IsaBIL: A Framework for Verifying (In)correctness of Binaries in Isabelle/HOL

Matt Griffin
University of Surrey, Guildford, UK
Imperial College London, UK

Brijesh Dongol
University of Surrey, Guildford, UK

Azalea Raad
Imperial College London, UK

Abstract

This paper presents IsaBIL, a binary analysis framework in Isabelle/HOL that is based on the widely used Binary Analysis Platform (BAP). Specifically, in IsaBIL, we formalise BAP's intermediate language, called BIL and integrate it with Hoare logic (to enable proofs of *correctness*) as well as incorrectness logic (to enable proofs of *incorrectness*). IsaBIL inherits the full flexibility of BAP, allowing us to verify binaries for a wide range of languages (C, C++, Rust), toolchains (LLVM, Ghidra) and target architectures (x86, RISC-V), and can also be used when the source code for a binary is unavailable.

To make verification tractable, we develop a number of big-step rules that combine BIL's existing small-step rules at different levels of abstraction to support reuse. We develop high-level reasoning rules for RISC-V instructions (our main target architecture) to further optimise verification. Additionally, we develop Isabelle proof tactics that exploit common patterns in C binaries for RISC-V to discharge large numbers of proof goals (often in the 100s) automatically. IsaBIL includes an Isabelle/ML based parser for BIL programs, allowing one to *automatically* generate the associated Isabelle/HOL program locale from a BAP output. Taken together, IsaBIL provides a highly flexible proof environment for program binaries. As examples, we prove correctness of key examples from the Joint Strike Fighter coding standards and the MITRE database.

2012 ACM Subject Classification Theory of computation → Program reasoning

Keywords and phrases Binary Analysis Platform, Isabelle/HOL, Hoare Logic, Incorrectness Logic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.14

Related Version *Extended Version*: `https://arxiv.org/abs/2504.16775` [21]

Supplementary Material *Other (Isabelle/HOL)*: `https://doi.org/10.5281/zenodo.15261359` [20]

Funding *Brijesh Dongol*: Supported by VeTSS and EPSRC grants EP/X037142/1, EP/X015149/1, EP/V038915/1, and EP/R025134/2.
Azalea Raad: Supported by the UKRI Future Leaders Fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1 and by VeTSS.

1 Introduction

Analysing binary code is highly challenging and is essential for decompilation and verification when one does not have access to the source code (e.g., proprietary code). It is also a key tool for finding security vulnerabilities and exploits and is often the only way to (dis)prove properties about programs [46]. Many tools for binary analysis such as angr [52], LIEF [49] and Manticore [36] lift machine code to higher-level, human-readable representations. These tools have been developed for different goals covering different languages (Rust, C++), hardware architectures (e.g., x86, RISC-V), types of binaries (ELF, smart contracts) and security analysis (malware analysis, reverse engineering).

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 14; pp. 14:1–14:30
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The *Binary Analysis Platform* (BAP) [8] provides a generic approach to binary analysis. BAP natively supports several languages (C, Python and Rust) and hardware architectures (x86, Arm, RISC-V, MIPS) and additionally can be integrated with toolchains such as LLVM and Ghidra to analyse programs on less common architectures such as Atmel AVR [2]. BAP is well-supported and widely used with an active user community. The platform can analyse binaries derived from a particular source language such as C/C++ and Rust, as well as closed-source functions. BAP provides a disassembler that lifts binaries into a unified intermediate language called *BIL (Binary Instruction Language)*, thus enabling architecture-agnostic analysis. While BAP also provides a set of static analysis tools, they do not provide mechanisms for formal verification, e.g., logics for (dis)proving correctness of programs. Recent works have covered subsets of BAP (called AIR) in UCLID5 [10] and use an extended semantics to enable reasoning about an information flow hyper-property known as *trace-property observational determinism* [10].

We present IsaBIL, a verification framework for low-level binaries encoded using the Isabelle/HOL theorem prover. IsaBIL provides a *deep embedding* of BIL programs into Isabelle/HOL, providing a *direct interface* between BAP and Isabelle/HOL without using any additional tools. We leverage the fact that BAP can generate BIL programs as abstract data types, providing a convenient hook for IsaBIL's ML-based parser.[1] Since BAP can lift binaries into BIL both with and without the source code being available, so can IsaBIL. Moreover, IsaBIL allows the generated BIL program to be connected to *any* operational semantics. We build on the semantics described by [8] (see § 3) to enable bit-precise analysis. Our approach generates an Isabelle *locale* [27], which provides an extensible interface that can be instantiated to different scenarios (see Figure 2). We demonstrate this by both proving (using Hoare logic [23]) and disproving (using Incorrectness logic [42]) properties for a number of key examples (see Table 1).

Table 1 IsaBIL examples and their sizes.

Test	**Size (LoC)**			
	C	**RISC-V**	**BIL**	**Isabelle**
AV Rule 17	13	34	80	94
AV Rule 19	10	16	36	64
AV Rule 20	18	52	118	393
AV Rule 21	16	45	105	210
AV Rule 23	11	35	81	90
AV Rule 24	15	40	107	74
AV Rule 25	24	99	233	578
DF (good)	27	19	44	366
DF (bad)	27	19	43	171
read_data (bad)	74	131	296	474
sec_recv (bad)	38	122	272	752
Total	**273**	**612**	**1415**	**3266**

Scalability. Our approach is *compositional* in that we can analyse and (dis)prove properties about a *code fragment*, say a function f, *in isolation* and then *reuse* these properties (without reproving them) in bigger contexts where f is used. To show this, we use our technique to detect (using incorrectness logic) a *double-free vulnerability* in the cURL library (used primarily for transferring data over the internet). As we demonstrate in § 7, although this

[1] Note that this is a significant advancement over the encoding by Griffin and Dongol [19], which used an external Python tool to translate BAP outputs to Isabelle/HOL.

Listing 1 Double Free (Bad).

```
void bad() {
    int *p = malloc(42);
    if (MyTrue) {
        free(p);
        // ...
    }
    free(p);
}
```

Listing 2 Double Free (Good).

```
void good() {
    int *p = malloc(42);
    if (MyTrue) {
        free(p);
        return;
    }
    free(p);
}
```

vulnerability is "hidden" in an internal function (not exposed in a header file) and is not visible to client applications, it is accessible by a call chain (comprising four separate functions) that can be triggered from an external function, i.e. it is accessible from within a client application and thus this vulnerability can be easily exploited. Indeed, this vulnerability has been previously documented as CVE-2016-8619.

Contributions. Our main contributions are as follows. **(1)** We develop IsaBIL, a flexible, semi-automated tool for verifying binaries (lifted to BIL) within the Isabelle/HOL theorem prover. Our mechanisation is complete with respect to the BIL language and semantics. During the mechanisation, we uncovered and fixed several inconsistencies in the official BAP documentation (see § 3). **(2)** We developed IsaBIL as a highly flexible and extensible system (see Figure 2) using Isabelle *locales* [3, 27]. We incorporate logics for (in)correctness, thus obtain methods for both proving and disproving properties. To the best of our knowledge, this is the first application of Incorrectness logic [42] in the analysis of low-level binaries. **(3)** We extend IsaBIL with a number of automation techniques, including reusable high-level big-step theorems, proof tactics that automatically discharge large numbers of proof goals, and architecture-specific proof optimisations. We also include a native parser for BIL_{ADT} programs, written in Isabelle/ML, to *automatically* transpile BAP outputs to an Isabelle/HOL locale suitable for verification. **(4)** We apply our methods to a number of examples (see Table 1), including key tests from the Joint Strike Fighter (JSF) C++ Coding Standards [32]. **(5)** We show the scalability of IsaBIL by detecting a double-free vulnerability in a large example (253 assembly LoC) in the cURL library.

2 Motivation and Workflow

In this section, we motivate our work and provide an overview of our overall approach using two running examples. We give the C program and their equivalent BIL in § 2.1. We present our workflow and an overview of the Isabelle/HOL mechanisation in § 2.2.

2.1 Two C programs and their BIL representations

We motivate our analysis using two variations of a double-free vulnerability (listed as CWE-415 in the MITRE database[2]). A double-free error occurs when a program calls `free` twice with the same argument. This may cause two later calls to `malloc` to return the same pointer, potentially giving an attacker control over the data written to memory.

The occurrence of the double-free vulnerability is straightforward to see in Listing 1, but of course, in a real program, the vulnerability may be much more difficult to identify. Listing 2 presents an alternative program that contains two occurrences of `free(p)` with no intervening `malloc`. However, Listing 2 does not contain a double-free vulnerability since the true branch returns from the method call before the second `free(p)` is executed.

[2] https://cwe.mitre.org/data/definitions/415.html

Listing 3 Pretty printed BIL of Listing 1.

```
sub bad(bad_result)

X10 := 0x2A
call @malloc with return 5
mem := mem with [X8 - 0x18, el]:u64 <- X10

X15 := mem[X3 - 0x7C8, el]:u32
when (X15 = 0) goto 13
goto 10

X10 := mem[X8 - 0x18, el]:u64
call @free with return 13

X10 := mem[X8 - 0x18, el]:u64
call @free with return 16

call X1 with noreturn
```

Listing 4 Pretty printed BIL of Listing 2.

```
sub good(good_result)

X10 := 0x2A
call @malloc with return 5
mem := mem with [X8 - 0x18, el]:u64 <- X10

X15 := mem[X3 - 0x7C8, el]:u32
when (X15 = 0) goto 13
goto 10

X10 := mem[X8 - 0x18, el]:u64
call @free with return 16

X10 := mem[X8 - 0x18, el]:u64
call @free with return 16

call X1 with noreturn
```

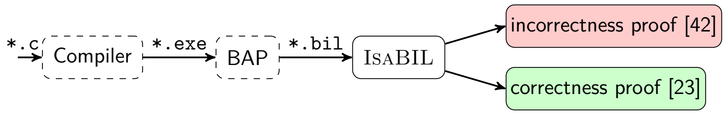

Figure 1 Translation of C source programs to an Isabelle/HOL proof.

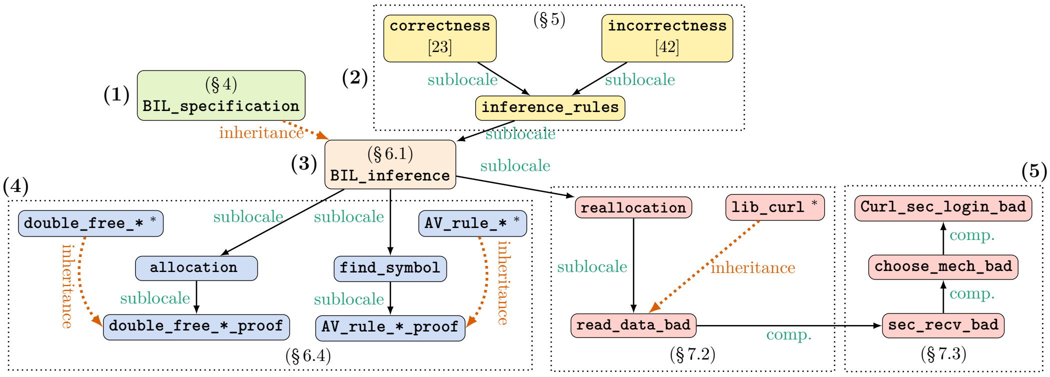

Figure 2 Structure of IsaBIL locales: theories indicated by * are auto-generated from a given `*.bil` file following the translation in Figure 1.

The focus of our work is to analyse such programs *without* relying on any high-level language semantics. Instead, we aim to develop a technique for reasoning about programs (e.g., those in Listings 1 and 2) at a lower level of abstraction by first compiling the programs then, lifting the resulting binaries to an intermediate representation (BIL). Although such analysis contains more detail, the analysis reflects the behaviour of the *actual* executable, thus is more accurate (e.g., does not require any additional compiler correctness assumptions).

2.2 Workflow and Mechanisation

Our overall workflow is given in Figure 1, where the dashed steps represent existing work. The first step to proving properties about our example program is to use BAP to generate a BIL representation of a binary executable (see Figure 1). For our example program in Listings 1 and 2, the BIL equivalent (expressed in BIR format) are given in Listings 3 and 4, respectively. BIR (Binary Intermediate Representation) is a structured refinement of BIL that significantly enhances human readability by presenting sequences of BIL as a single instruction. Take for example the BIR instruction `call @free with return 18` on line 16 in Listings 3 and 4. This represents two BIL statements, a jump to the address of the symbol "free", and the assignment of the return value 18 to the return pointer. Note that IsaBIL takes as input BAP's *abstract data type* format (see § 4), which we refer to as BIL_{ADT}, as opposed to the human-readable forms shown in Listings 3 and 4. This makes the BIL instructions simpler to parse when generating the corresponding Isabelle/HOL object (discussed in more detail below).

Extensions to Isabelle/HOL's logic [15, 5, 26, 18] often make use of Isabelle/HOL's *locale* system [27, 3], which provides convenient modules for building parametric theories. Locales are aimed at supporting the use of local assumptions and definitions for collections of similar theories, defined in terms of fixed variables and definitions. We refer to the fixed variables as parameters which must satisfy the local assumptions. For a locale $\mathcal{L}$, we give its definition as $\mathcal{L} = (p_1, p_2, \dots)$ where p_1, p_2 are the parameters of the locale. Each parameter may be any type derived from a set or a relation. From these variables, assumptions and definitions we can derive general theorems within the context of a locale. These theorems can be exported to the current proof context by *instantiation*, in which we provide values to one or all of the parameters and verify the fixed assumptions of these parameters. This effectively gives us the general theorems contained within each locale for free for a specific instantiation. Locales can be extended through *inheritance* or *interpretation*, enabling reusability and specialisation by introducing and refining a locale's parameters. Inheritance operates hierarchically, directly propagating the context of a parent locale to a child locale. This transfer includes general theorems, assumptions, definitions and fixed variables. Interpretation is compositional, associating a locale or definition with another locale. During interpretation, the assumptions of the parent locale must be satisfied as part of the mapping process. A locale that interprets another locale is referred to as a sublocale, denoted $\mathcal{L} \subseteq \mathcal{L}'$, where locale $\mathcal{L}'$ is the parent of $\mathcal{L}$. For IsaBIL, they provide a uniform system for representing both the program, and the underlying reasoning framework.

