Experiments on automation of formal verification of devices at the binary level

THOMAS LACROIX

Master in Computer Science
Date: 30 mai 2019
Supervisor: Mads Dam
Examiner: TODO
Computer Science department - INSA Lyon
Host company: Department of Theoretical Computer Science - KTH

Résumé

With the advent of virtualization, more and more work is put into the verification of hypervisors. Being low level softwares, such verification should preferably be performed at binary level. Binary analysis platforms are being developed to help perform these proofs, but a lot of the work has to be carried out manually.

In this thesis, we focus on the formal verification of a Network Interface Controller (NIC), more specifically we look at how to automate and reduce the boilerplate work from an existing proof. We base our work on the HolBA platform, its hardware-independent intermediate representation language BIR and supporting tools, and we experiment on how to perform this proof by leveraging existing tools.

We first replaced the existing NIC model written in HOL4 to an equivalent one written using BIR, enabling the use of HolBA tools. Secondly, we developed some visualization tools to help navigate and gain some insight in the existing proof and its structure. Thirdly, we experimented with the use of Hoare triples in conjunction with an SMT solver to perform contract verification. Finally, we proved a simple contract written in terms of the formal NIC model on the BIR implementation of this model, unlocking the way of performing more complex proofs using the HolBA platform.

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

Résumé

Avec la démocratisation de la virtualisation, de plus en plus d'efforts sont consacrés à la vérification des hyperviseurs. S'agissant de logiciels de bas niveau, une telle vérification devrait de préférence être effectuée au niveau binaire. Des plates-formes d'analyse binaire sont en cours de développement pour aider à réaliser ces preuves, mais une grande partie du travail doit encore être effectuée manuellement.

Dans cette thèse, nous nous concentrons sur la vérification formelle d'un Contrôleur d'Interface Réseau (NIC), plus spécifiquement sur la manière d'automatiser et de réduire le travial d'une preuve existante. Nous nous basons sur la plate-forme HolBA, son langage de représentation intermédiaire indépendant du matériel, BIR et ses outils de support, et nous nous intéressons à la manière de réaliser cette preuve en utilisant des outils existants.

Nous avons d'abord remplacé le modèle NIC existant écrit en HOL4 par un modèle équivalent écrit en BIR, permettant ainsi l'utilisation des outils de HolBA. Deuxièmement, nous avons développé des outils de visualisation pour vous aider à naviguer et à mieux comprendre la preuve existante et sa structure. Troisièmement, nous avons expérimenté l'utilisation des triplets de Hoare en conjonction avec un solveur SMT pour effectuer une vérification par contrat. Enfin, nous avons prouvé un contrat simple écrit en termes du modèle formel du NIC sur l'implémentation de ce modèle en BIR, ouvrant la voie à la réalisation de preuves plus complexes avec la plate-forme HolBA.

Mot-clés : binary analysis, formal verification, proof producing analysis, theorem proving

Todo list

Include :a) Git workflow for a team ;b) LogLib (and tracing in general) ; -> annexc) CI to track regressions + static analysis ;d) Simple interface for CFG lib ;

Table des matières

Chapitre 1

Introduction

This chapter serves as an introduction to the degree project and presents the background of the work along with this thesis objective. Delimitations to the project and the choice of methodology are also discussed.

1.1 Background

1.2 Intended readers

1.3 Thesis objective

1.4 Delimitations

1.5 Choice of methodology

Chapitre 2

Definitions and relevant theory

*This chapter intends to lay the concepts and theory that are essential to the reader in order to understand the problem that this degree project aims to explore. This includes an overview of virtualization and hypervisors, a presentation of Interactive Theorem Proving and formal proofs, relevant theory about Hoare Triple and Weakest Precondition analysis, and an introduction of the **HOL4 Binary Analysis Platform (HolBA)** framework.*

TODO : Define "proof-producing"

Chapitre 3

NIC model

Chapitre 4

Proving properties

4.1 Contract based verification

4.1.1 Hoare triples

Contract based verification is a powerful approach for verifying programs. For a given program $prog$ consisting of a list of instructions and two predicates P and Q called respectively pre- and postcondition, a Hoare triple $\{P\}\ prog\ \{Q\}$ states that when executing the program $prog$ from a state S terminates in a state S', if P holds in S then Q will hold in S' (Equation 4.1). Hereafter, we assume programs and states to be well-typed.

$$\{P\}\ prog\ \{Q\} \triangleq S' = exec(S, prog) \implies P(S) \implies Q(S') \tag{4.1}$$

For example, $\{P\}\ \varnothing\ \{P\}$ holds because an empty program doesn't change the state of the execution. $\{n = 1\}\ n := n + 1\ \{even(n)\}$, with $n \in \mathbb{N}$, holds because $1 + 1 = 2$, which is even.

In order to perform the verification, the Hoare logic introduces a set of axioms describing the effect of each instruction of a given language over the execution state [**hoare_axiomatic_1969**]. For an assignment $x := f$ where x is a variable identifier and f an expression without side-effects, Equation 4.2 defines the axiom of assignment, where $P[f/x]$ denotes the substitution of all occurrences of x by f in P.

$$\{P[f/x]\}\ x := f\ \{P\} \tag{4.2}$$

Introduce labels ? $\{l1 : P\}\ l1 -> \{l2, l3\}\ \{l2 : Q, l3 : Q'\}$

4.1.2 Weakest precondition derivation

While Hoare logic introduces sufficient preconditions, Dijkstra introduced the concept of necessary and sufficient preconditions, called "weakest" preconditions. Such weakest preconditions can be automatically derived from a program $prog$ and a postcondition Q. Let's call $WP(prog, Q)$ such a weakest precondition. Then, from Equation 4.1 follows :

$$\forall (prog, Q), \{\, WP(prog, Q)\}\ prog\ \{Q\} \tag{4.3}$$

For the program $n := n + 1$ mentioned above, we can generate the weakest precondition for the postcondition $even(n)$. First, we can rewrite $even(n)$ as $n\ MOD\ 2 = 0$ with MOD denoting the arithmetic modulo. Then, we derive the weakest precondition of the statement $n := n + 1$ by transforming the predicate $n\ MOD\ 2 = 0$ by substituting all occurrences of n by $n + 1$:

$$WP(\text{“}n := n + 1'', n\ MOD\ 2 = 0) = (n + 1\ MOD\ 2 = 0) \tag{4.4}$$

From the properties of the modulo, we can simplify $n + 1\ MOD\ 2 = 0$ to $n\ MOD\ 2 = 1$ or $odd(n)$. Therefore, $\{odd(n)\}\ n := n + 1\ \{even(n)\}$, i.e. incrementing the value of an odd integer variable by one makes it even.

