

# TrABin: Trustworthy Analyses of Binaries

Andreas Lindner<sup>a</sup>, Roberto Guanciale<sup>a</sup>, Roberto Metere<sup>b</sup>

<sup>a</sup>*KTH Royal Institute of Technology, Sweden*

<sup>b</sup>*Newcastle University, UK*

---

## Abstract

Verification of microkernels, device drivers, and crypto routines requires analyses at the binary level. In order to automate these analyses, in the last years several binary analysis platforms have been introduced. These platforms share a common design: the adoption of hardware-independent intermediate representations, a mechanism to translate architecture dependent code to this representation, and a set of architecture independent analyses that process the intermediate representation.

The usage of these platforms to verify software introduces the need for trusting both the correctness of the translation from binary code to intermediate language (called transpilation) and the correctness of the analyses. Achieving a high degree of trust is challenging since the transpilation must handle (i) all the side effects of the instructions, (ii) multiple instruction encodings (e.g. ARM Thumb), and (iii) variable instruction length (e.g. Intel). Similarly, analyses can use complex transformations (e.g. loop unrolling) and simplifications (e.g. partial evaluation) of the artifacts, whose bugs can jeopardize correctness of the results.

We overcome these problems by developing a binary analysis platform on top of the interactive theorem prover HOL4. First, we formally model a binary intermediate language and we prove correctness of several supporting tools (i.e. a type checker). Then, we implement two proof-producing transpilers, which respectively translate ARMv8 and CortexM0 programs to the intermediate language and generate a certificate. This certificate is a HOL4 proof demonstrating correctness of the translation. As demonstrating analysis, we implement a proof-producing weakest precondition generator, which can be used to verify that a given loop-free program fragment satisfies a contract. Finally, we use an AES encryption implementation to benchmark our platform.

*Keywords:* binary analysis, formal verification, proof producing analysis, theorem proving

---

---

*Email addresses:* [andili@kth.se](mailto:andili@kth.se) (Andreas Lindner), [robertog@kth.se](mailto:robertog@kth.se) (Roberto Guanciale), [r.metere2@ncl.ac.uk](mailto:r.metere2@ncl.ac.uk) (Roberto Metere)

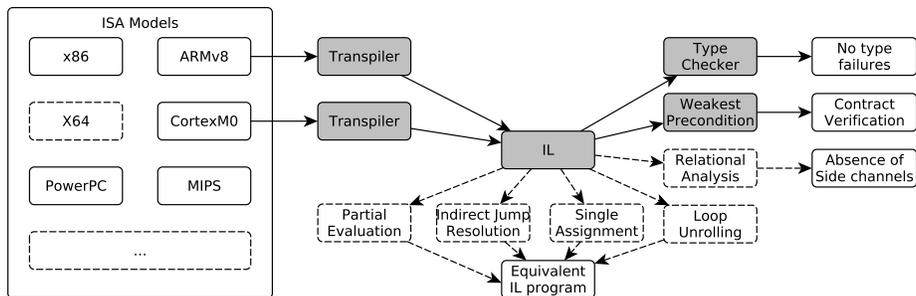


Figure 1: Architecture of the analysis platform. Gray components and non-dashed arrows are contributions described in this paper

## 1. Introduction

Despite the existence of formally verified compilers, the verification of binary code is a critical task to guarantee trustworthiness of systems. This is particularly necessary for software mixing high-level language with assembly (system software), using ad-hoc languages and compilers (specialized software), in presence of instruction set extensions (like for encryption and hashing), and when the source code is not available (binary blobs). This necessity is not only limited to the general-purpose computing scenario but also applies to connected embedded systems, where software bugs can enable a remote attacker to tamper with the security of automobiles, payment services, and smart IoT devices.

The need of semi-automatic analysis techniques for binary code has led to the development of several tools [1, 2, 3]. To handle the complexity and heterogeneity of modern instruction set architectures (ISA), all these tools follow a common design (see Figure 1): They have introduced a platform independent intermediate representation that allows to implement analysis independently of (i) names and number of registers, (ii) instruction decoding, (iii) endianness of memory access, and (iv) instruction side-effects (like updating conditional flags or the stack pointer). This intermediate representation is often a dialect of the Valgrind’s IR [4].

Even if the existing binary analysis platforms have been proved successful thanks to the automation they provide, their usage for verifying software introduces the need of trusting both the transpiler (i.e. the tool translating from machine code to intermediate language) and the analysis. Soundness of the transpiler should not be foregone: It may have to handle multiple instruction encodings (e.g. ARM Thumb), variable instruction length (e.g. Intel), and complex side effects of instructions (e.g. ARM branch with link and conditional executions). Clearly, a transpiler bug jeopardizes the soundness of all analyses done on the intermediate representation. Similarly, complex analyses involve program transformations (e.g. loop unrolling and resolution of indirect jumps) and simplifications (e.g. partial evaluation) that are difficult to implement correctly.

To handle these issues we implement a binary analysis toolkit whose results are machine checkable proofs. The prototype toolkit consists of four components: (i) formal models of the Instruction Set Architectures (ISAs), (ii) the formal model of the intermediate language, called Binary Intermediate Representation (BIR), (iii) a proof-producing transpiler, and (iv) proof-producing analysis tools. As verification platform, we selected the interactive theorem prover HOL4, due to the existing availability of formal models of commodity ISAs [6, 7]. Here, we chose ARMv8 [8] and CortexM0 [9] as demonstrating ISAs. For the target language, we implemented a deep embedding of a machine independent language, which can represent effects to registers, flags, and memory of common architectures and it is relatively simple to analyse. Verification of the transpilation is done via two HOL4 proof producing procedures, which translate respectively ARMv8 and CortexM0 programs to IL programs, and yield the HOL4 proof that demonstrates the correctness of the result. The theorem establishes a simulation between the input binary program and the generated IL program, showing that the two programs have the same behavior. Our contribution enables a verifier to prove properties of the generated IL program (i.e. by directly using the theorem prover or proof-producing analysis techniques) and to transfer them to the original binary program using the generated simulation theorems. As demonstrating analysis, we implement a proof-producing weakest precondition propagator, which can be used to verify that a given loop-free program fragment satisfies a contract.

*Outline.* We present the state of the art and the previous works relating to our contribution in Section 2. Section 3 introduces the HOL4 formal models of the ISAs and the BIR language. Section 4 and Section 5 present the two proof-producing tools: the certifying transpiler and the weakest precondition generation. We demonstrate that the theorems produced by the proof producing tools can be used to transfer verification conditions in Section 6. In Section 7, we test and evaluate our tool. We give concluding remarks in Section 8.

*New contributions.* We briefly describe the new contributions of this paper with respect to [10]. The weakest precondition generation and the corresponding optimization is a new proof producing tool. Therefore Section 5 was not present in [10]. Technically, the BIR model and the transpiler have been heavily re-engineered, for this reason Sections 3.2 and 4 have been adapted to introduce some of the new concepts (e.g. the weak transition relation) and tools (e.g. the type checker, the pre-verified theorems that speed up the transpilation). Also, Section 7 has been rewritten, since it evaluates the new transpiler and the weakest precondition procedure. Finally, the transpiler has been extended to support CortexM0. This allows us to demonstrate that the transpiler can be easily adapted to support new architectures and the requirements for specific proofs for the transpiler are limited.

## 2. Related work

Recent work has shown that formal techniques are ready to achieve detailed verification of real software, making it possible to provide low-level platforms with unprecedented security guarantees [11, 12, 13]. For such system software, limiting the verification to the source code level is undesirable. A modern compiler (e.g. GCC) consists of several millions of lines of code, in contrast to micro-kernels that consist of few thousand lines of code, making it difficult to trust the compiler output even when optimization is disabled<sup>1</sup>.

To overcome this limitation, formally verified compilers [14, 15, 16] and proof/producing compilers [17] have been developed. Similarly to our work, these compilers use detailed models of the underlying ISA to show the correctness of their output. This usually involves a simulation theorem, which demonstrates that the behavior of the produced binary code resembles the one specified by the semantics of the high level language (e.g. C or ML). These theorems permit properties verified at the source-level to be automatically transferred to the binary-level. For instance, CompCert has been used in [18] to verify security of OpenSSL HMAC by transferring functional correctness of the source code to the produced binary.

Even if formally verified compilers obviate the need for trusting their output, they do not fulfill all the needs of verified system software. Some of these compilers target languages that are unsuitable for developing system software (e.g. ML cannot be used to develop a microkernel due to its garbage collector). Also, they do not support mixing the high-level language with assembly code, which is necessary for storing and restoring the CPU context or for managing the page table. Some of the effects of these operations can break the assumptions made to define a precise semantics of the high level language (e.g. a memory write can alter the page table which in turn affects the virtual memory layout). Also, some properties (e.g. absence of side channels created by non-secure accesses to the caches) cannot be verified at the source code level; the analysis must be aware of the exact sequence of memory accesses performed by the software. Finally, binary blob analysis is imperative for verifying memory safety of binary code whose source code is not available (e.g. the power management of ARM trusted firmware).

Unfortunately, detailed formal specifications of machine languages (e.g. the ones used to verify compiler correctness [19]) consist of thousands of lines of definitions. The complexity of these models makes them unusable to directly verify any binary code that is not a toy example. Moreover, the target verification tools, usually interactive theorem provers, provide little or no support for either automatic reasoning or reuse of algorithms among different hardware models. To make machine-code verification proofs reusable by different architectures, Myreen et al. [20] developed a proof-producing decompilation procedure. Those

---

<sup>1</sup>An example of a bug found in GCC: [https://gcc.gnu.org/bugzilla/show\\_bug.cgi?id=80180](https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80180)

tools have been implemented in the HOL4 system and have been used by the seL4 project to check that the binary code produced by the compiler is correct, permitting to transfer properties verified at the source code level to the actual binary code executed by the CPU [21]. The same framework has been used to verify a **bignum** integer library [22]. However, the automation provided by this framework is still far from what is provided by today’s binary analysis platforms (e.g. [1, 2, 3]). These provide tools to compute and analyze control-flow graphs, to perform abstract interpretation and symbolic execution, to verify contracts, to verify information flow properties [23], and to analyze side channels [24]. On the other hand, their usage requires to trust both the transpiler and the implementation of the analysis. Due to the complexity of writing a transpiler for each architecture, recent work has been done to synthesize the transpiler from compiler backends [25]. However, this requires to trust both: the synthesis procedure and the compiler backend.

Regarding trustworthy weakest precondition generation, which is our demonstrating analysis, Vogels et al. [26] verified the soundness of an algorithm for a simple imperative while language in Coq. However, their work does not fit the needs of a trustworthy verification condition generator for a verification toolkit, since the target language is not designed to handle unstructured binary programs.