The overall structure of the IsaBIL development is given in Figure 2, and comprises four main parts (which highlight our four main contributions). Overall, we have: **(1)** a locale (see § 4), representing a complete formalisation of the BIL specification from § 3, **(2)** a set of locales (see § 5) that encode Hoare Logic (to verify correctness) and Incorrectness Logic (as developed by O'Hearn), **(3)** a locale (see § 6.1) that combines formalisation of the BIL specification with (in)correctness logics, **(4)** a set of locales (see § 6.4) that apply the resulting verification framework to prove correctness of a number of examples from the literature and **(5)** a set of locales (see § 7.2 and 7.3) that demonstrate scalability of the approach. In particular, § 7.2 demonstrates the use of subroutines to enable one to focus on key functions of interest, even when starting with large BIL files (> 100k LoC). Then § 7.3 describes how verified subroutines can be used as subcomponents to prove (in)correctness of a larger system.

Our framework is highly flexible and extensible. For instance, the **`BIL_inference`** locale combines the **`BIL_specification`** and **`inference_rules`** locales. In case one wishes to change the specification (for a different operational semantics), or use a different set of inference rules (to perform a type of analysis), one can simply change the specification and/or inference rule locales. Similarly, a locale can be extended with additional semantic features: our proofs extend the **`BIL_inference`** locale with an **`allocation`** locale (that models memory allocation) and a **`find_symbol`** locale (that tracks a symbol table) to enable reasoning about different program features.

Our encoding of BIL (§ 4) is the *first* full mechanisation of BIL in Isabelle/HOL. This mechanisation is a standalone contribution and can be used by others outside of our reasoning framework. Our encoding uncovers some inconsistencies in the official specification, which are addressed by the mechanisation (see § 3). Moreover, given that BIL is an intermediate language for many different architectures, future versions could be further optimised to a particular architecture. As an example, we present optimisations for RISC-V (see § 6.2.3), which is the architecture that we focus on.

To maximise flexibility, we provide an encoding of the inference rules parameterised by an operational semantics, i.e., an *instance* of the correctness and incorrectness rules can be generated for *any* operational semantics. This instance automatically inherits all of the rules of Hoare and Incorrectness logic that we prove generically in the **`correctness`** and **`incorrectness`** locales (§ 5).

The BIL semantics from § 4 and the inference rules from § 5 are combined to form a new locale, **`BIL_inference`**, which provides facilities to verify BIL programs. Then, as discussed above, to verify particular types of examples, we extend the locales with models of allocation and symbols to verify particular sets of examples. These proofs represent the *first* proofs of both correctness and incorrectness for BIL programs in Isabelle/HOL. As we shall see in § 3, the BIL semantics is highly detailed, with the reductions related to even single load command potentially splitting into a large number of *small-step* transitions. This has the potential to increase the verification complexity making the proofs of even small programs intractable. To address this, we use BIL's big-step semantics (§ 6.1) and introduce a number of high-level lemmas that allow one to discharge such proofs generically. These lemmas are reusable across all of the examples that we verify.

Our proofs cover key examples from the JSF coding standards [32] and the CWE database[2]. This task is greatly simplified using IsaBIL's parser, written in Isabelle/ML [50], that *automatically generates* a locale corresponding to a BIL program written in BIL_{ADT} format. Note that Isabelle isolates Isabelle/ML programming, and hence our parser cannot interfere with soundness of Isabelle's core logic engine. The generated locale is then combined with **`BIL_inference`** to provide a context in which (in)correctness proofs can be carried out.

3 BIL Syntax and Semantics

Before proceeding with the presentation of our proofs, it is essential to acquire a basic understanding of BIL. This section serves as a succinct introduction to BIL, emphasizing its type system, memory model and operational semantics. We encourage interested readers to explore the BIL manual [8].

3.1 Syntax

The basic syntax of BIL programs is given in Figure 3.

Statements. A BIL statement may assign an expression e to a variable var, transfer control to a given address e, interrupt the CPU with a given interrupt num, be an instruction with unknown semantics, a while loop, or an if-then-else conditional (with an optional else clause). Note that each BIL statement (including a compound statement) corresponds to a single machine instruction, thus side effects such as setting a status register must be captured in the statement definition (see § 3.4).

Statements, where *str* denotes a string (of type *string*) and $i \in \mathbb{Z}$

$$\begin{aligned} \mathit{stmt} \ni s &::= \mathit{var} := e \mid \textbf{jmp}\ e \mid \textbf{cpuexn}(i) \mid \textbf{special}(\mathit{str}) \mid \textbf{while}\ (e)\ \mathit{seq} \\ &\quad \mid \textbf{if}\ (e)\ \mathit{seq} \mid \textbf{if}\ (e)\ \mathit{seq}_1\ \textbf{else}\ \mathit{seq}_2 \\ \mathit{bil} \ni \mathit{seq} &::= \mathit{stmt}^* \end{aligned}$$

Expressions, where *id* denotes a variable name literal (of type *string*)

$$\begin{aligned} \mathit{exp} \ni e &::= v \mid \mathit{var} \mid e_1[e_2, \mathit{ed}] : \mathit{sz} \mid e_1\ \textbf{with}\ [e_2, \mathit{ed}] : \mathit{sz} \leftarrow e_3 \mid e_1\ \mathit{bop}\ e_2 \mid \mathit{uop}\ e \\ &\quad \mid \mathit{cast} : \mathit{sz}[e] \mid \textbf{let}\ \mathit{var} = e_1\ \textbf{in}\ e_2 \mid \textbf{ite}\ e_1\ e_2\ e_3 \mid e_1\ @\ e_2 \mid \textbf{extract}\colon \mathit{sz}_1 \colon \mathit{sz}_2[e] \end{aligned}$$

$$\mathit{bop} ::= \mathit{aop} \mid \mathit{lop} \qquad \mathit{var} ::= \mathit{id} : \mathit{type} \qquad \mathit{endian} \ni \mathit{ed} ::= \textbf{el} \mid \textbf{be}$$

$$\mathit{cast} ::= \textbf{low} \mid \textbf{high} \mid \textbf{signed} \mid \textbf{unsigned}$$

Types and Values, where $\mathit{sz}, \mathit{nat} \in \mathbb{N}$

$$\mathit{type} \ni t ::= \textbf{imm}\langle \mathit{sz} \rangle \mid \textbf{mem}\langle \mathit{sz}_{\mathit{addr}},\ \mathit{sz}_{\mathit{val}} \rangle \qquad \mathit{word} \ni w ::= \mathit{nat} :: \mathit{sz}$$

$$\mathit{value} \ni v ::= w \mid v[w \leftarrow v',\ \mathit{sz}_{\mathit{val}}] \mid \textbf{unknown}[\mathit{str}] : t$$

Figure 3 BIL syntax and types.

Expressions. Memory loads and stores are represented at the expression level as $e_1[e_2, \mathit{ed}] : \mathit{sz}$ for loads, and e_1 **with** $[e_2, \mathit{ed}] : \mathit{sz} \leftarrow e_3$ for stores. In these expressions, the evaluation of e_1 represents the target storage being accessed, while the evaluation of e_2 specifies the address. Since a memory operation can span multiple addresses, both the endianess (*ed*) and the size (*sz*) of the operation are provided. For stores, the evaluated value of e_3 is written to memory at the given address. Binary operations now differentiate arithmetic (*aop*) and logical (*lop*) operations. Expressions in BIL also include casts, let bindings, if-then-else expressions, concatenations and extractions. The extraction operation, denoted as $\textbf{extract}\colon \mathit{sz}_1 \colon \mathit{sz}_2[e]$, first evaluates the expression e to a word w and then extracts a slice from w using BIL's existing extraction mechanism. Specifically, $\textbf{ext}\ w \backsim \textbf{hi} : \mathit{sz}_1 \backsim \textbf{lo} : \mathit{sz}_2$ represents the bits of w from bit sz_1 to bit sz_2. Thus, for example, $\textbf{ext}\ 01001011 \backsim \textbf{hi} : 5 \backsim \textbf{lo} : 2$ is 0010. Expressions are side-effect free and are evaluated wrt a state $\Delta : \mathit{var} \rightarrow \mathit{val}$ mapping variables to values.

Types and Values. The highly expressible nature of BIL is due to its type system, which defines two distinct types for its values, the irreducible subset of a BIL expression. These types are $\textbf{imm}\langle \mathit{sz} \rangle$ (describing an *immediate type* of size *sz*) and $\textbf{mem}\langle \mathit{sz}_{\mathit{addr}},\ \mathit{sz}_{\mathit{val}} \rangle$ (describing a *storage type* with address size $\mathit{sz}_{\mathit{addr}}$ and value size $\mathit{sz}_{\mathit{val}}$). A value can be bound to one of the three possibilities. **(1)** A machine word (w). **(2)** An abstract storage ($v[w \leftarrow v',\ \mathit{sz}_{\mathit{val}}]$), which allows us to define a memory as a chain of mappings, where $[w \leftarrow v', \mathit{sz}_{\mathit{val}}]$ defines a single mapping from an address (w) to a value (v') of size $\mathit{sz}_{\mathit{val}}$. Here, v could either be the root of the chain (in which case it is $\textbf{unknown}[\mathit{str}] : t$) or another storage. Technically, v could also be a word, but we disallow this by introducing a typing context (see below). Similarly, v' could be a storage according to the grammar, but this possibility is also eliminated by the typing context. When v or v' is an unknown value, we require that type of v is **mem** and the type of v' is **imm**. **(3)** An "unknown" value ($\textbf{unknown}[\mathit{str}] : t$), which is obtained from the evaluation of a BIL expression holding some string information *str* of some type t.

The type of a value can be obtained using the function `type`, which is define as follows:

$$\begin{aligned} \texttt{type}(\mathit{nat} :: \mathit{sz}) &= \textbf{imm}\langle \mathit{sz} \rangle \\ \texttt{type}(v[(\mathit{nat} :: \mathit{sz}_{\mathit{addr}}) \leftarrow v',\ \mathit{sz}_{\mathit{val}}]) &= \textbf{mem}\langle \mathit{sz}_{\mathit{addr}},\ \mathit{sz}_{\mathit{val}} \rangle \\ \texttt{type}(\textbf{unknown}[\mathit{str}] : t) &= t \end{aligned}$$

In [21] we show the lifted BIL equivalent of a RISC-V program using the syntax presented in this section.

3.2 Typing Rules and Typing Context

BIL's type system facilitates the need for typing rules to ensure type correctness. At the lowest level, the rules ensure that types are correctly defined with the predicate t **is ok**. The type of variables must be tracked during symbolic execution to ensure that they do not change and only values of the correct type can be assigned. This is achieved via a *typing context*, defined as $\Gamma ::= \mathit{var}^*$, where type correctness of Γ is expressed by overloading the predicate Γ **is ok**. The rules for t **is ok** and Γ **is ok** are given below:

$$\text{(TWF\_IMM)}\ \frac{sz > 0}{\mathbf{imm}\langle sz\rangle\ \textbf{is ok}} \qquad \text{(TWF\_MEM)}\ \frac{sz_{addr} > 0 \qquad sz_{val} > 0}{\mathbf{mem}\langle sz_{addr},\ sz_{val}\rangle\ \textbf{is ok}} \qquad \text{(TG\_NIL)}\ \frac{}{[]\ \textbf{is ok}}$$

$$\text{(TG\_CONS)}\ \frac{\mathit{str} \notin \mathit{dom}(\Gamma) \qquad t\ \textbf{is ok} \qquad \Gamma\ \textbf{is ok}}{(\mathit{str} : t)\ \#\ \Gamma\ \textbf{is ok}}$$

These are extended to expressions, then to the level of statements (see [8] for details). As part of our formalisation, we discover a missing typing rule for empty sequences, which would otherwise be needed to ensure the sequencing rules are type correct. We have created a pull request in the BAP repositories for this issue (see `https://github.com/BinaryAnalysisPlatform/bap/pull/1588`), which has now been merged.

3.3 Expression Semantics

Expression evaluation requires the repeated application of small-step semantics rules until the expression is reduced to a value. An expression step in BIL is formally expressed with the $\Delta \vdash e \rightsquigarrow e'$, where multiple steps can be reduced using the reflexive transitive closure $\Delta \vdash e \rightsquigarrow^* e'$. Recall that loads and stores occur at the expression level, and are sensitive to endian orderings. We provide the load rules here, and refer the interested reader to the BIL manual [8] for further details.

A store instruction (semantics not shown) targets a single unit of addressable memory, typically 8-bits in size. When a storage operation exceeds this size, it is converted into a sequence of 8-bit stores, organised in big-endian order. The reduction rules for load operations is given in Figure 4. First, all sub-expressions of a memory object are reduced to values using the rules LOAD__STEP__ADDR and LOAD__STEP__MEM. Then, the resulting object is recursively deconstructed using the LOAD__WORD__EL and LOAD__WORD__BE rules, depending on the endian. This process continues until one of LOAD__BYTE, LOAD__UN__MEM or LOAD__UN__ADDR is used. LOAD__BYTE reduces the expression to the value, v', when the memory object is a storage of an immediate (known) value, w. LOAD__UN__MEM and LOAD__UN__ADDR both reduce the expression to an unknown value when the memory or address being loaded is unknown, respectively.