While the triple $\{n = 1\}\ n := n + 1\ \{even(n)\}$ uses a sufficient precondition for establishing its postcondition, the triple $\{odd(n)\}\ n := n + 1\ \{even(n)\}$ uses the weakest precondition. The later being the weakest precondition of the former, the two contracts are in relation :

$$n = 1 \implies odd(n) \tag{4.5}$$

More generally, for a triple $\{P\}\ prog\ \{Q\}$ to hold, P must be stronger than the weakest precondition, i.e. we need to prove that $P \implies WP(prog, Q)$.

$$(P \implies WP(prog, Q)) \implies \{P\}\ prog\ \{Q\} \tag{4.6}$$

Here and above, should we mention/explain termination ?

4.1.3 Using SMT solvers to prove contracts

From Equation 4.6 we see that, in order to prove that a triple $\{P\}\ prog\ \{Q\}$, we need to prove $P \implies WP(prog, Q)$. While multiple methods exist to perform such proofs, **SMT** solvers offer a convenient and automatic solution.

Satisfiability Modulo Theories (**SMT**) problem is a decision problem for logical formulas with respect to combinations of background theories such as arithmetic, bit-vectors, arrays, and uninterpreted functions [**nikolaj_bjorner_programming_2019**]. **Satisfiability Modulo Theories (SMT)** problem is a generalization of **Boolean SATisfiability Problem (SAT)** problem supporting more theories. When given a formula, a **SMT** solver decides if the formula is satisfiable, i.e. if there exist a valuation of its variables where the formula evaluates to true. As a **SMT** solver can fail to decide a given instance, there are three possible outputs : "satisfiable", "unsatisfiable" and "unknown". Another useful feature of some **SMT** solvers is the ability to ask for a satisfying model, which represents a counter-example of a false predicate.

A predicate P holds if it evaluates to true for all possible values of its variables. Alternatively, the negation of a predicate $\neg P$ holds if there exist no valuation of its variables where the predicate evaluates to true, i.e. if the instance is unsatisfiable. Therefore, if a **SMT** solver report that $\neg P$ is "unsatisfiable", then P holds.

isn't that the definition of tautologies ?

Another way of thinking about how to prove logical formulas with **SMT** solvers is by using De Morgan's Laws : we know that $\neg(P \implies WP) \equiv (P \wedge \neg WP)$. Therefore, proving that $\neg(P \implies WP)$ is "unsatisfiable" using an **SMT** solver can be seen as proving that there exist no model where P holds and WP doesn't.

Getting started with the BitVectors theory

To reason about fixed-size integers, **SMT** solvers often implement a "BitVector", or "FixedSizeBitVectors", theory. In order to understand its particularities, we can try to prove Equation 4.7. Hereafter, we will use Z3, a popular and efficient **SMT** solver implemented by Microsoft Research [1], and SMT-LIB 2.0, which is a standard format for **SMT** solvers. Listing 4.1 shows the SMT-LIB 2.0 representation of this proof attempt.

$$\forall x.\ x + 1 > x, \text{ with } x \text{ an unsigned 32-bit integer} \tag{4.7}$$

1. Z3 is available on GitHub at https://github.com/Z3Prover/z3/.

Listing 4.1 – SMT-LIB 2.0 representation of Equation 4.7.

```
(declare-const x (_ BitVec 32))
(assert (not
  (bvugt (bvadd x (_ bv1 32)) x)))
(check-sat)
(get-model)
```

When given Listing 4.7 as input, Z3 gives the following output :

Listing 4.2 – Z3 output for Listing 4.1.

```
sat
(model (define-fun x () (_ BitVec 32) #xffffffff))
```

Z3 is telling us that Equation 4.7 is false, and gives a counterexample : $x = 2^{32} - 1$. Indeed, with this value of x, $x+1$ wraps around and result in 0 which is smaller that $2^{32} - 1$. This behaviour is due to the bounded nature of fixed-size integers. The correct equation here would be :

$$\forall x.\ x \neq 2^{32} - 1 \implies x + 1 > x, \text{with } x \text{ an unsigned 32-bit integer} \quad (4.8)$$

Listing 4.3 and 4.4 show the input and output of Z3 used to successfully prove Equation 4.8.

Listing 4.3 – SMT-LIB 2.0 representation of Equation 4.8.

```
(declare-const x (_ BitVec 32))
(assert (not
  (bvugt (bvadd x (_ bv1 32)) x)))
(assert (not (= x #xffffffff)))
(check-sat)
```

Listing 4.4 – Z3 output for Listing 4.3.

```
unsat
```

4.1.4 Contract based verification in HolBA

HolBA provides a proof-producing tool for automatically deriving weakest preconditions on loop-free **BIR** programs whose control flow can be statically identified [**lindner_trabin:_2019**]. This tool is proof-producing in that it proves Theorem 4.9 which is the instantiation of Definition 4.10, with $(p,\ entry\_l,\ end\_ls)$ defining the program, wp the derived weakest precondition, $post$ the given postcondition.

