` information computed on each node, the information keeps growing. If the set of dataflow facts (`data`) is finite-value then convergence of the algorithm is ensured.

**Basic frame-write.** This forward dataflow analysis simply collects the set of `locations` (token, register and memory) that are written in the program. In the basic version, memory writes are all squashed as `memory`, avoiding complications inherent to memory analysis. The adequate configuration for this dataflow analysis is given in Alg. 2.

**Algorithm 1:** Data-flow analysis template

```
Require: StartNode, SuccEdges, Init, Transfer, Join

result[node] ⟵ Init(node) foreach node in cfg
stack ⟵ SuccEdges(StartNode(cfg))
while stack ≠ ∅ do
  | edge ⟵ Pop(stack)
  | data ⟵ Transfer(edge, result[edge.src])
  | data ⟵ Join(data, result[edge.dst])
  | if data ≠ result[edge.dst] then
  |   | result[edge.dst] ⟵ data
  |   | stack ⟵ stack ∪ SuccEdges(edge.dst)
end
```

**Algorithm 2:** Basic frame-write configuration

```
type data : location set
define StartNode(cfg) = cfg.entryNode
define SuccEdges(node) = node.forwardEdges
define Init(node) = ∅
define Transfer(edge, data) =
  | match edge.instr with
  |   | case v ← _ : data ∪ { v }  // register or token
  |   | case @[_] ← _ : data ∪ { memory }  // store
  |   | otherwise : data  // branches
define Join(data, data') = data ∪ data'
```

**Basic frame-read.** Known as *liveness* analysis, this backward dataflow analysis collects the set of every location (token, register and memory) that are read but not set. In the basic version, memory read are again squashed as `memory` (see the frame-write case) and expression dependencies are computed at word level, as shown in function Deps.

**Function** Deps(expr) : world level dependencies

```
match expr with
  | case v : { v }  // register or token
  | case @[a] : Deps(a) ∪ { memory }  // load
  | case _ x : Deps(x)  // unary
  | case x _ y : Deps(x) ∪ Deps(y)  // binary
  | case x ? y : z : // ternary
  |   | Deps(x) ∪ Deps(y) ∪ Deps(z)
  | otherwise : ∅  // constant
```

Data-flow configuration is given in Alg. 3 where `Outputs` are the tokens declared in the interface as $B^{\circ}$ (output tokens).

**Algorithm 3:** Basic frame-read configuration

```
type data : location set
define StartNode(cfg) = cfg.exitNode
define SuccEdges(node) = node.backwardEdges
define Init(node) = if node.isExit then Outputs else ∅
define Transfer(edge, data) =
  | match edge.instr with
  |   | case v ← x when v ∈ data : // live
  |   |   | ( data \ {v} ) ∪ Deps(x)
  |   | case @[a] ← x : // store
  |   |   | data ∪ Deps(a) ∪ Deps(x)
  |   | case if x then goto _ else goto _ :
  |   |   | data ∪ Deps(x)
  |   | otherwise : data
define Join(data, data') = data ∪ data'
```