### 3. Formal Models

#### 3.1. The ARMv8 and CortexM0 models

In our work, we use the ARMv8 and CortexM0 models developed by Fox [6], which are constructed from the pseudocode described in the ARM specifications [8, 9]. These models provide detailed HOL4 formalization of the effects of the instructions, taking into account the different execution modes, flags, and other characteristics of the processor behavior.

We start describing the ARMv8 model. The system state is modeled as a tuple  $s = \langle r, sr, p, m \rangle$ . Here,  $r$  represents a sequence of 64-bit general purpose registers. We identify the  $i$ -th register with  $r(i)$ . The tuple  $sr = \langle pc, sp, lr \rangle$  contains the special registers representing the program counter, the stack pointer, and the link register respectively. The tuple  $p$  represents the current processor state and contains the arithmetical flags. The 64-bit addressable memory is modeled as the function  $m : \mathbb{B}^{64} \rightarrow \mathbb{B}^8$ . Finally, the system behavior is represented by the deterministic transition relation  $s \rightarrow s'$ , describing how the ARMv8 state  $s$  reaches the state  $s'$  by executing a single instruction. The transition relation models the behavior of standard ARMv8 ISA, including fetching four bytes from memory, decoding the instruction, and applying its effects to registers, flags, and memory. Hereafter, we use  $.$  to access tuple fields (e.g.  $s.sr.pc$  states for the program counter of the state  $s$ ) and  $S$  to represent all possible states.

The CortexM0 model has a similar flavour, with main differences consisting of the general purpose registers being 32-bit and the memory being 32-bit

addressable memory. Also, CortexM0 has variable encoding, allowing each instruction to use either two or four bytes. Hereafter, when needed, we use subscripts  $v8$  and  $M0$  to respectively identify ARMv8 and CortexM0 models, i.e.  $\rightarrow_{M0}$  is the transition relation of the CortexM0 model.

The HOL4 machine models consist of hundreds of definitions and their complexity makes it difficult to analyze large programs. To simplify the analyses, the models are equipped with a mechanism to statically compute the effects of a single instruction via the *step* function. Let  $i$  be the binary encoding of an instruction and  $ad$  be the address where the instruction is stored, then the function  $step(i, ad)$  returns a list of step theorems  $[st_1, \dots, st_n]$ . Each theorem  $st_j$  has the following structure:

$$\forall s. \text{fetch}(s.m, s.sr.pc) = i \wedge s.sr.pc = ad \wedge c_j(s) \Rightarrow s \rightarrow t_j(s)$$

where *fetch* is a function that reads the instruction from the memory. Intuitively, each step theorem describes one of the possible behaviors of the instruction and consists of the guard condition  $c_j$  that enables the transition and the function  $t_j$  that transforms the starting state into the next state. We use three examples from the ARMv8 model to illustrate this mechanism.

Let the instruction stored at the address `0x1000000c` be the addition of the registers  $x0$  and  $x1$  into the register  $x0$  (whose encoding is `0x8b000020`), the step function produces the following step theorem:

$$\begin{aligned} \forall s. \text{fetch}(s.m, s.sr.pc) = \text{0x8b000020} \wedge s.sr.pc = \text{0x1000000c} \Rightarrow \\ s \rightarrow ( \lambda s'.s' \quad \text{with } r(0) = s'.r(0) + s'.r(1) \text{ with } sr.pc = s'.sr.pc + 4 ) s \end{aligned}$$

(where  $s'$  with  $r(0) = v$  updates the register  $x0$  of the state  $s'$  with  $v$ ). In this case, only one theorem is generated, and there is no guard condition (i.e.  $c_1$  is a tautology).

Some machine instructions (i.e. conditional branches) can have different behavior according to the value of some state components. In these cases, the step function produces as many theorems as the number of possible execution cases. For example, the output of the step function for the Signed Greater Than branch instruction consists of the following two theorems:

$$\begin{aligned} \forall s. \text{fetch}(s.m, s.sr.pc) = \text{0x54fffe8c} \wedge s.sr.pc = \text{0x1000000c} \\ \wedge s.p.Z = 0 \wedge s.p.N = s.p.V \Rightarrow \\ s \rightarrow (\lambda s'.s' \text{ with } sr.pc = s'.sr.pc - \text{0x30})s \end{aligned}$$

$$\begin{aligned} \forall s. \text{fetch}(s.m, s.sr.pc) = \text{0x54fffe8c} \wedge s.sr.pc = \text{0x1000000c} \\ \wedge \neg (s.p.Z = 0 \wedge s.p.N = s.p.V) \Rightarrow \\ s \rightarrow (\lambda s'.s' \text{ with } sr.pc = s'.sr.pc + 4)s \end{aligned}$$

That is, if the test succeeds (i.e.  $c_1 = s.p.Z = 0 \wedge s.p.N = s.p.V$  holds) then the jump is taken (in this case jumping back in a loop to the address  $pc - 0x30$ ), otherwise (i.e.  $c_2 = \neg(s.p.Z = 0 \wedge s.p.N = s.p.V)$  holds) the jump is not taken (the program counter is updated to point to the next instruction). Notice that for every state  $s$  the condition  $c_1 \vee c_2$  hold.

Finally, some instructions (i.e. memory stores) can have unsound behavior if some conditions are not met. In these cases, the step function generates the step theorems only for the correct behaviors; for a given instruction, let  $st_1, \dots, st_n$  be the generated theorems and  $c_1, \dots, c_n$  the corresponding guards, the behavior of the instruction is soundly deduced by the step function for every state  $s$  such that  $\bigvee_j c_j(s)$  holds and can not be deduced otherwise. For example, the output of the step function for a memory store consists of the theorem:

$$\begin{aligned} & \forall s. \text{read}_{32}(s.m, s.sr.pc) = \text{0xf90007e0} \wedge s.sr.pc = \text{0x1000000c} \\ & \wedge \text{aligned}(s.sr.sp + 8) \Rightarrow \\ & s \rightarrow \left( \begin{array}{l} \lambda s'.s' \text{ with } m = \text{write}_{64}(s'.m, s'.sr.sp + 8, s'.r(0)) \\ \text{with } sr.pc = s'.sr.pc + 4 \end{array} \right) s \end{aligned}$$

Intuitively, the step function can predict the behavior only for states having the target address (i.e.  $s.sr.sp + 8$ ) aligned.

### 3.2. The BIR model

Our platform uses the machine independent Binary Intermediate Representation (BIR). In this representation, a statement has only explicit state changes, i.e. there are no implicit side effects, and it can only affect one variable.

BIR's syntax is depicted in Table 1. A program is a list of blocks, each one consisting of a uniquely identifying label (i.e. a string or an integer), a list of block statements, and one control flow statement. In the following we assume that all programs are well defined, i.e. they have no duplicate block labels. A statement can affect the state by (i) assigning the evaluation of an expression to a variable, (ii) terminating the system in a failure state if an assertion does not hold. A control flow statement can (conditionally or unconditionally) modify the control flow. As usual, labels are used to refer to the specific locations in the program and can be the target of jump statements.

BIR expressions are built using constants (i.e. strings and integers), conditionals (i.e. **ifthenelse**), standard binary and unary operators (ranged over by  $\diamond_b$  and  $\diamond_u$  respectively) for finite integer arithmetic and casting, and accessing variables of the environment (i.e. **var**). Additionally, two types of expressions can operate on memories. The expression **load**( $exp_1, exp_2, \tau_{reg,n}$ ) reads  $n$  bytes from the memory  $exp_1$  starting from the address  $exp_2$ . The expression **store**( $exp_1, exp_2, exp_3, \tau_{reg,n}$ ) returns a new memory in which all the locations have the same values as the initial memory  $exp_1$  except the addresses  $exp_2 + i$  where  $i \in [0 \dots n - 1]$  that contain the chunks of  $exp_3$ . Figure 2 provides an example of a BIR program.

Hereafter, we use  $\Delta$  to represent the set of all possible strings. These can be used to identify both labels and variable names. We use  $\tau$  to range over BIR data types; let  $n \in \{1, 8, 16, 32, 64\}$ , the type for words of  $n$ -bits is denoted by  $\tau_{reg,n}$  and the type for memories addressed using  $n$ -bits is denoted by  $\tau_{mem,n}$ . We use  $T$  and  $V$  to represent the set of all BIR types and values respectively.

A BIR environment  $env$  maps variable names (given as strings) to pairs of type and value;  $env : \Delta \rightarrow (T \times V)$ . Types of variables are immutable and any

$pr$	$:=$	$block^*$
$block$	$:=$	$(string \mid integer, bst^*, cfst)$
$bst$	$:=$	$\mathbf{assign}(string, exp) \mid \mathbf{assert}(exp)$
$cfst$	$:=$	$\mathbf{jmp}(exp) \mid \mathbf{cjmp}(exp, exp, exp)$
$exp$	$:=$	$string \mid integer \mid$ $\mathbf{ifthenelse}(exp, exp, exp) \mid$ $\diamond_u exp \mid exp \diamond_b exp \mid \mathbf{var} string \mid$ $\mathbf{load}(exp, exp, \tau) \mid \mathbf{store}(exp, exp, exp, \tau)$

Table 1: BIR's syntax

wrongly typed operation produces a run-time failure. The semantics of BIR expressions is modeled by the evaluation function  $eval$ : It takes an expression  $exp$  and an environment  $env$  and yields either a value having a type in  $T$  or  $\bullet$ . The evaluation intuitively follows the semantics of operations by recursively evaluating the sub-expressions given as operands. The value  $\bullet$  results when operators and types are incompatible, thus modeling a type error.

A BIR state  $bs = (env, p) \in BS$  is a pair of an environment  $env$  and a program counter  $p \in \Delta \cup \mathbb{B}^{64} \cup \{\perp, \bullet\}$ . While executing a program, the program counter is  $\Delta \cup \mathbb{B}^{64}$  and is the label of the executing block. In the cases of either type mismatch or failed assertion, the execution terminates setting the program counter to either  $\bullet$  (type mismatch) or  $\perp$  (failed assertion). Notice that the program is not part of the state, disallowing run-time changes to the program.

The system behavior is modeled by the deterministic transition relation  $pr : bs \rightsquigarrow bs'$ , which describes the execution of one BIR block. In HOL4, this relation is modeled by the execution function  $exc$ , which defines the small step semantics of an entire block. Hereafter, we use  $bs' \in \{\perp, \bullet\}$  when the program counter of the resulting states represents one of the possible errors. The relation  $\rightsquigarrow$  is defined on top of two other functions:  $bst : env \rightarrow env'$  models the environment effects of a single block statement  $bst$ , and  $cfst : env \rightarrow p'$  models the program counter resulting by executing a single control flow statement  $cfst$ . Both functions can return  $\perp$  and  $\bullet$  in case of violated assertions and type errors respectively.