$$bir\_exec\_to\_labels\_triple\ prog\ entry\_l\ end\_ls\ \mathbf{wp}\ post \tag{4.9}$$

$$\begin{aligned}
\vdash\ & \forall (prog : \alpha\ bir_program_t)\ (entry_l : bir_label_t)\ (end_ls : bir_label_t \to bool) \\
& (pre : bir_exp_t)\ (post : bir_exp_t). \\
& bir_exec_to_labels_triple\ prog\ entry_l\ end_ls\ pre\ post \Leftrightarrow \\
& \forall\ (s\ : bir_state_t)\ (r\ : \alpha\ bir_execution_result_t). \\
& \qquad bir_env_vars_are_initialised\ s.bst_environ\ (bir_vars_of_program\ prog) \\
& \quad \Longrightarrow\ s.bst_pc.bpc_index\ =\ 0\ \wedge\ s.bst_pc.bpc_label\ =\ entry_l \\
& \quad \Longrightarrow\ s.bst_status\ =\ BST_Running \\
& \quad \Longrightarrow\ bir_is_bool_exp_env\ s.bst_environ\ pre \\
& \quad \Longrightarrow\ bir_eval_exp\ pre\ s.bst_environ\ =\ bir_val_true \\
& \quad \Longrightarrow\ bir_exec_to_labels\ end_ls\ prog\ s\ =\ r \\
& \quad \Longrightarrow\ \exists\ (obs\ : \alpha\ list)\ (step_count\ : num)\ (pc_count\ : num)\ (s'\ : bir_state_t). \\
& \qquad r\ =\ BER_Ended\ obs\ step_count\ pc_count\ s' \\
& \qquad \wedge\ s'.bst_status\ =\ BST_Running \\
& \qquad \wedge\ bir_is_bool_exp_env\ s'.bst_environ\ post \\
& \qquad \wedge\ bir_eval_exp\ post\ s'.bst_environ\ =\ bir_val_true \\
& \qquad \wedge\ s'.bst_pc.bpc_index\ =\ 0\ \wedge\ s'.bst_pc.bpc_label\ \in\ end_ls
\end{aligned} \tag{4.10}$$

It is to be noted that this tool doesn't produce a theorem stating that the generated expression is actually *the* weakest precondition. However, this theorem isn't needed to perform contract-based verification if the generated "weakest" precondition is weak enough so that our precondition can imply it. However, without this theorem it is impossible to prove that a given precondition P isn't strong enough to establish the postcondition. We will still use the term "weakest precondition" as it is in practice how we are using this tool.

Definitions 4.10 introduces additional conditions about well-typedness and initialization that are needed in BIR today [2], as well as the notion of "Block Program Counter" for multi-statement blocks.

This tool doesn't provide a simple interface to compute weakest preconditions for a given program and postcondition, nor does it provide and support for proving the relation between the precondition and the weakest precondition. Then, in order to prove that the Hoare triple holds from this generated Theorem 4.9, we need to prove :

$$bir\_exec\_to\_labels\_triple\ p\ entry\_l\ end\_ls\ \mathbf{pre}\ post \tag{4.11}$$

Assuming well-typedness and initialization, after rewriting the definition of $bir\_exec\_to\_labels\_triple$, we have to show $bir\_eval\_exp\ \mathbf{wp}\ s.bst\_environ\ =\ bir\_val\_true$ in order to prove our goal using the *modus ponens* with Theorem 4.9. This correspond to proving the following implication :

$$\begin{aligned} &bir\_eval\_exp\ \mathbf{pre}\ s.bst\_environ\ =\ bir\_val\_true \\ \Longrightarrow\ &bir\_eval\_exp\ \mathbf{wp}\ s.bst\_environ\ =\ bir\_val\_true \end{aligned} \tag{4.12}$$

In Equation 4.12 we can recognize Equation 4.6 that we discussed how to prove using **SMT** solvers in Section 4.1.3. However, the expressions are expressed as BIR expressions. We then have to find a way to use an SMT solver. This is the focus of the following of this thesis. Section 4.2 will use a non proof-producing method for translating those BIR expressions into an equivalent formula that SMT solvers can work on, then focus on automating the whole verification process. Section 4.3 will complete this proof and use it to lift properties that have been proved on the BIR implementation to the **Network Interface Controller (NIC)** model. The following Section 4.1.5 will discuss how to make proofs about BIR memories using **SMT** solvers.

4.1.5 BIR memories and SMT solvers

HOL4 features a library for interfacing **SMT** solvers and HOL4, called *HolSmtLib*. This library supports Yices 1 and Z3 as external provers. Yices 1 being

2. Removal of the need of initialization is being discussed at the time of the writing, because actual hardware registers and memories are in facts always initialized : https://github.com/kth-step/HolBA/issues/63

an abandoned project that doesn't support SMT-LIB 2.0, we will focus on Z3 and the standard format SMT-LIB 2.0. While *HolSmtLib* supports export for some SMT-LIB 2.0 theories, it doesn't support the *ArraysEx* theory and doesn't know about BIR. In Section 4.1.4, we discussed the translation from BIR expressions to *wordsTheory*. However, this theory doesn't contain anything about memories or arrays in general. Therefore, some modifications are needed.

Cite SMT-LIB 2.0

BIR memories are semantically defined as functions from addresses to values.

There exist five types of BIR expressions operating directly on memories (cf. Listing 4.18 for the list of BIR expressions, and Section 4.2.1 for a more precise discussion of the BIR semantic) :

— `BExp_Den` : this operation enables reading values from the environment. It is analogous to reading registers or the memory in assembly programs. This operation semantically equivalent to free variables that *HolSmtLib* already support.
— `BExp_MemEq` : this operation is the equality binary operation on BIR memories. This operation is semantically equivalent to equality between its operands. *HolSmtLib* already supports this operation.
— `BExp_Store` : this operation is used to represent memory writes. It is semantically defined as successive function updates of consecutive segments of the word being stored, because the length of the memory value-type can be less or equal than the length of values stored in the memory. A function update in *combinTheory* is defined with Definition 4.13. *HolSmtLib* cannot currently export function update operations.
— `BExp_Load` : this operation is used to read from memories. BIR memories are semantically defined as functions from addresses to values. A function application in *combinTheory* is defined with Definition 4.14. Then, a memory load operation is the concatenation of multiple function application of consecutive addresses. *HolSmtLib* supports function application of uninterpreted functions only.

$$\vdash \forall a\ b.\ a =+ b = (\lambda f\ c.\ \text{if } a = c \text{ then } b \text{ else } f\ c) \tag{4.13}$$

$$\vdash \forall x\ f.\ x :> f = f\ x \tag{4.14}$$

We saw in the previous list that we need to implement the support for *combinTheory* function update and application in the *HolSmtLib* SMT-LIB 2.0

exporter. Since we only need two operations on the memory, load and store, the *ArraysEx* theory is a good fit.

