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

Thomas Lacroix — thomas.lacroix@insa-lyon.fr

Département Informatique
INSA de Lyon

2018/2019

Sous la responsabilité de :
Mads Dam : Division of Theoretical Computer Science – KTH
Pierre-Édouard Portier : Département Informatique – INSA Lyon

Abstract

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 into 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 travail répétitif 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 nous 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

1 Introduction

This section 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 the methodology are also discussed.

1.1 Background

Embedded systems are becoming more and more common with the current advent of IoT and mobile computing platforms, such as smartphones. Those systems are fully-fledged computers with powerful hardware, complete operating systems, and access to the Internet. Such systems can run security-critical services, such as a building security system or automatic toll gates, or carry valuable information as it is the case for personal smartphones. Therefore, these two characteristics make them targets of choice for attackers.

The PROSPER project [1] aims to develop a secure and formally verified hypervisor for embedded systems. Hypervisors are thin layers running directly on top of hardware providing the ability to run virtualized applications, such that operating systems or real-time control systems. Those virtualized applications then don't have privileged access to the hardware and have to go through the hypervisor. This allows different applications to share the same hardware while providing strong isolation between them, thus ensuring confidentiality and security. Moreover, security not only means protection from external attacks but also resilience to bugs. If multiple critical systems are running on the same hardware, bugs or crashes in some systems shouldn't affect the others from behaving correctly.

Previous work in the PROSPER project achieved to formally verify a simple separation kernel [2], which later resulted into an implementation of a working hypervisor. Then, they achieved to run both Linux and FreeRTOS on top of it. Finally, they formally verified memory isolation for virtualized applications [3]. However, hardware devices were not included in the verification, so devices, like Network Interface Controller (NIC), cannot be used by the virtualized applications reducing the value of the hypervisor. In order to solve this issue, verification of hardware devices is being carried out.

A formal model of a NIC device has already been produced, on which some security theorems have been proved [4]. These theorems can be seen as high-level proofs relying on a layer of lower-level lemmas. This layer provides an abstraction over the raw formal model. This is illustrated in the left-hand side of Figure 1. However, there exist more devices that are of interest to verify, so it is very desirable to automate formal verification of devices and use a standard model for reasoning about them.

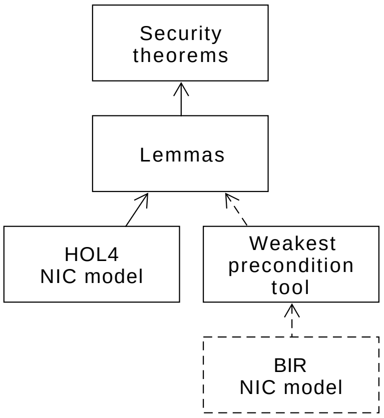

Figure 1: Formal v. BIR NIC models. The left hand side already exists. The dashed elements represent the work done during this project.

The team is now developing a new framework for performing binary analysis in HOL4, an interactive theorem prover, named HolBA [5]. This framework features a machine-independent Binary Intermediate Representation (BIR), a proof-producing transpiler from ARMv8 and Cortex-M0 assembly code to BIR, called the lifter, a proof-producing weakest precondition generator for loop-free programs, and supporting tools [6, 7].

The idea of this work is to translate the formal NIC model of [4] using BIR, then use HolBA's proof-producing weakest precondition tool to prove the same lower-level lemmas than the formal model. With all the lemmas proved, the security properties are implied. Figure 1 gives an overview of this idea: using the proof-producing weakest precondition tool to bind together a newly written BIR NIC model and the work done on the formal model.

1.2 Thesis objective

The goal of this thesis project is to explore verification techniques in order to automate parts, if not all, of the verification process of hardware devices using the HolBA platform. The formal NIC model of [4] is used as working example.

1.3 Delimitations

This work being about exploration of techniques towards automation of verification techniques, instead of being about producing an actual complete proof of a hardware device, some implementations have not been completed in order to save time to explore in more areas. Additionally, this work mainly concerns the HolBA platform that is developed in the team where this thesis took place.

1.4 Choice of methodology

This work has been carried out step-by-step toward an ideal goal, i.e. re-establishing all the security properties. On the road, needs have been identified and tools have been implemented in order to tackle them. This approach made sense in this particular work because the needs were not known in advance, and therefore needed to be identified. This thesis presents the steps taken during this work, the motivations of each tool that have been implemented, and discusses their limitations and future work in the conclusion.

1.5 Related work

1.5.1 Secure execution platforms

The PROSPER project isn't the only project focused on high-security execution platforms. Platforms such that seL4, Microsoft Hyper-V, INTEGRITY Multivisor or even MINIX 3 are examples of platforms used in production and providing strong security properties.

seL4 is a recent L4-based microkernel created in 2006 with the goal to produce a completely formally verified implementation of an L4 microkernel. This has been achieved in 2009 [8]. At this time, seL4 consisted of 8700 lines of C code and 600 lines of assembler. The implementation of seL4 has been formally proved from its abstract specification down to its C implementation. Later, the validity of the generated assembly code has been proved, removing the need of trusting the compiler [9].

Microsoft Hyper-V is Microsoft's hypervisor, widely used today within the Microsoft Azure cloud platform. It has been released in 2008. Hyper-V is a huge codebase, as we can read on VCC's website [1]: "Hyper-V consists of about 60 thousand lines of operating system-level C and x64 assembly code, it is therefore not a trivial target". Microsoft has put a lot of work in formal verification[2]of Hyper-V down to machine code [10]. However, they don't appear to include device drivers in their formal verification.

INTEGRITY Multivisor is a commercial real-time operating system and hypervisor developed by Green Hills Software. Although not much information seems to be publicly available, Green Hills Software has done considerable formal verification work [11]. Multivisor has several certifications, including, for example, ISO 26262 ASIL D automotive electronics, NSA-certified secure mobile phones or FAA DO-178B Level A-certified avionics controlling life-critical functions on passenger and military aircraft [3].

MINIX 3 is an operating system whose design is focused on high reliability and security. It is based on a tiny microkernel with the least responsibility possible, and the rest of the operating system is running an a number of isolated user processes. MINIX 3 has not been formally verified, but intends to provide strong security guarantees by design.

