Universität Leipzig

Fakultät für Mathematik und Informatik
Abteilung Technische Informatik

Model Checking Information Flow Control Policies on Instruction Set Architectures

Masterarbeit

Leipzig, Februar, 2020

vorgelegt von:
Felix Linker
M.Sc. Informatik

Betreuender Hochschullehrer:
Prof. Dr. Martin Bogdan
Abteilung Technische Informatik

Zweitbetreut durch:
Jörn Hoffmann
Abteilung Technische Informatik

Abstract

This thesis proposes an approach to formally verifying instruction set architectures (ISAs) against higher-level properties using the model checker nuXmv. This was first proposed by [Rei17]. We use nuXmv to perform information flow tracking at architecture level. Thus, the higher-level properties will be given by information flow properties. The concepts behind information flow tracking stem from [Fer+17] where information flow control (IFC) was applied to hardware description languages (HDLs).

The threat model that is assumed in this thesis is that user-mode is adversarial to machine-mode and wholly compromised. In this scenario, we consider whether a) user–mode can gain access to confidential data held by machine-mode and b) user-mode can gain control over machine-mode. Timing channels are excluded.

We develop a simplified version of the RISC-V architecture, the MINRV8 architecture, to which the aforementioned approach will be applied. Three information flow properties applying to this architecture will be developed and verified using nuXmv. The result of this are eight assumptions that grant the absence of any information flow property violation by any program running on the MINRV8 architecture. These results are tested by showing that our approach can detect the cache poisoning [WR09] and SYSRET vulnerabilities [12a; Dun12] applying to the x86 architecture without any manual intervention besides the implementation of the vulnerabilities.

Acknowledgements

First and foremost, I would like to thank Alastair Reid, formerly employed at Arm Ltd., now working for Google Research, for all the time and help he put into this thesis. I was fortunate to be able to continue the work he and I did during my internship at Arm. I am very grateful that, even after my internship at Arm ended, he put his free time into my work such that I was able to write a thesis in the field of formal verification. Without his guidance, I would not have been able to write, not even start this thesis.

Secondly, I want to thank Jörn Hoffmann of the department of Technische Informatik, who agreed to supervise me at Leipzig University. Without having known me or having a direct affiliation with the topic of my thesis, he allowed me to write this thesis officially and gave guidance whenever I was not sure about the direction of my thesis.

Also, I want to thank Prof. Dr. Andreas Maletti of the department of Algebraische und logische Grundlagen der Informatik at Leipzig University. When I was working on my thesis but did not yet have supervision by someone at my university, he helped me to keep motivated whenever I felt lost because of the physical distance to my unofficial supervisor.

And last but not least, I want to thank my proofreader and friend Friedrich Schöne who not only read my thesis but also contributed towards it by friendship and exchange.

Contents

Acronyms

x**cause** exception cause

x**epc** exception program counter

x**ie** interrupt enable

x**ip** interrupt pending

x**status** status

x**tval** trap value

x**tvec** trap-vector base-address

2BMC Bounded Model Checking-Based Model Checking

ABI application binary interface

AEE application execution environment

ALU arithmetic logic unit

API application programming interface

AUIPC add upper immediate to PC

BDD Boolean decision diagram

BMC Bounded Model Checking

CEGAR Counterexample Guided Abstaction Refinement

CISC complex instruction set computer

CSR control and status register

CSRRW atomic read/write word CSR

CTL computation tree logic

EBREAK environment breakpoint

ECALL environment call

hart RISC-V hardware thread

HDL hardware description language

IFC information flow control

ISA instruction set architecture

LR link register

LTL linear temporal logic

mcause machine cause

medeleg machine exception delegation

mepc machine exception program counter

mideleg machine interrupts delegation

mie machine interrupt enable

mip machine interrupt pending

MMU memory management unit

MRET machine trap return

mscratch machine scratch

mstatus machine status

mtval machine trap value

mtvec machine trap-vector base-address

OS operating system

PC program counter register

PMA physical memory attribute

pmacfg physical memory attributes configuration

PMP physical memory protection

pmpaddr physical memory protection address

pmpcfg physical memory protection configuration

PSL property specification language

RISC reduced instruction set computer

SBI supervisor binary interface

SEE supervisor execution environment

SP stack pointer register

SRL shift right logically

ucause user cause

uepc user exception program counter

uie user interrupt enable

uip user interrupt pending

uscratch user scratch

ustatus user status

utval user trap value

utvec user trap-vector base-address

Part I.

Introduction

Chapter 1.

Introduction

This thesis is in the realm of formal verification, i.e. the attempt to verify a system by the use of formal methods such as SAT solvers[1], interactive theorem provers and model checkers. With these tools, formal verification engineers strive to prove the correctness of systems such as general computer programs, operating systems (OSes), compilers or hardware designs. For a system to be correct means that it complies with a specification. Formal verification of a system is the attempt to prove that it is free of errors, i.e. meets all properties imposed by some specification. "Specification" here might refer to documents specifying standards of the industry, but it might also refer to more abstract properties like the absence of memory leaks or race conditions in parallel programming. Thus, formal verification complements testing. The relation between these two approaches often is illustrated by a famous quote of Edsger Dijkstra:

> "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dij72, p.6]

Whereas testing is a quick and efficient way of finding bugs in the development process of a system, formal verification is a more complex but complete (regarding the properties verified) process of proving the absence of bugs.

In "Who Guards the Guards? Formal Validation of the Arm V8-m Architecture Specification" [Rei17], Reid stressed the need for verifying specifications themselves as opposed to simply verifying implementations against specifications. To tackle this, he proposed to check specifications against higher-level properties and used this methodology to verify the specification of the ARM M-Class architecture.

The following analogy by Reid gives an intuition for this line of research: think of a specification as the law and of an implementation as some action that is touched by the law. In our everyday life, we rely on everybody to obey the law in their activities. If we want to find out whether some action is just (correct), it is checked whether it violates the law. The same applies to implementations. If we want to check whether some implementation is correct (just), it is checked whether it complies with the respective specification. However, the law can have loopholes. Sometimes what *we want* from the

[1] Tools that can solve the Boolean satisfiability problem, i.e. whether a formula of propositional logic has a model.

law does not align with what it actually says. This is because the law is big and complex, and some cases get overlooked. Therefore, there also is a constitution that expresses the fundamental properties of each and every statute of the law. To carry this analogy over to the world of formal verification: the constitution is given by the higher-level properties the specification itself is verified against.

In this thesis, we continue two lines of Reid's research: a) the main contribution of this thesis also is to propose an approach for verifying specifications against higher-level properties and b) this methodology will be applied to an ISA specification as well.

The higher-level properties an ISA will be verified against, mainly stem from the work of Ferraiuolo et al. in "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis" [Fer+17]. As the title indicates, their work also falls into the domain of verification; the authors propose a new way of verifying HDL implementations. Their core idea is to use a type system in which types serve as security annotation of information. By applying typing rules to expressions in the HDL code, these annotations are propagated, and thus, information is tracked through the code. Certain type conversions, however, are prohibited and mark a security vulnerability. Thus, their approach can be summarized as tracking and controlling information flow in HDL code.

In this thesis, the idea of tracking and controlling information flow as proposed in [Fer+17] will be lifted to the level of the ISA by annotating the registers of the architecture with information flow labels that are propagated by instructions. The information flow control that in [Fer+17] was given by a type system will be implemented using high-level properties expressed in linear temporal logic (LTL) that constrain information flow generally; thus following the line of research of [Rei17].

We claim that the approach that will be developed in the course of this thesis is

- viable, i.e. the hardware requirements are low and the work can be reproduced with limited time investment,
- effective, i.e. the approach successfully uncovers issues in architectures, and
- supplemental, i.e. it enhances on related work such as [Rei17].

As a proof of concept, this approach will be applied to the RISC-V architecture using the model checker nuXmv. RISC-V is a recent and open-source ISA that has first been published in 2011 [Wat+11]. A basic architecture inspired by the RISC-V architecture will be modelled in nuXmv. nuXmv is a general-purpose model checker that supports different specification languages and verification algorithms. Additionally, using nuXmv allows for enhancing on the approach proposed by Reid in a key aspect: In his work, Reid focused on higher lever properties that were limited to making specifications about a single transition of the processor only, i.e. that only take the pre- and post-state of a single cycle of the processor into account. nuXmv, on the other hand, allows considering infinite sequences of instructions, i.e. sequences of processor-transitions, of unbounded length.

1.1. Thesis Structure

In chapter 2, the background of this thesis will be introduced. This includes both key papers of Reid and Ferraiuolo et al. as well as ISAs and model checkers. An introduction to reduced instruction set computer (RISC) architectures and their ecosystem will be discussed, concluding with a threat model for this thesis. After this, model checkers will be explained. At the end of this chapter, the methodology of this thesis will be discussed in more detail.

In chapter 3, the RISC-V architecture and a new, minimal, RISC-V-inspired architecture called MINRV8 will be developed. It will be described how the MINRV8 architecture was implemented in nuXmv, following up on this.

Chapter 4 details what information flow tracking means in the context of an ISA, how specifically it can be applied to MINRV8, and how it was implemented in nuXmv, i.e. it will be discussed how we applied the work of Ferraiuolo et al. [Fer+17] to ISAs leading finally to the higher-level properties that implement information flow control for the MINRV8 architecture.

The evaluation of our approach is given in chapter 5-8. First, we will present the results of the verification process in chapter 5. Then, we discuss these results in chapter 6. In chapter 7, we will compare the results and the approach in general to related work, and in chapter 8, we will give a final conclusion.

1.2. Contributions

The contributions of this thesis are twofold. Firstly, a general and architecture-independent approach to verifying ISAs specifications will be developed that relies on a model checker and higher-level information flow properties. This approach takes unbounded numbers of instructions into account, is non-redundant and is meant to be used to detect vulnerabilities that must be combatted by architectural changes or to verify rules, e.g. OSes and compilers can obey to be not vulnerable themselves. These rules will be:

- practical and verifiable themselves,
- concise, and
- stable.

Secondly, this approach will be applied to the MINRV8 to give a prototype. This prototype will be able to detect both the cache poisoning and the SYSRET vulnerability the x86 architecture was or is susceptible to. Whereas the cache poisoning attack was an actual vulnerability to the x86 architecture, the SYSRET vulnerability affected e.g. OSes running on Intel's version of the x86 architecture. Therefore, the latter vulnerability is a prime example of why verifying ISAs must also take into account software running on it. Hence, it is taken into account that not every vulnerability can be fixed by changing the specification.

Chapter 2.

Background

In the introduction, the core motivation of this thesis was given, which is to formally verify ISAs by model checking information flow control properties. In this section, the background of this thesis will be set up.

The first two sections, 2.1 and 2.2, will introduce the foundational work of Reid [Rei17] and Ferraiuolo et al. [Fer+17]. The actual subject of verification, ISAs, and the technical details of the verification process will be outlined, following up on this. There will be not only an introduction to common ISAs, especially RISC-V, (section 2.3) but also to the ecosystem of processors which allows to set up a threat-model (section 2.4). Then, a general explanation of model checking will be given, resulting in the description of the nuXmv model checker that will be used in this thesis (section 2.5).

In each of these subsections, a collection of ISAs or model checkers, respectively, will be touched to a) give a better feel for the field as a whole and b) meaningfully argue for one of the presented options to work with in this thesis. However, the sections on RISC architectures other than RISC-V and model checker other than nuXmv can be skipped by readers that are more interested in the results of this thesis.

As a final part of the background, in section 2.6, a summary of all aforementioned sections will be given, which will allow for an outline of this thesis's methodology. This methodology has been sketched in the introduction but can be given precisely only after the background has been introduced.

2.1. Formal Verification of Specifications

As already mentioned, the work of Reid in "Who Guards the Guards? Formal Validation of the Arm V8-m Architecture Specification" [Rei17] focusses on verifying specifications, in particular ISAs, against high-level properties. The key motivation of this work is best summarized by the conclusion of the paper:

> "Formal verification of programs is becoming more and more practical but, if the verification is to be meaningful, it must be based on correct architecture specifications for the hardware that the programs run on. That is, the specifications are a critical part of the Trusted Computing Base. Unfortu-

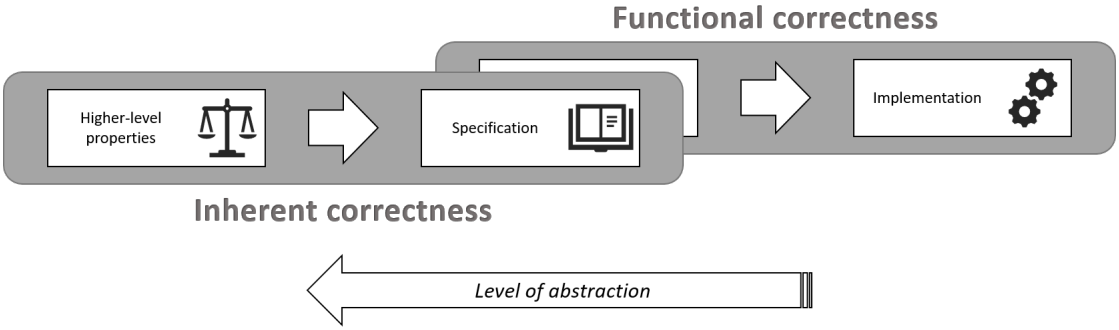


Figure 2.1.: Verification of specifications in relation to verification of implementations

> nately, the size and complexity of architecture specifications is such that it seems inevitable that specifications will contain bugs and our previous work confirms this supposition.
>
> While it is common to debug specifications by *testing* the specification, this paper proposes a different approach: we define a set of formal properties that should hold for the specification and we *formally verify* that the architecture specification satisfies these properties. We think of the relationship between the properties and the specification as being like the relationship between a nation's constitution and a nation's laws: the constitution is concise enough that everyone can read them while the laws are too large for effective review; the constitution can be used to test whether existing or proposed laws are compatible with high-level goals; and the constitution is stable and changes very, very slowly." [Rei17, p.88:22]

The relation between verifying specifications against higher-level properties and verifying implementations against specifications is depicted in figure 2.1. This figure illustrates that both of these levels of verification are parallel to each other: Higher-level properties take the same role for specifications as specifications do for implementations.

In the work of Reid, the high-level properties are given by "cross-cutting features" [Rei17, p.88:2] which have the following properties:

> "The central design challenge we face is to create a set of properties that:
>
> - express the major guarantees that programmers depend on;
> - are concise so that architects can easily review and remember the entire set of properties;
> - are stable so that architectural extensions do not invalidate large numbers of rules;
> - and that describe the architecture differently from existing specification to reduce the risk of common-mode failure." [Rei17, pp.88:2-3]

.

Cross-cutting features have the downside that they are usually hard to grasp for humans since they require a thorough understanding of the system as a whole, yet are powerful since they often (indirectly) touch many aspects of the specification.

The aspects of the conciseness and stability of the properties implement the idea of a "constitution" for an ISA linking back to the paragraphs quoted above. The first point, however, plays a greater role for the practical applicability of the verification process. If properties, the architecture is verified against, express "major guarantees that programmers [can] depend" on, they can be used to aid in the usage of the platform, e.g. by OS engineers to utilize the mechanisms of the architecture correctly and thereby to write secure OSes. Lastly, the requirement aimed at reducing the "risk of common-mode failure" aims at combatting the downsides of other approaches to verify specifications. Reid lists three commonly used ways of verifying specifications: "by testing specifications against existing implementations; by testing specifications using testsuites used to test implementations; or as a side effect of attempting to formally verify an implementation against a specification" [Rei17, p.88:2]. These approaches, however, all face the risk of issues with the specification being undetected because the implementation and specification both are affected by the same issue, i.e. the risk of issues being undetected because of common-mode failure. This risk can be mitigated if the properties the specification will be verified against actually express something *new*.

Reid applies his approach to the ARM M-Class architecture, for which not only a natural language but also a machine-readable specification is available. The machine-readable specification of the ARM M-Class architecture provides the function `TopLevel` which implements the transition of the architecture from one state to another, i.e. a cycle of the processor. The natural language specification on the other hand mainly consists of rules that constrain the behaviour of the architecture in different situations, i.e. constrain the transition relation between architectural states that implicitly is given by the `TopLevel` function.

In principle, these two forms of specification also face the risk of common-mode failures. In practice, however, it turned out that the natural language specification included several rules that described the behaviour of the architecture on a very high-level, thus, meeting aforementioned requirements. Reid was able to find 12 bugs in the ARM M-Class specification that despite prior testing had been undiscovered up to this point using these rules as high-level cross-cutting properties.

In summary, the aspects of [Rei17] that are relevant for this thesis are:

a) the paper argues for why the verification of specifications themselves is important; and

b) the paper proposes requirements for properties to verify an ISA against.

Additionally, the paper successfully uses its approach to verify the ARM M-Class architecture partially. A downside of this work, however, is that it is limited to verifying single processor transactions only. This approach was chosen due to the nature of the

properties from the natural language specification. These only constrain single transitions of the processor. To meet these, the formalized properties come in two forms: invariants and properties. These two types of properties form an inductive proof. First, it is proven that the initialization procedure of the ARM architecture establishes the invariants, then, it is proven that assuming the invariants no transition of the processor either invalidates an invariant or violates an actual property. Next to resembling the structure of the natural language specification, this approach promises to be more performant than unbounded model checking.

In this thesis, the work of [Rei17] will be continued: high-level properties will be established to verify ISAs. The work in this thesis, therefore, also is subject to the requirements that have been imposed on the higher-level properties in [Rei17]. However, the work of [Rei17] will be enhanced by verifying properties applying to multiple processor transitions, i.e. by performing unbounded model checking. This enhances three downsides of bounded model checking:

1. Counter-examples to properties of [Rei17] are states that also implicitly give a transition into the next state, i.e. an instruction to be executed. This makes a counter-example difficult to understand as this state is just some of many during operation of the architecture. In contrast, counter-examples in this thesis will be given by an initial state and a series of instructions executed subsequently from this state on, i.e. an actual program. It, therefore, will be easier to understand what exactly happens when a property is violated.

2. Unbounded model checking gives higher assurance of completeness of the model. In [Rei17], the verification procedure ultimately sets up an inductive proof that allows making the statement: There is no state that either invalidates an invariant or violates a property. In this thesis, the verification procedure will allow for the following statement to be made: There is no program that violates a property. This is much broader and there are fewer abstractions to grasp in order to understand what the verification procedure actually does.

3. Most importantly, the properties in this thesis will be more expressive. While in [Rei17], bounded model checking does limit the scope the properties apply to, the properties themselves are limited - they can only constrain a single transition at a time. In this thesis, however, ideas like: "If first X and then Y happens, P must hold." which can hardly[1] be expressed using bounded model checking.

[1] One can think of ways to implement such a property using bounded model checking, e.g. first establish a post-condition for when X happens and then assume this post- as a pre-condition for Y happening. However, in this a scenario, finding such conditions is non-trivial, it must be further verified that the respective conditions are adequate, e.g. they might be to weak, and establishing these conditions would introduce the need to reverse-engineer how certain actions influence the state. This would make them instable and non-concise and, thus, conflict with the self-imposed requirements for the properties.

2.2. Information Flow Control

In "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis" [Fer+17] by Ferraiuolo et al. the authors verify an implementation of the ARM TrustZone architecture by statically type-checking an information flow policy. To achieve this, they enhance the already existing HDL language SecVerilog, extending its type system to allow more fine-grained modelling of information flow policies. Therefore, the authors verify an implementation against higher-level properties as opposed to verifying a specification as done by Reid in [Rei17]. This thesis takes this general approach and lifts it to be applied to specifications rather than HDLs.

HDLs are very much like ordinary programming languages only that they are used to specify electronic circuits. This use case introduces several novelties to HDLs in contrast to programming languages, e.g. type systems are not as rich and usually comprise registers (multiple bits) and wires (one bit). i.e. they resemble much more low level systems. In the context of this thesis, think of an HDL as a programming language with a very specific use case.

Ferraiuolo et al. define an *information flow policy* as a lattice of security labels $(M, \sqcup, \sqcap)$; with $\sqsubseteq$ being the partial order on M as induced by the lattice operations supremum $\sqcup$ and infimum $\sqcap$. Intuitively speaking, information labelled with $a \in M$ is allowed to flow to a sink labelled with $b \in M$ if and only if $a \sqsubseteq b$, i.e. information is always allowed to flow up in the lattice but not down.

Example 1. To verify an ARM TrustZone implementation, Ferraiuolo et al. introduce four information flow labels $\Sigma = \{\mathrm{PT}, \mathrm{PU}, \mathrm{CT}, \mathrm{CU}\}$ which stand for public/trusted, public/untrusted, confidential/trusted and confidential/untrusted. These labels form the information flow policy lattice $(\Sigma, \sqcup, \sqcap)$ which is depicted in figure 2.2 as a Hasse diagram. As we can see, each of these four labels comprises two tokens from two domains of the information flow policy. The first of these domains is the one of *confidentiality*, i.e. each label states that a piece of information is either public (P) or confidential (C). The second of these domains is the one of *integrity*, i.e. each label states that a piece of information is either trusted (T) or untrusted (U) where "trusted" marks whether the values we deal with are trusted or in other words not malicious.

It can be seen in the diagram depicting the lattice of security labels (fig. 2.2) that public information can flow to confidential sinks but not vice versa and that trusted information can flow to untrusted sinks but not vice versa. Intuitively, this means that trusted parts of the architecture must only receive trusted data and that confidential data must be handled by confidential parts of the architecture only.