SMT-LIB 2.0 [**smtlib**] defines the *ArraysEx* theory using the three following axioms :

Listing 4.5 – SMT-LIB 2.0 axioms of the *ArrayEx* theory.

```
(forall ((a (Array s1 s2)) (i s1) (e s2))
  (= (select (store a i e) i) e))

(forall ((a (Array s1 s2)) (i s1) (j s1) (e s2))
  (=> (distinct i j)
      (= (select (store a i e) j) (select a j))))

(forall ((a (Array s1 s2)) (b (Array s1 s2)))
  (=> (forall ((i s1)) (= (select a i) (select b i)))
      (= a b)))
```

If those axioms hold in *combinTheory* then the translation is sound. *combinTheory*'s *UPDATE_APPLY* theorem in Equation 4.15 is equivalent to the first two theorems, the first and the second conjunct corresponding respectively to the first and the second axiom. The third axiom can be proved using both *combinTheory*'s *APP_def* theorem (Theorem 4.14) and *boolTheory*'s *EQ_EXT* theorem (Theorem 4.16)

$$\begin{aligned} \vdash \quad & \forall a\ x\ f.\ (a =\!+ x)\ f\ a = x \\ & \wedge \forall a\ b\ x\ f.\ a \neq b \implies ((a =\!+ x)\ f\ b = f\ b) \end{aligned} \tag{4.15}$$

$$\vdash \forall f\ g.\ (\forall x.\ f\ x = g\ x) \implies (f = g) \tag{4.16}$$

Since this translation is sound, it has been added in *HolSmtLib*. The translation is direct : from `:>` and =+ to respectively `select` and `store`. Section 4.2.4 presents a test using BIR memories.

4.2 Implementation of a non proof-producing automatic contract verification library

In the previous section, we learned about contract verification and the current status of **HolBA**'s implementation. To perform verification on the **NIC** model, we would like to automate the process as much as possible. **HolBA** currently offers tools for automatic weakest precondition generation, therefore we need to close the gap between BIR expression and SMT solvers, as well as to implement a convenient interface on top.

4.2.1 Exporting BIR expressions to SMT solvers

As an intermediate language for formal verification, **BIR** possesses a precise semantic. The semantic of BIR expressions is expressed as a set of definitions describing what are the equivalent operations using *wordsTheory* and *combinTheory*. These theories contains definitions and theorems about "words", i.e. bounded N-bit integers that are used to reason about integer types in programming languages and hardware memory in general, and function application and update used for BIR memories as already discussed in Section 4.1.5. For example, the semantic of binary operators in BIR is defined with the following theorems [3] :

axioms ?

$$\begin{aligned} &\vdash bir\_bin\_exp\_GET\_OPER\ BIExp\_And = words\_and \\ &\quad \wedge bir\_bin\_exp\_GET\_OPER\ BIExp\_Or = words\_or \end{aligned} \tag{4.17}$$

$$\begin{aligned} &\vdash \forall (bin\_op : bir\_bin\_exp\_t)\ (w1 : word64)\ (w2 : word64). \\ &\quad bir\_bin\_exp\ bin\_op\ (Imm64\ w1)\ (Imm64\ w2) \\ &\quad = Imm64\ (bir\_bin\_exp\_GET\_OPER\ bin\_op\ w1\ w2) \end{aligned} \tag{4.18}$$

Similarly, a set of definitions and theorems describe the semantic of operations on BIR memories. However, correct handling of endianness, alignment and genericity over the size of memories cells and addresses, these definitions and theorems are pretty complicated to work with. The same is true for the semantic of operations on BIR variables, because of well-typedness and initialization.

3. Theorems 4.17 and 4.18 have been reduced to only two operators and well-typed 64-bit expressions.

For this reason—i.e. writing proof-producing code is costly—, I decided to write a non-proof producing function `bir_exp_to_words` that translates BIR expressions to the equivalent words expression. The obvious downside of such a function is that we now have to trust the translation to be sound, because we no longer get any guarantee from the theorem prover. However, development time is dramatically decreased and offers more time for experimenting. Moreover, this function can later be implemented in a proof-producing way for more trustful verification. Then, in order to have a high confidence of correctness, software engineering practices apply :

— write small and understandable pieces of code and compose them, and
— write a comprehensive suite of tests.

BIR expressions are defined as a HOL4 algebraic data type in Listing 4.6. Hence, in order to translate BIR expressions to words expressions, we need to handle every variant. This has been done [4] using an exhaustive `if-then-else` statement [5]. The code is mostly destructuring HOL4 terms and creating new *wordsTheory* and *combinTheory* terms. In order to obtain an easily reviewable code, a balance between expressivity and conciseness has to be carefully decided. Table 4.1 shows that the length of each variant is relatively small in terms of lines of codes, from 1 line for constants to 51 for memory store expressions. This achieves the first point of the previous list.

Testing of `bir_exp_to_words` has been done using a set of $(bir\_exp, expected)$ couples with increasing complexity, where `bir_exp_to_words` is used to translate each bir_exp and the result is compared to $expected$. Then, BIR expression being defined as an algebraic data type, nesting of BIR expressions follow naturally. This achieves the second point of the previous list.