The execution of  $\mathbf{assign}(X, exp)$  assigns the evaluation of the expression  $exp$  to the variable  $X$ . Let  $v = eval(exp, env)$  and  $t$  be the type of  $v$ , the value of the variable is updated in the context  $(env[X \leftarrow (t, v)])$ . The statement fails in case of a type mismatch:  $v = \bullet$  or  $env(X) = (t', -) \wedge t \neq t'$ . The statement  $\mathbf{assert}(exp)$  has no effects if the expression evaluates to true (i.e.  $eval(exp, env) = (\tau_{reg, 1}, 1)$ ) and terminates in an error state otherwise.

The execution of  $\mathbf{jmp}(exp)$  jumps to the referenced block, by setting the program counter to  $eval(exp, env)$ . If the type of  $exp$  is neither string nor integer then the statement fails. The statement  $\mathbf{cjmp}(exp_c, exp_1, exp_2)$  changes the control flow based on the condition  $exp_c$ . The statement fails if the type of

$$\left[ \left( \begin{array}{l} 0x400000, \\ \left[ \begin{array}{l} \mathbf{assign}(R1, \mathbf{load}(\mathbf{var}(MEM), \mathbf{var}(SP), \tau_{reg,32})) \\ \mathbf{assign}(SP, \mathbf{var}(SP) + 4) \end{array} \right], \\ \mathbf{jmp}(0x400004) \end{array} \right), \right. \\ \left. \left( \begin{array}{l} 0x400004, \\ \left[ \begin{array}{l} \mathbf{assign}(MEM, \mathbf{store}(\mathbf{var}(MEM), \mathbf{var}(SP), \mathbf{var}(R1), \tau_{reg,32})) \\ \mathbf{assign}(SP, \mathbf{var}(SP) - 4) \end{array} \right], \\ \mathbf{jmp}(0x400008) \end{array} \right) \right) \right]$$

This BIR program pops and pushes a register from/to the stack. The register is modeled by the variable  $R1$ , the stack pointer by the variable  $SP$ , and the memory by the variable  $MEM$ . The program consists of two blocks, which are labeled  $0x400000$  and  $0x400004$ . The first block assigns to  $R1$  the content of  $MEM$  starting from  $SP$ , then it increases the stack pointer. The second block saves  $R1$  into the stack and decreases the stack pointer. Notice that  $\mathbf{store}(\mathbf{var}(MEM), \mathbf{var}(SP), \mathbf{var}(R1), \tau_{reg,32})$  returns a modified copy of  $MEM$  and does not directly modify  $MEM$ . Therefore, to model a memory write, the new memory must be explicitly assigned to the variable  $MEM$ .

Figure 2: Example of a BIR program

the condition is not  $\tau_{reg,1}$  or the targets (i.e.  $eval(exp_1, env)$  or  $eval(exp_2, env)$ ) are not valid labels. Notice that the targets of the jump are evaluated using the current context, allowing BIR to express indirect jumps that are resolved at run-time.

The HOL4 model is equipped with several supporting tools and definitions, which simplify the development of the transpiler and analyses. A weak transition relation is defined to hide some executions steps. Let  $LS$  be a set of labels,  $\rightsquigarrow_{LS}: BS \mapsto BS$  is a partial function built on top of the single-block small step semantics, which yields the first reachable state that is an error state or that has the program counter in  $LS$ . The function is undefined if no error and no label in  $LS$  are reachable. A formally verified type checker permits to rule out run-time type errors. Well typed programs cannot have wrongly typed expressions (i.e. arguments of binary operators must have the same type, values to store in memory must match the specified type, etc.), use only one type per variable name, use expressions with correct types in statements (i.e. conditions in assertions and conditional jumps must be boolean expressions). A pair consisting of a program and an environment is well typed if the program is well typed and if for every variable the type of variable in the environment matches the usage of the variable in the program. Well typed program fragments that start from well typed environments cannot cause type errors and can only reach well typed environments.

## 4. Certifying Transpiler

The translation procedure uses a mapping of HOL4 machine states to BIR states. Every machine state field must be mapped to a BIR variable or to the program counter. Our framework provides one default mapping for each supported architecture: ARMv8 and CortexM0. In both cases, the  $i$ -th register is mapped to the variable  $R\langle i \rangle$  (i.e.  $R0$  represents the register number zero), the variable  $MEM$  represents the system memory, the BIR program counter reflects the machine’s program counter, and every flag is mapped to a proper variable. This mapping induces a simulation relation  $\sim \subseteq BS \times S$  that relates BIR states to machine states.

To transform a program to the corresponding BIR fragment, we need to capture all possible effects of the program execution in terms of affected registers, flags and memory locations. The generated BIR fragment should emulate the behaviour of the instructions executed on the machine. This goal is accomplished by reusing the *step* function and the following two HOL4 certifying procedures.

- A procedure to translate HOL4 word terms (i.e. those having type  $\mathbb{B}^{64}$ ,  $\mathbb{B}^8$ ,  $\mathbb{B}$  etc.) to BIR expressions. This procedure is used to convert the guards of the step theorems and the expressions contained in the transformation functions.
- A procedure to translate a single instruction to the corresponding BIR fragment. This procedure computes the possible effects of an instruction using the transformation functions of the step theorems.

To phrase the theorem produced by the transpiler we introduce the following notations. A binary program  $BIN_{pr}$  is represented by a finite set of pairs  $(ad_j, i_j)$ , where each pair represents that the instruction  $i_j$  is located at the address  $ad_j$ . The predicate  $stored(s, BIN_{pr})$  states that the program  $BIN_{pr}$  is stored in the memory of the state  $s$  (formally,  $stored(s, BIN_{pr}) \stackrel{\text{def}}{=} \forall(ad_j, i_j) \in BIN_{pr}. \text{fetch}(s.m, ad_j) = i_j$ ). A single machine instruction can be transpiled to multiple blocks. The transpiler uses a naming convention to distinguish the label of the first block produced for an instruction from the labels of the other blocks. Hereafter, we use  $LS_1$  to identify the set of labels that represents the entry point of instructions. We denote  $n$  transitions of machine states with  $\rightarrow^n$  and  $n$  transitions of BIR visiting  $LS_1$  with  $\rightsquigarrow_{LS_1}^n$ . The translation procedure generates a theorem that resembles compiler correctness<sup>2</sup>:

**Theorem 1** *Let  $ad_0$  be the entry point of the program  $BIN_{pr}$ . For every state  $s$  and BIR state  $bs$ , if  $stored(s, BIN_{pr})$ ,  $s.sr.pc = ad_0$ , and  $bs \sim s$ , then*

---

<sup>2</sup>The ISA and BIR transition systems are deterministic, thus the transition relations are functions. For this reason we omit quantifiers over the states on the right hand side of transitions, since they are unique.

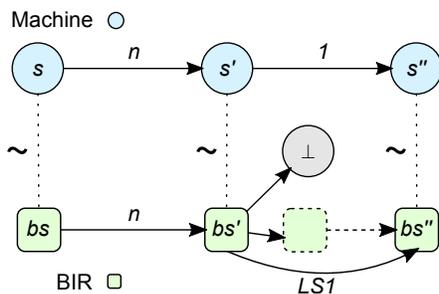


Figure 3: The theorem demonstrated by the transpiler

1. for every  $n > 0$  if  $s \rightarrow^n s'$  then  
 $pr : bs \rightsquigarrow_{LS_1}^n bs' \wedge (bs' = \perp \vee bs' \sim s')$ , and
2. for every  $n > 0$  if  $pr : bs \rightsquigarrow_{LS_1}^n bs' \wedge bs' \neq \perp$  then  
 $s \rightarrow^n s' \wedge bs' \sim s'$ .

The meaning of the transpiler theorem is depicted in Figure 3. Each machine instruction is translated to multiple blocks, the first one having a label in  $LS_1$ , and each block consisting of multiple statements. Assuming that the program is stored in machine memory, the state is configured to start the execution from the entry point  $ad_0$  of the program, and the initial HOL4 machine state resembles the initial BIR state, then (1) for every state  $s'$  reachable by the ISA model, there is an execution of the BIR program  $pr$  that results (after visiting  $n$  blocks whose labels are in  $LS_1$ ) in either an error state ( $bs' = \perp$ ) or in a state  $bs'$  that resembles  $s'$ , and (2) for every state  $bs'$  reachable by the BIR program after reaching the first block of an instruction, there is an execution of the machine that re-establishes the simulation relation.

Error states permit to identify if an initial configuration can cause a program to reach a state that cannot be handled by the transpiler (e.g. self-modifying programs or programs containing instructions whose behavior cannot be predicted by the step function). It is worth noticing that these cases cannot be identified statically without knowing the program preconditions (e.g. misaligned memory accesses can be caused by the initial content of the stack where pointers are stored) and must be ruled out when verifying the program.

#### 4.1. Translation of expressions

In order to build the transpiler on top of the step function, the HOL4 expressions occurring in the guards and the transformation functions must be converted to BIR expressions. For example, while translating the binary instruction `0x54fffe8c` of Section 3.1 to a conditional jump, the expressions  $s.p.Z = 0 \wedge s.p.N = s.p.V$  and  $s'.sr.pc - 0x30$  must be expressed in BIR to generate the condition and the target of the jump respectively.

Let  $e$  be a HOL4 expression, the output of the transpiler is the theorem  $\forall env.A(env) \Rightarrow (eval(exp, env) = e)$ , stating that, if the environment satisfies

the assumption  $A$ , then the evaluation of  $exp$  is  $e$ . These assumptions usually constrain the values of the variables in the environment to match the free variables of the HOL4 expressions. For instance, for the expression  $s.p.N = s.p.V$  the transpiler generates the theorem  $\forall env, s. (env("N") = (\tau_1, s.p.N) \wedge env("V") = (\tau_1, V)) \Rightarrow (eval((\mathbf{var} "N" = \mathbf{var} "V"), env) = (N = V))$ .

If a HOL4 operator has no direct correspondence in BIR, the transpiler uses a set of manually verified theorems to justify the emulation of the operator via a composition of the primitive BIR operators. This is the case for expressions involving bit extractions (i.e. most significant bit, least significant bit, etc), alignment, reversing endianness, and rotation.