To implement the actual *control* of such an information flow policy, Ferraiuolo et al. introduce the HDL language SecVerilogBL that builds upon the HDL SecVerilog. The core of both of these languages when it comes to information flow control is a type-system that adds syntax to the core HDL allowing for annotating variables with security labels. This enables verifying an information flow policy by static type-checking. In SecVerilog, variables are typed by labels of the respective information flow policy. The type of a variable is described either by a label $l \in M$ directly, by the conjunction $\sqcap$ or disjunction $\sqcup$

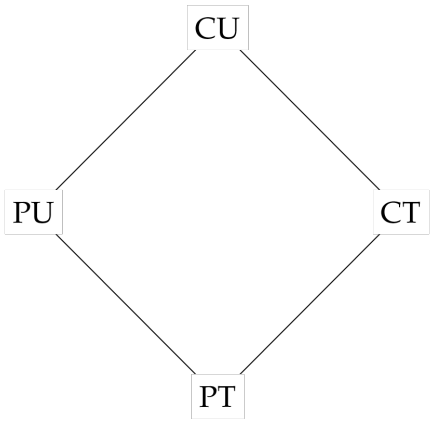


Figure 2.2.: A lattice of security labels for verifying an ARM TrustZone implementation displayed as Hasse diagram [Fer+17]

of two labels or by a type-level function f that maps some program variable v to a type. SecVerilogBL enhances on this by tracking security labels not only per variable but also per bit and array element, therefore, allowing for more fine-grained annotation of the HDL code but also adding a mechanism to downgrade information. Since information is not allowed to flow "down" in the lattice, downgrading can be understood best as a typecast that allows such behaviour. In some HDL models, it might be necessary to permit insecure flows of information (in regard to the information flow policy). In those cases, a programmer can use `downgrade(e, τ)` to annotate the expression `e` as being of type τ, regardless of any prior annotations. This effectively *grades* the label of a bit of information *down* such that information is not routed to a sink annotated with a label smaller than the information.

In total, SecVerilogBL supports seven ways to express types:

$$\text{Types } \tau ::= l \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2 \mid x \mapsto \tau \mid f(x) \mid \mathit{if}\ e^{\tau}\ \tau_t\ \tau_f \mid \mathit{case}\ e^{\tau}\ \tau_1 \ldots \tau_n$$

As can be seen above, types are either given as simple labels of the information flow policy $(M, \sqcup, \sqcap)$, as the disjunction or conjunction of two types, as a mapping from an index variable x to some type, as the application of some program variable x to the function f or as an *if*- or *case*-expression where e^{τ} is a pure-expression, i.e. side effect free. Types of the form $f(x)$ are called *dependent types*. e^{τ} in *if*- or *case*- expressions might also contain program variables; if it does, these expressions also form dependent types.

Types are of a certain kind:

$$\text{Kinds } k ::= l \mid \texttt{int} \to k$$

Any type is of kind l. Furthermore, the type $x \mapsto \tau$ is of kind $\texttt{int} \to k$ where k is the kind of τ. However, when type-checking is performed, types of kind l are lifted to be of kind $\texttt{int} \to l$ such that every type is a partial function from bit indices to information flow labels. This construct spares programmers from writing type maps for arrays and

bit vectors where the type does not change per bit. This way, a constant type τ of kind l can be applied to a whole bit vector as for type-checking, τ is lifted to $i \mapsto \tau$ which is of kind `int` $\rightarrow l$.

Example 2. The following snippet shows the type annotation of parts of a register file definition in SecVerilogBL:

```
1 reg [0:31] {world(ns)} read;
2 reg [0:31] { i -> j -> world(reg_ns[i]) } mem[0:1023];
3 reg {PT} reg_ns [0:1023];
4 ...
5 if(ns == 1) begin
6     read = (reg_ns[read_addr] == 1) ?
7         mem[read_addr] : 32'b0;
8 end else begin
9 ...
```

Snippet 2.1: A register file code segment [Fer+17]

In this example, the variable `ns` refers to the *security state* of the ARM TrustZone architecture. The ARM TrustZone architecture supports not only different privilege but also security states which can either be secure-state (meant to handle confidential data) or non-secure-state (meant to handle only non-confidential data). The variable `ns` in the implementation of the ARM TrustZone architecture by Ferraiuolo et al. indicates whether the architecture currently is in secure- or non-secure-state.

The variables `read` and `reg_ns` illustrate the purpose of lifting types to higher kinds. Both types `world(ns)` and `PT` are of kind l, yet they are used to annotate variables that store more than one bit. Because `read` is a register with 32 bits, `world(ns)` is lifted to `i -> world(ns)`, i.e. each element of `read` is of the dependent type `world(ns)`. Similarly, since `reg_ns` is an array of 1024 registers of width 1, `PT` is lifted to `i -> j -> PT`, i.e. each array element is of type `j -> PT` and each array element's bits are of type `PT`.

Starting in line 5, it can be seen how dependent types work in practice. The function `world` maps 0 to `CT` and 1 to `PU`. The `if`-statement first checks whether `ns` equals 1, in which case `read` would be of type `PU`. If `ns`, in fact, equals 1, furthermore, the register `read` shall be written with `mem[read_addr]` but only if the type of that also is of `PU` which is checked by ensuring that `reg_ns[read_addr]` also equals one. Since each element's type of `mem` depends on `reg_ns`, this assignment is safe since, obviously, a variable annotated with type `PU` can be written with a value of the same type.

If this check was omitted and `ns == 1`, `read` would be written with `mem[read_addr]` when `reg_ns[read_addr] == 0`, a variable of type `PU` would be written with a value of type `CT`. But since CT $\not\sqsubseteq$ PU, this would result in a type-checker error.

The details of this type-system are given by a collection of typing rules. These rules

are written as proofs of sequent calculus.

Example 3. Consider this example of a proof written in sequent calculus:

$$\frac{A \qquad B}{C}$$

This proof states that the formula C can be derived from the formulas A and B or more formally that: $A, B \vdash C$.

In the context of type systems, formulas used in proofs of sequent calculus often are so-called "typing judgements" [Fer+17] where some *environment* types some *expression*, e.g., if X is a type environment, x an expression and τ a type, one possible typing judgement would be $X \vdash x : \tau$ which means that x is of type τ under the environment X. In [Fer+17] three such environments are introduced:

Standard environment Γ maps variables to their respective type τ.

Width environment $\mathcal{W}$ maps variables to their respective bit vector length.

Kind environment Θ maps types τ to their respective kind k.

Ferraiuolo et al. introduce eight typing rules; these are for constant expressions and variables, logical and arithmetic expressions, bit vector concatenation and shifting as well as array indexing. An overview of some typing rules of [Fer+17] is depicted in figure 2.3[2]. The most simple type system rule probably is T-Const which types constant expressions n. These are always of type $\bot$, i.e. PT, and of the corresponding width w. The other type system rules share a pattern. They assume one or two expressions (e or e_1 and e_2) to be typed in $\Gamma; \mathcal{W}; \Theta$ and ensure that their corresponding types (τ or τ_1 and τ_2) are of the right kind in Θ. Then, they create a new type (τ or τ') and infer a type judgement for a new expression involving e, e_1 or e_2 and in some cases a constant n. This structure of type system rules follows the pattern of an inductive proof or inductive definition of a set where T-Const and T-Var (the typing rule for variables) are the start of induction.

When the SecVerilogBL code is being type-checked, it is verified whether all right sides of assignments are $\sqsubseteq$-greater than the respective left sides taken together with the label of the program counter. This means that information is only considered trusted if also the program counter can be considered to be trusted and data is considered to be confidential if either the data itself or the program counter is confidential.

In summary, the lattice of security labels taken together with the type system built on top of such a lattice implements three concepts:

- A lattice $(M, \sqcup, \sqcap)$ implements an information flow policy
- The typing rules implement information flow tracking
- Type-checking implements information flow control

[2]It will turn out that only these typing rules are relevant in context of this thesis.

$$\frac{}{\Gamma; \mathcal{W}; \Theta \vdash n : \bot, w}$$

(a) T-Const

$$\frac{\Gamma; \mathcal{W}; \Theta \vdash e_1 : \tau_1, w \qquad \Gamma; \mathcal{W}; \Theta \vdash e_2 : \tau_2, w \qquad \Theta \vdash \tau_1 : \texttt{int} \to l \qquad \Theta \vdash \tau_2 : \texttt{int} \to l \qquad \tau = i \mapsto \tau_1(i) \sqcup \tau_2(i)}{\Gamma; \mathcal{W}; \Theta \vdash e_1 \circ e_2 : \tau, w}$$

(b) T-Logical for $\circ \in \{\wedge, \vee\}$

$$\frac{\Gamma; \mathcal{W}; \Theta \vdash e_1 : \tau_1, w \qquad \Gamma; \mathcal{W}; \Theta \vdash e_1 : \tau_2, w \qquad \Theta \vdash \tau_1 : \texttt{int} \to l \qquad \Theta \vdash \tau_2 : \texttt{int} \to l \qquad \tau = i \mapsto \bigsqcup_{j \in [1,i]} (\tau_1(j) \sqcup \tau_2(j))}{\Gamma; \mathcal{W}; \Theta \vdash e_1 \circ e_2 : \tau, w}$$

(c) T-Arith for $\circ \in \{+, -\}$

$$\frac{\Gamma; \mathcal{W}; \Theta \vdash e : \tau, w \qquad \Theta \vdash \tau : \texttt{int} \to l \qquad \tau' = i \mapsto \texttt{if } (i < n)\ \tau(i - n + 1)\ \bot}{\Gamma; \mathcal{W}; \Theta \vdash e \ll n : \tau', w}$$

(d) T-LShift

$$\frac{\Gamma; \mathcal{W}; \Theta \vdash e : \tau, w \qquad \Theta \vdash \tau : \texttt{int} \to l \qquad \tau' = i \mapsto \texttt{if } (i > w - n)\ \bot\ \tau(i + n)}{\Gamma; \mathcal{W}; \Theta \vdash e \gg n : \tau', w}$$

(e) T-RShift

Figure 2.3.: A selection of typing rules for SecVerilogBL expressions [Fer+17]

Ferraiuolo et al. used these three concepts to verify an implementation of the ARM TrustZone architecture. In doing so, they first implemented the architecture as a prototype and used the information flow policy to ensure that it did not contain any bugs. Additionally, they deliberately added bugs to their implementation that have been found in other implementations of the ARM TrustZone architecture and checked whether they were able to find these bugs. It turned out that their approach was able to detect all bugs besides those which were related to downgrading, i.e. the approach of type-checking information flow control gives strong security assurances with the exception of bugs due to downgrading expressions.

In this thesis, the concept of an information flow policy, information flow tracking and information flow control will serve to define higher-level properties to verify an ISA against. To achieve this, it must be tackled how such an information flow policy as proposed by Ferraiuolo et al. can be applied to ISAs such that a) the labels can be tracked throughout the architecture by a model checker, and b) the policy can be controlled by the model checker., i.e. how can the two concepts of information flow tracking and control be implemented using a model checker?

2.3. RISC-Architectures

Computer Organization and Design: The Hardware/Software Interface defines the term ISA as follows:

> "One of the most important abstractions [in computer design] is the interface between the hardware and the lowest-level software. Because of its importance, it is given a special name: the instruction set architecture, or simply architecture, of a computer. The instruction set architecture includes anything programmers need to know to make a binary machine language program work correctly, including instructions, I/O devices, and so on. Typically, the operating system will encapsulate the details of doing I/O, allocating memory, and other low-level system functions so that application programmers do not need to worry about such details." [PH13, p.22]

Modern architectures all provide instructions of these categories: instructions for general computation, i.e. bit vector arithmetic, load/store instructions and platform management instructions.

As a target of verification, this thesis will work with RISC architectures. In *Computer Architecture: A Quantitative Approach*, RISC architectures are defined by three main concepts [HP12, p.C-4]:

- RISC architectures are *load/store architectures*, i.e. only load and store instructions affect or touch memory
- All other instructions work on registers only and always modify the full width of registers

- There are few instructions and their encoding is mostly homogenous

The last aspect coined the name *reduced* instruction set computer architecture. RISC architectures are often times understood as the counterparts to complex instruction set computer (CISC) architectures which describe all architectures that do not implement aforementioned RISC-properties. The most prominent example of a CISC architecture is the x86 architecture which is implemented by most processors manufactured by Intel.

In this section, three RISC architectures will be introduced: the ARM architecture, MIPS and RISC-V. It will turn out that these architectures can mainly be differentiated by the balance of customizability vs feature-richness. Whereas the RISC-V architecture is most customizable, it also supports the fewest features (not including custom extensions). On the other hand, the ARM architecture is very feature-rich but not very much customizable. The MIPS architecture strikes a middle-ground of these two extremes.

The goal of this section is to illustrate the features and fundamental design concepts that come with contemporary RISC architectures. The decision towards focusing on RISC architectures as a target of verification was made because their simpler approach to ISAs makes them far easier to analyze and model than CISC processors. Using a RISC architecture will allow for implementing an architecture in scope of this thesis while the results will still be applicable and comparable to real-world ISAs that are in use as of today. Introducing more than one RISC architectures will allow us to choose parts to exclude in the verification model without harming our results.

2.3.1. Arm

The ARM architecture originally was developed by Acorn Computers Ltd in the 1980s, which later became Arm Ltd [Wal15]. In the beginning, Arm stood for "Advanced RISC Machines" but has become a self-contained name by now. The architecture's most prominent use-case lies in mobile computing, e.g. tablets, smartphones and smartwatches. The ARM architecture defines three architecture *profiles*:

- The generic *application profile* ARMv8-A,
- the *real-time profile* ARMv8-R designed for deployment scenarios in which real-time responses from a processor is crucial, e.g. in aviation,
- and the *microcontroller* profile ARMv8-M aimed at embedded systems.

Here, the ARMv8-A profile will be put into focus. The *Arm Architecture Reference Manual* [19] introduces the ARM A-class architecture as a RISC architecture with three key properties:

> "[...]
>
> - A large uniform register file.
> - A load/store architecture, where data-processing operations only operate on register contents, not directly on memory contents.

- Simple addressing modes, with all load/store addresses determined from register contents and instruction fields only." [19, p.A1-34]

These properties already meet most of the aspects of RISC architectures as defined in [HP12]. When it comes to the aspect of a homogenous instruction set, the ARM architecture makes some trade-offs on this aspect to support the three profiles available. The ARM architecture supports three instruction sets: A64, a fixed-length 32-bit encoded instruction set for 64-bit systems, A32, a fixed-length 32-bit encoded instruction set for 32-bit systems, and T32, a variable-length instruction set that uses both 16- and 32-bit encodings to support compressed instructions. The ARMv8-A architecture eases the challenge of correctly decoding these different instruction sets by the concept of *execution states*. The ARMv8-A supports two of these execution states: AArch64, a 64-bit execution state, and AArch32, a 32-bit execution state. An ARM A-class processor can receive A64 instructions when in the AArch64 execution state and A32 or T32 instructions when in the AArch32 execution state.

The ARMv8-A specification defines how the processor interacts with memory which includes the specification of a cache and a memory model covering virtual addresses, memory access order control, memory access synchronization, and application-level memory restrictions. Furthermore, it is defined how processors work in a multiprocessor setup and how debugging of the architecture works. The ARMv8-A architecture also supports floating-point and vector arithmetic.

If these features do not suffice for a specific use-case, there are nine optional extensions available that can be supported by an ARMv8-A processor including an extension for accelerating cryptographic tasks, for enhancing the reliability, availability and serviceability of the processor, for monitoring and profiling, for enhanced vector arithmetic, and for memory partitioning.

As for registers, the ARMv8-A architecture supports the most versatile types of registers of all RISC architectures to be compared in this thesis. Depending on the execution state, 13 to 31 32- or 64-bit registers are supported. Additionally, there is a program counter register (PC), a dedicated stack pointer register (SP) and some exception link registers. A general-purpose register is used as the standard link register (LR). Furthermore, there are 32 64- or 128-bit registers for floating-point and vector arithmetic and a collection of control and status registers (CSRs) that hold status information of the architecture.

2.3.2. MIPS

As opposed to the ARM architecture, the MIPS architecture supports a smaller range of registers and fewer features by default, i.e. without extensions. It is introduced in *MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS64 Architecture* [14] as: "based on a fixed-length, regularly encoded instruction set [...] [which] uses a load/store data model, in which all operations are performed on operands in processor registers, and main memory is accessed only by load and store instructions" [14, p.21]. As for the aspect of a large, uniform register file, MIPS supports 32 32- or 64-bit registers

the first of which is hardwired to zero, a PC and some CSRs. Contrary to the ARM architecture, there are no dedicated LRs or SP. As indicated by the options for register width, MIPS supports both a 32- and a 64-bit architecture and instruction sets.

Historically, MIPS stands for "Microprocessor without Interlocked Pipelined Stages" and, as the ARM architecture, was developed in the 1980s at Stanford University [PH13, p.4.16-4]. In the past, MIPS was primarily used in workstations and servers, but as of today, its application has shifted towards embedded scenarios [14].

MIPS comes with fewer features (not including custom extensions) than the ARM architecture; the specification only touches memory interaction including memory virtualization and cache, debugging and the interaction with a standard coprocessor that implements floating-point arithmetic. MIPS, however, is far more customizable than the ARM architecture. Whereas in the case of the latter, customizability is only given by choosing different extensions, a MIPS processor does not need to implement all features defined in the specification to be MIPS compliant. Certain features can be left out, which is called *subsetting* of the architecture. Additionally to that, MIPS allows extending the architecture by user-defined instructions, compressed instructions, enhanced media processing, enhanced geometry processing, smart card or object interaction, signal processing, multi-threading, OS virtualization, vector arithmetic and extensions aimed specifically at microcontrollers.

2.3.3. RISC-V

RISC-V was originally presented in the technical report *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA* in May 2011 [Wat+11] and has undergone many changes since then. It was developed at the UC Berkeley to be a completely open ISA that suits a wide variety of applications and implementations by not "'over-architecting' for a particular microarchitecture style" [WA17a].

RISC-V's architectural specification is divided into two volumes: volume 1 specifies the base user-level ISA which is separated into several modules whereas volume 2 defines the privileged architecture and instruction set but is still only available as a draft. The privileged architecture is not divided into modules but rather *functional groups* that define certain aspects of privileged computing. Such groups include, for example, machine- and supervisor-level ISAs or the description of a platform level interrupt controller.

This thesis works with version 2.2 of volume 1 [WA17a] and version 1.1 of volume 2 [WA17b]. RISC-V understands itself as a highly customizable architecture. Currently, there are four variants of a base integer instruction set - one of which must be implemented by any RISC-V processor - and 13 optional extensions. The base integer instruction sets include instructions for integer arithmetic, memory operations (loads and stores), control transfer (jumps and branches) and platform management (environment calls and status register operations). The base integer instruction sets mainly differ in the word-size and register number. To cope with these differences, they also make small adjustments to some functionalities but only slightly change the instruction set itself. These four base integer instruction sets are given by:

M	Integer multiplication and division
A	Atomic instructions
F	Single-precision floating-point arithmetic
D	Double-precision floating-point arithmetic
Q	Quad-precision floating-point arithmetic
L	Decimal floating-point arithmetic
C	Compressed instructions
B	Bit manipulation
J	Dynamically translated languages
T	Transactional memory
P	Packed-SIMD instructions
V	Vector operations
N	User-level interrupts

Table 2.1.: All RISC-V standard extensions

RV32I Base integer instruction for a 32-bit architecture

RV32E Base integer instruction set for embedded computing

RV64I Base integer instruction set for a 64-bit architecture

RV128I Base integer instruction set for a 128-bit architecture

The RISC-V manual differentiates between standard and non-standard extensions but only introduces standard ones. A standard extension is meant to be compatible with any other standard extension and should be "generally useful" [WA17a] whilst non-standard extensions might add support for more niche use-cases and do not need to be compatible with all standard extensions.

The set of functionalities offered by standard extensions begins at a low level with the probably most mundane being the standard extension for integer multiplication and division. You can find a list of all standard extensions in table 2.1.

A subset of the RISC-V ISA is described by a so-called "ISA naming string" [WA17a] which comes in the following form:

⟨*naming-string*⟩ ::= 'RV' ('32' | '64' | '128') ('I' | 'E') ⟨*extensions*⟩

An example of a naming string is the very common ISA-subset RV64IMAFD which officially is abbreviated by RV64G. This subset includes the integer multiplication and division, user-level interrupts, atomic instructions, single-precision and double-precision floating-point arithmetic extensions.

A RISC-V core consists of multiple RISC-V hardware threads (harts). Each hart controls its own set of registers including a PC and is equipped with its own instruction fetch unit. However, multiple harts share the same memory.

A hart might either control 16 or 32 general-purpose registers which are named from x0 to x31. All but one instruction sets support 32 general-purpose registers whereas

the exception to this is the base instruction set aimed at embedded computing, RV32E, which only supports 16 general-purpose registers.

Similarly to the MIPS architecture, the first of these general-purpose registers is hard-wired to zero and for example can be used as a target register when the result of an instruction is not needed. Also, there is no dedicated LR or SP. RISC-V supports up to 4096 CSRs.