1.5.2 Binary analysis platforms

For this project, we will use the HolBA framework. However, several other binary analysis platforms have been created for various purposes, such as formal verification or static analysis. A common characteristic of these platforms is their use of an Intermediate Representation (IR). IRs are designed to be simpler to use for each platforms' end purpose. As an example, the HolBA platform has BIR as its intermediate representation.

Microsoft Boogie is Microsoft's intermediate verification language. Boogie is the IR for multiple Microsoft tools, including VCC. Boogie as a tool can infer some invariants on the given Boogie program and then generate verification conditions that are passed to an SMT solver [4].

[1] https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c/

[2] Microsoft has several formal verification projects, many of which are freely available for non-commercial use: https://github.com/Microsoft?q=verifier

[3] https://ghs.com/products/rtos/integrity_virtualization.html

[4] https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/

Valgrind is a framework for building program supervision tools, such as memory checkers, cache profilers or data-race detectors [12]. As its core, Valgrind is a JIT x86-to-x86 compiler, translating binary programs into its IR called UCode. Then, *skins*—tools built using the Valgrind framework—are free to work with the IR in order to perform their analysis.

LLVM is a compiler infrastructure which supports a unique multi-stage optimization system [13]. LLVM is built around its IR, LLVM Virtual Instruction Set, which can be described as a strict RISC architecture with high-level type information. This IR made LLVM successful because it is a pragmatic IR suitable for optimizations at multiple stages (link-, post-link, and run-time) and supporting a wide variety of transformations. Leveraging this IR, an ecosystem grew around LLVM, providing tools such as symbolic execution (LLVM KLEE), benchmarking environments or static and dynamic analyzers.

Mayhem is a system for automatically finding exploitable bugs in binary programs and generating working exploits as proof of the discovered vulnerabilities [14]. It leverages BAP, the Binary Analysis Platform from Carnegie Mellon University (CMU BAP) [15], as its IR. It proceeds by fist JIT-ing each instruction to the BAP intermediate language (IL) and then performing a custom symbolic execution.

There are several other tools, platforms, and frameworks, such as Angr. An interesting note though is that BIR's design is based upon CMU BAP's intermediate language.

The novelty introduced in HolBA, however, is that proofs are performed directly on the generated assembly code, not at the source code level. Therefore, proofs can be performed on programs without needing their source code, and regardless on the programming language used as long as it can be compiled in assembly code in an ISA supported by the platform.

2 Definitions and relevant theories

This section intends to provide brief introductions for concepts and the foundation of theories that are essential in order to understand the problem that this project aims to explore.

2.1 Interactive Theorem Proving and HOL4

Interactive theorem provers are software producing formal proofs, in an interactive fashion, i.e. a human can step through the proof interactively while the proof assistant provides some automation (like rewriting of terms, arithmetic evaluation, or integration with external tools like SMT solvers). Coq, HOL4 or Isabelle are such tools.

HOL4 [16] stands for Higher-Order Logic. It is a programming environment deeply embedded into the SML programming language enabling to prove theorems and write proof-producing programs. HOL4 uses a very small kernel in order to provide very high guarantees of correctness.

2.2 HolBA's BIR

HolBA's Binary Intermediate Representation (BIR) [7], introduced in the Introduction, is a machine independent binary representation. It aims to be the simplest possible while still being able to represent all possible binary programs but self-modifying programs. It does so by having a limited syntax and by forbidding implicit side-effects. A statement can only have explicit state changes and can only affect one variable. Valid BIR programs must be well-typed.

This representation allows producing proofs more easily that with classical binary representations, whose design are focused on execution speed rather than offline analysis. Moreover, BIR doesn't have unspecified behavior.

BIR is implemented as a set of HOL4 data types, and possesses a completely defined semantic. Section 5.2 contains a more thorough discussion of the BIR semantic. Section 4.3 implements a toy BIR program and presents the concrete BIR syntax.

Among its supporting tools, HolBA features a tool to visualize the Control Flow Graph (CFG) of BIR programs.

2.3 Hoare Triples

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 1). Hereafter, we assume programs and states to be well-typed. A Hoare Triple is also called *a contract*.

$$\{P\}\ prog\ \{Q\} \overset{def}{=} \\ S' = exec(S, prog) \land P(S) \implies Q(S') \quad (1)$$

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.

2.4 Weakest preconditions

While Hoare logic introduces sufficient preconditions, Dijkstra introduced the concept of necessary and sufficient preconditions, called "weakest" preconditions [17]. Such weakest preconditions (WP) can be automatically derived from a program *prog* and a postcondition Q. Let's call $WP(prog, Q)$ such a WP. Then, from Equation 1 follows:

$$\forall (prog, Q), \{WP(prog, Q)\}\ prog\ \{Q\} \quad (2)$$

For the program $n := n+1$ mentioned in the previous section, the WP for the postcondition $even(n)$ is $odd(n)$, i.e. incrementing the value of an odd integer variable by one makes it even.

For a triple $\{P\}\ prog\ \{Q\}$ to hold, P must be stronger than the WP, i.e. we need to prove that $P \implies WP(prog, Q)$. While multiple methods exist to perform such proofs, Satisfiability Modulo Theory (SMT) solvers offer a convenient and automatic solution. With an SMT solver, proving R consist in checking that $\neg R$, its negation, is *unsatisfiable*. SMT solvers can also give counter-example R is false.

HolBA provides a proof-producing tool for automatically deriving WP on loop-free BIR programs whose control flow can be statically identified [7]. However, SMT solvers had never been used before this work.

3 Overview of the formal proof of the NIC model

This section will present the formal verification of [4] and present an attempt that have been made about visualization of the proofs.

The formal verification of a NIC in [4] represents the NIC as a transition system. composed of five finite state automata, each responsible for a different task: initialization, transmission, transmission teardown, reception and reception teardown. These automata have autonomous transitions that represent the standalone operation of the device. Communication with the CPU is represented with non-autonomous transitions. The model also contains a scheduler. Since this model has been realized using the public specification of the device, which is underspecified, the simulated state of the device is marked *dead* if the model is asked to describe any transition or operation that is not described by the specification. Being designed as a transition system, the whole model is loop free, which is convenient for contract-based verification, since that hard composition theorems would otherwise be needed.