For several ISA models, special care must be taken to convert expressions involved in updating status flags. For instance, both ARMv8 and CortexM0 use so called NZCV status flags for conditional execution, where

- **N**egative is set if the result of a data processing instruction was negative
- **Z**ero is set if the result is zero
- **C**arry is set if is set if an addition, subtraction or compare causes a result bigger than word size
- **oV**erflow is set if an addition, subtraction or compare produces a signed result bigger than 31/63 bit (for CortexM0 and ARMv8 respectively), i.e. the largest representable positive number

The expressions produced by the ISA models for these flags involve conversion of words to natural numbers and arithmetic operations with arbitrary precision. For example, following the pseudocode of the ARMv8 reference manual [8], the *carry flag* in 64-bit additions is computed by the expression  $[x] + [y] \geq 2^{64}$ , where  $x, y \in \mathbb{B}^{64}$  and  $[\cdot] : \mathbb{B}^{64} \rightarrow \mathbb{N}$  is their interpretation as natural numbers. Both the inequality and the addition cannot be directly converted as BIR expression, because BIR can only handle finite arithmetics<sup>3</sup>. For the *carry flag* the transpiler uses the theorem  $\forall n > 0. \forall x, y \in \mathbb{B}^n. ([x] + [y] \geq 2^n) \Leftrightarrow ((\sim x) <_w y)$ , where  $\sim$  and  $<_w$  are complement and unsigned comparison of bitvectors respectively.

#### 4.2. Translation of single instructions

The transpilation of a single instruction takes three arguments: the binary code  $i$  of the instruction, the address  $ad$  of the instruction in memory, and a set of memory address ranges  $MemR$ . The latter argument identifies which memory addresses should not be modified by the instruction and is used to guarantee that the program is not self-modifying. In fact, a self-modifying program cannot be transformed to equivalent BIR programs (due to BIR following the Harvard architecture). If an instruction modifies the program code then the translated BIR program must terminate in an error state. The addresses in  $MemR$  are

---

<sup>3</sup>This design choice simplifies the development of analyses for BIR and the integration of external tools, like SMT solvers supporting bitvectors

```

pr =      [block0, block1, block2]
block0 = (ad, smts0)
smts0 =  assert (expc)
          cjmp (expc1, "ad-1", "ad-2")
blocki = ("ad-i", smtsi)
smtsi =  assert(expm)
          assign (tmpF1, expF1)
          ...
          assign (F1, var tmpF1)
          ...
          jmp (exp)

```

Figure 4: BIR fragment generated to one instruction

used to instrument the instruction transpiler with the information about where the program code is stored. Hereafter we use  $pr = \text{transpile}(i, ad, MemR)$  to represent that the transpiler produces the program fragment (sequence of blocks)  $pr$ . Also, we use  $pr \in pr'$  to represent that a program  $pr'$  contains the fragment  $pr$ .

The transpiler uses the *step* function to compute the behavior of the input instruction  $i$  and to generate the step theorems  $[st_1, \dots, st_n]$ , where  $st_j$  is  $\forall s. \text{fetch}(s.m, s.sr.pc) = i \wedge s.sr.pc = ad \wedge c_j(s) \Rightarrow s \rightarrow t_j(s)$ . Hereafter we assume that the *step* function generates two theorems (i.e.  $n = 2$ ), which is the case for conditional instructions in CortexM0 and branches in ARMv8. We will comment on the other cases at the end of this section.

A single machine instruction can be translated to multiple BIR blocks, following the template of Figure 4. The label of the first block is equal to the address of the instruction and is the only block having an integer label. The other two blocks have string labels and represent the effects of the two step theorems.

The behavior of the instruction can be soundly deduced by the step function only if one of the  $c_j$  predicates holds (see Section 3.1). The transpiler simplifies the disjunction of the guards demonstrating  $\forall s. \bigvee_j c_j(s) = e_c$  (where  $e_c$  is a HOL4 predicate) and translates it to a BIR expression  $exp_c$  (demonstrating  $\forall env, s. ((env, p) \sim s) \Rightarrow (eval(exp_c, env) = e_c)$ ). The BIR statement **assert** ( $exp_c$ ) is generated as first statement of the first block. Intuitively, if a machine state  $s$  does not satisfy any guard, then every similar BIR state  $(env, p)$  does not satisfy the assertion, causing the BIR program to terminate in a error state. On the other hand, if the BIR state satisfies the assertion, then every similar machine state satisfies at least one of the guards, thus the instruction's behavior can be deduced by the step function.

The second task is to redirect the BIR control flow to the proper internal block according to the guards of the step theorems. The procedure translates  $c_1$  to a BIR expression  $exp_{c_1}$  ( demonstrating  $\forall env, s. ((env, p) \sim s) \Rightarrow$

( $eval(exp_{c_1}, env) = e_{c_1}$ ) and **cjmp**( $exp_{c_1}$ , "ad-1", "ad-2") is generated as last statement of the first block. Intuitively, a BIR state ( $env, p$ ) executes  $block_1$  if and only if the similar machine state  $s$  satisfies  $c_1$ .

The third task is to translate the effects of the instruction on every field of the machine state for every step theorem  $st_j$ . Let  $f$  be one field of the machine state (e.g.  $f = r(0)$  is the register zero) and let  $F$  be the corresponding variable of BIR according to the relation  $\sim$ . The transpiler computes the new value  $e_F$  of the field (and demonstrates  $\forall s.(t_j(s)).f = e_F$ ). If  $e_F = s.f$  then the machine state's field is not affected by the instruction and the corresponding variable  $F$  should not be modified by the generated BIR block, otherwise the variable  $F$  must be updated accordingly. The expression  $e_F$  is translated to obtain the theorem  $\forall env.eval(exp_F, env) = e_F$  and the BIR statement **assign**( $tmpF, exp_F$ ) is generated. The need of a temporary variable  $tmpF$  is due to the presence of instructions that can affect several variables, and whose resulting values depend on each other (i.e. imagine an instruction swapping registers zero and one, where  $t(s) = s$  with  $\{r(0) = s.r(1)$  and  $r(1) = s.r(0)\}$ ). After all field values have been computed and stored into the temporary variables, these are copied into the original variables via the statement **assign**( $F, var tmpF$ ).

Special care is needed for memory updates (i.e.  $f = m$ ). The BIR program should fail if the original program updates a memory location in  $MemR$ . The transpiler inspects the expression  $e_{MEM}$  to identify the addresses that can be changed by the instruction and extracts the corresponding set of expressions  $e_1, \dots, e_n$  (in CortexM0 and ARMv8 a single instruction can store multiple registers). The expression  $\bigwedge_i e_i \notin MemR$  (which guarantees that no modified address belongs to the reserved memory region) is translated to obtain the theorem  $\forall env.eval(exp_m, env) = \bigwedge_i e_i \notin MemR$  and the BIR statement **assert**( $exp_m$ ) is added as preamble of the block. If the machine instruction modifies an address in  $MemR$ , then the corresponding BIR state does not satisfy the assertion, causing the BIR program to terminate in an error state.

Finally, the program counter field is used to generate statements that update the control flow. The expression  $e_{pc}$  is translated to  $exp_{pc}$  and **jmp**( $exp_{pc}$ ) is appended to the BIR fragment. If possible,  $e_{pc}$  is first simplified to be a constant, which reduces the number of indirect jumps in the BIR program.

This procedure is generalized to handle arbitrary number of step theorems, using one block per theorem. Moreover, the transpiler optimizes some common cases. If the transformation function  $t_j$  modifies only the program counter (i.e. a conditional instruction, which behaves as NOP if the instruction condition is not met) then  $block_j$  is not generated and the translation of  $(t_j(s)).sr.pc$  is used in place of "ad-j" in **cjmp**( $exp_{c_1}$ , "ad-1", "ad-2"). If there is only one step theorem, then the  $block_j$  is merged with  $block_0$  and the conditional jump is removed. If an updated state field  $f$  is not used to compute the value of other fields then the temporary variable is not used.

A large part of the HOL4 implementation focuses on optimizing the verification that the generated fragment resembles the original machine instruction. This is done by preproved theorems about the template-blocks (i.e.  $block_j$ ), which enable the transpiler to use the intermediate theorems generated for ex-

pressions and the step theorems to establish the *instruction-theorem*. The proved theorems for the template-blocks also ensure that these are well-typed, statically guaranteeing that a generated fragment cannot cause a type error.

**Theorem 2** *Let  $pr = \text{transpile}(i, ad, MemR)$ . For every machine state  $s$ , BIR state  $bs$ , and BIR program  $pr'$  if  $\text{read}_{32}(s.m, s.sr.pc) = i$ ,  $s.pc = ad$ ,  $bs \sim s$ , and  $pr \in pr'$ , then*

1.  $\exists bs'. pr' : bs \rightsquigarrow_{LS_1} bs'$
2. if  $s \rightarrow s'$  and  $pr' : bs \rightsquigarrow_{LS_1} bs'$  <sup>4</sup> then  
 $((bs' = \perp) \vee (bs' \sim s' \wedge \forall a \in MemR. s'.m(a) = s.m(a)))$

The theorem shows (1) the BIR program either fails or reaches the first block of an instruction starting from the first block of the translated one (i.e. the internal blocks do not introduce loops), and (2) if the complete execution of the generated blocks succeeds then the BIR program behaves equivalently to the machine instruction and memory in *MemR* is not modified.

#### 4.3. Transpiling programs

The theorems generated for every instruction are composed to verify Theorem 1. Property (1) is verified by induction over  $n$ , using predicate  $MemR = \{ad \mid (ad, i) \in BIN_{pr}\}$ . This ensures that the program is in memory after the execution of each instruction, thus allowing to make the assumption of the translation theorem (i.e.  $\forall (ad_j, i_j) \in BIN_{pr}. \text{read}_{32}(s.m, ad_j) = i_j$ ) an invariant.

Property (2) is verified by induction over  $n$ . Since  $bs \sim s$  then the program counter of  $bs$  points to the one of the  $block_0$  produced by the transpiler. Therefore we can use the corresponding instruction-theorem to show that  $bs'$  exists. This and the fact that the ISA transition relation is total enable part (2) of the instruction-theorem, showing that the machine instruction behaves equivalently to the BIR block.

#### 4.4. Support for more architectures

In the following, we review the modifications of the certifying procedures needed to support other common computer architectures, like MIPS, x86 and ARMv7-A. The transpiler has three main dependencies: A formal model of the architecture, a function producing step theorems, and the definitions of a simulation relation. There exist HOL4 models for x86, x64, ARMv7-A, RISC-V, and MIPS, which are equipped with the corresponding step function. The simulation relation can differ for each architecture since it maps machine state fields to BIR variables. In fact, the name, the number, and the type of registers can be very different among unrelated architectures. However, defining the simulation relation is straightforward, since it simply requires to map machine state fields to BIR variables.

---

<sup>4</sup>We remark that  $\rightarrow$  is deterministic and left total, and  $\rightsquigarrow_{LS}$  is deterministic

The expression translation has to handle the expressions of guard conditions and transformation functions that are present in the step theorems. Since these use HOL4 number and word theories, independently of the architecture, big parts of the translation of Section 4.1 can be reused. There are two exceptions: One is the possible usage of different word lengths, and the other is the need of proving helper theorems to justify the emulation of operators that have no direct correspondence in BIR (e.g. for the computation of the carry flag in CortexM0 and ARmv8, or to support specialized instructions for encryption).