Listing 4.6 – bir_exp_t definition.

```
Datatype `bir_exp_t =
    BExp_Const       bir_imm_t
  | BExp_Den         bir_var_t

  | BExp_Cast        bir_cast_t bir_exp_t bir_immtype_t

  | BExp_UnaryExp    bir_unary_exp_t bir_exp_t
```

4. Code available here : https://github.com/kth-step/HolBA/commit/2fcca54dcb04a20716e7697f64b5a4578f8a8af9

5. Pattern matching would have been optimal, but isn't possible because of how HOL4 is embedded in SML.

```
  | BExp_BinExp      bir_bin_exp_t bir_exp_t bir_exp_t
  | BExp_BinPred     bir_bin_pred_t bir_exp_t bir_exp_t
  | BExp_MemEq       bir_exp_t bir_exp_t

  | BExp_IfThenElse  bir_exp_t bir_exp_t bir_exp_t

  | BExp_Load        bir_exp_t bir_exp_t bir_endian_t bir_immtype_t
  | BExp_Store       bir_exp_t bir_exp_t bir_endian_t bir_exp_t `
```

`bir_exp_t` variant	Lines of code
BExp_Const	1
BExp_Den	21
BExp_Cast	not implemented
BExp_UnaryExp	8
BExp_BinExp	9
BExp_BinPred	10
BExp_MemEq	10
BExp_IfThenElse	9
BExp_Load	48
BExp_Store	51

TABLE 4.1 – Length of each `bir_exp_t` variant in the implementation of `bir_exp_to_words`.

4.2.2 Pretty-printing to visualize huge BIR expressions

When working with complex constructs, the need of visualization techniques often arise. Generated weakest preconditions grow quickly with the number of statements in a program, linearly or exponentially depending on the type of statements—control flow statements produce exponential growth. While clever techniques can be implemented to keep their size reasonable [**lindner_trabin:_2019**], we often need to read and analyze them.

Printing of BIR terms in general is very verbose. For example, the expression 4.19 with a 64-bit x integer defined using the BSL code in Listing 4.7 yields the printing in Figure 4.1 using HOL4's default printing capabilities.

$$\text{if } (x \leq 100) \vee (y+1 > 10) \vee (x+y \leq 20) \text{ then } 2 \times x \text{ else } 3 \times y + 1 \quad (4.19)$$

Listing 4.7 – BSL code

```
bite (
  borl [
    ble ((bden o bvarimm64) "x", bconst64 100),
    bnot (ble (bplus ((bden o bvarimm64) "y",
                      bconst64 1),
                bconst64 10)),
    ble (bplus ((bden o bvarimm64) "x",
                (bden o bvarimm64) "y"),
         bconst64 20)
  ],
  bmult ((bden o bvarimm64) "x", bconst64 2),
  bplus (bmult ((bden o bvarimm64) "x", bconst64 3),
         bconst64 1))
```

```
BExp_IfThenElse
  (BExp_BinExp BIExp_Or
     (BExp_BinExp BIExp_Or
        (BExp_BinPred BIExp_LessOrEqual
           (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 100w)))
        (BExp_UnaryExp BIExp_Not
           (BExp_BinPred BIExp_LessOrEqual
              (BExp_BinExp BIExp_Plus (BExp_Den (BVar "y" (BType_Imm Bit64)))
                 (BExp_Const (Imm64 1w))) (BExp_Const (Imm64 10w)))))
     (BExp_BinPred BIExp_LessOrEqual
        (BExp_BinExp BIExp_Plus (BExp_Den (BVar "x" (BType_Imm Bit64)))
           (BExp_Den (BVar "y" (BType_Imm Bit64)))) (BExp_Const (Imm64 20w))))
  (BExp_BinExp BIExp_Mult (BExp_Den (BVar "x" (BType_Imm Bit64)))
     (BExp_Const (Imm64 2w)))
  (BExp_BinExp BIExp_Plus
     (BExp_BinExp BIExp_Mult (BExp_Den (BVar "x" (BType_Imm Bit64)))
        (BExp_Const (Imm64 3w))) (BExp_Const (Imm64 1w)))
```

FIGURE 4.1 – Default HOL4 printing

This expression is relatively small and yet the printed term is 17 lines long. Compared to the BSL expression that is 8 lines long [6], that is a two time increase in size. Moreover, lines are long and verbose : for example, a “less-than” binary expression is written as “`BExp_BinPred BIExp_LessOrEqual e1 e2`”. Comparatively, the math expression “$e1 \leq e2$” and BSL expression “`ble e1 e2`” are shorter and arguably more readable.

To answer to these kinds of issues, HOL4 provides the ability to implement

6. 8 lines correspond to the length in documents where line length is limited to 100 characters, instead of the 60 in the report.

"pretty-printers", which are custom printing functions for a given type. Four pretty-printers have been implemented to shorten the verbosity of the printed representation and to add colors to the output. Figure 4.2 shows the same expression printed with the pretty-printers enabled.

```
BExp_If
  (BExp_Or
     (BExp_LessOrEqual
        (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 100w)))
     (BExp_Not
        (BExp_LessOrEqual
           (BExp_Plus
              (BExp_Den (BVar "y" (BType_Imm Bit64))) (BExp_Const (Imm64 1w)))
           (BExp_Const (Imm64 10w))))
     (BExp_LessOrEqual
        (BExp_Plus
           (BExp_Den (BVar "x" (BType_Imm Bit64)))
           (BExp_Den (BVar "y" (BType_Imm Bit64))))
        (BExp_Const (Imm64 20w))))
BExp_Then
  (BExp_Mult (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 2w)))
BExp_Else
  (BExp_Plus
     (BExp_Mult
        (BExp_Den (BVar "x" (BType_Imm Bit64))) (BExp_Const (Imm64 3w)))
     (BExp_Const (Imm64 1w)))
```

FIGURE 4.2 – Same expression printed with the pretty-printer enabled

The pretty-printers introduce a set of features :