The state of the NIC is defined as a nested data type containing registers and a memory called *CPPI_RAM*.

The low-level lemmas of the verification (cf. Figure 1) are stated as Hoare Triples using invariants (Equation 3) that must hold for every possible transition:

$$I_{NIC} \overset{def}{=} \neg dead\ \land\ I_{init}\ \land\ I_{tx}\ \land\ I_{rx}\ \land\ \cdots \quad (3)$$

3.1 Visualizing proof dependencies

The model of the NIC consists of 1500 lines of HOL4 code and required around three man-months of work. The NIC invariant consists of 650 lines of HOL4 code and the proof consists of approximately 55 000 lines of HOL4 code including comments. Identifying the invariant and implementing the proof in HOL4 required around one man-year of work [18].

The proof being consequent and divided into several script files, it is difficult to identify what are the low-level lemmas to be reproved in this work. Therefore, a tool, called DepGraph, has been implemented in order to extract the dependency structure from proof files in the form of a graph. Then, the fringe of the graph represents the smallest set of lemmas that is enough to prove in order to imply the security properties by using the rest of the proofs unchanged.

DepGraph features two frontends that can extract dependencies between HOL4 theories (i.e. compiled SML files containing proofs of lemmas and theorems) and between definitions, theorems, and lemmas. However, this tool presents some critical shortcomings:

- The theory dependencies exporter uses files generated by Holmake, the HOL4 compile system, in order to get the dependencies between theories. However, those files don't really represent dependencies but the files to be loaded before this script can be loaded, in a recursive fashion. Therefore, they represent the transitive reduction of the dependency graph. Because of this fact, precious knowledge is lost and cannot be recovered by using this method: edges representing direct dependencies can be removed if the remaining edges still account for this dependency. Therefore, we are still able to tell which nodes depend on some node n, but we cannot identify the aforementioned fringe. In order to solve this problem, different approaches exist, such as implementing a simplified SML parser that looks only at dependencies, or injecting code inside the dependency resolution of an existing SML compiler. However, this would involve too much work that isn't the direct focus of this thesis.

- The definition, theorem and lemma dependencies exporter uses word-based heuristics in order to extract dependencies, and is as such not quite reliable and cannot give any guarantee. As above, there exist similar solutions in order to get multiple levels of guarantees, such that implementing a SML parser or injecting code inside HOL4 theory and definitions handling, but this would also require too much work. Destructuring HOL4 theories does not work because of how HOL4 has been designed. Moreover, such dependency graphs become quickly big, making them unusable in practice, and some additional work would be needed in order to represent them in a convenient way. Therefore, as above, no further work has been put into this exporter.

DepGraph's frontend for theory dependencies is still useful for documentation purpose, and has been used to visualize HolBA's architecture.

4 The BIR NIC model

This section presents the three different approaches made in order to implement an equivalent BIR model from the formal NIC model. Tools that have been implemented in order to build it will also be introduced.

Multiple approaches to the translation of the formal NIC model to an equivalent BIR program have been considered, from handwritting a BIR program (Sections 4.3 to 4.5), lifting a C program (Section 4.2) to developing a new device model specific IR (Section 4.1).

4.1 Using flowcharts as Intermediate Representation

The CFG of the transition system of the formal model looks like a tree. Therefore, using flowcharts could be a convenient way to represent such structures. An attempt has been made to design a flowchart representation. Figures A.1, A.2 and A.3 show respectively a preview of the scheduler, transmission automaton and a particular transition of this automaton.

However, while this visual representation is useful in order to get to know the formal NIC model, we encountered several shortcomings:

- Flowcharts of each transition rapidly grows in size with the complexity of its formal counterpart. Possible workarounds include the use of nested diagrams, as it is the case of Figure A.3 representing one node of Figure A.2, namely `fetch_next_bd`, or usage of shorter ways of representing common patterns, as it is the case for representing dead transitions on Figure A.3.
- It is hard to define a coherent visual language able to represent the full set of features needed in order to realize device models. Additionally, this language must be compatible or easily translatable to BIR.
- It is hard to design a textual representation of this visual language other than conventional programming languages, so using such representation would require a substantial implementation effort in order to implement all the tools needed to use it. Developing visualization tools for a conventional language appears to be a more reasonable approach than developing a visual language.

For those reasons, it has been decided to not go further with this visual representation, and to focus instead of existing tools of the HolBA platform.

4.2 Writing the model in C

One-to-one translation of the definitions related to the transmission automaton, scheduler, state, and *CPPI_RAM* of the formal model have been realized in C. However, when studying the compiled assembly code and lifted BIR program, we noticed that all the convenient naming that we can use in the formal or the C model is lost and replaced with abundant usage of the stack. While this is completely normal behaviour for a C compiler, this is not convenient when performing later proofs on the model. Reasoning about the stack would require more code than a complete rewrite of the model in BIR. This experiment made us realize that writing the model is a rapid operation and that we should rather focus on making the verification step as smooth as possible because it is the most difficult one to perform.

4.3 A toy BIR model

Before writing the whole NIC model by hand, we shall identify the structure of the model and develop tools that facilitate its implementation. Using well-designed tools can reduce the boilerplate work of the implementation, helping to focus only on the meaningful content of the implementation, and can also reduce the chance of introducing bugs as the code is factored and mechanically shorter.

A transition system similar to the one of the NIC model has been implemented, containing one scheduler and two automata. The two automata feature a simple linear transition system, and each of them has one non-autonomous transition, that is performed respectively by two external functions that represent memory accesses from the CPU in the NIC model.

This toy model has first been designed using the flowchart representation. Then, writing the BIR program has been a repetitive but straightforward step. The resulting BIR program is 450 lines of code long. The following issues have been identified:

- BIR, as a HOL4 embedded language, is very verbose. Simple operations like additions or assignments require long constructions. Section 4.4 presents BSL, a less verbose way of writing BIR programs.

- BIR features only one conditional statement controlling the control flow of a program: conditional jumps. Hence, BIR is not convenient for representing **if-then-else** statements with more than two branches. Section 4.5 presents some helper functions that have been implemented in order to be able to abstract the raw BIR code and work at a higher level.

4.4 BSL: BIR Simple Language

A library, named BSL, offering the same expressiveness than BIR but with shorter constructs, has been implemented. This library has been kept simple and will serve as the base layer of possible later abstractions. As such, it has been decided that no feature other than pure syntactic construct, such that type inference, would be included.