2.3.4. Summary

In the previous sections, it was shown how the ARM, MIPS, and RISC-V architectures implement the three main properties of [HP12] characterizing RISC architectures: a small and uniformly encoded instruction set, a large and uniform register file and a load-/store architecture. Whereas the ARM architecture supported customizability only by a list of extensions and MIPS added user-defined instructions and allowed subsetting its core specification, RISC-V took the idea of customizability and made it part of the core idea of the architecture. RISC-V supports the most simple yet fully specified processor in form of the RV32E. It was this degree of customizability and simplicity that led to the decision to use RISC-V as a target of verification in this thesis. There are no major features RISC-V lacks in that are crucial to the concept of a RISC architecture.

2.4. The Ecosystem of an ISA

ISAs are embedded in a larger context. The goal of this thesis is to verify an ISA against higher-level properties. For this undertaking to be worthwhile, it must be clear where the limitations of these properties lie.

An ISA is an interface offered by hardware which is used by binary programs and in particular OSes. It serves an analogous purpose to a programming language. The latter is used by programs and OSes as a specification of the input language to compilers. The former is used by compilers and OSes as a specification of the input language to the actual hardware. Yet, this thesis does not verify OSes, programs, compilers or hardware. With these boundaries in mind, it must be decided what will be incorporated in the model that will be developed over the course of this thesis.

2.4.1. Microarchitectures

The hardware implementation of an ISA is called *microarchitecture*. Such implementations usually incorporate sophisticated features that speed up operation of the chip such as pipelining. Instructions that are defined as part of an ISA are not atomic in regard to their execution; there usually are five steps that must be taken in order to execute an instruction, the so-called Von-Neumann cycle (cf. [PH13, p.286]):

1. Instruction fetch - load the instruction from memory
2. Instruction decode and register file read - Read the fetched instruction and read registers targeted by the instruction

3. Execution or address calculation - Use the arithmetic logic unit (ALU) to perform the instruction or to calculate a memory address

4. Data memory access - Access the memory for a read or write

5. Write back - Store the result of the instruction in a register

The microarchitecture comprises functional modules that implement each of these steps. To use the microarchitecture to the fullest, it is much more efficient to use these modules in a pipeline where for example after the first instruction has been fetched and handed to the module decoding it, the next instruction is fetched in parallel. In an ideal scenario, executing n instructions using a pipeline of length p only takes $n + p - 1$ cycles whereas without pipelining this would take $n * p$ cycles.

Such features introduce *sub-architectural state*. For example, pipelining introduces the *state* of the pipeline, which is sub-architecturally because it is independent of the ISA. An accurate model of an ISA that aims to be true to current implementations would need to include such features. However, since the goal of this thesis is to verify an ISA, the model to be verified must be implementation-independent. Furthermore, verifying such hardware features falls in the realm of functional correctness. Features such as pipelining face the risk that an implementation turns out to violate the ISA because instructions might execute not as they should. Adding to the first argument of the model not being implementation-independent, including sub-architectural state into the model would lead to an indirect verification of the functional correctness of such parts, i.e. whether they truly implement the ISA. This thesis, however, is not about verifying how pipelining can be implemented correctly but about *higher-level* properties of ISAs.

Additionally, aspects of the functional correctness of an implementation towards the ISA are served adequately by existing methods: In [Muk+16] attempts to verify hardware in general have been proposed and implementations of processors have been verified against higher-level properties in [Zha+15; BB94; Ber+98; TLM19] or against their actual specifications in the form of an ISAs in [Rei+16; Wol]. In particular, pipelining has been verified in [BD94].

2.4.2. Virtual Memory

Virtual memory is usually managed by OSes and mostly serves two purposes: "to allow efficient and safe sharing of memory along multiple programs [...] and to remove the programming burdens of small, limited amount of main memory." [PH13, p.428] Virtual memory is a system that replaces physical memory addresses with virtual ones. The virtual address space usually is bigger than the physical. Memory is divided into pages with a virtual address being made up of a page index and a page offset. On accessing an address, the page index is translated to a physical memory address which taken together with the page offset results in the respective physical memory address. This allows programs to treat their "personal" address space as continuous, whereas in reality, it comprises several pages of memory scattered all over physical memory.

Virtual memory usually is part of ISAs; e.g. the ARM architecture includes the specification of the memory management unit (MMU) which translates addresses and is programmed by an OS via system registers (cf. chapter D5 [19]). The RISC-V specification also includes a system for managing virtual memory with an address space of 32, 39, or 48 bits (cf. section 4 [WA17b]). Here, the setup of virtual memory also is delegated to an OS and can be configured by programming system registers.

Again, it is important that hardware modules that handle address translation work functionally correct. This has been covered in [DHP05] and will be left out in this context to keep the target of verification implementation-independent. It can be verified whether virtual memory management systems *correctly* isolate programs and correctly set up virtual memory. This problem, however, falls in the realm of OS verification and has been touched by [VS12] in detail. The reader is also referenced to the seL4 project [Kle+09] in which a complete kernel was formally verified.

With this research in mind, it was decided not to include virtual memory addresses in the model to be targeted for verification. Mechanisms that provide isolation between memory regions will be modelled. However, the model does not benefit from a simulation of virtual memory addresses. Addresses will be treated as physical ones throughout this thesis.

2.4.3. Threat Model

Now that two concepts that touch most architectures have been ruled out, namely subarchitectural hardware mechanisms or virtual memory, this section will give a positive description of what *will* be part of the model. This leads to the question of the threat model of the higher-level properties to be verified.

A key feature of modern architectures that is used by modern OSes is to have different privileges levels. For example, the RISC-V architecture supports up to three different privilege levels; from lowest to highest: user-mode, supervisor-mode and machine-mode. Higher privilege levels usually give access to more system control registers, specific instructions and oftentimes work hand in hand with specific semantics of system registers, e.g. RISC-V supports physical memory regions which support settings for each privilege level, i.e. certain regions of physical memory can be set readable to machine- but not user-mode, etc.

In a sense, privilege modes play a similar role to virtual memory. Both concepts are used by OSes in very fundamental ways and therefore are relevant for the security of a system as a whole. Yet, hardware only supports the foundation for virtual memory, i.e. address translation and the possibility to set properties of memory regions, whereas privilege levels and interrupts are systems offered to the OS and implemented in the hardware completely. OSes use memory management systems to offer virtual memory to applications but rely on privilege levels to function at all. While it was argued that verification of virtual memory systems should be targeted at OSes, the same cannot be said for the verification of privilege levels. Thus, this thesis will focus on the mechanics of privilege levels in the process of verification.

This links back to [Rei17]; recall that one goal of this work was to “express the major

guarantees that programmers depend on" [Rei17, p.88:2]. Since OSes heavily rely on correct implementations of privilege levels and especially the mechanics to transition between those, it is of specific interest to verify high-level properties about the workings of these mechanisms which will lead to insights in these guarantees.

Privilege levels go hand in hand with interrupt and exception handling. Interrupts generally are used in ISAs to transfer control to specific privilege modes or programs, e.g. whenever one presses a key on a keyboard, a keyboard interrupt is generated that is handled by the appropriate device driver and propagated by the OS to the correct application, e.g. your text editor or shell. Exceptions usually refer to error conditions that arise during computation. For example, coming back to physical memory regions in RISC-V, if user-mode would attempt to read from a region unreadable to it, an exception would be generated. It is the obligation of software running in machine-mode to handle such error conditions - usually, related software comes as part of an OS.

To put the mechanics of transitions between privilege levels to the test, the threat model taken as a basis of the verification process in this thesis is the following[3]:

> *Assume that user-mode has been completely compromised. Is there any way it can gain access to secret data handled by machine-mode or it can gain control over machine-mode?*

Intuitively, linking privilege modes back to OSes: Assume you have a malicious app on your computer. Is there any way it can use the ISA that is built into your computer to either access secret data or hijack your OS?

This goes beyond what can be verified purely in terms of hardware verification, as discussed in section 2.4.1. Even if it could be assumed that the hardware your computer is running on accurately implements the respective ISA - as long as the design of that ISA is flawed in the sense that it has security holes baked into it, i.e. into the application programming interface (API) that is used by the OS to give instructions to the hardware, any attempt to build a secure system is pointless[4].

In this threat model, timing channels are not considered. With recent microarchitectural-attacks (cf. Meltdown [Lip+18] and Spectre [Koc+19]) these become more and more relevant. However, when considering ISAs, they are of secondary importance.

2.5. Model Checking

Model checking is a technique that falls into the domain of formal verification. It is one way to prove that a given system complies with a given specification. Baier and Katoen introduce model checking in their book *Principles of Model Checking* [BK08] as:

[3] Since the decision to work with RISC-V in this thesis has already been made, the threat model was phrased with this in mind.

[4] Research verifying hardware against higher-level properties has been discussed in section 2.4.1. Without going into details on this research, the approach of this thesis still is an important addition to this research as it is more general, i.e. implementation-independent whereas hardware verification by its very nature is implementation dependent.

> "*Model-based* verification techniques are based on models describing the possible system behaviour in a mathematically precise and unambiguous manner. [...] This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration (model checking) to experiments with a restrictive set of scenarios in the model (simulation), or in reality (testing). [...]
>
> Model checking is a verification technique that explores all possible system states in a brute-force manner. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property. [...]
>
> Typical properties that can be checked using model checking are of a qualitative nature: Is the generated result OK?, Can the system reach a deadlock situation, e.g. when two concurrent programs are waiting for each other and thus halting the entire system? But also timing properties can be checked: Can a deadlock occur within 1 hour after a system reset?, or, Is a response always received within 8 minutes?" [BK08, p.7ff.]

Model checking, therefore, deals with two parts: Firstly, a model of some system, secondly, properties formalized on the basis of some specification. Systems to be model checked come in many forms. They range from software libraries over hardware designs to embedded controllers. The same is true for specifications. Those can be fully-fledged, *actual* specifications that describe the requirements to a system exhaustively or higher-level properties that generally should apply to systems such as deadlock freeness as mentioned in [BK08].

More technically, in model checking some system is taken and transformed into a model using a formal language like PROMELA (cf. section 2.5.1) and a specification is taken and transformed into a property usually expressed in some formal logic, e.g. LTL. Then it is verified via a model checker whether the system model models the formal property. Note that "model" in this context is ambiguous. On the one hand, this term refers to the model of the system as a simplified description; on the other hand, this term refers to the logical models-relation $\models$ which is actually being checked.

An overview of model checking is given in figure 2.4. There, the idea and purpose of model checking are depicted. In summary, model checking is a technique that allows solving the problem of ensuring that a system complies with a specification. By translating the system into a model and the specification into a formal property, this problem can be translated into *checking* whether the model *models* the formal property.

It has already been mentioned in the introduction of both the thesis and this section that the model checker to be used in this thesis is nuXmv. The aim of this subsection is to introduce nuXmv and two other tools that were taken into consideration for this thesis. These other two model checkers are called SPIN and SPACER. By introducing three model checkers, we want to give a feel for all the different flavours of model checkers

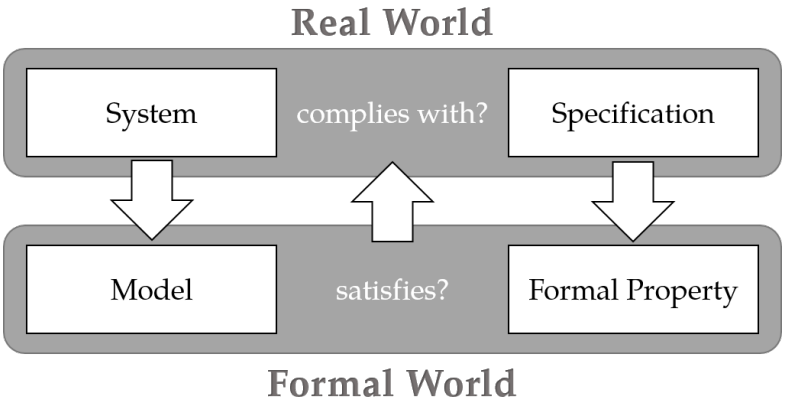


Figure 2.4.: Overview of Model Checking

and be able to argue for why nuXmv was chosen over others. However, this section does not intend to perform a thorough analysis of contemporary model checkers.

2.5.1. SPIN

SPIN (which stands for *S*imple *P*romela *IN*terpreter) is a model checker that has been originally developed by Bell Labs and made freely available in the nineties. As its name suggests, it uses PROMELA as input language. *The Spin Model Checker: Primer and Reference Manual* [Hol04] which has been used as the source for this section, describes SPIN initially as follows:

> "SPIN can be used to verify correctness requirements for systems of concurrently executing processes. The tool works by thoroughly checking either hand-built or mechanically generated models that capture the essential elements of a distributed system's design. If a requirement is not satisfied, SPIN can produce a sample execution of the model to demonstrate this." [Hol04, p.1]

As indicated by the quote, SPIN focusses heavily on the verification of distributed or parallel systems. This is reflected not only in its input language but also in the way how properties are expressed. For an introduction by example to PROMELA, cf. snippet 2.2. PROMELA relies on describing systems as sets of processes that run in parallel where parallel means that each process can advance its state (generally) independent of other processes[5].

In snippet 2.2, two processes of the same type `user` are declared in line 4. The idea of this snippet is to implement and verify an algorithm that grants these two processes

[5]We added the restriction *generally* to the claim because processes can be set up such that they deliberately wait for other processes to send them a message or set some shared state accordingly. However, such mechanisms where processes depend on each other always need to be implemented accordingly.

mutually exclusive access to the critical region spanning lines 25-27. This is ensured by the assertion in line 26 since variables in SPIN are always initialized to 0 - if two processes had access to the critical region at the same time, `cnt` would become 2 at some point.

The details of the protocol implemented in lines 8-23 are not relevant to this thesis. Their purpose is to grant mutual exclusive access to the critical region. However, this part of the code gives an indication of how models written in PROMELA look like. Besides `if` statements and labels similar to those in C (cf. line 7), PROMELA supports `do`-loops to control process execution flow. `do`-loops run indefinitely until they are exited manually by a `break` statement.

As data types, PROMELA supports three categories: processes, message channels and data objects. Data objects comprise atomic data types such as `byte`, `bool`, `int`, etc. as well as complex data type defined by `typedef` that define complex structures that have fields of data objects. Both atomic data types and complex data types are close to the data types of C.

Snippet 2.3 shows the output when verifying snippet 2.2 with SPIN where it is assumed that the latter snippet was written into a file called `mutex_flaw.pml`. The lines of the output are structured as follows: at the beginning of each line, a step index indicating the progress of all processes can be seen, whilst `proc X (NAME)` indicates the process that has made progress by its ID and name. The rest of the line `line X ... ↪[CODE]` shows the code executed by the process, along with the corresponding line number in the source file[6]. Notice, how processes 0 and 1 progress completely independently from each other, executing code on a line by line basis where each step of any process counts as a state transition of the whole model.

Assertions, however, are not the only way to express properties of PROMELA models for SPIN. Furthermore, there are:

- Labels
- Never claims
- Trace assertions

Labels have already been introduced informally along with snippet 2.2. To express properties about a model, certain types of labels can be used to give semantics to process state. For example, you can label a certain part of process code as end-state that might look like an idling-state to SPIN by default. Idling- and end-states are important to SPIN when it's checking for deadlock-freeness of systems. By default, if all processes are in an idling-state and wait for some kind of signal, this state is considered to be a deadlock. However, in some situations, it might be perfectly fine for some of the processes to idle in a specific state which should not contribute towards deadlocks. Labelling parts of code as end-state contributes towards this.

Never claims are processes themselves. They run like any other process but must not terminate otherwise they're regarded as a failure. This is the most complex way to express properties and in fact, falls together with writing LTL properties about a model

[6] In this case, line numbers do not fully align with snippet 2.2.

```
byte cnt;
byte x, y, z;

active [2] proctype user()
{
    byte me = _pid + 1;
again:
    x = me;
    if
    :: (y == 0 || y == me) -> skip
    :: else -> goto again
    fi;

    z = me;
    if
    :: (x == me) -> skip
    :: else -> goto again
    fi;

    y = me;
    if (z = me) -> skip
    :: else -> goto again
    fi;

    cnt++;
    assert(cnt == 1);
    cnt--;
    goto again
}
```

Snippet 2.2: Faulty Mutual Exclusion Algorithm Implemented in PROMELA [Hol04]

```
 1: proc  1 (user) line  5 ... [x = me]
 2: proc  1 (user) line  8 ... [(((y==0) || (y==me)))]
 3: proc  1 (user) line 10 ... [z = me]
 4: proc  1 (user) line 13 ... [((x = me))]
 5: proc  0 (user) line  5 ... [x = me]
 6: proc  0 (user) line  8 ... [(((y==0) || (y==me)))]
 7: proc  1 (user) line 15 ... [y = me]
 8: proc  1 (user) line 18 ... [((z = me))]
 9: proc  1 (user) line 22 ... [cnt = cnt+1]
10: proc  0 (user) line 10 ... [z = me]
11: proc  0 (user) line 13 ... [((x = me))]
12: proc  0 (user) line 15 ... [y = me]
13: proc  0 (user) line 18 ... [((z==me))]
14: proc  0 (user) line 22 ... [cnt = (cnt+1)]
spin: line 223 of "mutex_flaw.pml", Error: assertion
   ↪violated
spin: text of failed assertion: assert((cnt==1))
15: proc  0 (user) line 23 ... [assert((cnt==1))]
spin: trail ends after 15 steps
# processes: 2
    cnt = 2
    x = 1
    y = 1
    z = 1
15: proc  1 (user) line 23 "mutex_flaw.pml" (state 20)
15: proc  0 (user) line 24 "mutex_flaw.pml" (state 21)
2 processes created
```

Snippet 2.3: SPIN Example Output [Hol04]

which will be introduced in section 2.5.3. Therefore SPIN allows expressing such never properties in LTL directly via their command-line interface.

The last type of properties, trace assertions, solely deal with message passing and therefore are not relevant to this thesis.

It might be obvious at this point why we decided against using SPIN as the model checker of this thesis. Its focus on verifying distributed and parallel systems would make implementing an ISA in it very hard. In the *Primer and Reference Manual* for SPIN it is written:

> "[W]e saw that the emphasis in PROMELA models is placed on the coordination and synchronization aspects of a distributed system, and not on its computational aspects. [...] The specification language we use for systems verification is therefore deliberately designed to encourage the user to abstract from the purely computational aspects of a design and to focus on the specification of process interaction at the system-level." [Hol04, p.33]

However, the ISA we will attempt to verify a) will most likely not include components "interacting at the system-level" and b) will be verified on a component level, i.e. computationally as well - even in the case where multiple system-level components would be given.

Furthermore, it is unreasonable to assume that an implementation of instructions of an ISA could be implemented in a single statement of PROMELA. Yet, this should be the goal as otherwise as shown in snippet 2.3 what is regarded as state by SPIN would not fall together with what is regarded as state in an ISA. For an ISA, one typically would consider the *state* of registers and memory to be the state of the ISA whilst some instruction advances this state. However, for SPIN more complex state transitions would result in us not being able to differentiate architectural states of the ISA natively since the process implementing the ISA would change state on each line of code executed. This could be circumvented by using the `atomic` keyword offered by PROMELA which lets you wrap a group of PROMELA statements such that they're considered as one atomic statement that advances the process state only by one step. This, though, would lead to a model consisting of one process all of its code being wrapped by one `atomic` statement. This would massively contradict the key idea of SPIN of simple models being *abstracted* from computationally complex code. In this case, we would have skipped the whole step of abstracting from some computational model that has distributed components running in parallel. Not only could this lead to performance issues, but it also can be safely assumed that the work for this thesis would be cumbersome and might stumble over obstacles induced by abusing SPIN.

2.5.2. μZ & SPACER

μZ is a Datalog-engine that provides querying fixed points with constraints and has been proposed in [HBM11]. According to *An Introduction to Database Systems* [Dat00, p.790ff], Datalog is a descriptive and querying language that originated in the field of

database management systems. At its core, Datalog programs are sets of *rules* that combine predicates only using variables and constants to Horn-clauses, i.e. disjunctions with one positive literal at maximum. For example, consider the following rule π_0 which expresses the transitivity of a predicate P:

$$\pi_0 := P(a, b) \land P(b, c) \Rightarrow P(a, c)$$

Such programs are then used to *deduce* facts from the set of rules. To illustrate what this means, we introduce two other rules.

$$\begin{aligned}\pi_1 :=& P(0, 1) \Rightarrow \top \\ \pi_2 :=& P(a, b) \Rightarrow P(a + 1, b + 1)\end{aligned}$$

The Datalog-program $\Pi := \{\pi_0, \pi_1, \pi_2\}$ now induces the relation $<$ in the form of the predicate P on all natural numbers including 0. π_1 sets up a start of induction which, stating that 0 is smaller than 1. This is generalized for all natural numbers by π_2.

Whether or not this exact Datalog-program is supported by a given Datalog-engine depends on the extensions implemented by the respective engine. Our example relies on an extension for scalar operators since we use the $+$ operator.

As briefly mentioned, Datalog also supports queries. Datalog queries comprise only one predicate and a special head $?$.

$$q_0 := P(0, x) \Rightarrow ?$$

The result to a query is the set of all values for each variable that make the predicate true. In this case, the result set would be $\{1, 2, 3, \dots\}$. If no variables but only constants are given in the query, Datalog simply determines whether the given predicate can be derived for the given constants.

μZ now is a Datalog-engine that comes as part of the SMT solver z3 [MB08] and adds support for expressing Horn-clauses to it. For example, consider the implementation of the program Π for μZ as depicted in snippet 2.4. This example yields `sat` as a result when executed, meaning, that `goal` can be derived from the rules at hand.