Defining the simulation relation and extending the expression translation enable the transpilation of single instructions of Section 4.2 to support a new architecture, without requiring further modifications. In fact, the structure of the produced BIR blocks is architecture independent and is ready to support some peculiarities of MIPS and x64.

#### 4.4.1. Delay slots

On the MIPS architecture, jump and branch instructions have a “delay slot”. This means that the instruction after the jump is executed before the jump is executed. The HOL4 model for MIPS handles delay slots using the shadow registers BranchDelay ( $bd$ ), which can be either unset or the address of an instruction. This register can be mapped to BIR using a boolean variable ( $BD\_SET$ ), which holds if the shadow register is set, and a word variable ( $BD$ ), which represents the register value.

Let  $s$  be a MIPS state, the transition relation is undefined if  $s.sr.bd$  is set and the instruction makes a jump (i.e. jumps are not allowed after jumps). Otherwise it yields a state  $s'$  where

- $s'.sr.pc = s.sr.bd$  if  $s.sr.bd$  is set, otherwise  $s'.sr.pc = s.sr.pc + 4$
- $s'.sr.bd$  is the target of jump if the instruction modifies the control flow, otherwise  $s'.sr.bd$  is unset

The step theorems can be accordingly generated. Let  $c_1, \dots, c_n$  be the step theorem’s guards of an instruction and let  $c_{jmp}$  be the condition that causes the instruction to modify the control flow (i.e.  $c_{jmp}$  is always false if the current instruction is neither a jump nor a branch),  $\bigvee_j c_j(s)$  holds if  $s.sr.bd$  is unset or  $c_{jmp}$  does not hold. Following the template of Figure 4, the transpiler will produce the assertion **assert** ( $(\mathbf{var} \text{ "BD\_SET" } = F) \vee \neg exp_{jmp}$ ), where  $\forall env, s. ((env, p) \sim s) \Rightarrow (eval(exp_{jmp}, env) = c_{jmp})$ . Non-jump and non-branch instructions need multiple step theorems to model delay slots. These instructions could be translated via two BIR blocks: one executed when  $BD\_SET$  holds and that jumps to  $BD$ ; and one that jumps to the next instruction when  $BD\_SET$  does not hold. Finally, the variables  $BD\_SET$  and  $BD$  can be update like all other register variables.

#### 4.4.2. Different calling conventions

Intel x86 and x64 architectures have different calling conventions. In fact, in x64 a limited number of parameters can be passed via registers. It is worth

noticing that the transpiler does not make any assumption on the calling convention used. In fact, the transpiler procedure does not need to know symbols and can handle programs that violate standard calling convention. In practice, different calling conventions will result in different assembly instructions, whose behavior is captured by the step theorems.

## 5. Weakest precondition generation

Contract based verification is a convenient approach for compositionally verifying properties of systems. Due to the unstructured nature of binary code, a binary program (and therefore the corresponding BIR program) can have multiple entry and exit points. For this reason we adapt the common notion of Hoare triples. Hereafter we assume programs and environments to be well typed. Let  $\mathbb{P}$  and  $\mathbb{Q}$  be two partial functions mapping labels to BIR boolean expressions, we say that a BIR program  $pr$  satisfies the contract  $\{\mathbb{P}\}pr\{\mathbb{Q}\}$ , if the execution of the program starting from an entry point  $ad \in \text{dom}(\mathbb{P})$  and a state satisfying the precondition  $\mathbb{P}(ad)$  establishes the postcondition  $\mathbb{Q}(ad')$  whenever it reaches an exit point  $ad' \in \text{dom}(\mathbb{Q})$ , formally

**Definition 1 (BIR triple)**  $\{\mathbb{P}\}pr\{\mathbb{Q}\}$  holds iff for every  $env, ad \in \text{dom}(\mathbb{P})$ , if  $eval(\mathbb{P}(ad), env), pr : (env, ad) \rightsquigarrow_{\text{dom}(\mathbb{Q})} (env', p')$  then  $p' \neq \perp$  and  $eval(\mathbb{Q}(p'), env')$ .

There are well known semi-automatic techniques to verify contracts for program fragments that are loop free and whose control flow can be statically identified. A common approach is using precondition propagation, which computes a precondition  $\mathbb{P}'$  from a program  $pr$  and a postcondition  $\mathbb{Q}$ . Hence, the algorithm must ensure that the triple  $\{\mathbb{P}'\}pr\{\mathbb{Q}\}$  holds. Also, if for every  $ad \in \text{dom}(\mathbb{P}')$  the precondition  $\mathbb{P}'(ad)$  is the weakest condition ensuring that  $\mathbb{Q}$  is met, then the algorithm is called weakest precondition propagation. The desired contract holds if for every  $ad \in \text{dom}\mathbb{P}$  the precondition  $\mathbb{P}(ad)$  implies the computed precondition  $\mathbb{P}'(ad)$ . Therefore, the overall workflow is: specify a desired contract, automatically compute the weakest precondition, and prove the implication condition.

### 5.1. WP generation

Since the BIR transition relation models the complete execution of one block, we consider a block the unit for which the algorithm operate. The weakest precondition is propagated by following the control flow graph (CFG) backwards. The CFG is a directed acyclic graph (due to absence of loops) with multiple entry points and multiple exit points. For instance, Figure 5a depicts the CFG of a program with blocks having labels  $l_0$  through  $l_7$ , two entry points (i.e.  $l_0$  and  $l_1$ ) and two exit points ( $l_6$  and  $l_7$ ).

Our approach is to iteratively extend a partial function  $\mathbb{H}$ , which initially equals the postcondition partial function  $\mathbb{Q}$ . This function maps labels to their respective weakest precondition or postcondition in the case of exit points. Formally, the procedure preserves the invariant  $\{\mathbb{H} \downarrow_{\text{dom}(\mathbb{Q})}\}pr\{\mathbb{Q}\}$  and  $\mathbb{H} \downarrow_{\text{dom}(\mathbb{Q})} =$

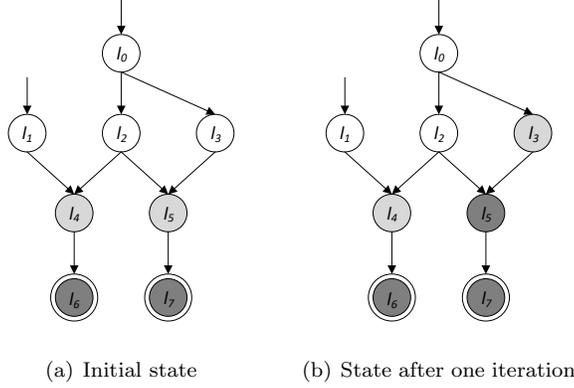


Figure 5: Iterations to compute the weakest precondition.

$\mathbb{Q}$  (where  $\downarrow_D$  is the restriction of a function to the domain  $D$  and  $\bar{D}$  is set complement). Once  $\mathbb{H}$  includes all entry points of the program, the weakest precondition is obtained by the restriction  $\mathbb{H} \downarrow_{\text{dom}(\mathbb{P})}$ . For each iteration, the algorithm uses the following rule:

$$\frac{\{\mathbb{H}\}pr\{\mathbb{Q}\} \wedge \{l \mapsto P\}pr\{\mathbb{Q}\}}{\{\mathbb{H} + \{l \mapsto P\}\}pr\{\mathbb{Q}\}}$$

It (1) selects the label  $l$  of a BIR block  $bl$  for which the domain of  $\mathbb{H}$  contains the labels of all successors (i.e. the weakest precondition of the successors have already been computed); (2) computes the weakest precondition  $P$  for the label  $l$ , demonstrating  $\{l \mapsto P\}pr\{\mathbb{Q}\}$ ; (3) extends  $\mathbb{H}$  as  $\mathbb{H} + \{l \mapsto P\}$ . Figure 5 depicts an example of this procedure. The partial function  $\mathbb{H}$  is defined for labels of dark gray nodes, and light gray nodes represent nodes that can be selected by (1). Figure 5a shows the initial state where only the exit nodes are mapped and Figure 5b shows the results of the first iteration if  $l_5$  is selected. The algorithm extends  $\mathbb{H}$  with the weakest precondition for  $l_5$ , making  $l_3$  available for selection in the next iteration. Notice that after the first iteration, we cannot directly compute the weakest precondition of  $l_2$ , since we do not know the weakest precondition of the child node labeled  $l_4$ .

To compute the weakest precondition of a block compositionally, the definition of triples is lift to smaller elements of BIR, i.e. the execution of single blocks and single statements:

**Definition 2** *Let  $bl$  be a block having label  $l$ ,  $bst$  be a statement, and  $cfst$  be a control flow statement:*

- $\{P\}bl\{\mathbb{Q}\}$  holds iff for every  $(env, l)$  if  $eval(P, env)$  and  $[bl] : (env, l) \rightsquigarrow (env', p')$  then  $p' \in \text{dom}(\mathbb{Q}) \wedge eval(\mathbb{Q}(p'), env')$

- $\{P\}bst\{Q\}$  holds iff for every env if eval  $(P, env)$  and  $bst : env \rightarrow env'$  then  $env' \neq \perp$  and eval  $(Q, env')$
- $\{P\}cfst\{Q\}$  holds iff for every env if eval  $(P, env)$  and  $cfst : env \rightarrow p'$  then  $p' \in dom(Q)$  and eval  $(Q(p'), env)$

To compute the weakest precondition for the label  $l$  (i.e.  $\{l \mapsto P\}pr\{Q\}$ ) the proof producing procedure uses the following rules:

$$\frac{\{\mathbb{H}\}pr\{Q\} \wedge pr[l] = bl \wedge \{P\}bl\{\mathbb{H}\}}{\{l \mapsto P\}pr\{Q\}}$$

$$\frac{\{P_{n+1}\}cfst\{\mathbb{H}\} \wedge \bigwedge_{i \in n \dots 1} (\{P_i\}bst_i\{P_{i+1}\})}{\{P_1\}([bst_1, \dots, bst_n], cfst)\{\mathbb{H}\}}$$

Assuming that  $\mathbb{H}$  is defined for every child of  $l$  and that the program block having label  $l$  is  $bl$ , the weakest precondition of  $l$  can be computed by propagating the preconditions  $\mathbb{H}$  through the block, by computing the weakest precondition of the execution of  $bl$  (i.e.  $\{P\}bl\{\mathbb{H}\}$ ). As usual for sequential composition, the block precondition is computed by propagating the postcondition backwards.