BSL is composed of a set of functions with short names (prefixed with the letter *b* in order to not clash with names in the global namespace) and a coherent interface offering interoperability with HOL4 quotation system and smart use of partial function application (e.g. for BIR types).

4.5 Implementing the real model

From the knowledge gained in 4.3, a set of helper functions has been implemented, mainly in order to facilitate reasoning about the state machine.

Because of time constraints, not every transition of the formal NIC model have been implemented in this new model. Instead, we decided to focus on only some transitions in two automata: initialization and transmission. This focus is pragmatic for two reasons: (a) we decided to write the model along with the proof, only needed transitions for the current proof, in order to grow the model at a reasonable pace and to no clutter it with unneeded features or at the contrary missing critical aspects during the first sketch (b) as the verification was performed at the same time, we decided to start with easier, but not obvious, transitions, hence the choice to postpone verification of the more complex reception automaton to later in the work.

5 Non-proof-producing automatic contract verification

This section presents the library that has been implemented in this work for automatic contract verification, after discussing about the problems that have been solved in order to build it.

5.1 Exporting BIR expressions to SMT solvers

In order to prove the implication needed to prove Hoare Triples, we must be able to export HOL4 goals to external SMT solvers. HOL4 features a library for interfacing SMT solvers and HOL4, called *HolSmtLib*, using the standard format SMT-LIB 2.0 [19]. However, for obvious reasons, *HolSmtLib* cannot export BIR expressions out of the box. Therefore, the BIR expression must be first translated.

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 HOL4's *wordsTheory* and *combinTheory*.

Because of the complexity of the BIR semantic, we deciced to implement a non proof-producing translation, in order to get more time for other experiments. 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, but a proof-producing version can still be implemented later.

bir_exp_to_words has been implemented as an exhaustive *match* statements on the expression type, each type being then translated to the corresponding non-BIR expression. Reccursive call is used for nested expressions.

5.2 BIR memories and SMT

In order to prove Hoare Triples containing BIR memories, we need to use *combinTheory*. However, it is not supported by *HolSmtLib*. In order to solve this issue, we proceeded in two steps:

1. we looked at how to translate BIR memory operations to SMT-LIB 2.0 and found *ArraysEx*. Then, we verified that this translation is sound by checking that *ArraysEx*'s axioms are correct in HOL4.

2. we extended *HolSmtLib* in order to support this theory, and wrote tests in order to ensure that the implementation is correct, since this library is not proof-producing [5].

5.3 Pretty-printing of BIR expressions

Generated WP grow in size quickly with the number of statements in a program, linearly or exponentially depending on the type of statements [6]. Additionally, the default printing of BIR terms is very verbose. In order to increase user-friendliness, the readibility of long BIR expressions should be improved. Therefore, four pretty-printers have been implemented, providing the following features:

- Simplification of verbose constructs.
- Different representation of **if-then-else** statements, simplifying reading the expression when either the condition or the **then** expression are very long.
- Consistent breaking of long expressions.
- 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 at the same level.
- Rainbow parenthesis, i.e. matching pairs of parenthesis are printed in the same color. This feature is really useful when reading long expressions in order to quickly identify where a sub-expression ends.

5.4 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 postcondition 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 1: Interface of "`prove_contract`"

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

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. Section 7).

[5] *HolSmtLib* does have proof-reconstruction capabilities, but they are using an outdated version of Z3, the SMT solver that we used.

[6] Control flow statements produce exponential growth. While clever techniques can be implemented to keep their size reasonable [7], we often need to read and analyze them.

Then, in order to test this function and check that the interface actually let us verify different contracts, we verified three different programs:

1. we verified $\{\top\}$ *prog* $\{y = 100\}$ on a simple program containing one conditional jump where each target assigns a different value to y, but where the condition is always true at runtime.

2. on a program storing a number N in memory at address A, then loading into x from address B, we verified $\{A = B\}$ *prog* $\{x = N\}$. This test allowed us to check that the extension of *HolSmtLib* works for basic cases.

3. on a program computing the sum of integers from 0 to a given n using a for loop, we verified the loop invariant on the body of the loop, using Gauss' formula.

For each of those tests, we also experimented with other pre- and postconditions, and checked that invalid contracts can indeed not be proved.

5.5 Simple automatized proofs on the NIC model

Since the base lemmas of the formal NIC proof are phrased in terms of invariants holding in each transition of the model, representable as Hoare Triple, they can be proved using the non-proof-producing contract verification library that has been implemented in this project, as long as the pre- and postconditions can be represented using BIR.

We used `prove_contract` in order to successfully verify two Hoare Triples (Equations 4 and 5). Those two Hoare Triple respectively represent that, stating from a non-dead initial NIC state, performing one autonomous transition of the transmission automaton doesn't end in an undefined state (i.e. a dead state) and performing one non-autonomous transition of the initialization automaton does end in an undefined state.

$$\{\neg NIC.dead \land NIC.tx.state \in \{tx2, tx3, tx5, tx6, tx7\}\}\ tx\_automaton\{\neg NIC.dead\} \quad (4)$$

$$\{\neg NIC.dead \land NIC.init.state \in \{it1, it3, it4\}\}\ init\_automaton\{NIC.dead\} \quad (5)$$

5.6 Limitations of this approach

While being convenient and working well for simple contracts, the contract verification library implemented in this chapter suffers from two limitations:

- this library isn't proof-producing. Using HOL4 requires a significant learning effort, and easier non proof-producing tools exist.

- this library, with its approach to contract-based verification, is limited by the expressiveness of BIR, since the pre- and post-conditions are BIR terms. BIR doesn't have quantifiers, and it is, therefore, impossible to prove contracts containing existential quantifiers. The existing formal proof on the NIC of [4] requires existential quantifiers in order to reason about the buffer descriptor queue in the *CPPI_RAM* memory of the NIC, and this part of the proof can therefore not be replaced by using `prove_contract`.

The following Section 6 explores a new proof-producing approach for performing contract-based verification on the BIR model that gives an answer to these issues.

6 A new approach for trustful analysis

In this section, we will perform a proof on a BIR program and then lift it to an abstract model.