SPACER [Kom+13] is an algorithm that combines two widely implemented approaches of currently available formal verification tools: Bounded Model Checking-Based Model Checking (2BMC) and Counterexample Guided Abstaction Refinement (CEGAR). In its implementation, SPACER uses μZ as a backend. The paper introducing SPACER, "Automatic Abstraction in SMT-Based Unbounded Software Model Checking" [Kom+13], describes those techniques as follows:

> "The key idea of 2BMC is to iteratively construct an under-approximation [(or refinement)] U of the target program P by unwinding its transition relation and check whether U is safe using Bounded Model Checking (BMC). If U is unsafe, so is P. Otherwise, a proof π_U is produced explaining *why* U is safe. Finally, π_U is generalized (if possible) to a safety proof of P. [...]
>
> The idea of [CEGAR] is to iteratively construct, verify, and refine an abstraction (*i.e.*, an over-approximation) of P based on abstract counterexamples." [Kom+13, p.1f]

```
 1  (declare-var a Int)
 2  (declare-var b Int)
 3  (declare-var c Int)
 4
 5  (declare-rel ge (Int Int))
 6  (declare-rel goal ())
 7
 8  (rule (ge 0 1))                        ; pi_1
 9  (rule (=>   (ge a b)                   ; pi_2
10              (ge (+ a 1) (+ b 1))))
11  (rule (=>   (and (ge a b) (ge b c)) ; pi_0
12              (ge a c)))
13
14  (rule (=>   (ge 10 15)
15             goal))
16  (query goal)
```

Snippet 2.4: Implementation of Π for μZ

The key to understanding how SPACER works is to understand what under- or over-approximation of programs are. For the sake of brevity, these concepts will be introduced by example. Consider the program P as depicted in snippet 2.5. P performs an integer division with remainder for a natural number `n` by a divisor `div` storing the results in `ratio` and `mod`. Intuitively speaking, $\hat{P}$ is an under-approximation (or abstraction) of P if for every part of P there is a corresponding part of $\hat{P}$ whose effects are logically entailed by the original part. Vice versa, P is an over-approximation of $\hat{P}$. An example for an over-approximation of P in snippet 2.7 and an example for an under-approximation in snippet 2.6. For abstracting P to $\hat{P}$ a couple of lines were removed whilst all other were left untouched[7].

With this example at hand, it can more easily be seen how 2BMC and CEGAR relate to under- and over-approximations respectively. Assume that we want to prove that always `0 <= ratio`. In the case of the refinement $\bar{P}$ in snippet 2.6 we could follow the idea of 2BMC and iteratively check whether `0 <= ratio` for finite traces of P. If a counterexample is found in $\bar{P}$ it is obvious that this counterexample must also apply to P. However, if no counterexample is found but a proof for the property at hand can be constructed, one can only hope that this proof can be generalized to P.

On the other hand, the converse holds for over-approximations, i.e. abstractions. If

[7] Note, that formally, under- and over-approximations are defined via control flow graphs. The intuition here is that a control flow graph G over-approximates another control-flow graph G' if G can be embedded into G', i.e. G resembles the same kind of control flow as G' but has fewer locations, i.e. G abstracts from G'.

```
 1  n = abs(input());
 2  div = abs(input());
 3  ratio = 0;
 4  mod = 0;
 5  n = n - div;
 6  while (0 <= n) {
 7      ratio++;
 8      n = n - div;
 9  }
10  mod = div + n;
```

Snippet 2.5: Program P

```
 1  n = abs(input());
 2  div = abs(input());
 3  ratio = 0;
 4  mod = 0;
 5  n = n - div;
 6  if (0 <= n) {
 7      ratio++;
 8      n = n - div;
 9      if (0 <= n) {
10          ...
11      }
12  }
```

Snippet 2.6: Refinement $\bar{P}$

```
 1  n = abs(input());
 2  div = abs(input());
 3  ratio = 0;
 4  mod = 0;
 5
 6  while (0 <= n) {
 7
 8      n = n - div;
 9  }
10
```

Snippet 2.7: Abstraction $\hat{P}$

a property is checked for the abstraction $\hat{P}$ of P and a positive result in the form of a correctness proof is given this proof will apply to P as well. However, when a counterexample is given, it is unclear whether it applies to P. If the counterexample does not apply to P, the idea of CEGAR [Cla+00] applies which is to refine the abstraction at hand in such a way that the counterexample does not apply to it anymore and re-iterate on the verification process.

SPACER uses both of these techniques to prove safety properties of programs expressed in Datalog using μZ. These safety properties are expressed using custom relations in Datalog such as `goal` in snippet 2.4. SPACER then tries either to construct a proof why the property at hand cannot be derived from the axioms of the Datalog program or it gives a counterexample illustrating a possible weakness of the program. In theory, Datalog could be used for the implementation of this thesis. A single predicate could be used to simulate the architectural state of the instruction set architecture at hand whilst each instruction could be modelled via a separate rule.

However, whereas the limiting factor for SPIN was the modelling language, the limiting factor for SPACER is the output of the tool. In case of a property failing to be verified, no counterexamples are given, but SPACER simply outputs that there is a trace of the program for which a given property does not hold without specifying the trace itself. μZ itself can be configured to give a more detailed result. However, this would still not include a *trace of derivations*. For SPACER or μZ to be usable in the context of this thesis, these tools would need to output a log of variable values or steps taken when applying rules. In theory, we could alter these tools such that the output is adjusted respectively. However, it is not clear whether this is actually possible and it might come with significant performance impacts or break the tool altogether.

2.5.3. nuXmv

nuXmv [Cav+14] is a symbolic model checker that is able to model check invariants, LTL and computation tree logic (CTL) formulas. Symbolic model checkers have been introduced in [Bur+92] and encode the state-space of the system to model check in Boolean decision diagrams (BDDs) which work analogously to binary decision trees with the major difference that BDDs are not trees but directed, acyclic graphs.

nuXmv supports both finite and infinite state transition systems. These systems are expressed in the form of *modules*. At its heart, a module consists of a collection of variables, **`INIT`** (i.e. initial) constraints and **`TRANS`** (i.e. transition) constraints. There are more syntactical constructs, e.g. macros or constants, but these do not add to the expressiveness of nuXmv but either serve as syntactic sugar or enhance on performance if used. A module in nuXmv induces a set of state transition systems. The state space of such a system is given by the variables of the module, i.e. an assignment to all variables is a state of the system. The initial state must model the **`INIT`** constraints the transition relation must model the **`TRANS`** constraints. Whenever nuXmv checks a property P for a module M, it is checked if the state transition system S given by M models P.

The input language to nuXmv is strongly typed. Variables can be of type Boolean, integer range, a custom enumeration, bit vector or array in the case of finite state tran-

```
1  MODULE main
2      IVAR
3          ped_button : boolean;
4      VAR
5          ped_waiting : boolean;
6          traffic_lights : { r, g };
7          ped_lights : { r, g };
8
9      INIT traffic_lights != ped_lights;
10     ASSIGN
11         init(ped_waiting) := FALSE;
12         next(ped_waiting) :=
13             (ped_lights = g ? FALSE : ped_waiting |
                 ↪ped_button);
14         next(ped_lights) :=
15             (ped_waiting ? g : r);
16         next(traffic_lights) :=
17             (next(ped_lights) = g ? r : g);
```

Snippet 2.8: An example of a nuXmv module

sition systems or additionally integers and reals which induce infinite state transition systems. **INIT** constraints are simple formulas of propositional logic supporting standard expression to deal with bit vectors, arrays, etc., such as basic arithmetic, array indexing, standard Boolean operators on both bit vector and Boolean level, but also case and if-then-else expressions. **TRANS** constraints also are formulas of propositional logic but additionally support the use of the `next(...)` operator, which is a reference to the value of the respective expression in the next state.

Example 4. Consider the example module depicted in snippet 2.8 modelling a pair of traffic lights; one for pedestrians crossing the street, the other for cars driving on the street.

This module uses an **IVAR** - *input variable* - declaration; a language construct that has not yet been introduced. Input variables have the characteristic that they are "read-only". Technically, every variable is read-only since they cannot be assigned directly but only constrained. **IVAR** variables, however, are "read-only" because they cannot be used in **INIT** constraints or inside `next(...)` expressions. In other words, the transition of input variables appears to be completely random as it is completely unconstrained and as such, can take any value in any step. In this case, the input variable to the module signals whether a pedestrian just pressed the button indicating the desire to cross the road.

Furthermore, the module comprises three ordinary variables: the Boolean variable

`ped_waiting` which indicates whether a pedestrian is waiting to cross the road as well as `traffic_lights` and `ped_lights` modelling the traffic or pedestrian lights and both are of an enum type that contains the symbols `r` for red and `g` for green.

Then, there is one **`INIT`** constraint asserting that one of the traffic lights is green and the other is red at startup. All other constraints are expressed in the **`ASSIGN`** statement, which is syntactic sugar for a group of **`INIT`** or **`TRANS`** statements. Its semantics should be clear from the example. `ped_waiting` is initialized to `FALSE` and becomes false after the traffic lights switched to green; otherwise, it can become `TRUE` whenever the input variable `ped_button` happens to be `TRUE`. The pedestrian lights switch to green whenever some pedestrian is waiting and otherwise switch to red. The traffic lights are coupled to the pedestrian lights and become red whenever the pedestrian lights are green and become green otherwise.

nuXmv supports four types of specifications: invariants, LTL or CTL formulas, and property specification language (PSL) expressions. Here, an introduction to PSL will be skipped since PSL is directly aimed at hardware verification (cf. [FMW05]).

Invariants are the simplest form of specification and can also be expressed in both LTL and CTL. In the context of nuXmv, an invariant is formally given by a next-expression, i.e. an expression that can contain the use of the *`next`* operator. Invariants specify that something (i.e. the respective next-expression) must be true in all reachable states of a model.

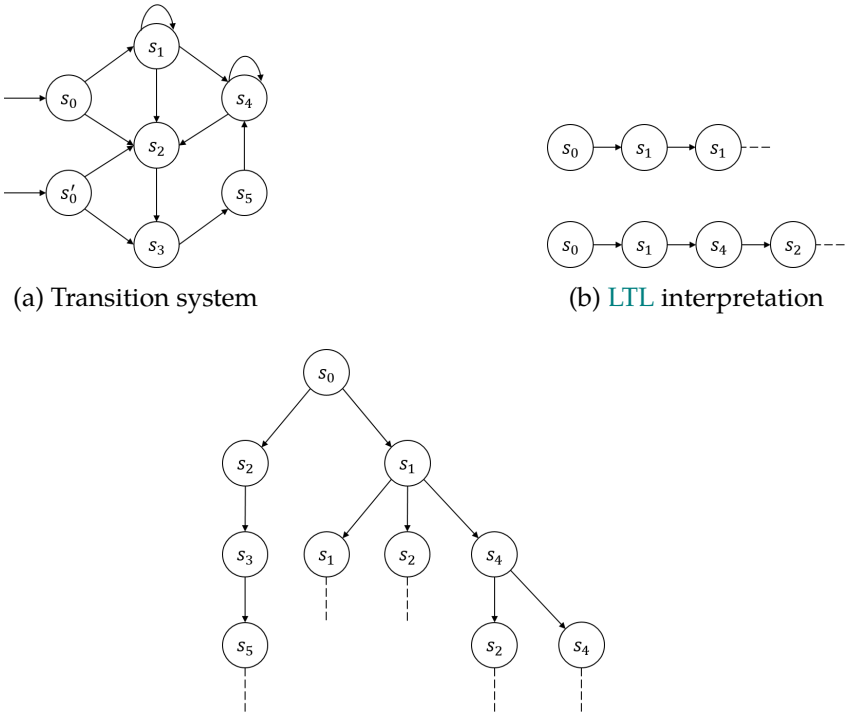
LTL, on the other hand, is a temporal logic and as such, supports the specification of properties over time. Both, LTL and CTL, will be introduced semi-formally here as presented in the *nuXmv User Manual* [Boz+16]. Interpretations and models of an LTL formula are infinite execution paths of a state transition system that in context of nuXmv are given by a module. A state transition system comprises a set of states and a transition relation amongst them with some dedicated initial states. Such a state can be understood as an assignment to the variables of the nuXmv module that is resembled by the transition system. An illustration of a transition system is given in figure 2.5a. The transition system comprises two initial states s_0 and s_0'. The set of all initial states is given by the set of all assignments satisfying all **`INIT`** constraints. Given a state s, the set of its successor states is given by the set of states $\{s^+ \mid s$ and s^+ satisfy all **`TRANS`** constraints $\}$.

Such an execution path now is an infinitely long trace of states $T = (t_1, t_2, \dots)$ such that t_1 is an initial state, t_1 and t_2 satisfy all **`TRANS`** constraints, etc. Intuitively, one can understand some execution as a random walk in the transition system starting at an initial state, i.e. some run of the system. Two example execution paths are depicted in figure 2.5b.

The most simple form of an LTL formula again is a next-expression `p`. T models `p` in LTL if `p` is true in t_1. Furthermore, there are five main temporal operators. LTL formulas are defined syntactically and semantically in the usual, recursive way where ψ, ψ' are LTL formula:

Next $\varphi :=$ **`X`** ψ: T models φ if ψ is true in t_2, i.e. ψ holds in the *next* state.

Globally $\varphi :=$ **`G`** ψ: T models φ if ψ is true for *all* t_i, $1 \leq i$, i.e. ψ is *always* true.



(a) Transition system

(b) LTL interpretation

(c) CTL interpretation

Figure 2.5.: nuXmv modules illustrated

Finally $\varphi :=$ **F** ψ: T models φ if ψ is true for *some* t_i, $1 \leq i$, i.e. ψ is true at *some point* from now on.

Until $\varphi := \psi$ **U** ψ': T models φ if ψ is true in t_1 to t_i, $1 \leq i$, and at t_{i+1}, ψ' is true, i.e. ψ is true *until* ψ' is true.

Releases $\varphi := \psi$ **V** ψ': T models φ if ψ' holds in t_1 to t_i, $1 \leq i$, and in t_i also ψ is true. Alternatively, i might not be bounded, i.e. ψ then is never and ψ' globally true. This means that ψ' is true *at least until* ψ becomes true (which also might be never).

When an LTL specification p is checked by nuXmv, it is verified that p is true for *all* execution paths of the transition system given by the main module. Consequently, an invariant p can also be expressed as the LTL formula **G** p.

Additionally, there are analogous operators for the past. **Y** p and **Z** p are the past-versions of **X** with the distinction that **Y** is false at t_1 and **Z** is true at t_1. The "Historically" operator **H** p is the counterpart to **G** and the "Once" operator **O** p is the counterpart to **F**.

CTL formulas share much of the idea of LTL formulas, but they differ in the interpretations and models for formulas. Whereas in LTL an interpretation of a formula is an infinitely long execution path of a transition system, in CTL an interpretation of a formula is a transition system itself or more specifically: the unwinded transition relation of a transition system as a tree (hence the name computation *tree* logic). Therefore, it is straightforward how CTL formulas are checked by nuXmv: it is checked whether the transition system unwinded as a tree models the formula. Consider the example tree depicted in figure 2.5c.

Again, the most simple form of a CTL formula is a next-expression p. Such a tree T models p if p is true in the root state s_0 of T. The operators supported by CTL are again defined inductively for some CTL formula ψ:

Exists globally $\varphi :=$ **EG** ψ: T models φ if there exists an infinite path $(s_0, \dots)$ in T such that ψ is true for all states on the path.

Exists next state $\varphi :=$ **EX** ψ: T models φ if there exists a direct successor state s to s_0 such that ψ is true in s.

Exists finally $\varphi :=$ **EF** ψ: T models φ if ψ is true in some successor state s to s_0.

For all globally $\varphi :=$ **AG** ψ: T models φ if for all infinite paths $(s_0, \dots)$ in T, ψ is true for all states on the respective path.

For all next state $\varphi :=$ **AX** ψ: T models φ if for all direct successor states s to s_0, ψ is true in s.

For all finally $\varphi :=$ **AF** ψ: T models φ if for all infinite paths $(s_0, \dots)$ there is some s on the path such that ψ is true in s.

It can be seen that the most important LTL operators **X**, **G** and **F** are adopted to CTL by existentially and universally quantifying each.

```
1 INVARSPEC traffic_lights != ped_lights;
2 LTLSPEC G (ped_button -> F ped_lights = g);
3 CTLSPEC (AF traffic_lights = g) & (AF ped_lights = g);
```

Snippet 2.9: An example of a specification in nuXmv

Example 5. To illustrate the relation of LTL and CTL consider some module M and its transition tree T as well as the set of all execution paths P. Now let `p` be some next expression. One has:

- T models **AG** `p` if and only if for all $p \in P$, p models **G** `p`
- T models **EG** if and only if there exists a $p \in P$ such that p models **G** `p`

The same applies to **X** and **F**, respectively. However, in general, LTL and CTL do not share the same expressive power and neither is strictly more expressive than the other.

Example 6. Consider the example invariant, LTL and CTL specifications depicted in snippet 2.9. These specify high-level properties about the traffic lights of example 4.

In nuXmv, specifications are introduced by **INVARSPEC**, **LTLSPEC**, or **LTLSPEC** and optionally may be preceded by a name which is omitted in this example. The invariant in this example states that at no point can the traffic and the pedestrian lights show green or red simultaneously. The LTL formula specifies that whenever the button for the pedestrians is pressed, at some point the pedestrian lights must show green. The CTL formula specifies that in unwindings of the transition relation, at some point the traffic lights must show green and at some point the pedestrian lights must show green.

In this example, both the invariant and the LTL specification are met by the model. The CTL specification though is false. nuXmv gives the following counter-example:

```
1 -- specification (AF traffic_lights = g & AF ped_lights =
    ↪g)  is false
2 -- as demonstrated by the following execution sequence
3 Trace Description: CTL Counterexample
4 Trace Type: Counterexample
5 -> State: 1.1 <-
6 ped_waiting = FALSE
7 traffic_lights = g
8 ped_lights = r
```

Counter-examples in nuXmv always show a finite fraction of an infinite execution path. For LTL formulas, this fraction loops at some point thus inducing an infinite trace. For CTL formulas, on the other hand, the counter-example marks the longest, unambiguous path in the transition tree to a subtree that violates the formula at hand. Only one state being depicted in the counter-example above indicates that the computation

tree with the initial state given above and in which no variable ever changes violates the CTL formula at hand. This tree illustrates a world in which the button for pedestrians is never pressed and therefore, the pedestrian lights will correctly never show green.

While it is not explicitly mentioned in the user manual how input variables relate to the states in the unwinding of the transition relation for the interpretation of a CTL formula, it is assumed here that the input variables are not part of the state space; otherwise, the counter-example given above would not apply since the state where `ped_button` becomes true trivially is reachable. This assumption is supported by the fact that in nuXmv, CTL formulas must not reference input variables. In other words, this means that one must not rely on input variables whilst verifying a system.

2.5.4. Summary

Whereas SPIN was problematic because its input language did not match the needs of this thesis, and SPACER or μZ were problematic because their output does not match the requirements of this thesis, nuXmv fulfils both aspects. The traces generated as counter-examples serve as an illustration what specifically went wrong on a property. The input language allows implementing modules that are limited in the description of the transition relation itself[8] but is powerful in expressing modules as a whole. Having variables and transition allows implementing pretty much any sequential system. Therefore the decision was made to use nuXmv as a verification engine. We are not the first to use model checking for hardware verification (cf. [KRS14; Irf+16; BS16]), however, to our knowledge, we are the first to implement information flow control using a model checker.

The idea for the implementation is straight-forward:

- Use input variables to model instructions to the architecture
- Use variables to model the architectural state, i.e. registers, etc.
- Use transition constraints to implement the behaviour of each instruction

We furthermore decided to express the information flow control of the architecture in LTL. nuXmv supports a plethora of techniques to check specifications, e.g. the *nuXmv User Manual* [Boz+16] lists 47 commands to check a specification. In the practical work for this thesis, after the decision to implement the IFC in LTL has been made, a couple of these techniques which apply to LTL-checking were tested. In this informal evaluation it was found that the K-Liveness algorithm proposed in [CS12] delivered best results and terminated quickly whenever the model met the specification. Whenever this was not

[8] Since transitions are constrained by propositional logic and expressions without loops only, there can be issues if complex transitions should be expressed. For example, in section 4.1 we will consider a situation where for the transition of a bit vector the first bit that is high needs to be found. In other programming languages, one would simply iterate over all bits of the vector and break the loop on the first high bit. However, when constraining the transition relation, there are no loops. This left us with the need to implement this search for the first high bit as a sequence of nested `( ? : )` expressions resembling the binary search tree for the respective bit.

the case, i.e. the specification was not met by the model which was usually indicated by a long runtime of the K-Liveness algorithm, a BMC approach computed counter-examples very efficiently. As the technique of BMC has already been introduced in section 2.5.2, now, the key idea behind the K-Liveness algorithm by Claessen and Sörensson shall be investigated.