Notice that, differently than blocks and control flow statements, internal block statements always have one successor, therefore their postcondition is a boolean expression instead of a partial function. The rules for the block statements **assign** and **assert** are standard (we use  $\{exp/v\}Q$  to represent the substitution of every occurrence of the variable name  $v$  in  $Q$  with the expression  $exp$ ):

$$\{\{exp/v\}Q\} \mathbf{assign}(v, exp) \{Q\} \quad \{exp \wedge Q\} \mathbf{assert}(exp) \{Q\}$$

The rules for the control flow statements **jump** and **cjump** are standard for unstructured programs. Both rules access  $\mathbb{H}$  for the successor labels, which requires to identify the CFG of the program statically (e.g. the program fragment is free of indirect jumps), and to have already computed the preconditions for the successor labels ( $\mathbb{H}(l)$ ,  $\mathbb{H}(l_1)$ , and  $\mathbb{H}(l_2)$ ).

$$\{\mathbb{H}(l)\} \mathbf{jump}(l) \{\mathbb{H}\} \quad \{exp \Rightarrow \mathbb{H}(l_1) \wedge (\neg exp) \Rightarrow \mathbb{H}(l_2)\} \mathbf{cjump}(exp, l_1, l_2) \{\mathbb{H}\}$$

Steps (2) and (3) of the weakest precondition procedure have been formally verified, by demonstrating soundness of the rules. Step (1) is a proof producing procedure, which dynamically demonstrates for each iteration that the selection of  $l$  is correct (i.e. that  $\mathbb{H}$  contains all successors of  $l$ ). This frees us from verifying that the algorithm to select  $l$  and compute the CFG graph is correct, enabling to integrate heuristics to handle indirect jumps, which can be incomplete and difficult to verify. Also, our procedure only demonstrates soundness of the generated precondition, but does not prove that it is the weakest one. This task requires to dynamically build a counterexample showing that any weaker condition is not a precondition. The lack of this proof does not affect the usage of the tools, since it is not needed for contract based verification.

## 5.2. Optimization

Weakest precondition propagation has two well known scalability issues. Firstly, branches in the CFG can cause an exponential blowup. For instance, in Figure 5 the precondition of node  $l_5$  is propagated twice (via  $l_2$  and  $l_3$ ) and occurs twice as sub-expression of the precondition of the branch node  $l_0$ . There exist approaches [27] to handle this problem, however they generate preconditions that are difficult to handle with SMT solvers, which can preclude their usage for practical contract verification. The second problem, which Section 7 demonstrates to be critical for our scenario, is that expression substitutions introduced by assignments can exponentially increase the size of preconditions. Consider the following program fragment:

```

assign ( $Y$ , var  $X + \mathbf{var}$   $X$ )
assign ( $Z$ , var  $Y + \mathbf{var}$   $Y$ )

```

The weakest precondition of  $Q$  is  $\{X + X/Y\}(\{Y + Y/Z\}Q)$ . This equals  $\{(X + X) + (X + X)/Z\}Q$  when external substitutions are expanded. This behavior is common in BIR programs that model binary programs, due to the presence of indirect loads and stores. For example, in the following fragment

```

assign ( $MEM$ , store (var  $MEM$ , var  $SP + 4$ , var  $R3$ ,  $\tau_{reg,n}$ ))
assign ( $R1$ , load (var  $MEM$ ,  $addr_1$ ,  $\tau_{reg,n}$ ))
assign ( $R2$ , load (var  $MEM$ ,  $addr_2$ ,  $\tau_{reg,n}$ ))
assign ( $MEM$ , store (var  $MEM$ , var  $R1$ , var  $R2$ ,  $\tau_{reg,n}$ ))

```

both  $R1$  and  $R2$  are loaded from  $MEM$ , leading the expression modeling the new value of the variable  $MEM$  to contain three occurrences of  $\mathbf{var} SP + 4$ .

A solution to this issue is using single dynamic assignment and passification. The program is transformed into an “equivalent” one, which ensures that each variable is only assigned once for every possible execution path. Then a second transformation generates a program that has assumptions in place of assignment. However, this approach requires proof-producing transformers and additional machinery to transfer properties from the the passified program to the original one.

For the scope of contract based verification, we can obtain the same conditions without applying these transformations. Our goal is to generate a precondition  $P'$  and to check that for every BIR state this is entailed by the contract precondition  $P$ , i.e. checking that  $P \Rightarrow P'$  is a tautology (we write this as  $P \vdash P'$ ). This drives our strategy: (1) we generate a weakest precondition  $P'$  that contains substitutions, but we do not expand them to prevent the exponential growth of  $P'$ ; (2) we take the precondition  $P$  and we generate a substitution free condition  $P''$  using a proof producing procedure, which ensures that  $P \vdash P'$  if and only if  $P \vdash P''$ ; (3) the original contract is verified by checking unsatisfiability of  $\neg(P \Rightarrow P'')$ .

For step (2) we developed a proof producing procedure that uses a set of preproved inference rules. The rules capture all syntactic forms which can be produced by the weakest precondition generation, i.e. conjunction, implication

and substitution. If the precondition  $P'$  is a conjunction, we can recursively transform the two conjuncts under the common premise  $P$  and combine the transformed preconditions:

$$\frac{P \vdash A \iff P \vdash A' \quad P \vdash B \iff P \vdash B'}{P \vdash (A \wedge B) \iff P \vdash (A' \wedge B')}$$

If  $P'$  is the implication  $A \Rightarrow B$ , we can include  $A$  in the premise and transform  $B$  recursively. Then we can restore the original implication form using the transformed predicate  $B'$ :

$$\frac{(P \wedge A) \vdash B \iff (P \wedge A) \vdash B'}{P \vdash (A \Rightarrow B) \iff P \vdash (A \Rightarrow B')}$$

If  $P'$  is the substitution  $\{E/v\}A$  and  $v$  is free in  $A$ , we apply the following rule.

$$\frac{v \in fv(A) \quad v' \notin (fv(P) \cup fv(E) \cup fv(A)) \quad v' \notin bv(A)}{P \vdash \{E/v\}A \iff P \vdash ((v' = E) \Rightarrow \{v'/v\}A)}$$

This rule prevents the blowup by avoiding applying the substitution  $\{E/v\}A$  and instead using the fresh variable  $v'$  as an abbreviation for  $E$  in  $\{v'/v\}A$ . Before continuing the application of transformation rules, we have to remove the substitution  $\{v'/v\}A$ . This ensures that each application of this transformation rule removes one substitution from  $P'$ . We achieve this by recursively applying the substitution  $\{v'/v\}$  until we reach another substitution  $\{E/v''\}B$ . Normally, the application of substitution would require the application of the inner substitution first, which would reintroduce the blowup problem. Instead, we rewrite the expression as follows:

$$\{v'/v\}(\{E/v''\}B) = \{(\{v'/v\}E)/v''\} \begin{cases} B, & \text{if } v = v'' \\ \{v'/v\}B & \text{otherwise} \end{cases}$$

This moves the inner substitution out by individually applying the substitution  $\{v'/v\}$  to every free occurrence of  $v$  in  $E$  and  $B$ . This means that the substitution should not be applied to  $B$  if  $v = v''$ .

Notice that the last tautology transformation rule cannot be applied if  $v \notin fv(A)$ , since it can introduce type errors. Consider the predicate substitution  $\{(x+1)/y\}(x = true)$ . Here, the two references to  $x$  have different types, i.e. integer and boolean. By using this rule and applying all substitutions, we would obtain  $P \Rightarrow (x = true) \vdash \iff P \vdash (y' = x + 1) \Rightarrow (x = true)$ . However, this equality does not hold, since the left side can be a tautology while the right side cannot, since it is a wrongly typed expression. Practically, if  $v$  is not free in  $A$  then the substitution has no effect and can be simply removed:

$$\frac{v \notin fv(A)}{P \vdash \{E/v\}A \iff P \vdash A}$$

This simplification procedure can be applied to the previous example, where the weakest precondition of  $Q$  is  $\{X + X/Y\}(\{Y + Y/Z\}Q)$ . Let  $P$  be a precondition, the simplification prevents the four repeated occurrences of the variable  $X$  in the final substitution:

$$\begin{aligned}
& P \vdash \{X + X/Y\}(\{Y + Y/Z\}Q) \\
\iff & P \vdash (y' = X + X) \Rightarrow \{y'/Y\}(\{Y + Y/Z\}Q) \\
& \quad \text{where } y' \notin fv(Q) \\
\iff & P \vdash (y' = X + X) \Rightarrow \{y' + y'/Z\}(\{y'/Y\}Q) \\
\iff & P \vdash (y' = X + X) \Rightarrow (z' = y' + y') \Rightarrow \{z'/Z\}(\{y'/Y\}Q) \\
& \quad \text{where } z' \notin fv(Q)
\end{aligned}$$

## 6. Applications

The TrABin’s verification work flow consists of three tasks: (1) transpile a binary program to BIR, (2) proving that the BIR program does not reach error states, (3) proving that the desired properties of the BIR program hold, and (4) using the refinement relation to transfer these properties to the original binary program.

Task (2) can be done following the strategy of Section 5. Let  $\mathcal{P}$  be the partial function whose domain is the program entry points and values are the corresponding preconditions, and let  $\mathbb{Q} = \{l \mapsto true\}$  be the partial function whose domain is the exit points of the program which all map to the constant true, error freedom of program  $pr$  can be verified by simply establishing the contract  $\{\mathbb{P}\}pr\{\mathbb{Q}\}$ .

For (3) and (4) we show that the transpiler output and BIR contract verification can be used for four common verification tasks: Control Flow Graph (CFG) analysis, contract-based verification, partial correctness refinement, and verification of termination.

Knowing the CFG of a program is essential to many compiler optimizations and static analysis tools. Furthermore, proving control flow integrity ensures resiliency against return-oriented programming [28] and jump-oriented programming attacks [29]. In its simplest form, the CFG consists of a directed connected graph  $G$ , whose node set is  $\mathbb{B}^{64}$ . The graph  $G$  contains  $(ad_1, ad_2)$  if the program can flow from the address  $ad_1$  to the address  $ad_2$  by executing a single instruction; The nodes  $EN \subseteq G$  represent the entry points of the program, which can be multiple due to binary programs being unstructured.

Analyzing the CFG of a binary program requires to deal with indirect jumps. Even if the source program avoids using function pointers, indirect jumps are introduced by the compiler, e.g. to handle function exits and exceptions (for example the ARM link register is used to track the return address of functions and can be pushed to and popped from the stack). For this reason, the correctness of the control flow depends on the integrity of the stack itself. Thus, verifying the CFG  $G$  of a program  $BIN_{pr}$  requires assuming a precondition  $P_{ad}$  for every entry point  $ad \in EN$ , which constraints the content of the heap, stack and registers.

**Definition 3 (Control flow graph integrity)** For every machine state  $s$  such that  $\text{stored}(s, \text{BIN}_{pr})$ ,  $s.\text{sr.pc} \in EN$ , and  $P_{s.\text{sr.pc}}(s)$ , for every  $n$ , if  $s \rightarrow^n s_1$  and  $s_1 \rightarrow s_2$  then  $(s_1.\text{sr.pc}, s_2.\text{sr.pc}) \in G$ .