In order to prove the feasibility of the approach, we decided to prove a simple property. We used as a formal state, on which we want to prove a property, a HOL4 data type composed of two variables: $NIC.dead : bool$ and $NIC.x : word32$. Equations 6, 7 and 8 present the property that we proved. Listing 2 contains a pseudocode representation of the BIR program on which we performed the verification. Figure 2 represents visually the structure of the verification and the steps that we took during the proof.

$$\vdash \forall nic.\ P_{NIC}\ nic \overset{def}{=} \neg nic.dead \land nic.x = 0w \quad (6)$$

$$\vdash \forall nic\ nic'.\ Q_{NIC}\ nic\ nic' \overset{def}{=} \neg nic'.dead \land nic'.x = nic.x + 1w \quad (7)$$

$$\vdash \forall nic\ nic'.\ (exec\_prog\ nic\ bir\_prog\ nic' \land\ P_{NIC}\ nic) \implies Q_{NIC}\ nic\ nic' \quad (8)$$

Listing 2: Pseudocode of the program used in this proof

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

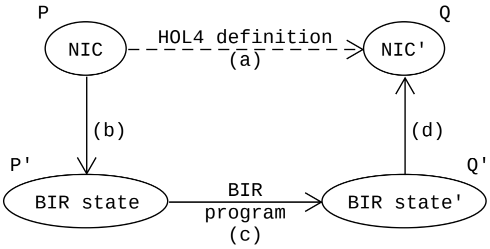

Figure 2: Visual structure of the proof. References like (a) to the arrows of this Figure are used throughout the proof to refer to a particular step.

Remark 1. *Q_{NIC} nic nic′ has been defined on both initial and final states, in order to be able to reason about the initial state in the postcondition. This allowed us to write $nic'.x = nic.x + 1w$ instead of $nic'.x = 1w$ for example.*

We used Definition 9 in order to establish a relation between the HOL4 definition (a) and the BIR program (c). Another solution would have been to generate a proof-producing lifter from the formal definition (a) to BIR program (c), but has been kept as future work.

$$\begin{aligned}
&\vdash \forall nic\ nic'.\ exec\_prog\ nic\ bir\_prog\ nic' \overset{def}{=} && \textbf{(a)}\\
&\quad \forall bir\_state\ bir\_state'.\ (R\ nic\ bir\_state\ \wedge && \textbf{(b)}\\
&\qquad bir\_state' = BIR\_exec\ prog\ bir\_state) && \textbf{(c)}\\
&\qquad \Longrightarrow R\ nic'\ bir\_state' && \textbf{(d)}
\end{aligned} \tag{9}$$

The relation R is used to express an equivalence between HOL4 states ("*nic*") and BIR states, and is defined as a simple mapping of the variables between the BIR state and the formal state.

Proof. For (b), we proved the injectivity of R, and added to the theorem some constraints on the initial BIR state needed as hypothesis for BIR_exec.

Then, we defined the equivalent pre- and postcondition on BIR, and proved Theorem 10.

$$\begin{aligned}
\vdash\ &\forall bir\_state\ (\exists nic.\ R\ nic\ bir\_state\ \wedge\\
&\quad P_{NIC}\ nic) \Longrightarrow P_{BIR}\ bir\_state
\end{aligned} \tag{10}$$

Limitation Q_{BIR} is a function of the end state only. Hence, in order to reason about the initial state, we need in general to introduce ghost variables postcondition. In this proof, since the contract that we are proving is simple, using the actual value of $nic.x$ is enough. However, this may pose a problem if we want to generalize the proof.

For step (d), we proved Theorem 11. We introduced bir_state and P_{BIR} in order to reason about both the initial and final state in the postcondition (cf. Remark 1).

$$\begin{aligned}
\vdash\ &\forall\ bir\_state'.\ Q_{BIR}\ bir\_state' \Longrightarrow\\
&\quad (\forall nic\ nic'\ bir\_state.\ P_{BIR}\ bir\_state\\
&\qquad \wedge\ R\ nic\ bir\_state\ \wedge\ R\ nic'\ bir\_state'\\
&\qquad \Longrightarrow Q_{NIC}\ nic\ nic')
\end{aligned} \tag{11}$$

Finally, we proved that the Hoare Triple holds on the BIR program:

$$\vdash\ \{P^{exp}_{BIR}\}\ bir\_prog\ \{Q^{exp}_{BIR}\} \tag{12}$$

To that end, we used HolBA's proof-producing WP tool to generate the implication $P \implies WP$ as a BIR expression. Then, to turn the goal into a *wordsTheory* expression, we showed the assumptions needed to use BIR semantic's rewrite theorems about well-typedness and initialization, as well as Lemmas 13 and 14 instantiated for all BIR variables. We were then able to substitute all occurrences of x_val into $BVal\_Imm\ (Imm32\ x\_word)$, and similarly for $dead\_val$.

$$\exists x\_imm.\ x\_val = BVal\_Imm\ x\_imm \tag{13}$$

$$\exists x\_word.\ x\_imm = Imm32\ x\_word \tag{14}$$

Finally, we were able to rewrite the goal using the full set of BIR semantic theorems and some rewriting rules into a form that *HolSmtLib* can handle.

To complete the proof of Theorem 8, we used the deduction rule with the injectivity theorem, Theorem 10, the definition of R, and Theorems 12, 9 and 11, in that order.

□

7 User experience and good practices

This chapter will present the care that has been put into user experience and good practices throughout this work.

7.1 Towards better user experience

Throughout our design and implementation process, great care has been put into having a good user experience.

BSL and the pretty-printer are two successful attempts to simplify HolBA users manipulation of BIR, respectively writing and reading BIR code. While HOL4 provides powerful quotation and printing systems, these are verbose by default. Those two libraries were not a hard need, but do offer a some relieve to users.

Error handling in SML and HOL4 offers a robust way of handling failure. However, sometimes some exceptions are raised in a hidden implementation of some function call and their message convey few information. To answer this issue, there are two immediate possibilities (in addition to fixing the actual error): (a) code could handle possible exceptions of code that it uses and that could produce some, and either handle the issue (if possible) or raising another more meaningful error, and (b) wrap the exceptions using HOL4's *Feedback* library in order to add context information and produce a result analogous to backtrace in other programming languages. We took great care to use both solutions during our implementations.s

A logging library , called *logLib*, has been implemented. It aims to provide a unified use of HOL4 tracing capabilities which it uses in order to manage verbosity level at runtime. It then gives five functions, one for each level, that log a message along with the location where it is issued and the message level of importance (*trace*, *debug*, *info*, *warn* and *error*).