▶ **Example 1.** Consider the instruction, `X10 := mem[X8-0x18, el]:u64`, from line 12 in Listing 3, which performs a little-endian load of a 64-bit word from address `X8-0x18` in the variable `mem`. We first reduce the address expression, `X8-0x18`, to a value using LOAD__STEP__ADDR, next we apply LOAD__STEP__MEM to read the value stored in `mem`. Next,

$$\textsc{(var\_\_in)}\ \frac{(var, v) \in \Delta}{\Delta \vdash var \leadsto v} \qquad \textsc{(load\_\_step\_\_addr)}\ \frac{\Delta \vdash e_2 \leadsto e_2'}{\Delta \vdash e_1[e_2, ed] : sz \leadsto e_1[e_2', ed] : sz} \qquad \textsc{(load\_\_step\_\_mem)}\ \frac{\Delta \vdash e_1 \leadsto e_1'}{\Delta \vdash e_1[v_2, ed] : sz \leadsto e_1'[v_2, ed] : sz}$$

$$\textsc{(load\_\_byte)}\ \frac{}{\Delta \vdash v[w \leftarrow v',\ sz][w, ed] : sz \leadsto v'} \qquad \textsc{(load\_\_byte\_\_from\_\_next)}\ \frac{w_1 \neq w_2}{\Delta \vdash v[w_1 \leftarrow v',\ sz][w_2, ed] : sz \leadsto v[w_2, ed] : sz}$$

$$\textsc{(load\_\_un\_\_mem)}\ \frac{}{\begin{array}{c}\Delta \vdash (\mathbf{unknown}[str] : t)[v, ed] : sz \leadsto \\ \mathbf{unknown}[str] : \mathbf{imm}\langle sz\rangle\end{array}} \qquad \textsc{(load\_\_un\_\_addr)}\ \frac{}{\begin{array}{c}\Delta \vdash v[w_1 \leftarrow v',\ sz'][\mathbf{unknown}[str] : t, ed] : sz \leadsto \\ \mathbf{unknown}[str] : \mathbf{imm}\langle sz\rangle\end{array}}$$

$$\textsc{(load\_\_word\_\_be)}\ \frac{\begin{array}{c} sz > sz_{mem} \qquad \mathtt{succ}(w) = w' \\ \mathtt{type}(v) = \mathbf{mem}\langle sz_{addr},\ sz_{mem}\rangle \end{array}}{\begin{array}{c}\Delta \vdash v[w, \mathbf{be}] : sz \leadsto \\ v[w, \mathbf{be}] : sz_{mem} \ @\ v[w', \mathbf{be}] : (sz - sz_{mem})\end{array}} \qquad \textsc{(load\_\_word\_\_el)}\ \frac{\begin{array}{c} sz > sz_{mem} \qquad \mathtt{succ}(w) = w' \\ \mathtt{type}(v) = \mathbf{mem}\langle sz_{addr},\ sz_{mem}\rangle \end{array}}{\begin{array}{c}\Delta \vdash v[w, \mathbf{el}] : sz \leadsto \\ v[w', \mathbf{el}] : (sz - sz_{mem}) \ @\ v[w, \mathbf{el}] : sz_{mem}\end{array}}$$

Figure 4 Reduction rules for Load.

we look up the value of mem in Δ in VAR__IN, which reduces mem to the corresponding storage. Next, we apply LOAD__WORD__EL to break down the 64-bit load into 8 separate loads in little-endian format. Each of the 8 byte-sized loads are individually read from memory using LOAD__BYTE and LOAD__BYTE__FROM__NEXT. Finally, we apply BIL's concatenation rule (not shown) to join each byte into a single 64-bit word.

IsaBIL Extensions. The previous sections present a summary of BIL's semantics from the BIL manual [8] with some minor corrections. In this section, we present extensions to BIL's big-step expression evaluation semantics. The reduction of expressions, e.g., in load and store instructions, results in a significant proof burden. To make verification tractable, IsaBIL defines additional big-step expression evaluation rules which are derived from a combination of BIL's existing small-step and big-step expression semantics. For example, the load expression in Example 1 requires the repeated application of over 150 small-step rules. Splitting the 64-bit read into individual reads in Example 1 (from LOAD__WORD__EL onwards) and the concatenation of the result can be combined into a single big-step rule. We first provide some syntactic sugar for a storage whose size is a multiple of 8.

The definition uses the function $\mathtt{succ}(w)$, which retrieves the successor of the word w such that $\mathtt{succ}(w) ::= w + 1$. Furthermore, we use $\mathtt{storage}_{en}\left(v, w, v', N\right)$ to refer to a contiguous storage of the value v' of size $N \in \{8i \mid i \in \mathbb{N}\}$ (i.e., N is a multiple of 8) at address w on the storage v:

$$\mathtt{storage}_{\mathbf{el}}\left(v, w, v', N\right) \doteq \begin{cases} \mathtt{storage}_{\mathbf{el}}\begin{pmatrix} v[w \leftarrow \mathbf{ext}\ v' \backsim \mathbf{hi} : 7 \backsim \mathbf{lo} : 0,\ 8], \\ \mathtt{succ}(w), \mathbf{ext}\ v' \backsim \mathbf{hi} : N-1 \backsim \mathbf{lo} : 8, N-8 \end{pmatrix} & \textbf{if } \boldsymbol{N > 8} \\ v[w \leftarrow v',\ 8] & \textbf{otherwise} \end{cases}$$

$$\mathtt{storage}_{\mathbf{be}}\left(v, w, v', N\right) \doteq \begin{cases} \mathtt{storage}_{\mathbf{be}}\begin{pmatrix} v[w \leftarrow \mathbf{ext}\ v' \backsim \mathbf{hi} : N-1 \backsim \mathbf{lo} : N-8,\ 8], \\ \mathtt{succ}(w), \mathbf{ext}\ v' \backsim \mathbf{hi} : N-9 \backsim \mathbf{lo} : 0, N-8 \end{pmatrix} & \textbf{if } \boldsymbol{N > 8} \\ v[w \leftarrow v',\ 8] & \textbf{otherwise} \end{cases}$$

For example, suppose B_1 and B_2 are two 8-bit words. We have:

$$\mathtt{storage}_{\mathbf{el}}\left(v, w, B_1B_2, 16\right) = v[w \leftarrow B_2, 8][\mathtt{succ}(w) \leftarrow B_1, 8]$$
$$\mathtt{storage}_{\mathbf{be}}\left(v, w, B_1B_2, 16\right) = v[w \leftarrow B_1, 8][\mathtt{succ}(w) \leftarrow B_2, 8]$$

We give a subset of these rules for little endian 64-bit load and store operations below.

(REFL_LOAD_EL_WORD64)

$$\frac{}{\Delta \vdash \texttt{storage}_{\mathbf{el}}\left(v, w_2, w_1, 64\right)[w_2, 64] : \mathbf{el} \leadsto^* w_1}$$

(REFL_STORE_EL_WORD64)

$$\frac{}{\Delta \vdash v \ \mathbf{with}\ [w_2, \mathbf{el}] : 64 \leftarrow w_1 \leadsto^* \texttt{storage}_{\mathbf{el}}\left(v, w_2, w_1, 64\right)}$$

▶ **Example 2.** Consider the load from Example 1. The first few steps proceed as before. However, instead of applying LOAD_WORD_EL, we can directly apply REFL_LOAD_EL_WORD64 to obtain the corresponding value $\Delta \vdash (\texttt{storage}_{\mathbf{el}}\left(v, w, v', 64\right))[w, el] : u64 \leadsto^* v'$.

All big-step rules from this section, such as REFL_LOAD_EL_WORD64 have been verified in Isabelle wrt to the small step semantics.

3.4 Statement semantics

In this section, we describe the operational semantics for evaluating statements. We also provide a correction to the sequencing rule from the manual [4]. Although each statement induces local control flow, in BIL, they are assumed to execute to completion in a single step.

The semantics of a statement is defined by $(\Delta,\ pc) \vdash seq \leadsto (\Delta',\ pc')$, which executes the *bil* statement *seq* from the variable store Δ to generate a new variable store Δ' and program counter pc'. The MOVE statement (see below) modifies Δ by generating a new variable binding, and JMP (not shown) affects program counter.

Example rules for MOVE (aka assignment) and IF_TRUE (for branching on a true guard are given below). In the move rule, the given expression e is evaluated to a value using the big-step semantics, and the value for the given variable *var* is updated in the variable state. In the IF_TRUE rule, the guard is evaluated (to *true*) then the statement seq_1 is executed. The sequencing rules describe execution of a list of statements to completion. The sequencing rules in manual [4] do not provide a means to reduce multiple statements in a single transition[3].

(MOVE)

$$\frac{\Delta \vdash e \leadsto^* v}{(\Delta,\ pc) \vdash var := e \leadsto (\Delta(var \mapsto v),\ pc)}$$

(IF_TRUE)

$$\frac{\Delta \vdash e \leadsto^* true \qquad (\Delta, pc) \vdash seq_1 \leadsto (\Delta', pc')}{(\Delta,\ pc) \vdash \mathbf{if}\ (e)\ seq_1\ \mathbf{else}\ seq_2 \leadsto (\Delta',\ pc')}$$

(SEQ_REC)

$$\frac{(\Delta_1,\ pc_1) \vdash s_1 \leadsto (\Delta_2,\ pc_2) \qquad (\Delta, pc_2) \vdash s_2 \ldots s_n \leadsto (\Delta_3, pc_3)}{(\Delta_1, pc_1) \vdash s_1 s_2 \ldots s_n \leadsto (\Delta_3, pc_3)}$$

(SEQ_NIL)

$$\frac{}{(\Delta, pc) \vdash \varepsilon \leadsto (\Delta, pc)}$$

▶ **Example 3.** Consider the load expression from Example 1, which appears on the right-hand side of the assignment statement given below:

$$(\Delta, (12 :: 64) + (1 :: 64)) \vdash \{\texttt{X10 := mem[X8-0x18, el]:u64}\} \leadsto (\Delta(X10 \mapsto v'), (13 :: 64))$$

To apply the statement semantics, we start by deconstructing it using SEQ_REC, which results in two sub-steps. For the first, we apply MOVE and the steps in Example 1 to reduce the expression `mem[X8-0x18,el]:u64` to a value, v'. Since the original sequence only contained a single statement, our recursive sequence (i.e., the second premise of SEQ_REC) is empty, which can be trivially reduced using SEQ_NIL.

[3] We have created a Git pull request in the BAP repositories for this issue (`https://github.com/BinaryAnalysisPlatform/bil/pull/12`.

3.5 BIL Step Relation

A machine instruction in BIL is represented by a named tuple, $insn = (\!|\ \mathbf{addr},\ \mathbf{size},\ \mathbf{code}\ |\!)$ (implemented later as an Isabelle `record`). Here **addr**, refers to the instruction's address in memory; **size**, the size of the instruction; and **code**, the semantics of the program represented by BIL statements. Instructions operate over the BIL machine state, (Δ, pc, mem), where Δ is the variable store, pc is the program counter and mem is a variable that denotes the memory. To decode the current instruction from a machine state, we assume an uninterpreted $\mapsto$ function, referred to as the *decode predicate*, which maps a machine state, (Δ, pc, mem), to the corresponding instruction, $(\!|\ \mathbf{addr},\ \mathbf{size},\ \mathbf{code}\ |\!)$. The BIL step relation $(\Delta, pc, mem) \leadsto (\Delta', pc', mem')$ defines the execution of a single BIL statement:

$$\textsc{(step\_prog)}\quad \frac{(\Delta, pc, mem) \mapsto (\!|\ \mathbf{addr} = pc, \mathbf{size} = z, \mathbf{code} = bil\ |\!) \qquad (\Delta, pc + z) \vdash bil \leadsto (\Delta', pc')}{(\Delta, pc, mem) \leadsto (\Delta', pc', mem)}$$

▶ **Example 4.** Consider the move statement from Example 3, which resides at the program address 12 and being of one-byte size. This is expressed as the BIL instruction below:

$$(\!|\ \mathbf{addr} = 12 :: 64, \mathbf{size} = 1 :: 64, \mathbf{code} = \{\texttt{X10 := mem[X8-0x18, el]:u64}\}\ |\!)$$

The instruction is obtained by decoding $(\mapsto)$ a machine state of the form $(\Delta, 12 :: 64, mem)$ where $X8, mem \in \texttt{dom}(\Delta)$. By using decode we can structure a step proof for the machine state as follows:

$$(\Delta, 12 :: 64, mem) \leadsto (\Delta(X10 \mapsto v'), 13 :: 64, mem)$$

By applying the STEP_PROG rule for programs, we are left with a reduction $(\Delta, (12 :: 64) + (1 :: 64)) \vdash \{\texttt{X10 := mem[X8-0x18, el]:u64}\} \leadsto (\Delta(X10 \mapsto v'), (13 :: 64))$, which we derived in Example 3.

4 Mechanisation

In this section, we describe the IsaBIL mechanisation, which utilises extensible *locales* to maximize reusability. In § 4.1, we describe the generic **`BIL_specification`** locale that formalises BIL's syntax and semantics, and in § 4.2 we describe the IsaBIL translation tool that automatically generates locales from BIL programs.

4.1 The `BIL_specification` Locale

IsaBIL's **`BIL_specification`** locale (see Figure 2) provides a complete mechanisation of BIL's syntax and semantics[4]. We encode BIL's operational rules (§ 3.5) as Isabelle's built-in *inductive predicates*, which allow one to formalise the structure of the syntax from Figure 3.