It is straightforward to show that CFG integrity can be verified by using the transpiler theorem, by defining a BIR precondition  $P'$  that corresponds to  $P$ , and by proving the following verification conditions.

**Verification Condition 1 (BIR control flow integrity)** For every  $(\text{env}, p)$  such that  $p \in EN$ ,  $\text{eval}(P', \text{env})$  and for every  $n$  if  $\text{pr} : (\text{env}, p) \rightsquigarrow_{LS_1}^n (\text{env}_1, p_1) \rightsquigarrow_{LS_1} (\text{env}_2, p_2)$ , then  $p_1 \neq \perp$ ,  $p_2 \neq \perp$ , and  $(p_1, p_2) \in G$ .

**Verification Condition 2 (Transfer of precondition)** For every  $bs$  and  $s$  such that  $bs \sim s$ , if  $P(s)$  then  $\text{eval}(P', bs)$ .

Contract based verification for a binary program consists of verifying that a program  $\text{BIN}_{pr}$  meets the contract  $\{\mathbb{P}\}\text{BIN}_{pr}\{\mathbb{Q}\}$ , when its executions start at an entry point  $ad \in \text{dom}(\mathbb{P})$  and end at one of the exit points  $ad' \in \text{dom}(\mathbb{Q})$ .

**Definition 4 (Contract verification)** For every  $s$  and  $n$  such that  $\text{stored}(s, \text{BIN}_{pr})$ ,  $s.\text{sr.pc} \in \text{dom}(\mathbb{P})$  and  $\mathbb{P}(s.\text{sr.pc})(s)$ , if  $s \rightarrow^n s'$  and  $s'.\text{sr.pc} \in \text{dom}(\mathbb{Q})$  then  $\mathbb{Q}(s'.\text{sr.pc})(s, s')$ .

This property can be verified using the theorem produced by the transpiler, by identifying BIR predicates  $(P', Q')$  for every  $(P, Q)$ , establishing the BIR contract  $\{\mathbb{P}'\}\text{pr}\{\mathbb{Q}'\}$ , and by proving the following verification condition:

**Verification Condition 3 (Transfer of contracts)** For every  $bs, bs', s, s', ad$  such that  $bs \sim s$  and  $bs' \sim s'$ , if  $\mathbb{P}(ad)(s)$  then  $\text{eval}(\mathbb{P}'(ad), bs)$  and if  $\text{eval}(\mathbb{Q}'(ad), bs')$  then  $\mathbb{Q}(ad)(s, s')$ .

Partial correctness is proved as a refinement of an abstract specification and by using contract verification. With composability of specifications in mind, we assume that the specification is phrased such that domain and codomain are the same. Let  $a_{out} = f_{spec}(a_{in})$  be a functional specification with the signature  $f_{spec} : A \rightarrow A$ .

**Definition 5 (Partial correctness - refinement)** For every  $s, a, n$  such that  $\text{stored}(s, \text{BIN}_{pr})$ ,  $s.\text{sr.pc} \in EN$ ,  $R(s, a)$ , if  $s \rightarrow^n s'$ , and  $s'.\text{sr.pc} \in EX$  then  $R(s', f_{spec}(a))$ .

Notice, that the refinement relation  $R(s, a)$  implicitly contains the mapping from  $a$  to  $s$  and an invariant that permits to preserve the refinement. Starting from  $R$  and  $f_{spec}$  we can derive a verification condition suitable for contract-based verification. The precondition  $P(s)$  is the invariant of the refinement relation; the postcondition  $Q(s, s')$  incorporates the functional specification  $f_{spec}$  with respect to the mapping of  $R$ .

Total correctness (or functional correctness) additionally requires termination:

**Definition 6 (Termination verification)** For every  $s$  such that  $\text{stored}(s, \text{BIN}_{pr})$ ,  $s.\text{sr.pc} \in \text{dom}(\mathbb{P})$  and  $\mathbb{P}(s.\text{sr.pc})(s)$ , exists  $n$  such that  $s \rightarrow^n s'$  and  $s'.\text{sr.pc} \in \text{EX}$ .

To prove this property, we use the theorem produced by the transpiler (i.e. the second clause of Theorem 1), identify an appropriate BIR precondition  $\mathbb{P}'$ , and prove Condition 2 and the following one:

**Verification Condition 4 (BIR termination verification)** For every  $bs$  such that  $bs.p \in \text{dom}(\mathbb{P}')$  and  $\text{eval}(\mathbb{P}'(bs.p), bs)$  exists  $n$  such that  $pr : bs \rightsquigarrow_{LS_1}^n bs'$  and  $bs'.p \in \text{EX}$ .

## 7. Evaluation

Our contribution counts  $\sim 33000$  lines of HOL4 code:  $\sim 3000$  lines for the model and semantics of BIR;  $\sim 2000$  lines for supporting tools, which includes the static type checker;  $\sim 10000$  lines for helper theorems, which includes validation of emulation of bitvector operators using primitive BIR operators and support for the weak transition relation;  $\sim 10000$  lines for the transpiler, which includes preproved theorems for computing the effects of template-blocks and compose them;  $\sim 2000$  lines of architecture-dependent proofs to handle peculiarities of CortexM0 and ARMv8;  $\sim 2000$  lines for the weakest precondition predicate transformer; and  $\sim 5000$  lines for the precondition simplifier.

### 7.1. Transpiler benchmarks

A large part of the proof engineering efforts focused on proving the architecture independent transpiler theorems which enable to reduce the run-time cost of establishing the instruction-theorem. This permits to translate (on a modern mobile Intel CPU) in average three instructions per seconds.

We experimented with various unmodified binary programs produced by standard compilers and using standard optimizations:

- An embedded SSL library WolfSSL for both ARMv8 and CortexM0:
  - The numlib used to implement asymmetric encryption
  - The modules for md5, sha, hmac, pkcs7, elliptic curve, des3, AES, RSA, and their dependencies
- The run-time of the embedded real-time operating system FreeRTOS for CortexM0
- Several binaries extracted from Ubuntu 18.04 for ARMv8
  - The embedded database SQLite
  - The interpreter of the high level language lua
  - The libc part of the standard run-time

	Instructions	Transpiling	Merging	Total	Rate
CortexM0 - numlib	9605	684 s	222 s	906 s	10.61 i/s
CortexM0 - crypto	21097	1245 s	1276 s	2521 s	8.37 i/s
CortexM0 - RTOS	6292	597 s	118 s	739 s	8.51 i/s
ARMv8 - numlib	5737	853 s	131 s	984 s	5.83 i/s
ARMv8 - crypto	13149	1808 s	643 s	2451 s	5.37 i/s
ARMv8 - lua	37026	10917 s	6614 s	17531 s	2.11 i/s
ARMv8 - SQLite	61134	23734 s	16470 s	40205 s	1.52 i/s
ARMv8 - libc	23362	6264 s	2425 s	8690 s	2.69 i/s
ARMv8 - vim	490398	40 h	time out		

Table 2: Transpilation benchmark.

	J	CJ	S	LB	O	LO
CortexM0 - numlib	8538	1067	33155	11	121682	71
CortexM0 - crypto	19449	1648	72396	11	269499	71
CortexM0 - RTOS	5605	654	21040	11	80404	71
ARMv8 - numlib	4972	765	14337	7	59284	85
ARMv8 - crypto	11879	1270	32009	7	128307	85
ARMv8 - lua	33429	3209	89840	7	321726	121
ARMv8 - SQLite	54086	6372	144367	7	521370	149
ARMv8 - libc	20069	2825	56912	7	226264	379

Table 3: Size of produced BIR programs. (J) number of jump statements; (CJ) number of conditional jump statements; (S) total number of statements; (LS) number of statements of the largest block; (O) total number of operators in the program expressions; (LO) total number of operators in expressions in the largest block.

– The general purpose application vim

The transpilation consists of two steps: the *transpiling step* that translates each instruction independently and produces *BIR* code and certificates; the *merging step* that composes the certificates to derive a single theorem for the entire program. The performance of the transpiler is reported in Table 2. The merging procedure is designed to translate single procedures and to enable their modular analysis. For this reason, it is not optimized to handle monolithic large binary blobs and the percentage of time spent in the merging step increases with the number of instructions. Despite this, the merging can handle binary blobs consisting of more than 50000 instructions. On the other hand, the time needed to transpile single instructions is independent of the size of the program. In Table 3 we report metrics regarding resulting BIR programs.

The transpiler relies on the external HOL4 model of the architecture. Therefore, it does not translate instructions that are not supported by the external model. For ARMv8, the model does not support floating point operations.

	Instructions	Unsupported	Supported
CortexM0 - numlib	9605	0	100.0%
CortexM0 - crypto	21097	0	100.0%
CortexM0 - RTOS	6292	33	99.47%
ARMv8 - numlib	5737	0	100.0%
ARMv8 - crypto	11879	0	100.0%
ARMv8 - lua	37026	1161	99.74%
ARMv8 - SQLite	61134	2028	98.68%
ARMv8 - libc	23362	1404	98.11%
ARMv8 - vim	490398	840	99.99%

Table 4: Frequency of unsupported instructions.

Frequency of these instructions are reported in Table 4.

In order to optimize transpilation, instruction theorems are cached. This permits to reduce the time needed for re-transpiling instructions. Clearly large programs benefit more from the caching mechanism. Table 5 reports cache hit for three programs. We split the data for each program in four fragments of equal size, in order to show that the frequency of cache hits increases while cache is filled with more instructions. It is worth noticing that cache hits are always less than 75%. This is connected with the fact that even if instructions are repeated, they occur with different arguments. For example both `add x1, x1, #0x9d8` and `add x1, x1, #0x9f8` add a constant to the same register (e.g. to set two different offsets on the stack), but these two instructions have different encoding (since the encoding includes the constants): `0x91276021` and `0x9127E021`. The design of the transpiler is oblivious to the decoding of a particular ISA. This prevents us to detect that these two instructions represent the same operation with different constants. On the other hand, this design allows the tool to be easily extended to support new architectures, since it delegates the inspection of the binary code to the step theorem of the external model.

## 7.2. Weakest Precondition benchmarks

For evaluating the weakest precondition procedures, we used a selection of the binaries transpiled: the numlib and the crypto library for both CortexM0 and ARMv8. The weakest precondition procedure is automatic only for loop-free programs, for this reason we extracted loop free fragments using an automatic procedure. Due to the time needed to execute the weakest precondition procedure, we limit the size of the extracted fragments to 150 blocks. We also report the benchmarks for the complete implementation of AES, which is part of the crypto library and is considerably larger than the limit imposed to the automatic extractor.