An SML exception pretty-printer has been implemented, called *pretty_exnLib*, in order to format exceptions in an easily readable way other: exception are otherwise just dumped on the output with bad formatting. It provides only two analogous functions that, given an exception as parameter, pretty-print the exception and return it unchanged.

7.2 Best practices for a complex platform

When building complex systems, special care should be taken in order to ensure that it grows in good conditions. Despite providing the best capabilities, a system with poor usability will encounter a slow adoption, as shown in the previous section. But when a system gets used by an increasing number of users, a different working organization is needed. This section will first briefly introduce a critical concept that should be mastered for the successful growth of a software platform. Then, we will study the CI system that has been set up for HolBA.

7.2.1 Developing simple interfaces

In 1995, the Gang of Four published a book about design patterns for software design [20]. "Program to an interface, not an implementation." is a key concept presented among the patterns. Since then, this sentence is considered as one of the most important design principles in software development.

This section doesn't intend to give a thorough explanation of the concept. However, some key insights shall be provided:

- An interface is a contract between a system (such as a function, a library, a platform) and its user. This contract states the behavior and guarantees that this system offers via its public components. It defines the interaction between the component, the system, and the user. But most importantly, the contract leaves out the implementation. An interface builds an abstraction over the implementation. Using explicit contracts is particularly useful when performing contract-based verification, as it can simplify reasoning about components and allow building a more resilient system.

- Libraries provide features, which sometimes have a very complex implementation. This is particularly true for trustworthy binary code analysis. Building interfaces helps to abstract away complex implementations and helps to compose systems together.

- Programming to an interface instead of an implementation makes the code more resilient to changes. Indeed, if an implementation must be changed in order to provide more features, fix critical bugs or deliver more performance, codes that are not aware of the implementation will not need to be modified.

7.2.2 Continuous Integration: tests and static analysis

Continuous Integration (CI) is a development practice where developers of a project *integrate* frequently their code and changes into a single shared source repository, hence the name of this practice. Each integration of any change into the shared repository should be automatically verified by automated builds and a full set of automated tests. This allows to immediately see if changes to the source repository break a feature or introduce a regression, thus allowing to fix it immediately. Since this practice imposes to run a whole battery of tests for each and every change, even the smallest one, automation is key. [21] gives a detailed introduction to this practice.

A CI system has been added to the HolBA repository, and its practices are being adopted by the research team. For each change, the CI system recompiles the whole platform from scratch and runs all the tests and executes a set of example scripts. Additionally, the CI system also performs some static analysis of the source code in order to give developers some key insight of the state of the code. Two static analyzers have been implemented, showing respectively the *cheats* used in the formal proofs and the "*TODO*" comments indicating left work to be done [7].

8 Conclusions

8.1 Discussion

The goal of this paper was to perform experiments about the automation of formal verification of devices at the binary level. The NIC model of [4] has served as a central theme throughout the work. Work in this thesis has been divided in practice into three parts, in addition to the learning process due to the very steep learning curve of HOL4:

1. **Implementation of the NIC BIR model**: the formal model of the NIC of [4] has been partially implemented as a BIR program, and BSL has been added to the HolBA platform in order to make this task more convenient.

2. **Non-proof-producing automatic contract verification library**: a user-friendly (adjective rarely used to describe HOL4), convenient and automatized library for contract verification have been implemented. This library brings HolBA one step closer to "one-button solutions" for performing software verification. It features a non-proof-producing translation of BIR expressions to expressions that can be exported to SMT solvers, and an extension of HOL4's SMT library has been realized in order to add reasoning about BIR memories.

3. **Formal proof of a BIR program on a HOL4 state**: a novel usage of BIR and method for transferring properties from BIR program to formal HOL4 specification has been carried out. Additionally, while not automated, the work needed in order to implement automatic HOL4 tactics has been clearly identified. Every automation work should be preceded by a manual implementation of the task, that this proof intends to be.

Throughout this work, a strong focus has been placed on ease of use and user-friendliness with BSL, the pretty-printer, the implementation of new logging utilities, guidelines about error handling in SML and with HOL4, and a friendly exception printer. We strongly believe that user experience of such complex platforms are of primary importance because it can dramatically reduce time spent in debugging and the need of exhaustive documentation. Furthermore, it can give users the desire to keep using the platform. If it is decided to put more work in HolBA in order to make it a powerful binary analysis platform, the research team is advised to consider this aspect while expanding HolBA capabilities.

Some failed experiments have also been realized, including the DepGraph tool that revealed to not be adapted to its planned usage or which would have needed to much work for too few benefits (though DepGraph's design is future-proof and the tool can easily be extended later if new needs or novel ideas appear), and the former two implementation attempts of the NIC model using flowcharts or C which have been found to not be suitable for device models.

8.2 Future work

The topic of software verification, while being as old as Computer Science, still have a lot of work

[7] At the time of writing, this feature has not been added yet to the CI system because of time constraints. However, it is ready to be integrated.

to be done and paths to be explored. This thesis opens some paths that should be considered by someone interested in the topic.

The pretty-printer implemented in this work is only a small experiment only scratching the surface of what can be done in HOL4. Further work in this domain should follow the concrete needs that arise with extended usage of BIR. Current limitations and possible further work of the pretty-printer presented in this document include:

- the generated output isn't parsable, which is against HOL4 conventions where every printed term should be parsable by HOL4 quoting system. A solution to this problem would be to implement a theory introducing definitions for the new abbreviations. However, some liberties taken in the pretty-printer cannot be parsed: (a) the flat printing of nested binary expressions of the same state, because they would need multiple arity functions which are not available in HOL4, and (b) **if-then-else** statements in the current representation.

- the pretty-printer doesn't use infix operators, feature widely employed in custom syntax used in some theories, like *wordsTheory*. Infix syntax for binary operators can dramatically reduce the verbosity of expressions, and give a more familiar representation that *could* be simpler to work with.