The **`BIL_specification`** $= (\mapsto)$ locale provides a single parameter: the *decode* predicate $(\mapsto)$, which will be uniquely instantiated for each binary that we wish to verify.

We provide rules for both *introduction* (which are used to introduce compound statements such as **if-then-else** to the proof goal) as well as *elimination* (which are used to eliminate compound statements from the proof assumptions) directly within the **`BIL_specification`** locale. For example, introduction and elimination rules for the STEP_PROG rule in § 3.5 which formalises a single step of the program are given below:

[4] As outlined in § 3 and provided in full in [21].

```
lemma step_progI[intro]:
assumes ‹(Δ, pc, mem) ↦
            ⦇addr = pc, size = sz, code =
    seq⦈›
    and ‹(Δ, pc + sz) ⊢ seq ⤳ (Δ', pc')›
  shows ‹(Δ, pc, mem) ⤳ (Δ', pc', mem)›
proof ...
```

```
lemma step_progE[elim]:
assumes ‹(Δ, pc, mem) ⤳ (Δ', pc', mem)›
    and ‹(Δ, pc, mem) ↦
            ⦇addr = pc, size = sz, code =
    seq⦈›
obtains ‹(Δ, pc + sz) ⊢ seq ⤳ (Δ', pc')›
proof ...
```

We prove introduction and elimination lemmas in this way for all of the rules in the official BIL specification [6] with respect to our changes (§ 3). In total, the **BIL_specification** locale consists of over 500 lemmas and rules.

4.2 The Program Locale

Recall that our workflow (Figure 1) proceeds by *Step 1*: compiling the given program; *Step 2*: feeding the resulting assembly into BAP to generate the corresponding $\text{BIL}_{\mathsf{ADT}}$; and *Step 3*: using IsaBIL to automatically generate an Isabelle/HOL locale for the given $\text{BIL}_{\mathsf{ADT}}$ input. Note that if a binary is distributed without source code, then Step 1 could be skipped and the given assembly could be fed directly into BAP to generate the $\text{BIL}_{\mathsf{ADT}}$.

The locale for a program prog is generated using IsaBIL's translation tool, which is written in Isabelle/ML and is invoked using the custom Isabelle/HOL commands **BIL** or **BIL_file** for inline or external BIL in $\text{BIL}_{\mathsf{ADT}}$ format, respectively. Both commands take as an input, a name for the locale and either a BIL string (for **BIL**) or a filename (for **BIL_file**). Within this locale, we automatically generate a decode predicate, $\mapsto_{\mathsf{prog}}$, that maps each machine state (Δ, pc, mem) to an instruction.

Note that, internally, Isabelle maintains the $\text{BIL}_{\mathsf{ADT}}$ format, however, we choose to represent it using `class` locales [50, 3, 41] to maintain human-readable syntax. For instance, without these syntax classes, expressing a direct jump to the fixed program address 3076 would require writing `RAX := Val(Imm(Word(3076,64)))`. By instantiating the word syntax ($nat :: sz$, see Figure 3), we can simplify the expression to the syntax `RAX := (3076 :: 64)`.

In addition to $\mapsto_{\mathsf{prog}}$, a program's locale defines *Step 1*: an *address set*, `addr_set` : $\mathbb{P}(word)$ (defining the set of addresses that have corresponding instructions) and *Step 2*: a *symbol table*, `sym_table` : $string \to word$ (mapping the binary's symbols, e.g. main, memcpy, free, as string literals to raw addresses).

Both `addr_set` and `sym_table` capture additional information about a binary that can be used later in a proof. For example, the `addr_set` is useful for determining whether the program counter points to a valid address and is particularly important for validating the correctness of indirect jumps. Many binary proofs verify that execution from some entry point, such as the `main` function, is either correct or incorrect with respect to some property. Symbols typically represent entry points to functions and although the address of an entrypoint may differ between binaries, the symbol will remain constant. Hence, referring to positions within the binary using symbol names, stored in the `sym_table` rather than program addresses, is often more convenient. Further details regarding the extraction of the `addr_set` and `sym_table` from a binary can be found in § 6.3.

▶ **Example 5.** Consider the `Binary_A` program below, written in $\text{BIL}_{\mathsf{ADT}}$, which corresponds to the BIL instruction `X8 := X2 + 32`. We assume the program has been compiled for RISC-V; for readability, $\text{BIL}_{\mathsf{ADT}}$ provides the original assembly at Example 5. The program locale corresponding to Example 5 starts on Example 5, which fixes the corresponding decode predicate.

```
BIL ‹
105dc: <test>
105dc: addi s0, sp, 32
(Move(Var("X8",Imm(64)),PLUS(Var("X2",Imm(64)),Int(32,64))))
› defining Binary_A

locale Binary_A
    fixes decode :: ‹(var ⇒ val option) × word × var ⇒ insn ⇒ bool›
    assumes decode_105dc: ‹(Δ, 0x105dc :: 64, mem) ↦prog
                ⦇ addr = 0x105dc :: 64, size = 4, bil = [X8 := X2 + (32 :: 64)] ⦈›
begin
definition addr_set :: ‹word set› where ‹addr_set = {(0x105dc :: 64)}›
definition sym_table :: ‹str ⇀ word› where ‹sym_table = ["test" ↦ (0x105dc :: 64)]›
end
```

Externally linked binaries. Binaries often include links to external code, typically in the form of function calls. Accurately modeling these external calls is crucial for understanding the behavior of the original binary. IsaBIL provides two methods for handling externally linked code, depending on the availability of that external code. If the external binary is available, an Isabelle/HOL locale can be generated for the external binary, which is then combined with the existing binary locale through inheritance. If the external binary is not available, we can make assumptions about its behavior and define an approximate implementation using locale assumptions. We give an example for both cases in [21].

5 (In)correctness

In this section, we present the IsaBIL encoding of Hoare and Incorrectness logic (see Figure 2). Our locale-based encoding combines both into a single proof system **inference_rules** locale, where we verify the AGREEMENT and DENIAL lemmas (see [42]).

Hoare and O'Hearn triples define (in)correctness of a command c over a pre-state σ satisfying the predicate P, resulting in a post-state τ satisfying the predicate Q. In the locale, we assume the existence of a big-step transition relation $(c, \sigma) \Rightarrow \tau$ defining the execution of c from state σ to termination, resulting in state τ. Thus, we have the well-known under- and over-approximating rules:

$$\text{(HOARE)}\ \frac{\forall \sigma, \tau.\ ((c, \sigma) \Rightarrow \tau) \implies (P(\sigma) \implies Q(\tau))}{\{P\}c\{Q\}} \qquad \text{(OHEARN)}\ \frac{\forall \tau.\ Q(\tau) \implies \exists \sigma.\ P(\sigma) \land ((c, \sigma) \Rightarrow \tau)}{[P]\, c\, [Q]}$$

Note that our definition of correctness follows partial correctness, meaning termination is not guaranteed. We encode Hoare and Incorrectness triples as locales **correctness** = ($\Rightarrow$) and **incorrectness** = ($\Rightarrow$), respectively, both of which are parameterised by $\Rightarrow$. Within the **correctness** locale, we verify standard Hoare logic rules, e.g.,

$$\text{(POST\_CONJ\_CORR)}\ \frac{\{P\}c\{Q_1\} \qquad \{P\}c\{Q_2\}}{\{P\}c\{Q_1 \land Q_2\}} \qquad \text{(PRE\_DISJ\_CORR)}\ \frac{\{P_1\}c\{Q\} \qquad \{P_2\}c\{Q\}}{\{P_1 \lor P_2\}c\{Q\}} \qquad \text{(STRENGTHEN\_WEAKEN\_CORR)}\ \frac{P' \implies P \qquad \{P\}c\{Q\} \qquad Q \implies Q'}{\{P'\}c\{Q'\}}$$

Similarly, within the **incorrectness** locale, we verify rules for Incorrectness logic, e.g.,

$$\text{(POST\_DISJ\_INCORR)}\ \frac{[P]\, c\, [Q_1] \qquad [P]\, c\, [Q_2]}{[P]\, c\, [Q_1 \lor Q_2]} \qquad \text{(PRE\_DISJ\_INCORR)}\ \frac{[P_1]\, c\, [Q] \qquad [P_2]\, c\, [Q]}{[P_1 \lor P_2]\, c\, [Q]} \qquad \text{(STRENGTHEN\_WEAKEN\_INCORR)}\ \frac{P \implies P' \qquad [P]\, c\, [Q] \qquad Q' \implies Q}{[P']\, c\, [Q']}$$

We then combine these to form a locale **`inference_rules`** $= (\Rightarrow)$, again parameterised by $\Rightarrow$. Within this combined locale, we are able to prove the AGREEMENT and DENIAL rules [42]:

$$\text{(AGREEMENT)}\quad \frac{[U]\,c\,[U'] \qquad U \Longrightarrow O \qquad \{O\}c\{O'\}}{U' \Longrightarrow O'} \qquad\qquad \text{(DENIAL)}\quad \frac{[U]\,c\,[U'] \qquad U \Longrightarrow O \qquad \neg(U' \Longrightarrow O')}{\neg(\{O\}c\{O'\})}$$

For proofs of incorrectness, it is typically not enough to simply state that if $[P]\,c\,[Q]$ holds for an error state Q then the program c is incorrect. A program cannot be incorrect if there is no valid state that satisfies Q and as such, we must ensure that Q is *non-trivial*, i.e., $\exists\sigma.\ Q(\sigma)$. Proving that Q is non-trivial can be difficult, as Q is often an over-approximation of the actual set of post-states (defined by some predicate Q') reachable from P. Therefore, proofs of incorrectness usually take the form $\exists Q'.\ [P]\,c\,[Q'] \wedge (\forall\sigma.\ Q'(\sigma) \implies Q(\sigma)) \wedge (\exists\sigma.\ Q'(\sigma))$. Note that this does not necessarily imply that $[P]\,c\,[Q] \wedge (\exists\sigma.\ Q(\sigma))$ since the implication is in the wrong direction to apply the STRENGTHEN_WEAKEN_INCORR rule.

6 Automation and Examples

In this section, we bring together the **`BIL_specification`** and the **`inference_rules`** locales to create a new locale, **`BIL_inference`** (see § 6.1), providing a proof environment for BIL programs. This in turn enables us to develop a large number of highly reusable proof automation procedures for verifying both correctness and incorrectness of BIL programs (see § 6.2). The final major component of IsaBIL automation is the BIL$_{\mathsf{ADT}}$ parser that automatically generates Isabelle/HOL locales for BIL programs (§ 6.3), providing a smooth and seamless pipeline from BAP to Isabelle/HOL proofs. We offer an overview of IsaBIL, discussing correctness and incorrectness proofs for a handful of examples.

6.1 The `BIL_inference` locale

`BIL_inference` interfaces with the **`BIL_specification`** and **`inference_rules`** locales (see Figure 2) and is later extended with instances of state models (e.g., **`allocation`** and **`find_symbol`**) to enable verification of generated BIL programs. Therefore, it is one of IsaBIL's most complex components.

Our first task is to define a big-step relation, $\underset{BIL}{\Longrightarrow}$, required by **`inference_rules`**, using the BIL step relation, $(\Delta, pc, mem) \leadsto (\Delta', pc', mem')$, defined in § 3.5. One option for defining $\underset{BIL}{\Longrightarrow}$ is to simply take the transitive closure of $\leadsto$. However, this would mean that the pre/postconditions that we define in our example would be predicates over the full variable store Δ, which is heavy-handed.

An alternative (which is the approach we take) is to define another transition relation $\underset{BIL}{\longrightarrow}$ that abstracts $\leadsto$, and define $\underset{BIL}{\Longrightarrow}$ as the transitive closure of $\underset{BIL}{\longrightarrow}$. As we shall see, this vastly increases the flexibility of our approach, which becomes **(1)** extensible, since $\underset{BIL}{\longrightarrow}$ can be used to cover additional program features such as memory allocation (see § 6.4.1), and **(2)** more efficient since the components of Δ that are not needed for the proof can be ignored.Thus, assuming $\mathcal{C} = (\Delta, pc, mem)$, we have:

$$\text{(PROGRAM\_BIG\_STEP)}\quad \frac{\begin{array}{c}\mathcal{C} = (\_, pc, \_) \qquad pc \in \mathtt{addr\_set} \\ \mathcal{C} \leadsto \mathcal{C}' \qquad (\mathcal{C}, \sigma) \underset{BIL}{\longrightarrow} \sigma' \qquad (\mathcal{C}', \sigma') \underset{BIL}{\Longrightarrow} \tau\end{array}}{(\mathcal{C}, \sigma) \underset{BIL}{\Longrightarrow} \tau}(\mapsto) \qquad \text{(PROGRAM\_BIG\_STEP\_LAST)}\quad \frac{\begin{array}{c}\mathcal{C} = (\_, pc, \_) \qquad \mathcal{C}' = (\_, pc', \_) \\ pc \in \mathtt{addr\_set} \qquad pc' \notin \mathtt{addr\_set} \\ \mathcal{C} \leadsto \mathcal{C}' \qquad (\mathcal{C}, \sigma) \underset{BIL}{\longrightarrow} \tau\end{array}}{(\mathcal{C}, \sigma) \underset{BIL}{\Longrightarrow} \tau}(\mapsto)$$

Here, we assume **`addr_set`** is the set of addresses that contain program instructions (see § 4.2), which will be later instantiated by a program's locale. A program may take a step as long as the current program counter corresponds to a program instruction. Note that the precise state model (i.e., type of σ, σ' and τ) that one uses in PROGRAM_BIG_STEP and PROGRAM_BIG_STEP_LAST will depend on the verification task at hand. If required, one could also take $\sigma = \Delta$ allowing the proofs to inspect the full variable state (at a low level of abstraction).