For our experiments we use the postcondition:  $Q = true$ . Even if minimal, establishing it is not trivial, since the weakest precondition entails all intermediate assertions generated by the transpiler: i.e. there is no memory error

Subset	Cache hits	Percentage of hits
lua (1/4)	5084	54.92%
lua (2/4)	5626	60.77%
lua (3/4)	6688	72.24%
lua (4/4)	6647	71.82%
SQLite (1/4)	8064	52.76%
SQLite (2/4)	10035	65.65%
SQLite (3/4)	9651	63.14%
SQLite (4/4)	10043	65.71%
libc (1/4)	1990	34.06%
libc (2/4)	2794	47.83%
libc (3/4)	3002	51.39%
libc (4/4)	3186	54.56%

Table 5: Cached instructions in ARMv8 binaries.

	fragments	avg. size	avg. time	max. size	time
CortexM0 - AES	3	177	372 s	269	568 s
CortexM0 - numlib	189	8	7 s	96	150 s
CortexM0 - crypto	299	10	16 s	131	529 s
ARMv8 - AES	2	268	494 s	282	511 s
ARMv8 - numlib	151	10	7 s	106	150 s
ARMv8 - crypto	245	10	8 s	72	109 s

Table 6: Benchmarks for weakest precondition.

that can lead to code injection and all memory accesses are correctly aligned (which is required to compute the step theorems). Therefore, establishing the postcondition *true* rules out the error states from transpilation theorems. Table 6 reports the number of fragments extracted, their average number of BIR blocks, the size of the largest block, and the time (average and maximum) to compute the weakest precondition. The average time to propagate a weakest precondition for one block is 1.2 seconds. This time refers to the first phase of the procedure, and does not include application of the optimization procedure and the substitutions.

Comparing the optimization procedure with respect to the naive expansion of the substitution is possible only for relatively small fragments, because the time required by the naive approach increases quickly.

We compared the two approaches for the last 70 instructions of the AES encryption procedure. In this case, the weakest precondition consists of 1170 BIR operators and includes 70 substitutions for ARMv8; and 1576 BIR operators and 133 substitutions for CortexM0. Naively applying the substitutions for the ARMv8 expression requires 49.0 minutes and produces a weakest precondition consisting of 7 million BIR operators. For CortexM0, the same approach requires 7.5 hours and produces an expression of 25 million BIR operators. The execution of our optimization procedure requires 1.7 minutes and produces an expression with 1377 BIR operators for ARMv8. For CortexM0 this takes 2.3 minutes and produces an expression with 1335 BIR operators.

Once the weakest precondition is computed, it can be verified to be entailed by the program precondition. In the case of the AES encryption procedure, the precondition constrains the stack to be separated from the program memory, and the function argument (i.e. the content of the stack containing the pointers to the block to be encrypted and the encryption key) to not overlap with the stack and the program memory. The validity of the tautology is checked using the external SMT solver Z3. We could not use the existing HOL4 Z3 export that provides proof reconstruction, since it cannot handle the theory of arrays, which is necessary for modeling memory loads and stores. For this reason, we developed a small untrusted export that supports the BIR operators.

## 8. Concluding remarks

We presented the main building blocks of TrABin, a platform for trustworthy analyses of binary code. These consist of the HOL4 formal model of the intermediate language BIR, the implementation of a transpiler for binary programs, and a weakest precondition predicate transformer. TrABin overcomes two of the main barriers in adopting binary analysis platforms to formally verifying binary code: the need for trusting translation soundness and the lack of a formal ground for correctness of the analyses.

We demonstrated the proof producing transpiler for two common architectures: ARMv8 and CortexM0. To handle other machine architectures (e.g. x86, x64, ARMv7, MIPS, RISC-V), new transpilers must be developed. Fortunately, among 12000 lines of the transpiler, only 2000 are architecture specific, which

permits to easily adapt the translation to support existing (and upcoming) HOL4 ISA models that are equipped with the step function.

The proof producing tool to generate weakest preconditions demonstrates that the platform can be used to create trustworthy analyses and is a key component for a trustworthy semi-automatic verification tool based on pre/post conditions for binary code. Such tools can be developed by developing a sound satisfiability solver for bitvectors to check if the precondition entails the weakest precondition. Böhme et al. [30] demonstrated HOL4 proof reconstruction for Z3 [31] capable of handling the theory of fixed-size bit-vectors. However, the current implementation lacks support for arrays, which are needed to handle memory loads and stores. Also, to make the use of the verification tool practical, some supporting analyses are needed, like loop unrolling and heuristics for indirect jump resolution. Fortunately, these analyses can be build on top of contract based verification, which allows us to reuse the existing infrastructure to prove correctness of their results.

An ongoing research activity is extending the BIR language and the supporting tools to handle non-functional aspects of hardware architectures, for instance to represent cache accesses performed by a binary program. This can enable the development of trustworthy static analysis of side channels, including timing and power consumption, in the style of CacheAudit [24].

### TrABin artifacts

The source code of the analysis platform, including the BIR model, the transpiler, the weakest precondition generator, and the binaries used for benchmarking, are available at <https://github.com/andreaslindner/HolBA>.

### Acknowledgments

We warmly thank Thomas Tuerk for his key contributions to build up the foundation of the binary analysis platform reported in this paper. This work has been supported by the TrustFull project financed by the Swedish Foundation for Strategic Research and by the KTH CERCES Center for Resilient Critical Infrastructures financed by the Swedish Civil Contingencies Agency.

### References

#### References

- [1] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, P. Saxena, BitBlaze: A new approach to computer security via binary analysis, in: International Conference on Information Systems Security, Springer, 2008, pp. 1–25.
- [2] D. Brumley, I. Jager, T. Avgerinos, E. J. Schwartz, BAP: A Binary Analysis Platform, in: Computer Aided Verification, Springer, 2011, pp. 463–469.

- [3] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, G. Vigna, SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis, in: Symposium on Security and Privacy, IEEE, 2016, pp. 138–157.
- [4] N. Nethercote, J. Seward, Valgrind: A program supervision framework, *Electronic notes in theoretical computer science* 89 (2) (2003) 44–66.
- [5] K. Y. Rozier, M. Y. Vardi, LTL satisfiability checking, *International journal on software tools for technology transfer* 12 (2) (2010) 123–137.
- [6] A. Fox, L3: A Specification Language for Instruction Set Architectures (Retrieved on 2015).  
URL <http://www.cl.cam.ac.uk/~acjf3/l3/>
- [7] A. Fox, Directions in ISA Specification, in: *Interactive Theorem Proving*, Springer, 2012, pp. 338–344.
- [8] A. Ltd., ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile) (2013).  
URL <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.h/index.html>
- [9] A. Ltd., ARMv6-M Architecture Reference Manual (2017).  
URL <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0419c/index.html>
- [10] R. Metere, A. Lindner, R. Guanciale, Sound transpilation from binary to machine-independent code, in: *Brazilian Symposium on Formal Methods*, Springer, 2017, pp. 197–214.
- [11] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al., seL4: Formal verification of an OS kernel, in: *Operating systems principles*, ACM, 2009, pp. 207–220.
- [12] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. W. Schirmer, A. Starostin, The Verisoft approach to systems verification, in: *Verified Software: Theories, Tools, and Experiments*, Springer, 2008, pp. 209–224.
- [13] M. Dam, R. Guanciale, H. Nemati, Machine code verification of a tiny ARM hypervisor, in: *Workshop on Trustworthy Embedded Devices, Co-located with CCS*, ACM, 2013, pp. 3–12.
- [14] X. Leroy, Formal verification of a realistic compiler, *Communications of the ACM* 52 (7) (2009) 107–115.
- [15] S. Boldo, J. Jourdan, X. Leroy, G. Melquiond, A Formally-Verified C Compiler Supporting Floating-Point Arithmetic, in: *Symposium on Computer Arithmetic*, IEEE, 2013, pp. 107–115.

- [16] R. Kumar, M. O. Myreen, M. Norrish, S. Owens, CakeML: a verified implementation of ML, in: SIGPLAN Notices, Vol. 49, ACM, 2014, pp. 179–191.
- [17] G. Li, S. Owens, K. Slind, Structure of a proof-producing compiler for a subset of higher order logic, in: European Symposium on Programming, Springer, 2007, pp. 205–219.
- [18] L. Beringer, A. Petcher, Q. Y. Katherine, A. W. Appel, Verified Correctness and Security of OpenSSL HMAC., in: USENIX Security Symposium, 2015, pp. 207–221.
- [19] A. C. Fox, M. J. Gordon, M. O. Myreen, Specification and verification of ARM hardware and software, in: Design and verification of microprocessor systems for high-assurance applications, Springer, 2010, pp. 221–247.
- [20] M. O. Myreen, M. J. C. Gordon, K. Slind, Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic, in: Formal Methods in Computer-Aided Design, IEEE Press, 2008, pp. 1–8.
- [21] T. A. L. Sewell, M. O. Myreen, G. Klein, Translation validation for a verified OS kernel, in: SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2013, pp. 471–482.
- [22] M. O. Myreen, G. Curello, Proof Pearl: A Verified Bignum Implementation in x86-64 Machine Code, in: Certified Programs and Proofs - Third International Conference, Springer, 2013, pp. 66–81.
- [23] M. Balliu, M. Dam, R. Guanciale, Automating information flow analysis of low level code, in: SIGSAC Conference on Computer and Communications Security, ACM, 2014, pp. 1080–1091.
- [24] G. Doychev, B. Köpf, L. Mauborgne, J. Reineke, Cacheaudit: A tool for the static analysis of cache side channels, ACM Transactions on Information and System Security (TISSEC) 18 (1) (2015) 4.
- [25] N. Hasabnis, R. Sekar, Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers, ACM SIGOPS Operating Systems Review 50 (2) (2016) 311–324.
- [26] F. Vogels, B. Jacobs, F. Piessens, A machine-checked soundness proof for an efficient verification condition generator, in: Symposium on Applied Computing, ACM, 2010, pp. 2517–2522.
- [27] K. R. M. Leino, Efficient weakest preconditions, Information Processing Letters 93 (6) (2005) 281–288.
- [28] H. Shacham, The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86), in: Conference on Computer and Communications Security, ACM, 2007, pp. 552–561.

- [29] T. Bletsch, X. Jiang, V. W. Freeh, Z. Liang, Jump-oriented programming: a new class of code-reuse attack, in: *Symposium on Information, Computer and Communications Security*, ACM, 2011, pp. 30–40.
- [30] S. Böhme, A. C. Fox, T. Sewell, T. Weber, Reconstruction of Z3’s Bit-Vector Proofs in HOL4 and Isabelle/HOL., in: *Certified Programs and Proofs: First International Conference*, Springer, 2011, pp. 183–198.
- [31] L. M. de Moura, N. Björner, Z3: An Efficient SMT Solver, in: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.