In LTL, two types of properties can be distinguished: *safety* and *liveness* properties. Safety properties are formulas for which every counter-example has finite length. Liveness properties on the other hand can[9] have counter-examples which are infinite and cannot be made finite. The background of the K-Liveness algorithm is that techniques for efficiently solving safety properties already exist, i.e. the IC3 algorithm proposed in [Bra11]. Claessen and Sörensson take this algorithm and present a decision procedure for model checking liveness properties on finite state transition systems, i.e. a general algorithm for checking LTL formulas on such systems. The algorithm works on LTL formulas of the canonical form: `F G q` where `q` is some LTL formula. The authors implicitly mention that any LTL formula φ can be transformed into this form but give no proof or reference for this. For such properties, Claessen and Sörensson present the following lemma:

Lemma ([CS12]). *If a state transition system models* `F G q`*, then there exists some integer* k *such that for any trace* T*,* `q` *is false at most* k *times in* T*.*

This leads to a straightforward algorithm that is sound and complete on positive instance of LTL formulas:

1. Let $k := 0$.
2. Construct a formula `q_new` that encodes: `q` is false at most k times. This obviously is a safety property.
3. Consult the IC3 algorithm and check whether `q_new` is true; if it is, return that `F G q` is true.
4. Otherwise; set $k := k + 1$ and go to step 2.

To have this algorithm also terminate for negative instances, i.e. where the property at hand is not modelled by the state transition system, the authors complement the basic K-Liveness algorithm with methods dedicated to finding counter-examples. Such an algorithm needs to construct bounded counter-examples for `q_new`-like properties and detect repeated states in which `q` is false. If a trace is found where such a state appears at least twice, there also is a trace in which this state occurs infinitely often thus disproving `q` since no k in accordance to the lemma can be found.

2.6. Summary & Methodology

The goal of this thesis is to evaluate an approach to formally verifying ISAs against higher-level information flow properties using a model checker. This combines the work

[9] Especially, one has that every safety property is also a liveness property but not vice versa.

of Reid [Rei17] and Ferraiuolo et al. [Fer+17]. This approach will be evaluated by applying it to a simplified version of the RISC-V ISA, MINRV8. We adopt the requirements to these properties from [Rei17], i.e. these properties should:

> "[...]
>
> - express the major guarantees that programmers depend on;
> - [...] [be] concise so that architects can easily review and remember the entire set of properties;
> - [...] [be] stable so that architectural extensions do not invalidate large numbers of rules;
> - [...] describe the architecture differently from existing specification to reduce the risk of common-mode failure." [Rei17, pp.88:2-3]

To achieve this, we will investigate the RISC-V architecture in more detail and define as well as implement the MINRV8 architecture in chapter 3. This architecture will stick to the RISC-properties as defined in [HP12] in that it will be a load/store architecture, instructions will operate on registers mainly, and there are few, homogenously encoded instructions available. A model of this instruction set architecture will be implemented in nuXmv.

As the next step in chapter 4, the terms of an information flow *policy* and information flow *control* will be applied to the MINRV8 architecture, thus, using the work of Ferraiuolo et al. [Fer+17] to define higher-level properties to verify the architecture against. Thereby, it will be investigated how the policy in [Fer+17] which applies to HDL code can be carried over to ISAs and implemented in nuXmv. By using nuXmv as a verification tool, we enhance on [Rei17] in the key aspect, that unbounded numbers of transitions can be constrained. When this has been dealt with, the actual IFC will be expressed as LTL properties. These properties will assume the user-mode of the system to be compromised and express that insecure flows of information between machine- and user-mode are impossible.

The results of this verification process will be discussed in chapter 5. How will these results look like?

The idea of IFC is to express properties like: "Information must not flow like ...". Whenever nuXmv is asked: "Does this model of the MINRV8 architecture allow for such flows of information?", it will either respond with yes, giving a sequence of instructions that violate the information flow property or respond with no. If the response happens to be a counter-example, the following questions must be investigated:

1. Is the counter-example legitimate or is it a false positive?
2. If the counter-example is legitimate, how can the uncovered vulnerability to the system be mitigated?

The former question arises because the model of the architecture itself might be flawed. To answer it, one simply must inspect the trace at hand and examine whether

all state transitions and variable changes are indeed valid in regard to the specification of MINRV8. If the answer to this question is no, the implementation has to be fixed accordingly and the proofs must be re-run.

The latter question leads to the actual results. If it is found that the counter-example at hand is genuine, it must be decided how the newly found vulnerability is handled. There are two ways to handle a vulnerability. Either it is a proper architectural vulnerability that must be mitigated accordingly by architectural measures or it shows a vulnerable *usage* of the architecture that cannot be mitigated by the architecture without removing critical features from it. For example, it is an obvious security issue, when machine-mode directly hands confidential data to user-mode. No sensible architectural mechanisms could prevent machine-mode from doing this as long as machine-mode is, in fact, able to hand data to user-mode - which it should be.

In the first case, where it is discovered that a counter-example comprises a proper architectural vulnerability, we will investigate mechanisms that mitigate the respective vulnerability and also attempt to verify the effectiveness of such mitigations by re-running the proofs. If however, we do not find the counter-example to uncover a vulnerability but rather to describe *vulnerable code*, we will phrase rules to be obeyed by, e.g. OS or compiler engineers, that prohibit such vulnerabilities. We also will then verify the effectiveness of these rules by implementing them in the model as assumptions and re-running the proofs as well.

When phrasing assumptions, we impose the following requirements upon them:

1. The assumptions must be non-trivial, i.e. they must not be contradictory and must not trivially entail the information flow properties to be verified,
2. the assumptions must only constrain machine-mode, and
3. the assumptions must be practical and verifiable.

The reasoning behind point 1 is straight-forward: Working with contradictory assumptions leads to proving "false $\Rightarrow$ properties" which is trivially true. Likewise, working with assumptions trivially entailing the properties leads to proving "properties $\Rightarrow$ properties" again being trivially true.

Point 2 was raised because it is important to note that the assumptions that will be introduced as a result of this thesis fit in the threat model, as discussed in section 2.4.3. Recall, that in the context of this thesis, it is assumed that user-mode is adversarial to machine-mode and compromised completely. Constraining user-mode in the assumptions, therefore, would contradict this threat model.

Lastly, we hope that the assumptions can serve as a verification target for OSes or compilers. This is expressed in point 3. If one was to write a collection of interrupt handlers for an OS, these assumptions might be used to verify that each rule given by these assumptions is implemented by the handlers. This would grant the absence of information flow related bugs in the context of the information flow properties as proposed in section 4.3. This, however, presupposes that it is actually possible to verify programs against these properties. As part of this thesis, we cannot give guarantees on the feasibility of this undertaking. Yet, we try to allow for it.

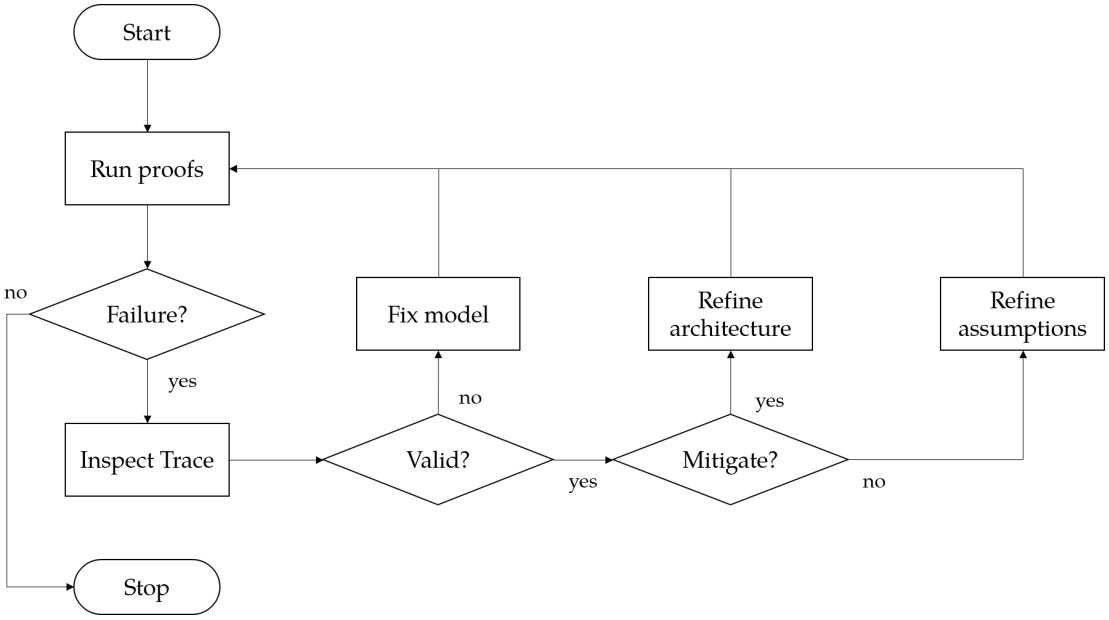


Figure 2.6.: Verification Process Flowchart

These steps constitute an interactive prove-refine loop that is depicted in figure 2.6. This loop is entered by running nuXmv in order to prove the absence of any IFC violations. If, however, there is such a violation, the model will be refined either by a fix or by a refinement of the architecture or by a refinement of the assumptions implementing rules to secure usage of the architecture. At some point, a set of respective fixes, architectural changes and assumptions will mark a fixpoint to this loop where no further refinement is necessary and finally, no IFC violations can be found.

This is what will be presented in chapter 5. From chapter 6 onwards, these results will be discussed, put into context of related work in chapter 7 and finally summarized in chapter 8.

Part II.

Main Part

Chapter 3.

Architecture

In this section, we will develop and implement the architecture to work with. As mentioned before, the RISC-V architecture will serve as a role model for the architecture to verify as part of this thesis. To derive a simpler version of the RISC-V architecture, which will be called MINRV8, we will introduce the RISC-V architecture itself in section 3.1. This introduction will conclude with a summary that mentions which parts will be adopted and which parts will be left out in the development of the MINRV8 architecture.

Consequently, in section 3.2, we will introduce the MINRV8 architecture. The development of the MINRV8 architecture went hand in hand with its implementation in nuXmv, which we will present in section 3.3. Hence, some design decisions of the model and the implementation will only be sensible once both have been introduced.

To conclude, section 3.4 will give a summary of what has been achieved in this chapter.

3.1. An Introduction to RISC-V

The core idea of the RISC-V architecture has already been introduced in section 2.3.3. The base integer instruction set comprises instruction of four categories: integer and bit vector arithmetic, memory operations, control transfer, and platform management. The instructions that are available to the RISC-V architecture specifically will be introduced in section 3.1.1 subsequently.

Afterwards, the privileged architecture of RISC-V will be introduced, which can be divided into three parts: The specification of:

1. Three levels of privilege
2. Memory attributes
3. Exceptions, traps and interrupts

In section 3.1.2 each of these parts will be introduced whilst the exceptions, traps and interrupts will be put into focus.

The overall goal of this section is to give a feel for how the RISC-V ISA works and how a more minimal, RISC-V inspired ISA could be distilled from it. A selection of features to be included in such a minimal architecture will be discussed in section 3.1.3.

3.1.1. The Base Integer Instruction Set

Integer arithmetic and Boolean instructions The base integer instruction set provides instructions for addition and subtraction, integer comparisons, bit-shifts and the Boolean operators AND, OR and XOR. Most of these instructions are implemented in two versions: a register version (or R-type instruction) and an immediate version (or I-type instruction). In the first, case the two operands are given as registers, and in the second case, one operand is given as register and the other as an immediate value read from memory. In either case, the value is stored in a destination register.

These basic instructions are all side effect free besides from advancing the program counter, which means that if the destination register is x0, each of them encodes a no-operation instruction. This makes them particularly easy to analyze. However, as a bonus, this means that there is no built-in support for overflow checks via flags in some dedicated CSR. There also is no special instruction to handle this.

Memory operations There are two types of instructions that deal with external memory. Firstly, simple load and store instructions that read from or write to memory. Secondly, FENCE instructions that synchronize memory access between multiple harts.

Memory in RISC-V is byte-addressable. Therefore, load and store operations in RISC-V can read or write words of size 8-, 16-, 32-, 64- or 128-bits, depending on the maximum word length of the base instruction set of choice. Read- and write-addresses do not have to be aligned by software before they can be used, however, if they are not, the architecture must first align them before fetching from or storing to them, i.e. reading a 32-bit word from memory at address 0x1 is impossible. If this address is given as an argument to a respective load instruction, it will be aligned to the address 0x0.

Control transfer instructions There are two types of instructions that can control program flow in RISC-V: jumps and branches. In short, jumps are performed unconditionally and can target a wider range in memory (at least $\pm 2^{20}$bits in case of a 32-bit architecture) whereas branches are performed conditionally, only can target a narrower range in memory ($\pm 2^{12}$bits in case of a 32-bit architecture) and always must branch to PC-relative addresses.

In contrast to memory operations, addresses targeted by jumps must be aligned correctly. If this is not the case, an exception will be generated.

Jumps are intended to call routines and functions. This is why they store a return address to some general-purpose register. Branches, on the other hand, are intended to control program flow, e.g. to implement an *if-else* structure or *for*-loops where code will not return from the branched execution. RISC-V is agnostic about the details of calling conventions for routines, e.g. how the stack is organized, which general-purpose

registers are used as LR or SP and leaves these details to be implemented by software running on the core.

Platform management instructions The RISC-V manual also knows this category and divides it into two subcategories: instructions that deal with CSRs and other instructions performing operations that potentially need some form of privilege. Examples for the former category are straight forward. RISC-V supports instructions to read or write complete CSRs but also knows operations to modify single bits only. For the latter category, RISC-V only knows two instructions at the base instruction set level, which are environment call (ECALL) and environment breakpoint (EBREAK) instructions. These instructions will be inspected in more detail in section 3.1.2.

Summary The base integer instruction set of RISC-V contains everything needed to implement a comfortable Turing-machine but pretty much nothing more. You can perform basic arithmetic with integers or bit vectors of Booleans, load and store results of your calculations and abstract your code by using jumps and branches. Abstractions other than from code, i.e. by routines implemented using jumps and branches, are not given.

Only the platform management instructions give a hint on the complexity that is offered by the RISC-V architecture to allow abstracting processing done on the core.

3.1.2. The Privileged Architecture

The basis of the privileged architecture is formed by the three levels of privilege: machine-, supervisor- and user-mode.

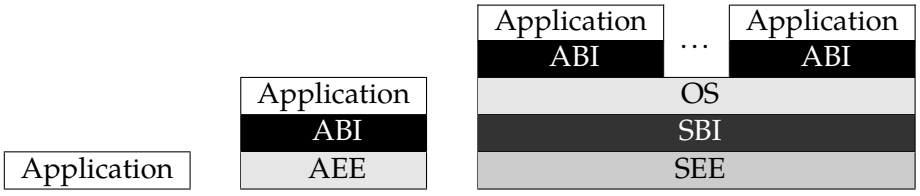
For example, the purpose of interrupts and exceptions is not only to handle error conditions that arise during runtime but also to communicate between these privilege layers. Also, memory attributes rely on these levels of privilege as they define which mode is eligible to do some operation on some region of memory.

This basis of the privileged architecture is controlled by the machine status (mstatus) register, which is the core of machine-mode. This register controls both parts of the memory attributes and interrupt and exception handling. There are more CSRs in the RISC-V architecture that will be introduced when necessary; however, note that the mstatus register is the most important of all CSRs.

In this section, the aforementioned concepts will be introduced subsequently while the respective parts of the mstatus register will be discussed whenever applicable.

Levels of Privilege RISC-V knows three privilege levels: user-mode, supervisor-mode and machine-mode[1]. Each implementation of the RISC-V specification must at least

[1] In an earlier version, there was a fourth privilege-mode, hypervisor-mode, that was more privileged than supervisor- but less privileged than machine-mode. However, this mode has been dropped from the specification as of now. Yet, RISC-V still needs to encode mode-relative bit-fields with two bits. Usually, 00 stands for user-, 01 for supervisor and 11 for machine-mode. 10 is reserved in most places and has been used for hypervisor-mode.



(a) Simple embedded system

(b) Secure embedded system

(c) System with Unix-like OS

Figure 3.1.: Privilege level combinations [WA17a]

provide machine-mode as a base mode of operation. Besides this, there are two other choices of privilege levels combinations supported:

1. Machine-mode only ("simple embedded system" [WA17a])
2. User and machine-mode ("secure embedded system" [WA17a])
3. User, supervisor and machine-mode ("[system] running Unix-like operating systems" [WA17a])

A visualization of these three combinations is given in figure 3.1 where they are depicted from left to right. A simple embedded system might be the cheapest to implement and manufacture, however, this system lacks any protection against malicious application code, i.e. one would only want to run such a system in very basic scenarios where full control over all source code and the device itself can be guaranteed.

A secure embedded system supports some application to be run on it while an application execution environment (AEE) running in machine-mode controls the execution of this application. The application communicates to the AEE via an application binary interface (ABI) that defines the possible interactions between an application and the AEE.

A system running a Unix-like OS enhances this by having an OS running in supervisor-mode. This OS communicates with multiple applications running in parallel via an ABI. The OS itself is managed by a supervisor execution environment (SEE) which communicates with the OS via an supervisor binary interface (SBI). The SEE runs in machine-mode.

To keep things simple, the focus of this thesis lies in secure embedded systems[2]. This means that the architecture to be worked with supports two modes: user- and machine-mode. Machine-, supervisor- and user-mode do not simply form a linear order of privilege attributes of code execution. They differ in the concepts that are available to them and therefore, must be handled separately. However, as the focus will lie on secure embedded systems, many parts of the specification which are about supervisor-mode

[2]This will be justified in more detail in section 3.1.3 where we introduce the architecture that will be verified as part of this thesis.

solely will be left out. Supervisor-mode will be handled wherever it fits in the bigger picture but not on its own.

Memory Attributes RISC-V offers two ways to configure physical memory: physical memory attributes (PMAs) and physical memory protection (PMP).

PMAs state "inherent properties of the underlying hardware and rarely change during system operation" [WA17b, p.40]. Examples for such "inherent properties" are supported access types, whether a region of memory supports memory access ordering or and cacheability. Depending on the implementation, these attributes might be hardwired for certain regions (or all) or modifiable by software. In general, PMAs constrain possible interactions with physical memory. PMAs can be accessed by software to determine what kind of behaviour it can expect from a physical region of memory.

The PMP setup serves as another layer of *protection* for regions of memory. This makes it similar to virtual memory. However, PMP offers more coarse-grained protection in comparison to virtual memory. PMP assigns read, write, or execute permissions per region of memory per privilege mode. These permissions are set up in the physical memory protection configuration (pmpcfg) registers of which there are four. Each of these comprises four PMP entries applying to one region of memory, i.e. there can be at most 16 regions of memory. The region of memory such a PMP entry applies to is specified by a corresponding physical memory protection address (pmpaddr) register - consequently, there are 16 pmpaddr registers. The details of these register match specific physical addresses will be left out here. The permissions specified about a region of memory by a PMP entry are enforced on user- but not machine-mode by default.

One exception to this is the *locking* of PMP entries. If a PMP entry is locked, not only can its contents not be changed anymore[3], the respective permissions also are enforced on machine-mode.

Another way of enforcing PMP checks onto machine-mode is to set the MPRV bit in the mstatus register. This bit is only available if more than one privilege mode is supported; otherwise, it is hardwired to zero. If MPRV is set to one, "load and store memory addresses are translated and protected as though the current privilege mode were set to MPP" [WA17b], i.e. they're performed with the privilege level that was preempted by taking the trap.

Additionally, the mstatus register supports the MXR field that allows reading executable memory regions. If MXR is not set, attempting to read from a memory region marked as executable will fail. If MXR is set, such reads can succeed if said region is also marked as readable. MXR is only applicable, though, if page-based virtual memory is in place.

Exceptions, traps and interrupts Volume 1 of the RISC-V specification [WA17a] introduces the concept of exceptions, traps and interrupts as follows:

[3] System resets may reset an entry - unlocking it. However, manufacturers still can decide to hardwire a PMP entry, e.g. to make it read-only. This, however could also be realized via PMAs.

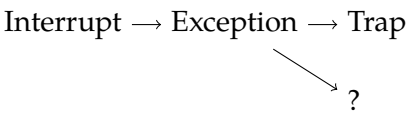


Figure 3.2.: Trap-Trigger-Hierarchy

> "We use the term *exception* to refer to an unusual condition occurring at run-time associated with an instruction in the current RISC-V thread. We use the term *trap* to refer to the synchronous transfer of control to a trap handler caused by an exceptional condition occurring within a RISC-V thread. Trap handlers usually execute in a more privileged environment.
>
> We use the term *interrupt* to refer to an external event that occurs asynchronously to the current RISC-V thread. When an interrupt that must be serviced occurs, some instruction is selected to receive an interrupt exception and subsequently experiences a trap.
>
> The instruction descriptions [...] describe conditions that raise an exception during execution. Whether and how these are converted into traps is dependent on the execution environment, though the expectation is that most environments will take a *precise* trap when an exception is signaled [...]." [WA17a]

This means that a trigger hierarchy for interrupts, traps and exceptions can be given, which is depicted in figure 3.2. In general, interrupts occur asynchronously and trigger synchronous exceptions to be generated for a hart which in turn may generate traps but also might be handled otherwise. The specification mentions the example of the floating-point extension in which exceptions not necessarily generate traps.

For the sake of simplicity, we will assume that every exception causes a trap. As we will not implement any floating-point arithmetic extension, the specification does not demand from us to handle exceptions differently and doing so does not serve any purpose at this point. In the following paragraphs, an introduction to the control flow of dealing with interrupts will be given. Said control flow is depicted in figure 3.3 and follows these general steps:

1. If an interrupt or exception is pending, determine the privilege mode to take the trap
2. Check if the interrupt is enabled
3. Take the trap
4. Return from trap handler

Interrupt handling can be understood as a generalized form of exception handling since exception handling in principle works the same but has fewer steps.