Overall, we obtain a locale **`BIL_inference`** $= (\mapsto, \xrightarrow[BIL]{},$ **`addr_set`**$)$. This locale extends the **`BIL_specification`** (from § 4) from which we inherit the decode predicate, $\mapsto$. Then, we use $\xrightarrow[BIL]{}$ and **`addr_set`** to define $\xRightarrow[BIL]{}$ as above. This allows us to interpret the **`inference_rules`** as sublocale, instantiating the parameter $\Rightarrow$ to $\xRightarrow[BIL]{}$.

Within **`BIL_inference`**, we can prove several rules of Hoare and Incorrectness logic in terms of the BIL semantics. These proofs are trivial at this level, but are nevertheless critical for proof automation since they can be used by all of our examples [21].

Stuckness and Undefined Behaviour. A BIL program is considered stuck if it reaches a state where no transition rules apply. Programs that become stuck are deemed correct, as they do not terminate.

The most common cause of stuckness in BIL is an attempt to decode an instruction at an address that does not exist within the program (see § 3.5). This typically occurs when a jump targets a non-existent address, resulting in a post-state with no valid instruction to execute. However, the big step semantics treat this as valid program termination rather than an explicit error. If the prover wishes to ensure that the program terminates with an expected PC they may assert its value in an (in)correctness triple's post-state Q.

Stuckness may also arise due to type errors, such as assigning a variable a value of an incompatible type. These types of stuckness are prevented by ensuring BIL statements are type correct, which can be achieved using IsaBIL's automated type checker (see [21]). In our proofs, we verify type correctness when necessary to prevent stuckness. More details, along with an example, can be found in the BIL manual [6].

Many traditional sources of stuckness in program semantics, such as dereferencing invalid memory or reading from uninitialized registers, result in undefined behaviour (UB) for BIL. In BIL, UB is represented using the **unknown**[str] : t value. For example, if an uninitialized register is used:

```
    mv a6, a5      # UB if a5 was never initialized
```

Since the value of `a5` (a 64 bit register) is unspecified, $\textbf{unknown}[str] : \textbf{imm}\langle 64\rangle$ will be set to `a6`. Evaluating BIL semantics that contain unknown values (such as `a6` $+ 5$) will propagate further unknowns unless avoided by control flow structures such as if-then-else. However, this does not necessarily lead to stuckness, except in cases where the target of a dynamic jump is unknown (i.e. $\textbf{jmp unknown}[str] : \textbf{imm}\langle 64\rangle$). In such cases, the program cannot take a step as it does not know which program address to jump to. In (in)correctness logic, unknowns that do not lead to stuckness may still prevent the discharge of the post-state (Q). Unknown values in BIL can be resolved by explicitly specifying constraints on the program and pre-state as required.

6.2 Proof Automation

Instruction set architectures (ISAs), despite their large scale, are inherently finite, and compilers employ common patterns for optimization. Additionally, developers leverage reusable components such as *functions* and *gadgets* to perform common tasks. Consequently, binaries often exhibit a significant degree of repetition. Once identified, this repetition can be verified without the need for human input. IsaBIL exploits this repetition to alleviate the burden associated with proof construction and verification. In this section, we outline the proof automation techniques employed by IsaBIL.

6.2.1 Eisbach

IsaBIL heavily leverages Eisbach [33], an Isabelle tactic framework for proof automation, to discharge proof goals automatically. We create Eisbach methods for symbolic execution (`sexc`) and type checking (`typec`) which utilise IsaBIL's *introduction* and *elimination* rules to solve both big- and small-step BIL statements, expressions and variables as well as proving type correctness. These methods employ a best-effort approach. If a case that cannot be solved automatically occurs, the methods will backtrack to a safe state and the partially completed proof context will be handed back to the human prover, whereby corrections or manual inputs can be made before resuming the automation (see [21] for details).

6.2.2 Human Readability

In assembly languages, registers are typically assigned specific names and types. However, in BIL, registers are represented as generic *variables* of type *var*. These variables are parameterised with names and types. For example, the RISC-V registers `R0-31` are represented by the variables ("R0-31" : $\mathbf{imm}\langle 64\rangle$). While this representation offers flexibility, it can be verbose. It is common knowledge for RISC-V developers that R0-31 are registers capable of storing 64-bit words. To enhance program readability, we define the set of registers for `riscv64` [55] as the 64bit registers R0-31 : $\mathbf{imm}\langle 64\rangle$. Additionally, we introduce syntax abbreviations for the 64-bit x86 architecture.

6.2.3 Architecture-Specific Proof Optimisations (for RISC-V)

Whilst IsaBIL provides sufficient granularity to handle many architectures out of the box, the speed and efficiency of proofs can be improved by tailoring optimizations to specific hardware architectures. This section describes the proof optimizations undertaken for RISC-V.

Instructions. Modern architectures comprise of large instruction sets. Proofs of programs can be optimised by verifying high-level lemmas corresponding to program steps of the most commonly used instructions directly in the **`BIL_inference`** locale. For example, IsaBIL provides step rules for the 32-bit (4-byte) RISC-V instructions `auipc`, `jalr` and `ld`.

$$\texttt{auipc}\ rd,\ \ imm = \llparenthesis\ \mathbf{addr} = pc, \mathbf{size} = 4 :: 64, \mathbf{code} = \{rd := pc + imm\}\ \rrparenthesis$$

The `auipc` (Add Upper Immediate to Program Counter) instruction facilitates the computation of an absolute address. It achieves this by adding the immediate value *imm* to *pc*, storing the result in the destination register *rd*. `auipc` is commonly used to create jump targets, such as those for external functions.

$$\texttt{jalr}\ rd,\ \ rs1,\ \ offset = \llparenthesis\ \mathbf{addr} = pc, \mathbf{size} = 4 :: 64, \mathbf{code} = \{rd := pc + 4;\ \mathbf{jmp}\ rs1 + offset\}\ \rrparenthesis$$

The `jalr` (Jump And Link Register) instruction performs an unconditional jump. It sets *pc* to the address stored in the source register *rs*1 with an optional immediate value *offset*. The address of the subsequent instruction is then stored in the destination register *rd*. This instruction is used for function calls, where it is important to preserve the return address.

$$\texttt{ld}\ rd,\ \ offset(rs1) = \llparenthesis\ \mathbf{addr} = pc, \mathbf{size} = 4 :: 64, \mathbf{code} = \{rd := mem[rs1 + offset, \mathbf{el}] : 64\}\ \rrparenthesis$$

```
00000000000104a0 <malloc@plt>:
   104a0: 00002e17            auipc   t3,0x2
   104a4: b78e3e03            ld      t3,-1160(t3) # 12018 <malloc@GLIBC_2.27>
   104a8: 000e0367            jalr    t1,t3
   104ac: 00000013            nop

00000000000104b0 <free@plt>:
   104b0: 00002e17            auipc   t3,0x2
   104b4: b70e3e03            ld      t3,-1168(t3) # 12020 <free@GLIBC_2.27>
   104b8: 000e0367            jalr    t1,t3
   104bc: 00000013            nop
```

Figure 5 PLT stubs for free and malloc.

The `ld` (Load Double) instruction is used to load a 64-bit word, referred to as a double word, from memory. First, `ld` computes an address by adding the contents of the source register $rs1$ to the immediate *offset*. The value at this address in memory is then stored in the destination register rd. `ld` is the primary means of retrieving data of this size from memory.

Representing the instructions `auipc`, `jalr` and `ld` as the high-level program step lemmas (given below) allows the prover to otherwise skip the lengthy reduction for a program step. The rules for these instructions are provided below:

$$\text{(AUIPC\_LEMMA)}\quad \frac{(\Delta, pc, mem) \mapsto \texttt{auipc}\ rd,\ imm}{(\Delta, pc, mem) \rightsquigarrow (\Delta(rd \mapsto pc + imm), pc + 4), mem)}$$

$$\text{(JALR\_LEMMA)}\quad \frac{(\Delta, pc, mem) \mapsto \texttt{jalr}\ rd,\ rs1,\ \mathit{offset} \qquad (rs1, addr) \in \Delta}{(\Delta, pc, mem) \rightsquigarrow (\Delta(rd \mapsto pc + 4), addr + \mathit{offset}, mem)}$$

$$\text{(LD\_LEMMA)}\quad \frac{(\Delta, pc, mem) \mapsto \texttt{ld}\ rd,\ \mathit{offset}(rs1) \qquad (mem, v) \in \Delta \qquad (rs1, addr) \in \Delta \qquad \Delta \vdash v[addr + \mathit{offset}, \mathbf{el}] : 64 \rightsquigarrow^* w}{(\Delta, pc, mem) \rightsquigarrow (\Delta(rd \mapsto w), pc + 4), mem)}$$

Further semantics for `addi`, `sd` and `ret` RISC-V instructions are provided in [21].

Execution. We also prove high-level big-step semantics rules for common gadgets in binary executables. For example, the *procedure linkage table* (PLT) acts as an intermediary between the current program and shared libraries, including C's standard library. The PLT facilitates indirect calls to external functions, whose locations are unknown until runtime when they are resolved by the dynamic loader. In the dump of the RISC-V binary for Listing 1 given in § 6.2.3, we can observe the presence of stubs within the PLT, representing entries for each external call. Specifically, in § 6.2.3, the stubs correspond to the functions `free` and `malloc` that are called by the program.

By leveraging the consistent pattern observed in PLT entries, we can construct universal big-step rules for PLT stubs, as demonstrated below.

$$\text{(PLT\_STUB\_LEMMA)}\quad \frac{\begin{array}{c} (\Delta_1, pc, mem) \mapsto \texttt{auipc}\ t3,\ 0x2 \qquad (\Delta_2, pc + 4, mem) \mapsto \texttt{ld}\ t3,\ \mathit{offset}(t3) \\ (\Delta_3, pc + 8, mem) \mapsto \texttt{jalr}\ t1,\ t3,\ 0 \qquad \{pc, pc + 4, pc + 8\} \subseteq \texttt{addr\_set} \\ \Delta_2 = \Delta_1(t3 \mapsto pc + 0x2) \qquad \Delta_3 = \Delta_2(t3 \mapsto w) \qquad \Delta_4 = \Delta_3(t1 \mapsto pc + 12) \\ ((\Delta_1, pc, mem), \sigma_1) \xrightarrow[BIL]{} \sigma_2 \qquad ((\Delta_2, pc + 4, mem), \sigma_2) \xrightarrow[BIL]{} \sigma_3 \qquad ((\Delta_3, pc + 8, mem), \sigma_3) \xrightarrow[BIL]{} \sigma_4 \\ ((\Delta_4, w, mem), \sigma_4) \xRightarrow[BIL]{} \tau \qquad (mem, v) \in \Delta_1 \qquad \Delta \vdash v[pc + \texttt{0x2} + \mathit{offset}, \mathbf{el}] : 64 \rightsquigarrow^* w \end{array}}{((\Delta_1, pc, mem), \sigma_1) \xRightarrow[BIL]{} \tau}$$

Now, if we encounter a PLT stub in a proof at a lower level, we can discharge the proof obligation efficiently using PLT_STUB_LEMMA. This lemma significantly reduces the proof effort required. Without PLT_STUB_LEMMA, one must apply over 150 rules to achieve the same result. Furthermore, by applying this approach to stack allocations and deallocations, we eliminate the need for approximately 50 rules in the case of allocation and 250 rules in the case of deallocation. Further details on this are provided in [21].

6.3 BIL to IsaBIL transpiler

A key component of our IsaBIL framework is a verified BIL parser that transpiles BIL$_{\text{ADT}}$ programs into an Isabelle/HOL locale to enable verification.

Parsing occurs in two distinct phases. The first phase involves the translation of BIL$_{\text{ADT}}$ into BIL$_{\text{ML}}$, a format suitable for manipulation within Isabelle/ML. This intermediate step enables analysis tasks in Isabelle/ML, such as calculating the size of instructions. The second phase involves translating BIL$_{\text{ML}}$ into an Isabelle/HOL representation, which is the starting point for verification. Parsing can be invoked directly on a BIL$_{\text{ADT}}$ string with `BIL` or alternately on a file containing BIL$_{\text{ADT}}$ with `BIL_file`. An overview of the parsing process is given in Figure 6.

During the first phase, the parser iterates over the instructions in a BIL$_{\text{ADT}}$ input. It captures and stores all program addresses in `addr_set`. Moreover, the parser associates symbols with a program address in `sym_table`. Since the syntax of BIL$_{\text{ADT}}$ resembles a tree structure, it is intuitive to convert it into an Abstract Syntax Tree (AST) using a lexer. This AST is processed by a Recursive Descent Parser (RDP), which traverses each node in the tree depth-first, matching the node's value to a corresponding parsing function. For example, if the parser encounters a `Var` node, it will attempt to parse the first child as a `string` and the second child as a BIL type. This process transforms the input into BIL$_{\text{ML}}$, a structured data type closely resembling BIL$_{\text{ADT}}$ which can be manipulated within Isabelle/ML.

The second phase defines a locale within the current proof context with a fixed decode predicate. For each BIL$_{\text{ML}}$ instruction, an assumption is added to the locale, stating how a program with any variable state and memory, but with the instructions program address, decodes to an Isabelle/HOL representation of said instruction. To obtain the Isabelle/HOL representation, an RDP translator recursively converts each BIL$_{\text{ML}}$ instruction to Isabelle/HOL terms defined in the **`BIL_specification`** locale (see § 4).