— Simplification of verbose constructs as discussed before (e.g. `BExp_BinExp BIExp_Or` is written as `BExp_Or`).
— Different representation of "if-then-else" statements, simplifying reading the expression when either the condition or the "then" expression are very long.
— Consistent breaking—new lines—of long expressions, because the default printer isn't aware of the structure of printed expressions. In Figure 4.1, we can see inconsistent breaking in addition and multiplication binary operations.
— Highlighting of types, facilitating debugging when the expression isn't well-typed.
— Highlighting of all strings, facilitating reading labels and variable names.
— Gathering of nested binary expressions of the same type on the same level. We can see this feature in Figure 4.2 with the two nested "or" binary operators, where the three operands are printed on the same

Listing 4.8 – Ideal interface for "prove_contract"

```
fun prove_contract contract_name prog_def
    (precond_lbl, precond_bir_exp)
    postcond_lbl_and_bir_exp_list
```

Listing 4.9 – Actual interface for "prove_contract"

```
fun prove_contract contract_name prog_def
    (precond_lbl, precond_bir_exp)
    (postcond_lbl_list, postcond_bir_exp)
```

level.

— Rainbow parenthesis, i.e. matching pairs of parenthesis are printed in the same color. This feature is really useful when reading long expression in order to quickly identify where a sub-expression ends.

4.2.3 Implementing a convenient interface

In order to perform a high number of proofs on the **NIC** model, we want to hide as much as possible the implementation details of the contract verification procedure. Ideally, we want a function "prove_contract" taking a program fragment, a pre- and a post-condition as parameters, and producing a proof about the Hoare triple if the contract holds or a comprehensive and useful error message if it doesn't. Listing 4.8 shows the ideal interface that we would want, and Listing 4.9 shows the actual interface that have been implemented.

Interface in Listing 4.8 leverages the general idea of how the weakest precondition generation procedure works : it starts from end labels, setting the weakest precondition there to the postcondition, then propagate the weakest precondition of each node of the **Control Flow Graph (CFG)** to the previous nodes, and stops when it meets the entry label. Then, it is in theory possible to provide different postconditions to each end label, hence the last parameter being a list of $(end\_label, postcond\_exp)$ pairs. However, the current tool only supports using the same postcondition for the multiple end labels, therefore the

interface has been constrained [7].

When implementing this function, high attention has been paid to provide useful and comprehensive feedback in the case of failure. To that end, extensive use of exception wrapping has been made in order to give precise context to exceptions, and a logging library has been implemented (cf. Annex A).

When using **BSL** to express pre- and post-conditions, this function provides an automatic solution to prove contracts. In the following sections, we will then see usage of this function, first to test it and then to perform proofs on the NIC model.

4.2.4 Testing the automatic proof procedure

Performing simple proofs is needed in order to test that the proof procedure works. The following examples introduce two of the tests that have been implemented, focusing on the critical parts of each of them. To this end, some liberties have been taken in order to reduce the complexity for the reader. Moreover, even if the test on conditional jumps has been the last one introduced in chronological order [8], it will be presented first because of its relative simplicity.

Conditional jump

Here we are interested in testing the `prove_contract` function in the presence of a conditional jump with its condition being just an equality test. Feature-wise, this test contains only jump, conditional jump and assignment statements. Listing 4.10 gives a pseudocode representation of this test program, the conditional jump being represented with the `goto-if-then-else` construct.

In this test, we want to check that the triple $\{\top\}\ prog\ \{y = 100\}$ holds. Intuitively, this contract means "for every possible initial state S, executing the program will result in a state S' with $y = 100$". It is interesting to note that the precondition $\top$ means "for every initial state", analogous to the universal

7. A proposal is being discussed at the time of writing this report about making the weakest precondition generation and the Hoare triple definition more general, and possibly allowing this feature.

8. The test on conditional jumps has been introduced in order to fix a bug in the weakest precondition simplification library.

Listing 4.10 – Equivalent pseudocode of the *cjmp* test.

```
entry:
  x = 1;
  goto (if x=1 then assign_y_100 else assign_y_200)
assign_y_100:
  y = 100;
  goto end
assign_y_200:
  y = 200;
  goto end
end:
```

Listing 4.11 – Invocation of `prove_contract` for the *cjmp* test.

```
val thm = prove_contract "cjmp"
  cjmp_prog_def
  (* Precondition *) (blabel_str "entry", btrue)
  (* Postcondition *) (
    [blabel_str "end"],
    beq ((bden o bvarimm32) "y", bconst32 100)
  )
```

quantifier $\forall$ in logic. This comes from the fact that $\top$ is the weakest precondition possible : $\forall x.\ x \implies \top$. Thus, for this Hoare Triple to hold, the generated weakest precondition must be $\top$. Listing 4.11 shows the invocation of `prove_contract`.

Figure 4.3 shows the auto-generated BIR $P \implies WP$ expression, and Equation 4.20 shows the same expression after translation to a *wordsTheory* expression. This expression can be trivially simplified to $\top$, which SMT solvers can very efficiently do. Hence, this invocation to `prove_contract` succeeds.

$$\top \vee (\neg(x = 1w) \vee ((\neg(x = 1w) \vee 100w = 100w) \wedge (x = 1w \vee 200w = 100w))) \tag{4.20}$$

Load and store test

The **NIC** manipulating a buffer descriptor queue, represented in BIR using memories, we need to ensure that `prove_contract` works with programs