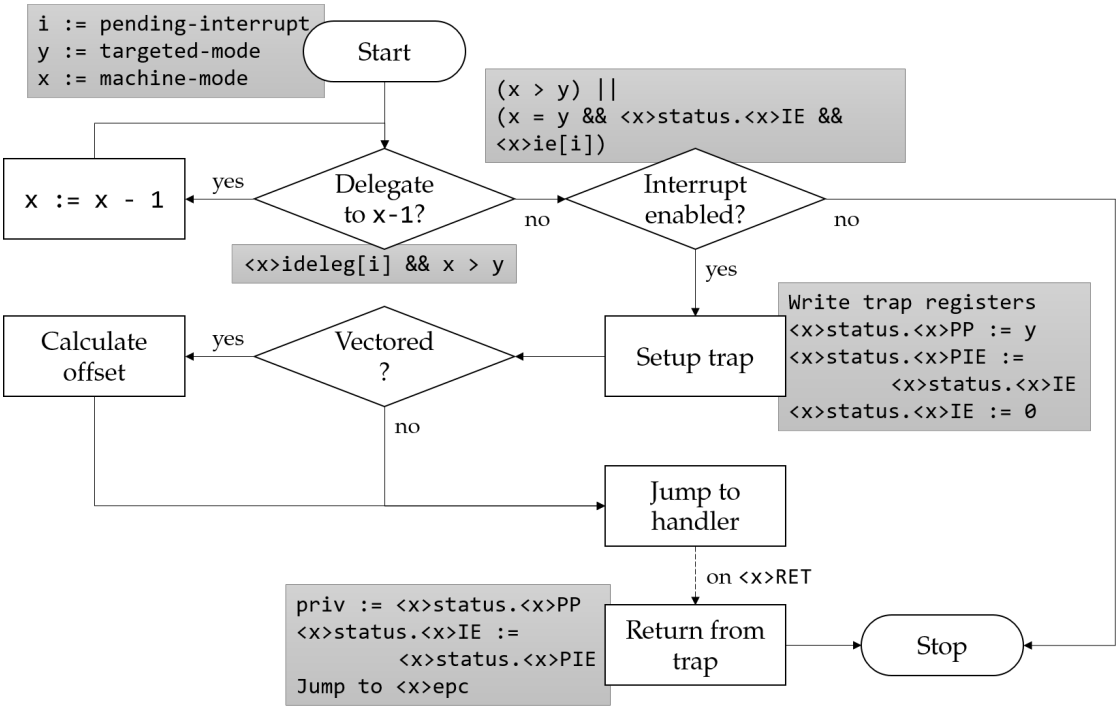


Figure 3.3.: Interrupt handling flow-chart

There are three categories of interrupts: external, timer, and software interrupts. External interrupts are used to signal exceptional events occurring on some external device that is connected to the core. Timer interrupts can be used to track time. Code can use certain CSRs to signal to the platform that it needs to be alerted when a given amount of real-world time has passed. Such an alert will be given by a timer interrupt. Software interrupts can be set pending by software running on the same hart with equal or higher privilege than the targeted privilege mode. Software interrupts have no predefined semantics and thus can be used by programmers or system designers in every imaginable way.

Each of these categories of interrupts can target any privilege-mode resulting in a total of six interrupts available to the platform in our case (or nine if supervisor-mode is supported). These interrupt-types are associated with indices in the range from 0_{10} to 11_{10}, also-called interrupt code[4]. Interrupts can be set pending individually in the machine interrupt pending (mip) register. An interrupt is set pending if the bit of the mip corresponding to the interrupts index is set to 1.

If an interrupt or exception is set pending, it must first be decided which privilege mode will take the trap to handle the exception generated. By default, all interrupts

[4]The reader might have noticed that these are more indices than needed to encode all interrupts available - even if supervisor-mode is present. Indices 2, 6 and 10 currently are reserved. These indices' two least significant bits are 10_2 which corresponds to the encoding of hypervisor-mode. When hypervisor-mode was removed from the specification, these bits were marked as reserved.

and exceptions will be handled by machine-mode. However, interrupts and exceptions can be delegated to less privileged modes via the machine interrupts delegation (mideleg) and machine exception delegation (medeleg) register[5]. Delegation of interrupts can be disabled either by software writing the bits of mideleg and medeleg accordingly or by an implementation not supporting these registers altogether. Similar to the mip register, each bit field of mideleg corresponds to the interrupt of the same index. Exceptions also have an exception code assigned and thus, fields of medeleg correspond to these codes. An interrupt or exception can only be delegated to modes equally or higher privileged than the mode the interrupt or exception was targeting originally. For example, if an interrupt is targeting supervisor-mode, it can optionally be delegated to supervisor-mode or must be handled by machine-mode but cannot be handled by user-mode. For each mode x ($x \in \{\mathrm{M}, \mathrm{S}, \mathrm{U}\}$, there are respective counterparts to machine-mode registers such as mip, e.g., if an interrupt is delegated to mode x, the corresponding bit in the respective interrupt pending ($x$ip) register reads accordingly and becomes writable. Otherwise, it appears to be hardwired to zero. From now on, registers related to interrupt and exception handling will be introduced for some arbitrary mode x.

After the mode to be targeted by the trap has been determined, it must be checked whether the given interrupt is actually enabled. Exceptions cannot be disabled and therefore will always be taken - as a consequence the following checks will be omitted for exceptions. Interrupts, on the other hand, can be dis- or enabled individually in the interrupt enable ($x$ie) register. Analogously to the $x$ip register, an interrupt is enabled if the bit corresponding to the interrupt's index is set to 1. Additionally, the status ($x$status) register can globally dis- or enable interrupt handling for a specific mode via its xIE bits. These xIE fields, however, are only taken into account if the hart operates at an equal or higher privilege mode than targeted, i.e. interrupts are always enabled globally if some interrupt is targeting a mode of higher privilege than the mode of current operation. Finally, an interrupt will be taken if interrupts are globally enabled *and* the interrupt itself is enabled in the $x$ie register.

If a pending interrupt is taken, the exception program counter ($x$epc), exception cause ($x$cause), trap value ($x$tval), and $x$status registers are written and the platform will jump to a base address held by the trap-vector base-address ($x$tvec) register. $x$tvec also provides a field that controls whether interrupts can be *vectored*, i.e. for interrupts the PC is not set to the base address for handling trap but to $(\text{base} + 4 \times \text{cause})$. $x$epc holds the address of the original instruction stream that was preempted, i.e. either the address of the instruction that caused the exception or the instruction that was next to be executed as an external interrupt was taken. $x$cause holds an identifier of the current interrupt or exception being handled by the trap. The most significant bit indicates whether an interrupt is currently being serviced and all other bits hold the exception or interrupt code. $x$tval serves as a register to hold arguments for a trap handler. Technically, it holds

[5]It would also be possible to delegate traps by using the machine trap return (MRET) instruction and setting all corresponding CSRs accordingly. However, this would come with a higher latency as a couple of jumps would need to be performed first.

Furthermore, if supervisor-mode is supported, there are supervisor-mode equivalents to the medeleg and mideleg registers to further delegate interrupts and exceptions to user-mode.

"exception-specific information to assist software in handling the trap" [WA17b]. One example for such "exception-specific information" is that $x$tval can contain the faulting bits of an instruction for an illegal instruction exception whereas $x$epc would only point to the instruction in memory as a whole therefore not giving specific information about which part of the instruction caused the exception.

$x$status also provides bits to "stack" certain values when handling exceptions or interrupts. Firstly, nested interrupts are only supported between privilege modes. To implement this, on a trap to handle an interrupt in mode x, interrupts are disabled for this mode and the value of the interrupt-enable bit is "stacked onto" a prior-interrupt-enable bit xPIE, i.e. $x\text{PIE} := x\text{IE}$ and $x\text{IE} := 0$. Secondly, there is a prior-privilege bit xPP that stores the privilege mode before the trap was taken.

Note that due to the small offset between vectored interrupt jump destinations, the instructions located there cannot perform more than jumping to the *real* trap handler. Furthermore, as exceptions cannot be vectored, any differentiation between those must be performed by software.

After the jump to the trap handler, at some point, the xRET instruction will be executed signalizing that the trap has been handled sufficiently. On execution of the xRET instruction, the architecture will reset the state according to the values stacked earlier, i.e. $x$status.xIE is written with $x$status.xPIE, the privilege mode is set to $x$status.xPP, the PC is set to $x$epc, and normal execution continues until the next interrupt or exception occurs.

Summary In the previous paragraphs, the basic ideas of the three levels of privilege, memory attributes and protection and interrupt and exception handling specified by the privileged architecture of RISC-V [WA17b] were introduced. The specification includes even more mechanisms then these, e.g. it was touched that there are timer CSRs, but there is much more. However, these mechanisms are beyond this thesis's scope as they a) often do not introduce new concepts to the architecture as a whole but build upon already introduced ones[6] and b) already exceed what will be incorporated in the MINRV8 architecture.

To summarize: PMAs and PMP entries make general properties of physical memory transparent to software, allowing it to adapt to possible interactions with respective regions correctly. PMP entries, however, can also be used for memory protection in addition to virtualization.

Table 3.1 gives an overview of all registers that are involved in trap handling. CSRs which are available to user-mode would also be available to supervisor-mode in this context. Registers written in italic font are not actual registers on their own but provide a restricted view on their respective machine-mode counterpart.

Summarizing the mechanisms that play into interrupt and exception handling, recall the four steps to interrupt and exception handling we introduced at the beginning of this section. These can now be filled with more details:

[6] A prominent exception to this is supervisor-mode adding support for virtualization.

Machine-mode	User-mode	Description
mstatus	*ustatus*	Status
mie	*uie*	Interrupt enable
mip	*uip*	Interrupt pending
mtvec	utvec	Trap-vector base address
mscratch	uscratch	Scratch
mepc	uepc	Exception program counter
mcause	ucause	Trap cause
mtval	utval	Trap value
medeleg		Exception delegation
mideleg		Interrupt delegation

Table 3.1.: Trap-handling CSRs

1. If an interrupt or exception is pending, determine the privilege mode to take the trap by checking the targeted privilege mode and any delegation settings.
2. If an interrupt is pending, check that interrupts are enabled globally and check that the specific interrupt is enabled as well.
3. Take the trap by jumping to the trap base vector and, if an interrupt is being serviced, take vectoring into account.
4. Return from trap handler.

The differences between handling exceptions and interrupts can be summarized by the following:

- Exceptions cannot be disabled therefore they will always be handled
- $x$cause$[31]$ is written with 0
- Exceptions cannot be vectored therefore the PC will always be set to $x$tvec

To adopt figure 3.3 accordingly, one would need to alter it by replacing most interrupt-related CSRs with their exception-related counterpart if possible. An exception to this is $x$status.xIE which still is set to 0, meaning that it is not possible to preempt handling an exception by handling an interrupt.

Finally, note that exceptions are handled as soon as they occur. This is necessary as exceptions always are closely related to the current stream of instructions and therefore need attention before said stream can continue to be executed. The only exception to this are other interrupts which are handled before any exception with the following decreasing order of priority: external interrupts, software interrupts, timer interrupts, exceptions. RISC-V knows no concept of settings exceptions pending, this, however, is not necessary. If an exception occurs while an interrupt is set pending, the interrupt can be handled with $x$epc being set to the address of the faulting instruction which either

will again fault after the interrupt has been handled or will succeed by some magical condition in which case it does need any further attention[7].

3.1.3. Feature Selection

From the introduction of the RISC-V architecture, we will now select the features to include in the MINRV8 architecture. The overall goal is to create an architecture that is simplified in comparison to the RISC-V ISA but still incorporates all types of features necessary to introduce a sufficient level of complexity, allowing the MINRV8 to be comparable to modern architectures.

The first decision is which combination of privilege levels to model. In section 3.1.2, it has been mentioned that in this thesis, the focus will be put on secure embedded systems, i.e. systems supporting user- and machine-mode but not supervisor-mode. The core features added by supervisor-mode are those of virtualization. Since we deemed memory virtualization as out of scope in this context (cf. sec. 2.4.2), we value simplicity over the completeness of the model and rule out supervisor-mode.

Additionally, to keep the state space small, we decided to model an architecture where the word size is eight bit. There is no reason to believe that an increase in word size introduces fundamentally new security or information flow issues to an architecture, so we decided to value efficiency over completeness in this case. Since the state space grows exponentially in word size, choosing a smaller word size should significantly improve the performance of the model.

From both parts of the RISC-V specification, the base integer instruction set and the privileged architecture, the MINRV8 architecture will include the very foundation of all categories introduced except jumps and branches.

Consequently, MINRV8 supports integer arithmetic, bitwise-logical instructions and bit-shifts, loads and stores, as well as instructions to read and write CSRs and to transition between privilege modes.

As for memory attributes, we decided to implement caching mechanisms in the architecture. Although we deemed recent side-channel-attacks such as Meltdown [Lip+18] and Spectre [Koc+19], which rely on cache, out of scope, there also are direct attacks leveraging the cache [WR09]. Caching undoubtedly plays a critical role in system security. We did not want to rule out a significant attack surface by assumption. Therefore, PMAs will be given in the form of cacheability settings but nothing more. The RISC-V specification is quite vague when it comes to other types of PMAs. It mentions them but does not formally specify their workings. We interpret this as PMAs applying more to specific implementations than RISC-V as a whole since each type of PMAs needs to be specified further when implemented. Therefore, it was decided to include only PMAs for which positive reasons can be given - applying only to caching PMAs.

We adopt PMP entries as well. There will be no configurable but fixed memory regions but each of these can be set readable, writable and locked freely. The decision towards fixed regions of memory was made during the prototyping phase where we

[7] While not being a citable source, credit should be given to Stack Overflow user Baard who pointed this out to us (cf. https://stackoverflow.com/a/56402079/7194995).

found that dynamic memory regions significantly impacted the performance of the model.

As for interrupts and exceptions, the MINRV8 architecture supports the call to machine-mode exception and external interrupts targeting machine-mode. An implementation of more types of interrupts adds additional complexity without serving a clear purpose besides the completeness of the model. Therefore, following the decision on supervisor-mode, simplicity was valued over completeness. Concerning the interrupt handling routine, it was decided to remove all optional parts from it, i.e. there is no interrupt delegation and there are no vectored interrupts. Furthermore, as all interrupts target machine-mode, there is no need to check the target of an interrupt or exception initially. This leaves MINRV8 with the following, very simple interrupt/exception handling routine:

1. If an interrupt is pending, check that interrupts are enabled globally and the specific interrupt is enabled as well
2. Take the trap
3. Return from trap handler

3.2. The MINRV8 Architecture

The architecture that will be model checked in this thesis is a minimal, RISC-V-inspired 8-bit architecture and shall be named MINRV8 from now on. A secure embedded system that implements the RV32E, i.e. base integer instruction set for embedded computing, was taken as a role model for this minimal architecture. The reasoning behind the architecture and features to include was given in section 3.1.3. In this section, we will introduce and briefly specify the architecture itself.

MINRV8 supports four general-purpose registers and has two privilege modes: machine- and user-mode. Besides this, it has 4-bytes of readable and writeable but non-executable memory divided into two regions the first of which ranges from addresses 0 to 1 whereas the other occupies the remaining addresses 2 to 3. MINRV8 supports basic instructions for memory reads and writes, integer arithmetic with $+$ and $-$, bit-shifts, bitwise-logical operations with AND and OR, a MOV instruction to move a word from one register to another, a comparison instruction, two instructions to switch privilege mode and instructions to read and write CSRs. A full list of all instructions can be found in table 3.2. Machine words are generally interpreted as signed-words. Thus, there will not be unsigned counterparts to integer arithmetic or the comparison instruction.

With this list of instructions, all instructions of the base integer instruction set of the RISC-V ISA are either left out deliberately, can be emulated, or have been made redundant by other design decisions of the MINRV8 ISA. As for what has been left out, it was already mentioned that jumps and branches will not be modelled. Since this removes the need for a PC, the add upper immediate to PC (AUIPC) instruction, which is supported by RISC-V, is made redundant as well. This decision is closely related to the way

the MINRV8 architecture was modelled. In section 3.2.1, an outlook on the reasoning behind this will be given.

Additionally, there is the EBREAK instruction that transfers control to a debugging environment. It was decided not to include debugging in the MINRV8 architecture since, as mentioned before, at its current state, the level of complexity offered by the architecture is sufficient to make it comparable to other modern RISC architectures. Finally, there is the FENCE instruction that allows to order memory access. As the PMAs do not support memory ordering, this instruction was left out as well.

For most computational instructions, RISC-V offers an immediate alternative that, instead of receiving two registers as arguments, receive one register and one immediate value from memory as arguments. These instructions can be emulated by first executing the LOADI instruction and then the respective non-immediate version of the instruction. Also, RISC-V supports the XOR, shift right logically (SRL), and atomic read/write word CSR (CSRRW) instructions that can both be emulated. XOR can be emulated by a combination of SUB[8], AND, and OR, SRL by SLL with a negative argument to shift the register by, and CSRRW by CSRRS and CSRRC executed subsequently.

Instructions that have been made redundant are all unsigned variants of computational instructions and load/store instructions of word sizes greater than bytes. These load/store instructions have been made redundant with the decision to limit the word size of MINRV8 to one byte.

All this gives strong assurance that the computational model of MINRV8 does not lack in any significant part in comparison to RISC-V.

Besides the general-purpose storage options, MINRV8 also includes three CSRs: an mstatus equivalent, a pmpcfg equivalent and a physical memory attributes configuration (pmacfg) register that implements the configuration PMAs as described in section 3.1.3. MINRV8 supports external interrupts as the only source of interrupts and an environment call to machine-mode as the only source of exceptions. Therefore, in the context of MINRV8, mstatus has 4 bits comprising two stacking bits for taking traps, i.e. MPP (machine-mode prior privilege) and MPIE (machine-mode prior interrupts enabled), a MEIP bit to signal pending external interrupts and an MIE bit to en- or disable interrupts globally. By this combination of CSRs, it was only included from RISC-V what was necessary to implement the features presented in section 3.1.3.

The pmacfg register defines cacheability of the two memory regions. Each memory region can either be set as uncacheable, write-back-cacheable, write-through-cacheable or write-protected-cacheable. These caching methods are inspired by the caching methods that are available in the x86 architecture from Intel as described in section 11.3 *Methods of Caching Available* of the *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide* [16]. This manual describes these

[8] SUB can be used to implement the bitwise, logical not, i.e. for a bit vector x of length n, the bitwise, logical not is given by $1^n - x$, where 1^n is the repeated concatenation of bit 1.

`LOAD rd, rs1`	Load the word stored in memory at the address stored in registers `rs1` modulo 4 into register `rd`
`STORE rs1, rs2`	Store the word located in register `rs2` into memory at address stored in register `rs1` modulo 4
`LOADI rd, imm`	Load the 8-bit immediate value `imm` into register `rd`
`ADD rd, rs1, rs2`	Set register `rd` to the sum of the values in registers `rs1` and `rs2`
`SUB rd, rs1, rs2`	Set register `rd` to the value of register `rs1` minus the value of register `rs2`
`AND rd, rs1, rs2`	Set register `rd` to the result of a bitwise-and of the values of registers `rs1` and `rs2`
`OR rd, rs1, rs2`	Same as `AND` but with the bitwise-or operation
`MOV rd, rs1`	Set register `rd` to the content of register `rs1`
`SLL rd, rs1, rs2`	Set register `rd` to the content of register `rs1` shifted left logically (i.e. without sign-extension) by the value in register `rs2`
`SRA rd, rs1, rs2`	Set register `rd` to the content of register `rs1` shifted right arithmetically (i.e. with sign-extension) by the value in register `rs2`
`SLT rd, rs1, rs2`	Set register `rd` to 0x01 if the value in register `rs1` is smaller than the value in register `rs2`
`ECALL`	Environment call to machine-mode
`MRET`	Machine-mode return from trap handler
`CSRRS rd, rs1, rs2`	Read the value of the CSR with index `rs1` modulo the number of all CSRs and store it in register `rd`; additionally, set all bits of the CSR high that are high in register `rs2`
`CSRRC rd, rs1, rs2`	Same as `CSRRS` but set all bits low that are low in register `rs2`

Table 3.2.: Instructions of the MINRV8 architecture

		Uncached	WB	WT	WP
Write	Hit	×	Fill	Fill if valid	Invalidate
	Miss	×	Fill	×	×
Read	Hit	×	Read	Read	Read
	Miss	×	Fill	Fill	Fill

× = no action

Table 3.3.: Caching Methods

methods of caching as follows:

> "[...]
>
> **Write-through (WT)** — [...] Reads come from cache on cache hits; read misses cause cache fills. [...] All writes are written to a cache line (when possible) and through to system memory. When writing through to memory, invalid cache lines are never filled, and cache lines are either filled or invalidated. [...]
>
> **Write-back (WB)** — [...] Reads come from cache lines on cache hits; read misses cause cache fills. [...] Write misses cause cache line fills [...], and writes are performed entirely in the cache, when possible. [...] The write-back memory type reduces bus traffic by eliminating many unnecessary writes to system memory. Writes to a cache line are not immediately forwarded to system memory; instead, they are accumulated in the cache. The modified cache lines are written to system memory later when a write-back operation is performed. Write-back operations are triggered when a cache lines need to be deallocated, such as when new cache lines are being allocated in a cache that is already full. [...]
>
> **Write-protected (WP)** — Reads come from cache lines when possible, and read misses cause cache fills. Writes are propagated to the system bus and cause corresponding cache lines on all processors on the bus to be invalidated. [...]" [16, p.11-7]

Cited paragraphs from the *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide* are summarized in table 3.3. In this setting, we decided to always fill the cache on write-hits for write-through-cacheable memory regions to make the caching methods as concise as possible, although the manual leaves it open to implementations to also invalidate the cache.