The first phase of the parser harnesses Isabelle/HOL's *code generation*, where specifications for both the lexer and parser were defined within Isabelle/HOL and subsequently translated into ML. BIL$_{\text{ML}}$ for example, is a direct ML representation of the BIL specification from the IsaBIL framework. Using *code reflection*, we import the generated ML code back into Isabelle/HOL as a plugin. Consequently, the lexer and parser are formally verified under the Isabelle/HOL framework. In contrast, the second phase employs non-code-generated ML. This decision stems from its necessity to interface with Isabelle/HOL's core ML library, which cannot be achieved via code-generated ML. To bridge this gap, we introduce a minimal layer of unverified ML code that lifts the BIL representation obtained from the verified parser into Isabelle/HOL's proof system. Below, as an example, we present a snippet of the translation code responsible for converting BIL$_{\text{ML}}$'s binary operations to IsaBIL's binary operations:

```
mk_exp (AstParser.BinOp (e1,bop,e2)) = @{term "BinOp"} $ mk_exp e1 $
    mk_bop bop $ mk_exp e2
```

Note that Isabelle isolates Isabelle/ML programming, and hence our parser cannot interfere with the soundness of Isabelle's core logic engine.

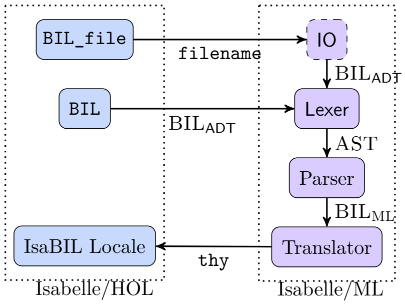

Figure 6 Overview of the BIL parser, entry-points via the `BIL` or `BIL_file` commands.

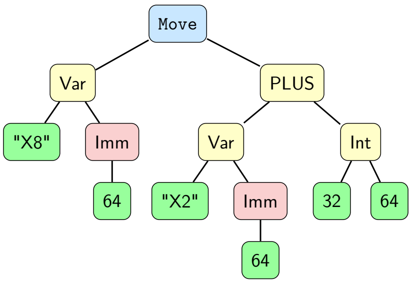

Figure 7 AST representation of the BIL$_{ADT}$ statement `Move(Var("X8", Imm(64)), PLUS(Var("X2", Imm(64)), Int(32, 64)))`.

Table 2 Overview of theories and lemma for each of IsaBIL's components.

Component	Lemmas
Correctness	13
Incorrectness	3
Inference Rules	2
BIL Specification (including syntax)	594
BIL Inference	13
RISC-V optimisations	74
Alloc Model	6
Find Symbol	0
Double Free (total for both examples)	77
AV Rules (total for all seven examples)	187

Table 3 AV rules with description.

AVR	Description (the following shall not be used)
17	The error indicator errno
19	`<locale.h>` and the setlocale function[5]
20	The `setjmp` macro and the `longjmp` functions
21	The signal handling facilities of `<signal.h>`
23	The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>`
24	The library functions `abort`, `exit`, `getenv` and system from library `<stdlib.h>`
25	The time handling functions of library `<time.h>`

6.4 Example Proofs

The majority of the development consists of general tactics and lemmas which can be reused across multiple developments. We give the number of lemmas for each component in Table 2.

All of the sublocales and inherited locales can reuse the proofs of the higher-level (more abstract) locales. Thus, for instance, the 594 lemmas of the **`BIL_specification`** locale are applicable to **`BIL_inference`** and all of the examples, which improves re-usability. The RISC-V optimisations help improve performance and are reusable across all examples. The theories for **`double_free_*`** and **`find_symbol`** are general and apply to all corresponding examples. Finally, each example comprises a step lemma for each line of code. Using the earlier optimisations, solving them is trivial and only requires running the automated tactics.

To motivate IsaBIL, we employ a combination of correctness and incorrectness proofs, focusing on illustrative examples frequently referenced in BAP literature [8]. These examples represent patterns commonly found within much larger programs/libraries.

6.4.1 Double Free

We detail the (in)correctness proofs for CWE-415: Double Free outlined in §2. To classify this vulnerability, we require a memory allocation model, which is not part of the BIL semantics (§3). Memory allocation in C is reflected as a PLT stub in the binary. In particular, we are only required to track the pointer (memory address) that is allocated (and later freed). To this end, we develop a locale, **`allocation`**, that extends the BIL semantics (see Figure 2) with an abstract allocation model inspired by earlier works [34, 31].

$$\text{(ALLOC)}\ \frac{\texttt{is\_alloc}(\mathcal{C}) \qquad w = \texttt{next\_ptr}(\omega_{\mathcal{A}}) \\ sz = \texttt{get\_sz}(\mathcal{C}) \\ \omega'_{\mathcal{A}} = \omega_{\mathcal{A}}.[\texttt{alloc}(w, sz)]}{(\mathcal{C}, \omega_{\mathcal{A}}) \xrightarrow[\mathcal{A}]{} \omega'_{\mathcal{A}}} \qquad \text{(FREE)}\ \frac{\texttt{is\_free}(\mathcal{C}) \qquad w = \texttt{get\_ptr}(\mathcal{C}) \\ \omega'_{\mathcal{A}} = \omega_{\mathcal{A}}.[\texttt{free}(w)]}{(\mathcal{C}, \omega_{\mathcal{A}}) \xrightarrow[\mathcal{A}]{} \omega'_{\mathcal{A}}} \qquad \text{(SKIP)}\ \frac{\neg\texttt{is\_free}(\mathcal{C}) \\ \neg\texttt{is\_alloc}(\mathcal{C})}{(\mathcal{C}, \omega_{\mathcal{A}}) \xrightarrow[\mathcal{A}]{} \omega_{\mathcal{A}}}$$

Figure 8 Allocation semantics.

The `allocation` locale. We define **`allocation`** by instantiating the small-step transition relation $\xrightarrow[BIL]{}$ of **`BIL_specification`** (see §6.1). We start by defining two memory operations:

$$memop ::= \texttt{alloc}(w, sz) \mid \texttt{free}(w)$$

where w is the memory being allocated or freed and sz is the size. Additionally, we assume an abstract allocation function `next_ptr` : $memop^* \rightarrow word$ that serves the purpose of selecting suitable memory addresses for allocation and retrieves the next address from the allocator based on the given history (sequence) of allocations and deallocations. For each allocation, we use `next_ptr` to obtain the next available memory address. Consequently, in the state of **`allocation`**, we must track the sequence of *memop* operations, which is provided to `next_ptr` as an input whenever an allocation occurs. For the Double Free example, no other variables need to be tracked, hence we simply assume the state to be this sequence, which we denote $\omega_{\mathcal{A}} : memop^*$.

Recall that by using the $\mapsto$ predicate (§ 3.5), the BIL instruction corresponding to $\mathcal{C} = (\Delta, pc, mem)$ can be uniquely identified. We use predicates `is_free` : $prog \rightarrow bool$ and `is_alloc` : $prog \rightarrow bool$, which hold when the next instruction to be executed in the given $\mathcal{C} \in prog$ corresponds to a free and allocation instruction, respectively. We assume that an instruction that frees memory cannot simultaneously perform an allocation and vice versa, i.e., $\neg\texttt{is\_free}(\mathcal{C}) \vee \neg\texttt{is\_alloc}(\mathcal{C})$. When the instruction for $\mathcal{C}$ is an allocation, we utilise the function `get_sz` : $prog \rightarrow sz$ to obtain the size allocated and when the instruction for $\mathcal{C}$ is a deallocation, we utilise the function `get_ptr` : $prog \rightarrow word$ to obtain the address that is freed. The size and location can be determined using the $\mapsto$ function on the given $\mathcal{C}$. Thus, we define a small-step transition relation, $\xrightarrow[\mathcal{A}]{}$, over allocations in Figure 8.

We encode this *allocation model* as the locale **`allocation`**, fixing the functions `next_ptr`, `is_free`, `is_alloc`, `get_sz` and `get_ptr`. This locale is derived as an *interpretation* of the locale **`BIL_inference`** retaining the decode predicate $\mapsto$ and address set `addr_set` as abstract, but overriding $\xrightarrow[BIL]{} = \xrightarrow[\mathcal{A}]{}$. Thus, we have the signature **`allocation`** $= (\mapsto,$ `addr_set`, `next_ptr`, `is_free`, `is_alloc`, `get_sz`, `get_ptr`$)$, where parameters `next_ptr`, `is_free`, `is_alloc`, `get_sz` and `get_ptr` are used to derive $\xrightarrow[\mathcal{A}]{}$ within the locale.

The `double_free_bad` and `double_free_good` locales. These locales correspond to the double free program and are *auto-generated* from the corresponding BIL programs (shown in Listings 3 and 4). This step is trivial when using IsaBIL's `BIL_file` command, which we have developed using Isabelle/ML. In particular, we invoke the Isabelle/HOL commands:

```
BIL_file ‹double-free-bad.bil.adt›  defining double_free_bad
BIL_file ‹double-free-good.bil.adt› defining double_free_good
```

where `double-free-bad.bil.adt` is a file containing the BIL for Listing 1 in BIL$_{\text{ADT}}$ format and `double_free_bad` is the name of the locale to be generated. The good version is similar.

The `double_free_*_proof` locales. We combine the locales generated from the BIL programs with our allocator locale (**`allocation`**) to produce new locales **`double_free_bad_proof`** and **`double_free_good_proof`**. Recall from § 4.2 that for each locale generated from a BIL_{ADT} program, we have access to a symbol table `sym_table` mapping strings (i.e., external function calls) to the program address at which the symbol appears. Note that when a BIL program calls a function, it typically stores the return address, then jumps to the function. The symbol corresponding to the program counter after the jump contains the name of the external function, which is stored in the symbol table. In RISC-V convention, the first argument of a function call is stored in the register `X10`. To allocate memory, function `malloc` is called with the size as the first argument whereas to deallocate a pointer, function `free` is called with the pointer as the first argument. Therefore, we can obtain the size allocated by a call to `malloc` and pointer freed by a call to `free` by retrieving the value stored in register `X10`.

We therefore instantiate the predicates `is_free`, `is_alloc`, `get_sz` and `get_ptr` from the **`allocation`** locale as follows:

$$\begin{array}{rcl} \texttt{is\_free}((\_, pc, \_)) = (pc = \texttt{sym\_table}(\text{“}\texttt{free}\text{”})) & & \texttt{get\_ptr}((\Delta, \_, \_)) = \Delta(\texttt{X10}) \\ \texttt{is\_alloc}((\_, pc, \_)) = (pc = \texttt{sym\_table}(\text{“}\texttt{malloc}\text{”})) & & \texttt{get\_sz}((\Delta, \_, \_)) = \Delta(\texttt{X10}) \end{array}$$

The exact implementation of `next_free` occurs at a lower level of abstraction, and depends on the allocation model. At this level, we provide the main requirement axiomatically, i.e., we require

$$w = \texttt{next\_ptr}(\omega_{\mathcal{A}}) \implies (\forall i.\ \omega_{\mathcal{A}}(i) = \texttt{alloc}(w, sz) \implies \exists j.\ j > i \wedge \omega_{\mathcal{A}}(i) = \texttt{free}(w))$$

This presents us with all the components necessary to prove (in)correctness of our examples. The following predicate captures the double-free over the memory state defined above:

$$\begin{array}{rl} \texttt{double\_free\_vuln} = \lambda \omega_{\mathcal{A}}.\ \exists i, j, w.\ & i < j \wedge \omega_{\mathcal{A}}[i] = \texttt{free}(w) \wedge \omega_{\mathcal{A}}[j] = \texttt{free}(w) \wedge \\ & (\forall k.\ i < k < j \implies \omega_{\mathcal{A}}[k] \neq \texttt{alloc}(w, sz)) \end{array}$$

Note that the predicate captures *only* the double-free vulnerability and does not formalise other pointer misuse vulnerabilities, e.g., use-after-free or free-before-alloc.

▶ **Theorem 6.** *Let $\mathcal{C}_{bad}$ and $\mathcal{C}_{good}$ be the programs corresponding to* **`double_free_bad`** *and* **`double_free_good`**, *respectively. Both of the following hold*

1. $\exists Q.\, [\neg\texttt{double\_free\_vuln}]\, \mathcal{C}_{bad}\, [Q] \wedge (\forall \sigma.\, Q(\sigma) \Longrightarrow \texttt{double\_free\_vuln}(\sigma)) \wedge (\exists \sigma.\, Q(\sigma))$
2. $\{\neg\texttt{double\_free\_vuln}\}\ \mathcal{C}_{good}\ \{\neg\texttt{double\_free\_vuln}\}$

Note that in $\mathcal{C}_{bad}$, the memory error does not lead to early termination via the allocation semantics in Figure 8. The program continues executing as normal, but generates a history that can be shown to contain a double-free vulnerability at termination.

6.4.2 AV rules

In a similar manner, we prove incorrectness of the AV rules from Table 3. Details of these proofs are provided in [21].

7 Case Study: Double-Free Vulnerability in cURL (CVE-2016-8619)

cURL is an open-source library primarily used for transferring data over the internet using various protocols, including TLS. It has a large user base, supporting multiple operating systems and architectures (including RISC-V), and thus ensuring the security of cURL is critically important.

```
static __attribute__ ((noinline)) CURLcode read_data(struct connectdata *conn,
                                                     curl_socket_t fd,
                                                     struct krb5buffer *buf)
{
  int len;
  void* tmp = NULL;
  CURLcode result;

  result = socket_read(fd, &len, sizeof(len)); // result = 0 iff socket_read successful
  if(result)
    return result;
```

```
                                              if (len) {
  len = ntohl(len);                             len = ntohl(len);
  tmp = realloc(buf->data, len);                tmp = realloc(buf->data, len);
                                              }
```

```
  if(tmp == NULL)
    return CURLE_OUT_OF_MEMORY;

  buf->data = tmp;
  result = socket_read(fd, buf->data, len);
  if(result)
    return result;
  buf->size = conn->mech->decode(conn->app_data, buf->data, len,
                                 conn->data_prot, conn);
  buf->index = 0;
  return CURLE_OK;
}
```

Figure 9 The `read_data` function of the cURL library in `security.c`. The snippet highlighted in red shows the vulnerable code in version 7.50.3, and that in green shows the corrected vulnerability in version 7.51.0.