- the choice of colors could be reviewed in order to more closely follow HOL4 convention and usage. For example, BIR types are highlighted in the same color than free variables. While creating a very limited chance of ambiguities, users familiar with HOL4 conventions can be annoyed with this inconsistency.

The non-proof-producing contract verification library implemented in this work can be really useful for prototyping, but it cannot be used for trustful verification. The limitations it suffers could be reasonably fast to overcome, and they should be fixed especially if HolBA were to become a strong binary analysis platform. Limitations include (a) the impossibility to compose contract in order to reason about longer functions, to compose functions or to prove properties on not loop-free programs, and (b) the obvious limitation of not being proof-producing. Work is ongoing at the time of writing about weak-correctness and contract composition, answering to (a). The main blocker of (b) is the absence of a proof-producing ways of using SMT solvers for proving contracts (Equation **??**). The proof of Chapter 6 and its SML implementation identify what automatic procedures should be added to HolBA in order to perform step (b) and progress towards proof-producing automatic verification. Additionally, the signature of `prove_contract` should be reworked in order to enable mixing it with proof-producing code, after both (a) and (b) are solved. For example, it should work with HOL4 definitions instead of raw terms.

References

[1] *PROSPER: Provably Secure Execution Platforms for Embedded Systems.* URL: http://prosper.sics.se/ (visited on 02/04/2019).

[2] Mads Dam et al. "Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel". In: 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS'13),November 4 - 8, 2013 Berlin, Germany. ACM Press, 2013. (Visited on 02/04/2019).

[3] Hamed Nemati et al. "Trustworthy Memory Isolation of Linux on Embedded Devices". In: *Trust and Trustworthy Computing.* Ed. by Mauro Conti, Matthias Schunter, and Ioannis Askoxylakis. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 125–142. ISBN: 978-3-319-22846-4.

[4] Jonas Haglund. "Formal verification of systems software". Master's Thesis. KTH Royal Institute of Technology, Nov. 5, 2016. 290 pp.

[5] *HolBA - Binary analysis in HOL.* original-date: 2017-11-29T00:55:48Z. May 21, 2019. URL: https://github.com/kth-step/HolBA (visited on 05/21/2019).

[6] Roberto Metere, Andreas Lindner, and Roberto Guanciale. "Sound Transpilation from Binary to Machine-Independent Code". In: *Formal Methods: Foundations and Applications.* Ed. by Simone Cavalheiro and José Fiadeiro. Lecture Notes in Computer Science. Springer International Publishing, 2017, pp. 197–214. ISBN: 978-3-319-70848-5.

[7] Andreas Lindner, Roberto Guanciale, and Roberto Metere. "TrABin: Trustworthy analyses of binaries". In: *Science of Computer Programming* 174 (Apr. 1, 2019), pp. 72–89. ISSN: 0167-6423. DOI: 10.1016/j.scico.2019.01.001. (Visited on 02/05/2019).

[8] Gerwin Klein et al. “seL4: formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. the ACM SIGOPS 22nd symposium. Big Sky, Montana, USA: ACM Press, 2009, p. 207. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. (Visited on 02/15/2019).

[9] *What is Proved and What is Assumed | seL4*. URL: https://sel4.systems/Info/FAQ/proof.pml (visited on 06/03/2019).

[10] Dirk Leinenbach and Thomas Santen. “Verifying the Microsoft Hyper-V Hypervisor with VCC”. In: *FM 2009: Formal Methods*. Ed. by Ana Cavalcanti and Dennis R. Dams. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 806–809. ISBN: 978-3-642-05089-3.

[11] Raymond J. Richards. “Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel”. In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Ed. by David S. Hardin. Boston, MA: Springer US, 2010, pp. 301–322. ISBN: 978-1-4419-1539-9. DOI: 10.1007/978-1-4419-1539-9_10. (Visited on 02/18/2019).

[12] Nicholas Nethercote and Julian Seward. “Valgrind: A Program Supervision Framework”. In: *Electronic Notes in Theoretical Computer Science*. RV '2003, Run-time Verification (Satellite Workshop of CAV '03) 89.2 (Oct. 1, 2003), pp. 44–66. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)81042-9. (Visited on 02/18/2019).

[13] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. Master's Thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[14] Sang Kil Cha et al. “Unleashing Mayhem on Binary Code”. In: *2012 IEEE Symposium on Security and Privacy*. 2012 IEEE Symposium on Security and Privacy (SP) Conference dates subject to change. San Francisco, CA, USA: IEEE, May 2012, pp. 380–394. ISBN: 978-1-4673-1244-8 978-0-7695-4681-0. DOI: 10.1109/SP.2012.31. (Visited on 02/18/2019).

[15] David Brumley et al. “BAP: A Binary Analysis Platform”. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 463–469. ISBN: 978-3-642-22110-1.

[16] *HOL Interactive Theorem Prover*. URL: https://hol-theorem-prover.org/ (visited on 02/04/2019).

[17] Edsger W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. (Visited on 02/15/2019).

[18] Jonas Haglund and Roberto Guanciale. “Trustworthy Isolation of a Network Interface Controller”. In: KTH Royal Institute of Technology, Stockholm, Sweden: To be published.

[19] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. 2016. URL: http://www.smt-lib.org.

[20] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. OCLC: 31171684. Reading, Mass.: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0.

[21] Martin Fowler. *Continuous Integration*. martinfowler.com. URL: https://martinfowler.com/articles/continuousIntegration.html (visited on 06/04/2019).