The pmpcfg register allows to set read and write privileges per memory region and allows to lock the settings of individual regions.

3.2.1. Restrictions of the MINRV8 Architecture

Someone experienced in the field of microcontrollers and processors might find MINRV8 to lack the crucial features of jump and branch instructions and the specifi-

cation of executable memory. These points were left out or left unspecified deliberately. The development of the MINRV8 architecture was tightly coupled to its implementation in nuXmv, which will be presented in section 3.3. In the implementation, it was decided to model the stream of instructions to the architecture as input variables. This makes a model of jump and branch instructions or executable memory pointless since input variables cannot be constrained. A more thorough discussion of this problem will be given in section 6.1.1 where the scope of the MINRV8 architecture will be analyzed - both in terms of what it tries to achieve and what it actually is capable of representing, i.e. how close it comes to real-life architectures.

3.3. Implementation of MINRV8 in nuXmv

With the MINRV8 ISA being set up, this section presents its implementation in nuXmv. Cf. section 2.5.3 for an introduction to this tool. The implementation of MINRV8 was split into two modules: the first module implements the general mechanisms of the ISA, the second implements caching. In analogy to this, section 3.3.1 will describe how the main module was implemented and section 3.3.2 will explain the implementation of the module related to caching. In each of these sections, the variables and their corresponding types will be stated declared as part of the modules after which the transition relation for each of the respective variables will be outlined.

The goal of this section is not to give a full description of the whole implementation but to give an overall understanding of the core ideas behind it[9]. The implementation includes practical solutions to certain downsides of nuXmv, and generalizations to ease the implementation as a whole. All these aspects are not relevant to the goal of this thesis. It should be noted that the macro processing engine pyexpander [17] was used to remove redundancies. nuXmv is undoubtedly not the most low-level language imaginable. However, programming nuXmv still leads to many redundancies in the codebase. pyexpander, in principle, serves the same purpose as the C-preprocessor but is much more powerful and allowed to get rid of most aforementioned redundancies.

3.3.1. Core Functionality

Variables The core of the MINRV8 architecture is modelled using four variables:

- `priv : boolean`
- `csrs : array 0..1 of unsigned word[8]`
- `regs : array 0..3 of signed word[8]`
- `memory : array 0..3 of signed word[8]`

[9] If the full implementation is of interest, the repository containing it is accessible via: https://github.com/felixlinker/ifc-rv-thesis

As the names suggest: `priv` indicates the current privilege mode where `TRUE` means that the hart is in machine-mode and `FALSE` means that the hart is in user-mode. `csrs` hold the three CSRs of the MINRV8 architecture. Technically, only 16 bits are needed to implement all CSRs of the MINRV8 architecture, which is why only two "physical" CSRs are modelled. This allows to implement a homogeneous interface for memory/register reads and writes and does not introduce unused bits to the implementation, i.e. dead state space that still requires traversal from nuXmv. mstatus is located in the lower half (bits 0 to 3) of `csrs[0]`, pmacfg in the upper half (bits 4 to 7) of `csrs[0]` and pmpcfg in `csrs[1]`. The variables `regs` and `memory` implement the registers and memory in a straight-forward fashion.

Furthermore, the model knows five input variables:

- `op`
- `rd : 0..3`
- `rs1 : 0..3`
- `rs2 : 0..3`
- `m_external_interrupt : unsigned word[1]`

`op`, `rd`, `rs1` and `rs2` represent a fully decoded instruction where `op` is the opcode as a symbolic constant and the other arguments point to registers with per-instruction determined semantics. When an instruction does not support some or all of the arguments to the instruction, the values of these input variables simply are ignored. The variable `m_external_interrupt` signals the event of a pending external interrupt from some source outside of the hart.

There is no input variable for an immediate value which might be needed by the LOADI instruction because otherwise, we would waste extra state space. Instead, the content of the destination register in the case of a LOADI instruction being executed is not constrained. This means that nuXmv can choose any value in the next state for register `i` if it happens to be targeted by instruction LOADI. Thereby, the value written to the destination register by the LOADI instruction is encoded implicitly in the transition relation rather than in the state space.

An overview of the implementation's modules is given in figure 3.4. On the left-hand side, the main module and its variables are depicted. Arrows indicate whether one variable influences another[10]. Sometimes, arrows are labeled to indicate the instruction on which a variable changes its value. Here, `Comp.` represents computational instructions such as ADD and `Sys.` stands for platform management instructions such as ECALL.

Transitions nuXmv allows constraining the transition of variables by arbitrary formulas expressed in propositional logic. The system can make a transition if it satisfies all formulas given as constraints. Therefore - if full control over the transition relation is desired - it often is not sufficient to simply constrain transitions by a group of

[10]For clarity reasons, it was omitted that CSRs can be written to and with registers.

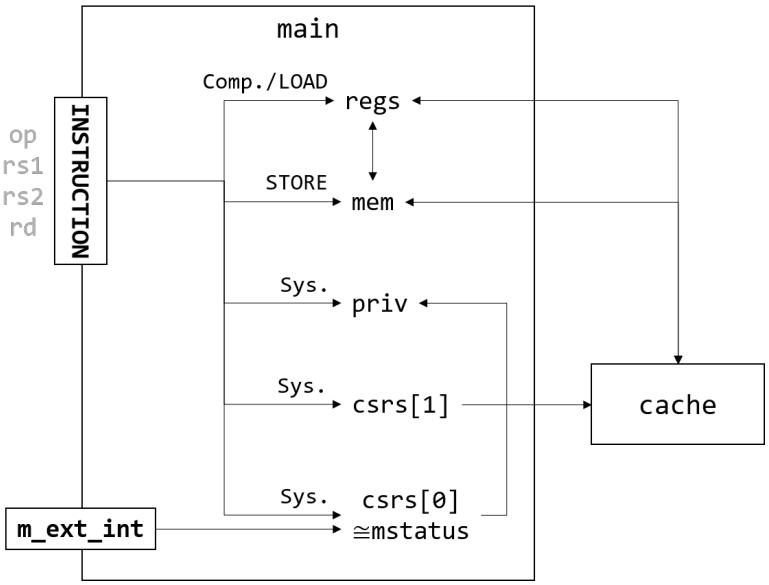


Figure 3.4.: Overview of all modules of the implementation

implications[11]. In the implementation for nuXmv of the MINRV8 architecture, `case`-expressions that always close with a default statement were used to ensure that transitions of variables are always fully constrained.

In general, all variables besides `priv` change their value in accordance with the flow-chart depicted in figure 3.5 with some exceptions depending on the variable. At first, it is checked whether a trap occurred. All variables besides `mstatus`, which is mapped to `csrs[0]`, are stable in this case. Then, it is checked whether the current instruction is targeting the respective variable: for example, for an ADD instruction, some index `i`, and the variable `regs`, check whether `rd == i`. Finally, the variable must only change its value, if the current instruction applies to the variable, e.g. a STORE instruction will only change values of `memory` but not `regs`. If all checks succeed, the respective instruction is evaluated and the variable is assigned with the instruction's result.

In the following, the "formal idea" of the variable transitions of the model will be introduced. Whenever there is an exception to the flow-chart depicted in figure 3.5, it will be made explicit. Examples for such exceptions are changes of `memory` on cache-line evictions or `mstatus` being influenced by the input variable `m_external_interrupt`. In this context, it is assumed that variables are stable unless otherwise mentioned.

The transitions of the `priv` variable are simple. There are two conditions by which `priv` can change. Firstly, whenever there is an exception - be it an exception triggered by a pending external interrupt or an environment call to machine-mode exception - the hart switches to machine-mode and does not execute the current instruction, i.e.

[11] The exception to this are groups of implications where the antecedents form a total case differentiation, i.e. where one antecedent is always true.

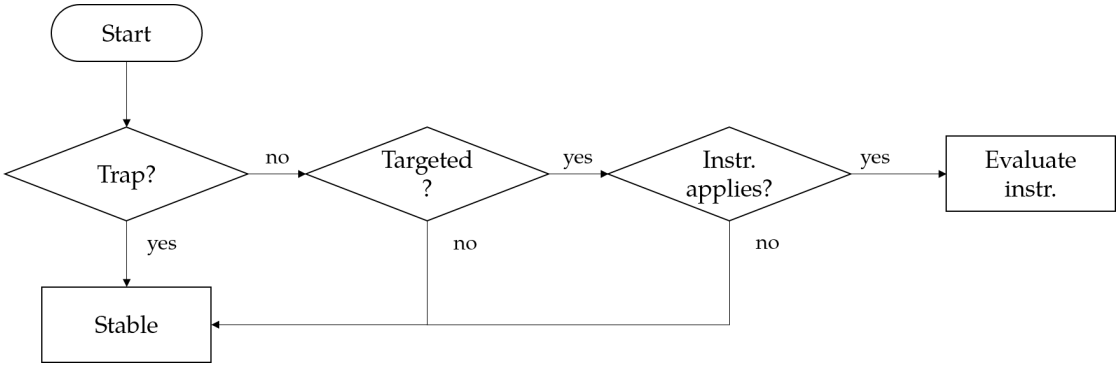


Figure 3.5.: Variable changing flow-chart

the value of `op`, `rd`, `rs1` and `rs2` are ignored. Secondly, when the MRET instruction is executed, the hart sets `priv` to the value stored in mstatus.MPP (cf. section 3.1.2).

mstatus changes under the same conditions, which leads to the transitions of `csrs`. On a trap, MPP (machine-mode prior privilege) and MPIE (machine-mode prior interrupts enabled) are written accordingly while MEIP (machine-mode external interrupt pending) and MIE (machine-mode interrupts enabled) are set to zero. MEIP can be set to zero although the trap might have been triggered by an ECALL instruction because interrupts have higher priority in RISC-V than synchronous exceptions. To understand why this is correct, there are two cases to consider. Either an external interrupt was set pending: then it must be serviced immediately and MEIP must be cleared, or it was not set pending: then writing MEIP with 0 does not change the state since MEIP was 0 to begin with. On return from machine-mode, i.e. execution of MRET, MIE is written with MPIE, MPIE is set to 1 and MPP is set to 0; all other bits are left unchanged. An exception to the rules mentioned above is the field MEIP which can be set high on every transition based on the value of `m_external_interrupt`.

All CSRs, pmacfg, pmpcfg, and partially mstatus, can also be written by the CSRRC or CSRRS instructions. In order to control read- and write-access to the CSRs, the model knows two constant arrays:

```
1 __csrs_read_privs := [ 0h_0F, 0h_00 ];
2 __csrs_write_privs := [ 0h_FF, 0h_FF ];
```

These arrays provide bitmasks that indicate the least privilege level necessary to read or write the CSR of the corresponding index at the corresponding bit-position. All CSRs besides mstatus can be read but no CSR can be written by user-mode. These constants are transformed to the bitmasks `csr_read_mask` or `csr_write_mask` respectively by the per-bit b implication of $b \rightarrow$ `priv` based on the index of the currently targeted CSR. Bits of CSRs are changed by the CSRRS or CSRRC instruction only if `csr_write_mask` is high at the corresponding index with two exceptions:

- mstatus.MEIP cannot be written by software

- Once a pmpcfg register has been locked, it cannot be changed anymore

The transition relation of the registers and memory is easier to understand. `regs` changes at index `i`, if it is targeted by some instruction and no trap is taken. Then, `regs[i]` is written with a value determined by the instruction semantics[12]. These per-instruction semantics include considering the `csr_read_mask` bitmask on `CSRRS` and `CSRRC`.

`memory` is written at index `i` if it is targeted by the `STORE` instruction, no trap occurs, and the write privilege settings in pmpcfg allow the current privilege mode to write the register. Otherwise, `memory` does not change. It does not mark an error, should the current privilege level not suffice to write the targeted memory address. No exception will be generated - the memory simply will not be written. The same holds for read privileges and `regs` on a `LOAD`.

3.3.2. Caching

The cache is implemented as its own module that gets instantiated by the main module. It increases the complexity of the memory's and registers' transition relation. The module comprises three variables:

- `addr : 0..3`
- `valid : boolean`
- `line : signed word[8]`

The type of the `line` variable indicates that this cache can only store one word, i.e. one byte, at a time. These variables transition depend on the cache-configuration of the respective memory region in pmacfg. As already introduced in section 3.2, a memory region can be either set as uncacheable, write-back-cacheable or write-through-cacheable. If a memory region is set as uncacheable, the cache will ignore reads and writes to this region and not make any transition.

Reads If a memory region is set as cacheable[13], reads will trigger a cache-fill. On a `LOAD`, if the cache is invalid or currently holds the contents of a different memory address than targeted, `addr` gets written with `regs[rs1] mod 4`, i.e. the address to read from, `valid` becomes `TRUE` and `line` is written with `memory[addr]`[14], i.e. the value to read. This process is independent of the current privilege level, e.g. user-mode attempting to read from a memory address causes a cache-fill regardless of whether user-mode can access the region. In this case, though, user-mode still will not be able to read the respective word as reads are privileged controlled.

[12] This only applies to instructions that *have* semantics targeting registers, e.g. `ECALL` will not change values of any register.

[13] In context of this thesis, "cacheable" means that a memory region is marked as write-back-cacheable, write-through-cacheable or write-protected-cacheable.

[14] `addr` here refers to the already updated value of that variable, i.e. `regs[rs1] mod 4`.

If, however, on a LOAD the cache already holds the targeted word, is valid and the current mode suffices to read the respective memory address, `line` instead of `memory[regs[rs1] mod 4]` will be written into register `rd`.

Writes From the cache's perspective, writes to regions which are set cacheable work the same way, regardless of the specific region's cacheability settings. For such a region, on a STORE, `line` is written with `regs[rs2]`, i.e. the word to write, `address` with `regs[rs1] mod 4`, i.e. the address to write to, and `valid` becomes `TRUE`. However, if the region is set to be write-through-cacheable, the write will also be reflected in memory whereas, for a write-back-cacheable region, the write will be reflected in cache only.

This brings the need to persist writes to the cache in memory when cache lines are evicted. For example, consider the following program:

```
1 STORE 1, 0   # store regs[0] at memory[regs[1] mod 4]
2 LOAD 0, 2    # read memory[regs[2] mod 4] to regs[0]
```

In general (i.e. when `regs[1] mod 4 != regs[2] mod 4`[15]), if the memory region targeted by `regs[1]` is set to be write-back-cacheable the write in the first line will not be persisted in memory but on cache level only. If furthermore the memory region targeted by `regs[2]` is set to be cacheable as well, the read in the second line will overwrite changes of the first instruction which is why the cache content must first be written to memory. Therefore, if the cache holds some not persisted values of write-back-cacheable memory regions and either the cache's target changes or the pmacfg attributes change such that the cache is not write-back-cacheable anymore, the content of the cache will be written to memory as well.

3.4. Summary

In this section, the core of the RISC-V ISA was introduced from which the MINRV8 architecture was derived (cf. section 3.1). MINRV8 supports machine- and user-mode as privilege levels and 15 instructions to perform general computation, read/write memory, load immediate values from memory, transition between privilege modes and read-/write CSRs (cf. table 3.2). It was shown that this list of instructions is complete in comparison to RISC-V minus very few but negligible instructions that were not included and instructions made redundant by design decisions or that can be emulated. Furthermore, MINRV8 supports caching of memory regions and PMP entries for memory regions, i.e. read- and write-permissions configurable for user-mode and to some extent for machine-mode (cf. section 3.2).

It was argued that MINRV8 provides a sufficient amount of complexity to make it comparable to modern RISC architectures besides not including a model of executable

[15] If this is not assumed, the LOAD instruction would simply read the word that has just been written from cache. The problematic instances are where the cache line is evicted because of the LOAD instruction.

memory and not supporting jumps and branches. This will be discussed in section 6.1.1 and is related to the decision to model the stream of instructions as input variables in nuXmv.

Finally, it was presented how MINRV8 was implemented in nuXmv (cf. section 3.3). The implementation was conducted by hand with the aid of the preprocessor engine pyexpander [17]. The file containing the model overall includes 630 lines of code where, for the most part, no line exceeded 80 characters.

Chapter 4.

Information Flow Control

With an architecture implemented in nuXmv, we can now specify information flow properties about this architecture or its model, respectively, to define how information is tracked in the model. The general idea of information flow tracking will be adopted from [Fer+17], which has been presented in section 2.2. To recapture, Ferraiuolo et al. defined three concepts:

- An information flow policy in the form of a lattice of security labels
- Information flow tracking via type-system rules
- Information flow control by type-checking

The information flow policy can be adopted directly. However, since the model checker is meant to perform the information flow tracking and control, the idea of using type-checking to implement it cannot be adopted directly as well. In this section, it will be discussed how information flow tracking and control was lifted to the architectural level and implemented in nuXmv.

In section 4.1, we will adopt the type system rules to instructions of the MINRV8 architecture. Section 4.2 then explains how information was tracked in nuXmv. Finally, in section 4.3, the information flow properties that constitute the information flow control itself will be given.

4.1. Information Flow Semantics for Instructions

In order to apply the aforementioned type system rules of [Fer+17] to instructions, the idea behind each rule will be investigated next to generate information flow semantics for instructions from those. We introduce an interpretation function $\mathcal{I}$ that maps each instruction to its information flow semantics. We give the semantics for instruction as functions. These functions receive the input to the instruction as arguments and map them to the new security labels generated by the instruction. Such input values are abstract entities. We denote the integer value of some input x by $v(x) \in \mathbb{Z}$ and its information flow tracking labels by $l(x)$, i.e. x must be understood as an abstract entity. The security labels in the architecture are tracked bitwise as words of length 8 over the

alphabet $\Sigma = \{\text{PT}, \text{PU}, \text{CT}, \text{CU}\}$, i.e. $l(x) \in \Sigma^8$. $\bot$ refers to the least element of Σ in regard to $\sqcup$, i.e. $\bot = \text{PT}$.

Example 7. The `SLL` instruction receives two arguments: the word to shift (`rs1`) and the word indicating the amount to shift the former by (`rs2`). It assigns the result of this operation to some register. The information flow tracking labels associated with the result of `SLL` are denoted by:

$$\mathcal{I}(\textsc{Sll})(\texttt{rs1}, \texttt{rs2})$$

Note, that $\mathcal{I}$ does not hold semantics for *where* information will flow; $\mathcal{I}$ solely determines *how* information flows, i.e. how labels change.

As we will be working with formal words, we will introduce some definitions to work with these first. For $w \in \Sigma^*$, $|w|$ denotes its length, $\cdot$ is used as word concatenation with its generalization over indices $\prod$, and $\times$ is the repeated concatenation of a word. As we try to stay close to bit vectors, words of security labels are zero-indexed and sliced from right to left, i.e. $abcd[0] = d$ and $abcd[1:0] = cd$.

Example 8. Let $w = abcd \in \Sigma^*$.

- $ab \cdot cd = abcd$
- $\prod_{i \in (|w|,0]} w[i:0] = abcd \cdot bcd \cdot cd \cdot d$
- $3 \times abc = abc \cdot abc \cdot abc$

Furthermore, we introduce $\ll\colon \Sigma^8 \times \mathbb{Z} \to \Sigma^8$ as well as $\gg\colon \Sigma^8 \times \mathbb{Z} \to \Sigma^8$ as bit shift counterparts for words; these word shift operators are recursively defined on Σ as follows:

Let $w \in \Sigma^8$ and $w', w'' \in \Sigma^*$ such that $a \cdot w' = a \cdot w'' \cdot a' = w$ and $1 < n$.

$$\begin{aligned}(a \cdot w') \ll 1 &= w' \cdot \bot \\ w \ll n &= (w \ll 1) \ll (n-1)\end{aligned}$$

$$\begin{aligned}(a \cdot w'' \cdot a') \gg 1 &= a \cdot a \cdot w'' \\ w \gg n &= (w \gg 1) \gg (n-1)\end{aligned}$$

To extend this definition for shifts by amounts in $\mathbb{Z}$ we say that $w \ll 0 = w \gg 0 = w$ and:

Let $n < 0$.

$$\begin{aligned}w \ll n &= w \gg |n| \\ w \gg n &= w \ll |n|\end{aligned}$$

As you can see, the left shift behaves as to be expected and shifts in the least element of Σ. For actual bit vectors, this would be the 0. The right shift operation, on the other hand, does a sign extension because MINRV8 knows signed words only.