Unfortunately, however, cURL has been known to contain vulnerabilities. Consider the function `read_data` in Figure 9, which underpins the Kerberos authentication protocol in cURL (prior to version 7.51.0), where § 7 reads the length of an incoming data packet from a socket. If this succeeds, `socket_read` will return a 0, causing the if statement on § 7 to fall through. Then, on § 7 the call to `ntohl` converts the data length (`len`) from network byte order (big-endian) to host byte order (big- or little-endian). Note that this method does not sanitise its input. Next, `realloc` on § 7 resizes the `data` buffer in `krb5buffer`, and returns either a new pointer to the resized memory, or `NULL` if an error occurs. The rest of the method then reads and decodes the remaining data from the socket.

The caller is responsible for managing the `krb5buffer` pointer by allocating it before and freeing it after calling `read_data`. Here we assume `malloc` is called directly before `read_data` and free is called directly after, as follows: `malloc(buf, sz); read_data(conn, fd, buf); free(buf);`

If receiving data from the socket on § 7 yields a length of 0 (`len` $= 0$), the call to `realloc` with `len` $= 0$ results in *zero reallocation*, leading to *undefined behaviour*[6]. Most implementations of `realloc` will return NULL and free the pointer following a *zero reallocation*. As such, since the `krb5buffer` buffer is already freed before `realloc` returns and the caller is expected to free it after the function returns, this will lead to a *double-free vulnerability*, identified by the vulnerability enumeration CVE-2016-8619.

[6] https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2464.pdf

$$\text{(REALLOC)}\ \frac{\texttt{is\_realloc}(\mathcal{C}) \qquad w = \texttt{get\_ptr}(\mathcal{C}) \\ sz = \texttt{get\_sz}(\mathcal{C}) \\ \omega'_{\mathcal{A}} = \omega_{\mathcal{A}}.[\texttt{alloc}(w, sz)]}{(\mathcal{C}, \omega_{\mathcal{A}}) \xrightarrow[\mathcal{R}]{} \omega'_{\mathcal{A}}} \qquad \text{(ZERO\_REALLOC)}\ \frac{\texttt{is\_realloc}(\mathcal{C}) \qquad w = \texttt{get\_ptr}(\mathcal{C}) \\ sz = \texttt{get\_sz}(\mathcal{C}) \qquad sz = 0 \\ \omega'_{\mathcal{A}} = \omega_{\mathcal{A}}.[\texttt{free}(w)]}{(\mathcal{C}, \omega_{\mathcal{A}}) \xrightarrow[\mathcal{R}]{} \omega'_{\mathcal{A}}} \qquad \text{(SKIP)}\ \frac{\neg\texttt{is\_free}(\mathcal{C}) \\ \neg\texttt{is\_alloc}(\mathcal{C}) \\ \neg\texttt{is\_realloc}(\mathcal{C})}{(\mathcal{C}, \omega_{\mathcal{A}}) \xrightarrow[\mathcal{R}]{} \omega_{\mathcal{A}}}$$

Figure 10 Reallocation semantics.

This vulnerability is resolved in version 7.51.0 of cURL by preempting the zero-allocation that leads to it: as highlighted in green in Figure 9, placing a guard around the call to `realloc`[7] ensures that it is not called with a zero length. As such, if `len` $= 0$, then the call to `realloc` is skipped, `TMP` remains `NULL` and returns 0. This ensures consistent behaviour across all C implementations.

7.1 Scalability of the Parser

We cross compile cURL 7.50.3 for RISC-V on Linux using the GNU RISC-V compiler with the default options. We modify the source code for `read_data` by adding the `noinline` attribute, which ensures the function is not inlined. Note that `noinline` is not a requirement for verification, but facilitates compositional reasoning (see § 7.3).

Using the generated BIL directly creates a scalability challenge. The assembly corresponding to version 7.50.3 contains *63,592 RISC-V instructions*, which equates to *127,023 LoC in BIL* with *63,592 BIL instructions*. Naively generating the corresponding IsaBIL locales (see § 6.3) takes approximately 14 minutes, which is impractical for verification. Of this, the lexer takes 4s, the parser takes 2s and the translator takes approximately 13 minutes. The wait time in the translator arises from instantiating the Isabelle/HOL locale in the proof context, an operation which cannot be avoided.

To address this, we extend the `BIL_file` command with a *new option*, `with_subroutines`, to focus only on the subroutines of interest (and their dependencies). For example, to generate the IsaBIL locale for `read_data` in version 7.50.3, we use the following command:

```
BIL_file ‹libcurl.7.50.3.bil.adt› defining read_data_7_50_3
  with_subroutines read_data and ...
```

where "..." contains functions that `read_data` calls (e.g., `ntohl`). Running `BIL_file` now takes only 6s, with 131 instructions to verify. We next show how we detect this vulnerability in version 7.50.3 using an incorrectness proof.

7.2 Incorrectness of the `read_data` Subroutine

`read_data` contains a double-free vulnerability. However, unlike our prior example (§ 6.4.1), the vulnerability arises from a reallocation of memory. Thus, we first describe the construction of a `reallocation` locale, which reuses components from the `allocation` locale. Then, we prove the incorrectness of the `read_data` function in version 7.50.3.

The `reallocation` locale. We define `reallocation` by instantiating the small-step transition relation $\xrightarrow[BIL]{}$ of `BIL_specification` (see § 6.1). Whilst we do not extend `allocation` from § 6.4.1 directly, as it already defines small step semantics ($\xrightarrow[\mathcal{A}]{}$) that are incompatible

[7] `https://github.com/curl/curl/commit/3d6460edeee21d7d790ec570d0887bed1f4366dd`

with reallocation, we can borrow: (1) the definition of *memop*, (2) our abstract allocator `next_ptr`, (3) our memory trace $\omega_{\mathcal{A}}$, (4) the getter for the size `get_sz`, (5) the getter for the pointer to free `get_ptr`, (6) the predicate that denotes a program is freeing memory `is_free` and (7) the predicate that denotes a program is allocating memory `is_alloc`. These concepts are explained in detail in § 6.4.1.

We use the predicate `is_realloc` to determine whether the next instruction to be executed in the given $\mathcal{C}$ corresponds to a `realloc` instruction. We assume that an instruction that frees or allocates memory cannot simultaneously perform an reallocation and vice versa, i.e., $\neg\texttt{is\_free}(\mathcal{C}) \vee \neg\texttt{is\_alloc}(\mathcal{C}) \vee \neg\texttt{is\_realloc}(\mathcal{C})$.

When a program reallocates memory it has two parameters, a pointer to the memory to be resized, and the new size of that memory. We utilise the existing functions `get_sz` to obtain the size reallocated and `get_ptr` to obtain the address that is reallocated. The size and location can be determined using the $\mapsto$ function on the given $\mathcal{C}$. The size and location are then used to add an $\texttt{alloc}(w, sz)$ to the $\omega_{\mathcal{A}}$. However, if the size of a reallocation is 0, this represents a zero-reallocation, and hence a $\texttt{free}(w)$ is added to the $\omega_{\mathcal{A}}$. This results in a small-step transition relation, $\underset{\mathcal{R}}{\longrightarrow}$, over reallocations as given in Figure 10. Note that we retain the ALLOC and FREE rules from Figure 8, and adapt the SKIP to encompass is reallocations. Finally, we obtain the signature

$$\texttt{reallocation} = (\mapsto, \texttt{addr\_set}, \texttt{next\_ptr}, \texttt{is\_free}, \texttt{is\_alloc}, \texttt{is\_realloc}, \texttt{get\_sz}, \texttt{get\_ptr})$$

where the parameters `next_ptr`, `is_free`, `is_realloc`, `is_alloc`, `get_sz` and `get_ptr` are used to derive $\underset{\mathcal{R}}{\longrightarrow}$ within the locale.

The `read_data` proof locale. We combine the locale generated for cURL binaries to produce a new locale `read_data_7_50_3_proof`. Reallocations are handled by calls to a PLT stub for `realloc` so we can proceed in the same way as `malloc`/`free`. In § 6.4.1, we discussed how the first argument of a subroutine call is stored in the register `X10`. Reallocations take two arguments: (1) a pointer to memory, and (2) a size. The second argument (2) is stored in the return register (`X11`), thus we instantiate the parameters of `reallocation` as follows for both locales:

$$\texttt{is\_realloc}(\_, pc, \_) = (pc = \texttt{sym\_table}(\text{“}\texttt{realloc}\text{”}))$$

$$\texttt{get\_sz}(\Delta, pc, \_) = \begin{cases} \Delta(\texttt{X11}) & \textbf{if } pc = \texttt{sym\_\_table}(\text{“}\texttt{realloc}\text{”}) \\ \Delta(\texttt{X10}) & \textbf{otherwise} \end{cases}$$

The remaining definitions of `get_ptr`, `is_free` and `is_alloc` match `allocation` (see §6.4.1).

▶ **Theorem 7.** *Let $RD_{7.50.3}$ be the program corresponding to* `read_data_7_50_3`*. Then*

$$\exists Q.\ [\neg\texttt{double\_free\_vuln}]\ RD_{7.50.3}\ [Q] \wedge (\forall\sigma.\ Q(\sigma) \implies \texttt{double\_free\_vuln}(\sigma)) \wedge (\exists\sigma.\ Q(\sigma)).$$

7.3 Compositionality

The proof in Figure 9 covers the `read_data` subroutine in the cURL library. However, this subroutine is an *internal* function (i.e. not defined in a header file), making the vulnerability difficult to exploit directly in practice. Nevertheless, as shown in the call graph in Figure 11, there is a path from an *external* function (defined in a header file), namely `Curl_sec_login`,

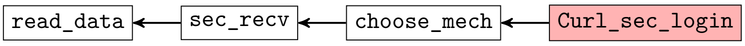

Figure 11 A partial call graph in cURL demonstrating a path from an external function (highlighted), `Curl_sec_login`, to the internal function `read_data`.

to `read_data`. That is, a call to `Curl_sec_login` includes a call to `choose_mech`, which in turn includes a call to `sec_recv`, which itself calls `read_data`. As such, the vulnerability in `read_data` *could potentially* be exploited by a call to `Curl_sec_login`.

Note that this may not always be the case, as freeing the data buffer pointer is left up to the caller, not the callee, and thus further incorrect behaviour to by caller (in this case a function above `read_data` in the call graph) could rectify this vulnerability by forgetting to call free, or perhaps reallocating the pointer again before freeing it. Thus, we should aim to establish that each of these methods are (in)correct formally.

We do this in a *compositional* fashion through *proof reuse*: we reuse the (in)correctness specifications we prove at lower levels of the call graph in higher levels. For instance, when proving that `sec_recv` is (in)correct, we reuse the specification of `read_data` in Theorem 7 as a lemma. Indeed, this is precisely why we used the `noinline` attribute when compiling `read_data`: this allows us to reuse its specification; otherwise each call to `read_data` would be inlined, preventing us from proof reuse.

In Theorem 8 below, we prove incorrectness specifications for the $SR_{7.50.3}$, $CM_{7.50.3}$ and $CSL_{7.50.3}$ functions, showing that each of these functions do indeed exhibit a double-free vulnerability. Our proof is compositional in that the proof for each function reuses the specification of the function below it, as discussed above (e.g. the proof of $SR_{7.50.3}$ reuses the incorrectness specification of `read_data` in Theorem 7 as a lemma).

▶ **Theorem 8.** *Let* $C \in \{SR_{7.50.3}, CM_{7.50.3}, CSL_{7.50.3}\}$*, be a program corresponding to the 7.50.3 versions of* `sec_recv`*,* `choose_mech` *and* `Curl_sec_login`*. Then:*

$$\exists Q.\ [\neg\texttt{double\_free\_vuln}]\ C\ \ [Q] \land (\forall \sigma.\ Q(\sigma) \implies \texttt{double\_free\_vuln}(\sigma)) \land (\exists \sigma.\ Q(\sigma)).$$

Theorem 8 demonstrates a scalable proof for the incorrectness specifications of the functions in Figure 11. We can apply similar techniques to verify their corresponding correctness specifications for version 7.51.0; we have elided these proofs as they are similar.

8 Conclusions and Related Work

Program analysis in the absence of source code techniques typically focus on a combination of hardware (e.g., x86, ARM), intermediary representations, e.g., LLVM-IR, high-level language models (e.g., C/C++). Our work adds to the growing literature on low-level verification using Hoare logic, and is the first to also apply incorrectness logic to verify the absence and presence of bugs in binaries. While many works in the literature focus on particular architectures, e.g., RISC-V, x86 or ARM, our work is on the generic BAP platform, and hence inherits its generality. BAP is a production ready system with 1000s of users but lack a verification component that IsaBIL provides. We have developed a complete formalisation of BIL, and developed the first (in)correctness logics for BIL. Our formalisation of the BIL specification uncovered bugs in that specification, which have been reported to the developers and corrected. We provide automation to enable better proof scalability, and demonstrate compositional proofs.

The importance of binary-level analysis has meant that there is increasing work on theorem provers being applied to verify program binaries.