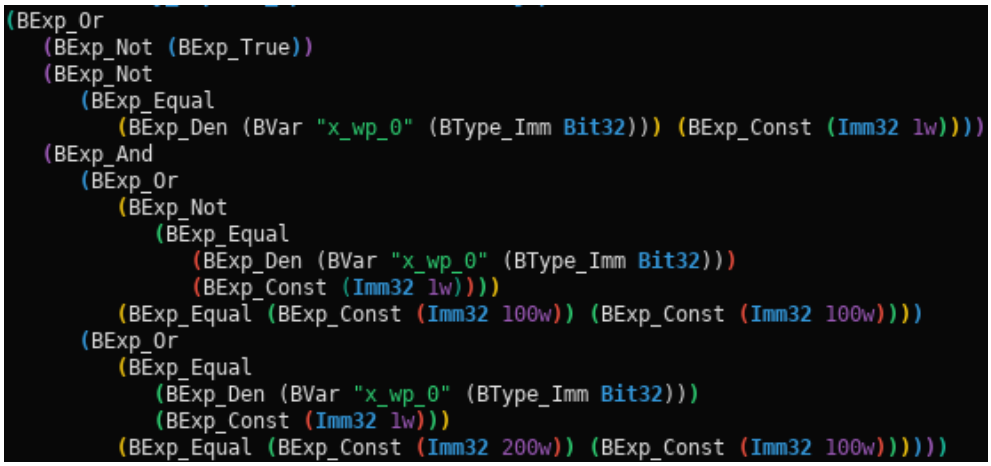

FIGURE 4.3 – Auto-generated $P \implies WP$ BIR expression for the *cjmp* test.

containing memories. In this test, we will store a number N in memory at address A, then load a number into x from address **B**. We want to check the following Hoare Triple : $\{A = B\}\ prog\ \{x = N\}$. Listing 4.12 shows the equivalent pseudocode of the test program, Figure 4.4 the auto-generated $P \implies WP$ expression, Figure 4.5 the same expression translated in *wordsTheory* and Listing 4.13 the auto-generated SMT-LIB 2.0 instance featuring `select` and `store` operations.

Listing 4.12 – Equivalent pseudocode of the *load and store* test

```
MEM = store(MEM, ADDR1, 42)
x = load(MEM, ADDR2)
```

This expression is harder to prove manually. However, SMT solvers can report very efficiently that the negated expression is unsatisfiable, proving the contract. Therefore, we see that contracts involving BIR memories can be proved, thanks to the work of Section 4.1.5.

Other preconditions have been tested to verify that `prove_contract` doesn't prove false contracts and succeeds to prove true ones. With the current program, the precondition $load(MEM, B = N) \wedge B = A + 2$ can establish the postcondition. The second conjunct is important because N is stored in two

Listing 4.13 – Autogenerated SMT instance for the *load and store* test

```
(set-info :source |Automatically generated from
   ↪ HOL4 by SmtLib.goal_to_SmtLib.
Copyright (c) 2011 Tjark Weber. All rights reserved
   ↪ .|)
(set-info :smt-lib-version 2.0)
(declare-fun v0_ADDR1 () (_ BitVec 32))
(declare-fun v1_ADDR2 () (_ BitVec 32))
(declare-fun v2_MEM_wp_0 ()
  (Array (_ BitVec 32) (_ BitVec 8)))
(declare-fun v3_MEM ()
  (Array (_ BitVec 32) (_ BitVec 8)))
(assert
 (not
  (=
   (bvor
    (bvnot (ite (= v0_ADDR1 v1_ADDR2)
                (_ bv1 1) (_ bv0 1)))
    (bvor
     (bvnot
      (ite
       (= v2_MEM_wp_0
        (store
         (store
          v3_MEM
          (bvadd v0_ADDR1 (_ bv1 32))
          ((_ zero_extend 0) ((_ extract 15 8) (_
             ↪ bv42 16))))
         (bvadd v0_ADDR1 (_ bv0 32))
         ((_ zero_extend 0) ((_ extract 7 0) (_
            ↪ bv42 16)))))
       (_ bv1 1)
       (_ bv0 1)))
     (ite
      (=
       (concat
        (select v2_MEM_wp_0 (bvadd v1_ADDR2 (_ bv1
           ↪ 32)))
        (select v2_MEM_wp_0 (bvadd v1_ADDR2 (_ bv0
           ↪ 32))))
       (_ bv42 16))
      (_ bv1 1)
      (_ bv0 1))))
   (_ bv1 1))))
(check-sat)
(exit)
```

```
(BExp_Or
   (BExp_Not
      (BExp_Equal
         (BExp_Den (BVar "ADDR1" (BType_Imm Bit32)))
         (BExp_Den (BVar "ADDR2" (BType_Imm Bit32)))))
   (BExp_Not
      (BExp_MemEq (BExp_Den (BVar "MEM_wp_0" (BType_Mem Bit32 Bit8)))
         (BExp_Store (BExp_Den (BVar "MEM" (BType_Mem Bit32 Bit8)))
            (BExp_Den (BVar "ADDR1" (BType_Imm Bit32))) BEnd_BigEndian
            (BExp_Const (Imm16 42w)))))
   (BExp_Equal
      (BExp_Load (BExp_Den (BVar "MEM_wp_0" (BType_Mem Bit32 Bit8)))
         (BExp_Den (BVar "ADDR2" (BType_Imm Bit32))) BEnd_BigEndian Bit16)
      (BExp_Const (Imm16 42w))))
```

FIGURE 4.4 – Auto-generated $P \implies WP$ BIR expression for the *load and store* test.

```
⊢ ¬(if ADDR1 = ADDR2 then 1w else 0w) ‖
  ¬(if
     MEM_wp_0 =
     MEM |+ (ADDR1 + 1w,(15 >< 8) 42w) |+ (ADDR1 + 0w,(7 >< 0) 42w)
   then
     1w
   else 0w) ‖
  (if MEM_wp_0 ' (ADDR2 + 1w) @@ MEM_wp_0 ' (ADDR2 + 0w) = 42w then 1w
   else 0w) =
  1w
```

FIGURE 4.5 – $P \implies WP$ expression translated in *wordsTheory* for the *load and store* test. `><` is the bitwise extraction operation, `@@` the word concatenation operation, `|+` the memory update operation and `'` the memory load operation. Bit extraction and concatenation is needed because we are working with 16-bit words in a 8-bit memory.

consecutive 8-bit memory locations. Interestingly, the precondition $\bot$ works for all contracts, because $\forall x.\ \bot \implies x$, and it works in this particular case.