Appendix. *Figures*

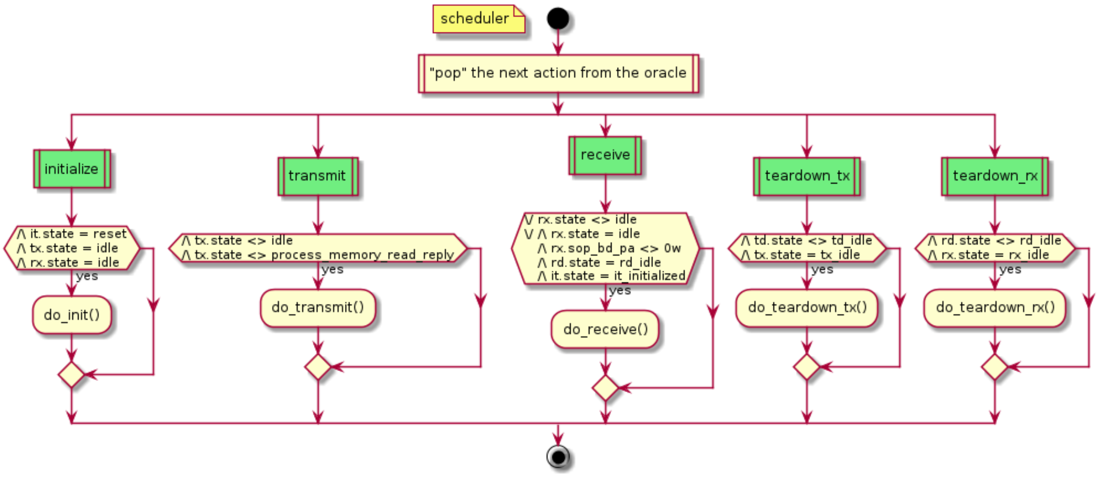

Figure A.1: Flowchart of the scheduler of the NIC model. Green nodes represent condition statements, here they represent the value of the popped action from the oracle. The full dot represents the entry point, and the other point the exit point.

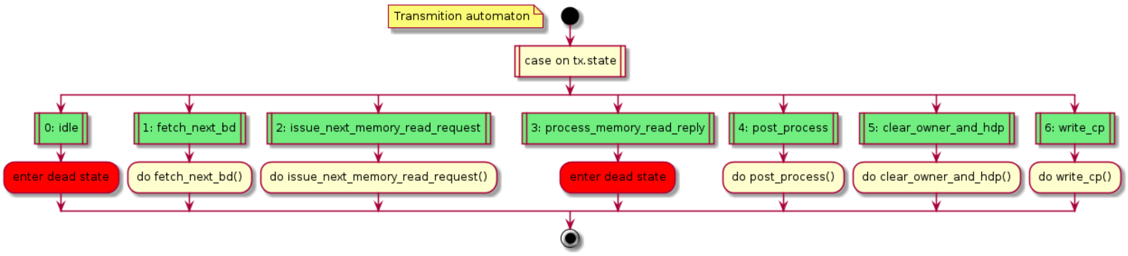

Figure A.2: Flowchart of the transmission automaton of the NIC model. Green nodes are similar to the ones of Figure A.1. Red nodes represent non autonomous transitions leading to dead states.

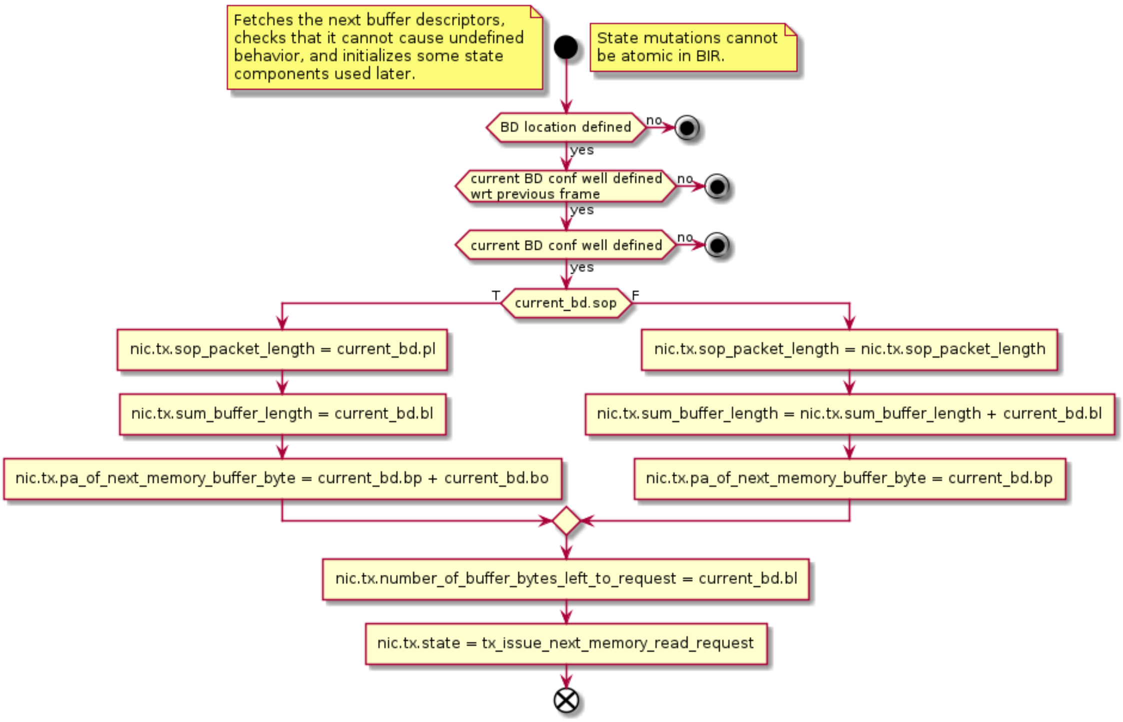
Fetches the next buffer descriptors, checks that it cannot cause undefined behavior, and initializes some state components used later.
State mutations cannot be atomic in BIR.
BD location defined
no
yes
current BD conf well defined wrt previous frame
no
yes
current BD conf well defined
no
yes
current_bd.sop
T
F
nic.tx.sop_packet_length = current_bd.pl
nic.tx.sum_buffer_length = current_bd.bl
nic.tx.pa_of_next_memory_buffer_byte = current_bd.bp + current_bd.bo
nic.tx.sop_packet_length = nic.tx.sop_packet_length
nic.tx.sum_buffer_length = nic.tx.sum_buffer_length + current_bd.bl
nic.tx.pa_of_next_memory_buffer_byte = current_bd.bp
nic.tx.number_of_buffer_bytes_left_to_request = current_bd.bl
nic.tx.state = tx_issue_next_memory_read_request

Figure A.3: Flowchart of the `fetch_next_bd` transition of the transmission automaton of the NIC model. The full dot represent the entry point, ⊗ represents the exit point and the other dots are shorthands to represent dead transitions (the symbols have been changed because of technical limitations of the tool used to draw the diagram).