Verified machine code. Closest to our work is the PICINÆ framework [22], which lifts BIL to Coq for verifying properties like termination and functional correctness. Unlike IsaBIL, its lifter is unverified and adds an extra step via a custom intermediate representation (IR) with unclear semantics. The work is preliminary, focusing on small examples like memset and string comparisons.

Early machine-code verification includes Myreen et al.[37, 38], who used Hoare logic on high-level Arm models. Similarly, Jensen et al. [25] developed separation logic for verifying low-level x86 code via abstractions, with the framework being encoded in Coq. Later work on *decompilation into logic* [39, 40] targets x86, ARM, and PowerPC by translating binaries into tail-recursive functions in HOL4 [24], avoiding conventional lifters like BAP. Verification is done via Hoare triples over these functions, shifting trust to mechanised ISA models [13, 30, 16, 17]. However, these models were not originally designed for decompilation and may not reflect modern ISAs.

The Islaris framework [44] builds on Sail-based Armv8-A and RISC-V semantics [47, 48, 1], using higher-order separation logic in Coq with automatic Sail-to-Coq translation. Unlike Islaris, which focuses on reasoning directly over ISA models, our approach targets lifted assembly to enable a unifying, architecture-agnostic proof layer. Our automation (§ 6) is reusable across architectures supported by BAP. MiniSail [54] encodes a subset of Sail in proof assistants like Isabelle/HOL, enabling reasoning over lifted Sail models similar to IsaBIL. While useful for verifying Sail's type soundness, it lacks IsaBIL's automation and higher-level reasoning capabilities.

Verified compilation. Notable compilation correctness efforts include CakeML [28] and CompCert [7], targeting ML and C, respectively. Sewell et al.[45] verify seL4 C code down to gcc-generated binaries. Bedrock [11] uses an intermediate representation and strongest postcondition reasoning in Coq for C macros, unlike IsaBIL, which allows verification of binaries that may not be generated by a compiler. Overall, these approaches verify functional correctness at the source level, requiring compiler models and assumptions about architecture and optimisation levels. Moreoever, they are unsuitable when source code is unavailable, whereas IsaBIL, via BAP, operates directly on binaries.

Verified lifting. Verbeek et al [51] have formalised and integrated a subset of the x86-64 instruction set (i.e., the 64-bit subset of x86) within Isabelle/HOL and used it to verify so called *sanity properties* (e.g., the integrity of a return address) for a large codebase. However, the focus of this verification is correctness of the lifting, as opposed to properties about the program itself (e.g., termination, functionality). IsaBIL employs type verification, however, this method only verifies that the lifted BIL is correct according to its specification, and not the specification of the assembly it was lifted from. Lam and Coughlin [29] developed a verified BAP lifter for ARMv8. BAP is already accompanied by an extensive set of validation tools that uses trace files to verify the lifter's accuracy [8], that are checked "per instruction". We consider formal verification of BAP lifters to be future work.

Verified instrumentation. Armor [56] presents a checker for memory safety and control flow integrity of Arm binaries, which instruments a check before particular operations, then uses HOL to prove that the modified binary is correct. Similarly, RockSalt [35] is a

[8] `https://github.com/BinaryAnalysisPlatform/bap-veri`

Coq-based checker for a subset of x86 that is used to verify software-based fault isolation implementations. RockSalt is designed to check specific sandbox policies, e.g., that the code will only read/write data from specified contiguous segments in memory. Unlike IsaBIL, which has the full flexibility of Hoare and O'Hearn's logics for proving (in)correctness, both Armor and RockSalt focus on a subset of properties.

Security analysis. An overview of security properties, exploits and tools for binary analysis has been given by Shoshitaishvili et al [46]. Above, we have focussed mainly on safety, though also describe how BAP can be extended to reason about transient execution vulnerabilities (e.g., as exploited by Spectre and Meltdown) [53, 10, 19]. Cheang et al [10] cover model checking (using UCLID5), while Wang et al [53] develop a taint tracking tool based on BAP to check correctness of Spectre mitigations. Griffin and Dongol [19] consider proofs of speculative execution over BAP outputs, but only cover a subset of the BIL language called AIR. They apply Hoare-style proofs to verify a hyperproperty [12] known as TPOD [10] directly over AIR without any of the modularity offered by IsaBIL. Nevertheless, it would be interesting to extend our work to cover proofs of (in)correctness of speculative execution [9, 14].

(Incorrectness) separation logic. In IsaBIL we do not currently have support for separation logic (SL) or its incorrectness analogue, ISL [43]. While the absence of (I)SL support does not impede the scalability of our approach, it is an interesting direction of future work to pursue. Doing so, however, is non-trivial as the memory representation in BIL is incompatible with that of (I)SL. As described in § 3.1, memory in BIL is represented as a "history" of updates (mutations) that must be "replayed" to ascertain its contents. For instance, the BIL memory $[x \leftarrow 1][y \leftarrow 2][x \leftarrow 3]$ describes *any* memory obtainable after applying (replaying) the listed updates in order, namely by first updating location x to hold 1, then updating y to 2 and finally re-updating x to 3. While this confers the existence and the final values of locations x and y, it has no bearing on the existence or the values of other memory locations.

Representing the memory in BIL as described above is a design choice we *inherited* from BIL, and it is not immediately conducive to "separation" as in (I)SL. As such, to add support for (I)SL in BIL we would have to fundamentally change the memory representation in BIL, an undertaking which is far from trivial. While we believe this a worthy direction of work in the future, doing so is beyond the scope of our work here.

References

1 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for armv8-a, risc-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL):71:1–71:31, 2019. doi:10.1145/3290384.

2 Atmel. Avr instruction set manual, 2016. URL: https://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf.

3 Clemens Ballarin. Locales and locale expressions in isabelle/isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *LNCS*, pages 34–50. Springer, 2003. doi:10.1007/978-3-540-24849-1_3.

4 BAP. Bap toolkit examples, 2019. URL: https://github.com/BinaryAnalysisPlatform/bap-toolkit/blob/master/av-rule-21/descr.

5 Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. Cosmedis: A distributed social media platform with formally verified confidentiality guarantees. In *SP*, pages 729–748. IEEE Computer Society, 2017. doi:10.1109/SP.2017.24.

6 A formal specification for BIL: BIL instruction language (v0.3), 2018. URL: `https://github.com/BinaryAnalysisPlatform/bil/releases`.

7 Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 460–475. Springer, 2006. `doi:10.1007/11813040_31`.

8 David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 463–469. Springer, 2011. `doi:10.1007/978-3-642-22110-1_37`.

9 Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In Alastair F. Donaldson and Emina Torlak, editors, *PLDI*, pages 913–926. ACM, 2020. `doi:10.1145/3385412.3385970`.

10 Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *CSF*, pages 288–303. IEEE, 2019. `doi:10.1109/CSF.2019.00027`.

11 Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In Greg Morrisett and Tarmo Uustalu, editors, *ICFP*, pages 391–402. ACM, 2013. `doi:10.1145/2500365.2500592`.

12 M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010. `doi:10.3233/JCS-2009-0393`.

13 Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In Franz Baader, editor, *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2003. `doi:10.1007/978-3-540-45085-6_9`.

14 Brijesh Dongol, Matt Griffin, Andrei Popescu, and Jamie Wright. Relative security: Formally modeling and (dis)proving resilience against semantic optimization vulnerabilities. In *CSF*, pages 403–418. IEEE, 2024. `doi:10.1109/CSF61375.2024.00027`.

15 Chelsea Edmonds and Lawrence C. Paulson. Formal probabilistic methods for combinatorial structures using the lovász local lemma. In Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy, editors, *CPP*, pages 132–146. ACM, 2024. `doi:10.1145/3636501.3636946`.

16 Anthony C. J. Fox. Formal specification and verification of ARM6. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40. Springer, 2003. `doi:10.1007/10930755_2`.

17 Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010. `doi:10.1007/978-3-642-14052-5_18`.

18 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *CoRR*, abs/1707.01747, 2017. `arXiv:1707.01747`.

19 Matt Griffin and Brijesh Dongol. Verifying secure speculation in Isabelle/HOL. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *FM*, volume 13047 of *LNCS*, pages 43–60. Springer, 2021. `doi:10.1007/978-3-030-90870-6_3`.

20 Matt Griffin, Brijesh Dongol, and Azalea Raad. Isabelle/HOL files for "IsaBIL: A framework for verifying (in)correctness of binaries in Isabelle/HOL", 2025. `doi:10.5281/zenodo.15261359`.

21 Matt Griffin, Brijesh Dongol, and Azalea Raad. IsaBIL: A framework for verifying (in)correctness of binaries in Isabelle/HOL (Extended version), 2025. `arXiv:2504.16775`.

22 Kevin W. Hamlen, Dakota Fisher, and Gilmore R. Lundquist. Source-free machine-checked validation of native code in coq. In *FEAST*, FEAST'19, pages 25–30, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3338502.3359759`.

23 Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. `doi:10.1145/363235.363259`.

24 Graham Hutton. Book review: Introduction to HOL: A theorem proving environment for higher order logic by mike gordon and tom melham (eds.), cambridge university press, 1993, ISBN 0-521-44189-7. *J. Funct. Program.*, 4(4):557–559, 1994. doi:10.1017/S0956796800001180.

25 Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. High-level separation logic for low-level code. In Roberto Giacobazzi and Radhia Cousot, editors, *POPL*, pages 301–314. ACM, 2013. doi:10.1145/2429069.2429105.

26 Florian Kammüller and Manfred Kerber. Investigating airplane safety and security against insider threats using logical modeling. In *SP*, pages 304–313. IEEE Computer Society, 2016. doi:10.1109/SPW.2016.47.

27 Florian Kammüller, Markus Wenzel, and Lawrence C Paulson. Locales a sectioning concept for isabelle. In *TPHOLs*, pages 149–165. Springer, 1999. doi:10.1007/3-540-48256-3_11.

28 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 179–192. ACM, 2014. doi:10.1145/2535838.2535841.

29 Kait Lam and Nicholas Coughlin. Lift-off: Trustworthy armv8 semantics from formal specifications. In Alexander Nadel and Kristin Yvonne Rozier, editors, *FMCAD*, pages 274–283. IEEE, 2023. doi:10.34727/2023/ISBN.978-3-85448-060-0_36.

30 Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006. doi:10.1145/1111037.1111042.

31 Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1):1–31, 2008. doi:10.1007/s10817-008-9099-0.

32 Lockheed Martin. Joint strike fighter, air vehicle, c++ coding standard, 2005.

33 Daniel Matichuk, Toby C. Murray, and Makarius Wenzel. Eisbach: A proof method language for isabelle. *J. Autom. Reason.*, 56(3):261–282, 2016. doi:10.1007/s10817-015-9360-2.

34 Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL):67:1–67:32, 2019. doi:10.1145/3290380.

35 Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: better, faster, stronger SFI for the x86. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *PLDI*, pages 395–404. ACM, 2012. doi:10.1145/2254064.2254111.

36 Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *ASE*, pages 1186–1189. IEEE, 2019. doi:10.1109/ASE.2019.00133.

37 Magnus O. Myreen, Anthony C. J. Fox, and Michael J. C. Gordon. Hoare logic for ARM machine code. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 4767 of *LNCS*, pages 272–286. Springer, 2007. doi:10.1007/978-3-540-75698-9_18.

38 Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *LNCS*, pages 568–582. Springer, 2007. doi:10.1007/978-3-540-71209-1_44.

39 Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In Alessandro Cimatti and Robert B. Jones, editors, *FMCAD*, pages 1–8. IEEE, 2008. doi:10.1109/FMCAD.2008.ECP.24.

40 Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation into logic - improved. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 78–81. IEEE, 2012. URL: https://ieeexplore.ieee.org/document/6462558/.

41 Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002. doi:10.1007/3-540-45949-9.

42 Peter W. O'Hearn. Incorrectness logic. *POPL*, 4(POPL):1–32, 2019. doi:10.1145/3371078.

43 Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 225–252, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-53291-8_14.

44 Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: verification of machine code against authoritative ISA semantics. In Ranjit Jhala and Isil Dillig, editors, *PLDI*, pages 825–840. ACM, 2022. doi:10.1145/3519939.3523434.

45 Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In Hans-Juergen Boehm and Cormac Flanagan, editors, *PLDI*, pages 471–482. ACM, 2013. doi:10.1145/2491956.2462183.

46 Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE SP*, 2016.

47 Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in armv8-a. In Ilya Sergey, editor, *ESOP*, volume 13240 of *LNCS*, pages 143–173. Springer, 2022. doi:10.1007/978-3-030-99336-8_6.

48 Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. Armv8-a system semantics: Instruction fetch in relaxed architectures. In Peter Müller, editor, *ESOP*, volume 12075 of *LNCS*, pages 626–655. Springer, 2020. doi:10.1007/978-3-030-44914-8_23.

49 Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com, April 2017.

50 Christian Urban. The Isabelle cookbook: A gentle tutorial for programming Isabelle, 2019.

51 Freek Verbeek, Joshua A. Bockenek, Zhoulai Fu, and Binoy Ravindran. Formally verified lifting of c-compiled x86-64 binaries. In Ranjit Jhala and Isil Dillig, editors, *PLDI*, pages 934–949. ACM, 2022. doi:10.1145/3519939.3523702.

52 Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *IEEE SecDev*, pages 8–9. IEEE Computer Society, 2017. doi:10.1109/SecDev.2017.14.

53 Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Trans. Software Eng.*, 47(11):2504–2519, 2021. doi:10.1109/TSE.2019.2953709.

54 Mark Wassell, Alasdair Armstrong, Neel Krishnaswami, and Peter Sewell. Mechanised metatheory for the sail isa specification, 2018.

55 Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, University of California, Berkeley, 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

56 Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. Armor: fully verified software fault isolation. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *EMSOFT*, pages 289–298. ACM, 2011. doi:10.1145/2038642.2038687.