4.2.5 Simple automatized proofs on the NIC model

In Chapter 3 we implemented parts of the NIC model using BIR. + Prove invariants + say that we cannot prove everything that we need + conclude on this section

4.3 Trustful analysis on the NIC model

In the previous section, we implemented an automated non proof-producing contract verification library, the non proof-producing part being the translation from BIR expressions to the equivalent *wordsTheory* and *combinTheory* expression. Moreover, this verification library can only produce contracts on BIR programs. In order to perform trustworthy verification on the **NIC** model, we need to make proofs directly on the **NIC** state. In this section, we will perform a proof on a BIR program and then lift it to the **NIC** model.

In order to prove the feasibility of this approach, we will prove a simple property. Listing 4.14 contains the NIC state on which we want to prove a property, Equation 4.21 presents the property that we want to prove, and Listing 4.15 contains a pseudocode representation of the BIR program on which we will perform the verification. Figure 4.6 represents visually the structure of the verification and the steps that we will take during the proof.

Listing 4.14 – NIC state used in this proof

```
Datatype 'nic_state = <|
  dead : bool;
  x : word32
|>'
```

$$\begin{aligned} \vdash \forall nic\ nic'.\ (\neg nic.dead \wedge nic.x = 0 \quad & \{\text{P}\} \\ \wedge\ nic' = holprog\ nic) \quad & \text{prog} \\ \implies \neg nic'.dead \wedge nic'.x = nic.x + 1 \quad & \{\text{Q}\} \end{aligned} \tag{4.21}$$

Listing 4.15 – Pseudocode of the program used in this proof

```
nic.x := nic.x + 1
if nic.x > 10:
    nic.dead := true
```

We recognize in Equation 4.21 the equation of a Hoare Triple :

$\{P(nic)\}\ bir\_prog\ \{Q(nic, nic')\}$.

As already discussed in Section 3, this structure is very common in the base lemmas of the existing proof on the **NIC** model.

make sure it is

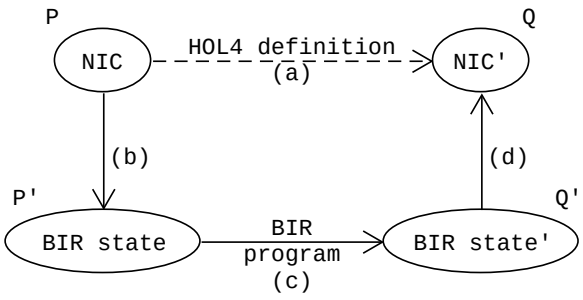

FIGURE 4.6 – Visual structure of the proof

Proof of Equation 4.21

***Note** : The proof has been simplified in order to focus on its global structure.*

All we should have to know is the NIC state definition and some equivalence theorem saying that "NIC' = prog NIC" (a) is equivalent to executing the BIR program from a state *bir_state* to a state *bir_state'* (c), where NIC is somehow equivalent to *bir_state* (b) and NIC' somehow equivalent to *bir_state'* (d).

To express the equivalence between HOL4 states ("NIC") and BIR states, we introduce a relation R. The relation $R\ nic\ bir\_state$ is defined as a simple mapping between the BIR state and the NIC state, as shown in Listing 4.16.

Listing 4.16 – Definition of the relation R

```
val R_def = Define `
  R (nic: nic_state) (bir_state: bir_state_t) <=>
    /\ (bir_env_lookup "nic_dead" bir_state.bst_environ
          = SOME (BType_Bool,
                  SOME (bool2bval nic.dead)))
    /\ (bir_env_lookup "nic_x" bir_state.bst_environ
          = SOME (BType_Imm Bit32,
                  SOME (w2bval32 nic.x))))`
```

We also need an equivalence theorem between the HOL4 definition (a) and the BIR program (c). In real proofs, this can be produced either by a lifter which generates the BIR program from a given input program and gives a "certificate", i.e. a theorem stating the equivalence, this can be a definition which would then mean that we trust that the BIR program is equivalent to the

HOL4 definition. For this proof, we decide to use a definition :

$$\begin{array}{lll} \vdash \forall nic\ nic'.(exec\_prog\ nic\ bir\_prog\ nic') & \text{(a)} & \\ \quad \Leftrightarrow \forall bir\_state\ bir\_state'. & & \\ \quad\quad (R\ nic\ bir\_state & \text{(b)} & (4.22) \\ \quad\quad \wedge\ bir\_state' = BIR\_exec\ prog\ bir\_state) & \text{(c)} & \\ \quad\quad \Longrightarrow\ R\ nic'\ bir\_state' & \text{(d)} & \end{array}$$

In step (a) of our proof, we will need a theorem about the injectivity of the relation R :

$$\vdash \forall nic.\ \exists bir\_state.\ R\ nic\ bir\_state \tag{4.23}$$

Chapitre 5

Conclusions

5.1 Results

5.2 Discussion

Pretty-printer, some work left :

- not parsable yet
- no infix operators
- should change color of types (blue = free vars)

One-button proofs :

- in binary verification, we work with ASM instead of clean C code. Therefore, pre-postconditions are harder to define. Hence, this one-button solution would need more support to easily define those expressions.
- prove_contract should take definitions instead of terms for P and Q
- gives access to lower level functions (generate wp and simp), so it is still possible to get some control either when the proof doesn't work or if we need to use them for other tasks and compose them differently

Mention the "future" meeting that we had on the 29/05/2019.

Sound satisfiability solver for bitvectors to check if the precondition entails the weakest precondition -> same for arrays :

- B ohme, S., Fox, A.C., Sewell, T., Weber, T. : Reconstruction of Z3's Bit-VectorProofs in HOL4 and Isabelle/HOL. In : Certified Programs

and Proofs : First Inter-national Conference. pp. 183–198. Springer (2011)
— https ://arxiv.org/pdf/1807.10664.pdf

Proof : automate

5.3 Future work

Annexe A

LogLib

Reference in Section 4.2.3, but the need arised in general when debugging HOL4 code for tracing code (we can leave trace functions using LogLib and define different levels of verbosity).

Introduce tracing in HOL4 in general