LOAD	×
STORE	
LOADI	
ADD	T-Arith
SUB	
AND	T-Logical
OR	
MOV	×
SLL	T-LShift
SRA	T-RShift
ECALL	×
MRET	
CSRRS	
CSRRC	

× = no mapping

Table 4.1.: Instruction to type system rule mapping

Example 9. Let $w \in \Sigma^5$.

$$\begin{aligned} \mathrm{PT} \cdot \mathrm{PU} \cdot w \cdot \mathrm{CT} \ll 2 &= \mathrm{PU} \cdot w \cdot \mathrm{CT} \cdot \bot \ll 1 \\ &= w \cdot \mathrm{CT} \cdot \bot \cdot \bot \\ &= w \cdot \mathrm{CT} \cdot \mathrm{PT} \cdot \mathrm{PT} \end{aligned}$$

$$\begin{aligned} \mathrm{CU} \cdot w \cdot \mathrm{PU} \cdot \mathrm{CT} \gg 2 &= \mathrm{CU} \cdot \mathrm{CU} \cdot w \cdot \mathrm{PU} \gg 1 \\ &= \mathrm{CU} \cdot \mathrm{CU} \cdot \mathrm{CU} \cdot w \end{aligned}$$

At last, the $\sqcup$-supremum operator on Σ is extended to Σ^* by applying it literal-wise to a word, e.g. $ab \sqcup cd = (a \sqcup c) \cdot (b \sqcup d)$.

With these tools at hand, the translation of type system rules to semantic functions will now be presented. Recall that MINRV8 offers 15 instructions. For many instructions, there are type system rules that can be mapped directly. E.g. there is no semantic difference between the ADD instruction and performing addition in code. However, there are some instructions that have no corresponding type-system rule in [Fer+17]. An overview of this mapping is given in table 4.1.

When a respective type system rule was assigned to an instruction, it should be obvious that the instruction actually resembles the type system rule. For all remaining instructions, consider the list of unused type system rules in [Fer+17]: T-Const (for constant expressions), T-Var (lifting types of variables), T-Concat (for bit vector concatenation) and T-ArrIndex (for array indexing). It is obvious that most unmapped instructions in table 4.1 can be mapped to any of the rules just listed. The only exception to this is the MOV instruction the information flow of which might be specified by the T-Var rule.

However, this rule does not type variable assignments which might be of interest but variables itself.

In the following paragraphs, we will give the information flow semantics of those instructions that have a corresponding type system rule. After that, we discuss the semantics for all remaining instructions. It will turn out that although there are no type system rules to derive the semantics from, the information flow associated with these instructions is rather simple and therefore can be given directly.

Firstly, the type system rule T-Logical formalizing the information flow of logical operators shall be translated. This rule infers the type τ of the expressions $e_1 \wedge e_2$ and $e_1 \vee e_2$ when e_1 and e_2 are of type τ_1 and τ_2 respectively by mapping each bit of the resulting word to the supremum of the labels of the argument words. This is intuitive as for $\wedge$ and $\vee$ each bit can only be influenced by the two bits of the same index in the argument words. This leads us to the following semantical function of AND and OR:

$$\mathcal{I}(\text{AND}) = \mathcal{I}(\text{OR}) = (w, w') \mapsto l(w) \sqcup l(w')$$

The translation of the T-Arith rule which will apply to the interpretation of the instructions ADD and SUB is analogously straight-forward. In this case, the inferred type τ of $e_1 + e_2$ and $e_1 - e_2$ is defined as:

$$\tau = i \mapsto \bigsqcup_{j \in [1,i]} (\tau_1(j) \sqcup \tau_2(j))$$

Although [Fer+17] not explicitly mentions whether bit vectors are one- or zero-indexed, it can be assumed for them to be one-indexed as the authors write: "The rule T-Arith must track the bits that are propagated by carry bits. The i^{th} bit of the result is affected by all bits below i from both inputs." [Fer+17] As the definition of τ uses indexing over the interval $[1, i]$ it can be assumed for bit vectors to be one indexed as the alternative, i.e. zero-indexing, would contradict each bit being affected by "all bits below" - the definition would skip bit zero.

The label of each bit of the result is inferred by taking the supremum of all lower bits in order to deal with the possible carrying of bits. This leads to a partially ordered sequence of security labels, i.e. we have $\forall i_1 < i_2 . \tau(i_1) \preceq \tau(i_2)$ where $\preceq$ denotes the partial order on Σ induced by $\sqcup$. This notion is adopted to words of labels by the helper function extendSup, which is defined as follows:

$$\text{extendSup} = x \mapsto \prod_{i \in (|x|,0]} \left(\bigsqcup_{j \in [0,i]} x[j] \right)$$

extendSup implements the idea of the mapping τ from the rule T-Arith and subsequently *extends* the label of each index i to all indices greater than i which map to a $\preceq$-smaller label.

Example 10.

$$\text{extendSup}(\text{CU} \cdot \text{PU} \cdot \text{PT}) = \text{CU} \cdot \text{PU} \cdot \text{PT}$$

$$\text{extendSup}(\text{PU} \cdot \text{CU} \cdot \text{PU} \cdot \text{PT} \cdot \text{PU}) = \text{CU} \cdot \text{CU} \cdot \text{PU} \cdot \text{PU} \cdot \text{PU}$$

With extendSup, the semantics of the ADD and SUB instructions can be defined.

$$\mathcal{I}(\textsc{Add}) = \mathcal{I}(\textsc{Sub}) = (w, w') \mapsto \text{extendSup}\big(l(w) \sqcup l(w')\big)$$

Example 11. Consider the addition of two bit vectors which generally are considered to be public and untrusted with one bit being confidential and trusted.

$$\begin{aligned}
&\mathcal{I}(\textsc{Add})(\text{PU} \cdot \text{CT} \cdot \text{PU}, \text{PU} \cdot \text{PU} \cdot \text{PU}) \\
=&\text{extendSup}((\text{PU} \cdot \text{CT} \cdot \text{PU}) \sqcup (\text{PU} \cdot \text{PU} \cdot \text{PU})) \\
=&\text{extendSup}(\text{PU} \cdot \text{CU} \cdot \text{PU}) \\
=&\text{CU} \cdot \text{CU} \cdot \text{PU}
\end{aligned}$$

T-LShift and T-RShift infer the type of $e \ll n$ and $e \gg n$ by simply shifting the labels. The biggest part of these type system rules has already been translated in the definition of $\ll$ and $\gg$ on security label words with the only difference being that shifts in SecVerilogBL are unsigned. Therefore in the type system rules, $\bot$ is shifted in not only for left shifts but also for right shifts; this exception has been already coped with in the definition of $\gg$.

However, in [Fer+17] the authors also only know one source of labels, i.e. the word that is shifted, because shifts can only occur by constant amounts[1]. In the context of this thesis, the value of a register determines the amount a word will be shifted by, i.e. that amount is dynamic and has its own security labels. We, therefore, extend the interpretation of the SLL and SRA instructions by taking the maximum label of the amount the word will be shifted by and applying this label to all labels of the shifted word. The reasoning behind this is that each bit of the word shifted is influenced by the amount it will be shifted by.

$$\mathcal{I}(\textsc{Sll}) = (w, s) \mapsto \big(l(w) \ll v(s)\big) \sqcup \Big(|l(s)| \times \bigsqcup l(s)\Big)$$
$$\mathcal{I}(\textsc{Sra}) = (w, s) \mapsto \big(l(w) \gg v(s)\big) \sqcup \Big(|l(s)| \times \bigsqcup l(s)\Big)$$

Now, the only thing left is to give semantics for all the instructions that do not have an analogous type system rule, i.e. SLT, MOV, LOAD, STORE, CSRRS and CSRRC, LOADI.

SLT sets only one bit of the targeted register thus its semantics also should set only one literal of the resulting information flow tracking labels. As all bits are taken into account when comparing two bit vectors, the resulting bit is set to the maximum of all bits of both arguments.

$$\mathcal{I}(\textsc{Slt}) = (w, w') \mapsto (7 \times \bot) \cdot \Big(\bigsqcup l(w) \sqcup \bigsqcup l(w')\Big)$$

The semantics of MOV is rather simple. Since MOV only *moves* information, the labels of a moved word must not change:

$$\mathcal{I}(\text{Mov}) = w \mapsto l(w)$$

[1] Recall that constants in SecVerilogBL are of type $\bot$, i.e. do not need to be considered further as $x \sqcup \bot = x$.

Similar is true for LOAD and STORE. These instructions also simply move values through the architecture and therefore also simply cascade security labels. However, as memory addressing is dynamic via indices stored in registers the labels to propagate for a read are not associated with the inputs to the instruction itself but with the memory location that is read. The auxiliary function memoryLabel that maps memory addresses to the corresponding labels copes with this. It allows to define the semantics of LOAD and STORE as follows:

$$\mathcal{I}(\textsc{Load}) = a \mapsto \text{memoryLabel}\big(v(a)\big)$$
$$\mathcal{I}(\textsc{Store}) = (a, d) \mapsto l(d)$$

A similar auxiliary function is needed for the semantics of CSRRS and CSRRC. The auxiliary function csrLabel that maps CSR-indices to the corresponding information flow tracking labels deals with CSR reads as these happen via index as well.

$$\mathcal{I}(\textsc{Csrrs}) = \mathcal{I}(\textsc{Csrrc}) = (r, w) \mapsto \text{csrLabel}\big(v(r)\big)$$

Finally, the semantics of LOADI is coupled to the current execution context. Under the given threat-scenario (cf. section 2.4.3), it is sensible to assume that constants loaded by machine-mode are always labelled with CT and constants loaded by user-mode are always labeled with PU.

$$\mathcal{I}(\textsc{Loadi}) = c \mapsto \begin{cases} 8 \times \text{CT} & \text{if current mode is machine-mode} \\ 8 \times \text{PU} & \text{if current mode is user-mode} \end{cases}$$

Summary A final overview of all information flow semantics for instructions that have been introduced in this section can be found in figure 4.1. Note that not all labels of all inputs are used when generating new labels; these include[2]:

- The label of the address register a for the LOAD and STORE instructions
- The label of the CSR-write register w for the CSRRS and CSRRC instructions

The labels of these values have been left untouched purposefully. This does not mean that these labels do not matter to the system as a whole. However, they do not constitute the new labels of the actual *values* moved through the architecture. The labels of these values will be used by the properties the architecture will be verified against. More on this in section 4.3.

[2]The label of c for LOADI is also not being used but in this case, the constant c does not have a label assigned as it is not a proper value of the architecture. This is not made explicit by our formalisms as they do not differentiate between such *proper values* and hard-coded constants but is inline with the type system rules in [Fer+17].

$$
\begin{aligned}
\text{extendSup} &= x \mapsto \prod_{i \in (|x|,0]} \Big(\bigsqcup_{j \in [0,i]} x[j] \Big) \\
\mathcal{I}(\textsc{And}) &= \mathcal{I}(\textsc{Or}) = (w, w') \mapsto l(w) \sqcup l(w') \\
\mathcal{I}(\textsc{Add}) &= \mathcal{I}(\textsc{Sub}) = (w, w') \mapsto \text{extendSup}\big(l(w) \sqcup l(w')\big) \\
\mathcal{I}(\textsc{Sll}) &= (w, s) \mapsto \big(l(w) \ll v(s)\big) \sqcup \Big(|l(s)| \times \bigsqcup l(s)\Big) \\
\mathcal{I}(\textsc{Sra}) &= (w, s) \mapsto \big(l(w) \gg v(s)\big) \sqcup \Big(|l(s)| \times \bigsqcup l(s)\Big) \\
\mathcal{I}(\textsc{Slt}) &= (w, w') \mapsto \bot\bot\bot\bot\bot\bot\bot\bot \cdot \Big(\bigsqcup l(w) \sqcup \bigsqcup l(w')\Big) \\
\mathcal{I}(\textsc{Mov}) &= w \mapsto l(w) \\
\mathcal{I}(\textsc{Load}) &= a \mapsto \text{memoryLabel}\big(v(a)\big) \\
\mathcal{I}(\textsc{Store}) &= (a, d) \mapsto l(d) \\
\mathcal{I}(\textsc{Csrrs}) = \mathcal{I}(\textsc{Csrrc}) &= (r, w) \mapsto \text{csrLabel}\big(v(r)\big) \\
\mathcal{I}(\textsc{Loadi}) &= c \mapsto \begin{cases} 8 \times \text{CT} & \text{if current mode is machine-mode} \\ 8 \times \text{PU} & \text{if current mode is user-mode} \end{cases}
\end{aligned}
$$

Figure 4.1.: Information flow semantics for instructions

4.2. Implementation of Information Flow Tracking

In order to implement the information flow tracking semantics described in section 4.1 we define a binary representation of the lattice of security labels given by Σ and $\sqcup$. Let $\Sigma_{01} = \{00, 01, 10, 11\}$ with a mapping $\varphi : \Sigma \to \Sigma_{01}$ such that:

$$
\begin{aligned}
\varphi(\text{PT}) &= 01 \\
\varphi(\text{PU}) &= 00 \\
\varphi(\text{CT}) &= 11 \\
\varphi(\text{CU}) &= 10
\end{aligned}
$$

Let $\sqcup$ be defined on Σ_{01} such that φ is an isomorphism, e.g. $00 \sqcup 11 = 10$. This mapping is depicted in figure 4.2. Note that public and untrusted are encoded by a 0 whereas confidential and trusted are encoded by a 1. This abstraction can be formalized by examining two equivalence-relations or -classes on Σ and Σ_{01}. We define two partitionings $\Sigma/_{\equiv_C}$ and $\Sigma/_{\equiv_I}$ on Σ which induce corresponding equivalence classes of security labels modulo confidentiality $\equiv_C$ or modulo integrity $\equiv_I$, respectively.

$$
\begin{aligned}
\Sigma/_{\equiv_C} &= \big\{\{\text{PU}, \text{PT}\}, \{\text{CU}, \text{CT}\}\big\} \\
\Sigma/_{\equiv_I} &= \big\{\{\text{PU}, \text{CU}\}, \{\text{PT}, \text{CT}\}\big\}
\end{aligned}
$$

We coin the names $\text{P} = [\text{PU}]_{\equiv_C}$, $\text{C} = [\text{CU}]_{\equiv_C}$, $\text{U} = [\text{PU}]_{\equiv_I}$ and $\text{T} = [\text{PT}]_{\equiv_I}$ and lift $\sqcup$ to be defined on equivalence-classes in the usual fashion, i.e. $[x] \sqcup [y] = [x \sqcup y]$. Note

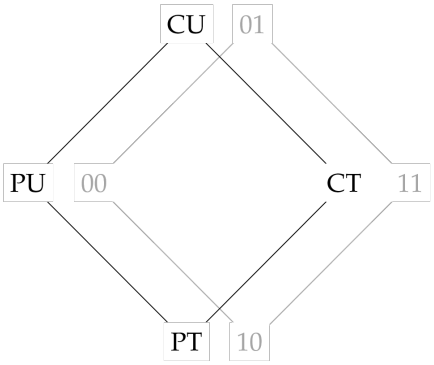


Figure 4.2.: Security lattice for SecVerilogBL [Fer+17] with binary equivalent

that this allows lifting of the $\preceq$ order on Σ as well, i.e. $P \preceq C$ and $T \preceq U$ where $P \preceq C$ if and only if there is no label $l_1 \in C$ such that for any $l_2 \in P$ one has $l_1 \preceq l_2$, and that all labels can be represented by the intersection of the respective equivalence classes,i.e. $\{PU\} = P \cap U$. Furthermore, it is possible to define $\sqcup$ solely by working with these equivalence classes:

$$\{x \sqcup y\} = \left(([x]_{\equiv_C} \sqcup [y]_{\equiv_C}) \cap ([x]_{\equiv_I} \sqcup [y]_{\equiv_I})\right)$$

Example 12.

$$\begin{aligned}\{PU \sqcup CT\} &= \left(([PU]_{\equiv_C} \sqcup [CT]_{\equiv_C}) \cap ([PU]_{\equiv_I} \sqcup [CT]_{\equiv_I})\right)\\ &= \left((P \sqcup C) \cap (U \sqcup T)\right)\\ &= (C \cap U)\\ &= \{CU\}\end{aligned}$$

This means that labels of the lattice $(\Sigma, \sqcup, \sqcap)$ can be tracked by considering the confidentiality and integrity parts individually. In other words: in general, one can calculate the supremum or infimum of two labels by performing the respective operation on the parts in the domains of confidentiality and integrity individually. This is illustrated in figure 4.3 where again the lattice of security labels is shown but there also are two axes which divide each domain in half. There, you can see why, e.g. $\{PT\} = P \cap T$ and also that $P \sqcup C = C$ or $T \sqcup U = U$.

Dealing with information flow labels in this way also is much more intuitive:

- Two sources of information combined are considered to be confidential if *any* of the sources are confidential. This is sensible as with some result of an operation at least partial knowledge about its confidential source can be inferred- depending on whether the public source or the operation are known.
- Two sources of information combined are considered to be trustworthy if *both* of the sources are trustworthy themselves. This is intuitive, as well. Non-trustworthy

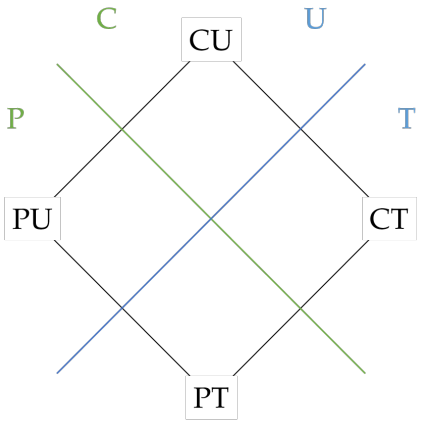


Figure 4.3.: Security lattice for SecVerilogBL [Fer+17] with equivalence-classes

sources of information are assumed to be controlled by an attacker. Some piece of information can only be trustworthy if it can be ensured that the attacker does not control it in any way. This is only given if both of the sources of an operation are not controlled by an attacker, i.e. are trustworthy.

The intuition behind $\sqcup$ as described above perfectly matches Σ_{01} as well. Recall that C is represented by a 1 and P by a 0 in the binary representation. That means that two labels $a = a_C a_I \in \Sigma_{01}$ and $b = b_C b_I \in \Sigma_{01}$ are in C, i.e. confidential, if a or b is confidential, i.e. if *one* of the bits representing the confidentiality of the labels is high, i.e. if $a_C \vee b_C$. In turn, these two labels are in T, i.e. trusted, if a and b are trusted, i.e. if *both* of the bits representing the integrity of the labels are high, i.e. if $a_I \wedge b_I$. This allows us to define $\sqcup$ on Σ_{01} again - this time not just semantically:

$$a_C a_I \sqcup b_C b_I = (a_C \vee b_C) \cdot (a_I \wedge b_I)$$

In summary, these paragraphs showed two things:

1. Information flow labels of information can be tracked by using separate labels for the domain of confidentiality and the domain of integrity which in turn can be stored independently of each other. This means that the implementation of the MINRV8 architecture does not need to work with the full lattice of information flow labels itself. It is sufficient to deal with equivalence classes which can be inferred from individual bits.

2. The $\sqcup$ operation on the binary representation Σ_{01} of the security labels can be implemented by concatenating the logical disjunction ($\vee$) of the confidentiality bits and the logical conjunction ($\wedge$) of the integrity bits.

This now allows for adding security label tracking to the model in nuXmv. Labels are assigned on a per-bit basis to all values in registers, memory, CSRs and the cache. For

Variable	Confidentiality Tracking	Integrity Tracking
`regs`	`regs_conf`	`regs_integrity`
`memory`	`memory_conf`	`memory_integrity`
`csrs`	`__csrs_conf`	`__csrs_integrity`
`cache.line`	`cache.conf`	`cache.integrity`

Table 4.2.: Information Flow Tracking Variables

each variable in the model, a confidentiality or integrity tracking counterpart as (arrays of) unsigned word(s) is added; find an overview of all such variables in table 4.2.

These variables transition under the same conditions as their base variable does, e.g. `regs_conf` does a transition if and only if `regs` transitions and `regs_conf` will deduct its content from sources parallel to those of `regs`. The values themselves change based on the current instruction and its information flow tracking semantics as described in section 4.1. The only exception to this is the information flow tracking of the variable `csrs` the labels of which do not change at all. CSRs are assigned constants labels because they do not store information but reflect the architectures state. Certain parts of the architectural state might be for example confidential but the characteristic of being confidential is not determined by the actual value that the respective CSR is written with but the kind of state that is defined by it.

In other words, the `..._conf` and `..._integrity` variables mirror the architectural part of implementation as described in section 3.3. The domain of confidentiality and the domain of integrity of the MINRV8 architecture are tracked in addition to simulating the pure computational world where variables implementing the information flow tracking change in parallel to their counterparts in the computational world.

4.3. Information Flow Properties

Finally, the information flow properties constituting the information flow control will be defined now. Each property is expressed as an LTL formula of the form `assumptions -> G expr`. `assumptions` is a macro that will be replaced by a conjunction of assumptions that assume programs running on the architecture to obey the rules to secure usage of the architecture as discussed in 2.6.

In the summary of section 4.1, it was mentioned that there are some arguments to the information flow semantic functions in $\mathcal{I}$ that are not used to determine the information flow label of some architectural word. These induce the first properties the MINRV8 architecture will be verified against.

Firstly, recall the definition of the LOAD and STORE semantics. The new tracking labels resulting from executing such an instruction solely depend on the labels of the word loaded or stored. What has not been used in the definition of information flow is the label of the target address of the memory operation. It is intuitive that the address does not influence the label of the information itself. This leads to the following property:

I. Whenever a memory operation is performed in privileged mode, the target address must be trusted.

```
1 LTLSPEC NAME MEMORY_OP_INTEGRITY :=
2     assumptions -> G (
3         priv & op in { LOAD, STORE }
4         -> (regs_integrity[rs1] & 0h_03) = 0h_03
5     );
```

Snippet 4.1: Implementation of property I.

Secondly, recall the definition of the semantics for CSRRS or CSRRC, respectively. These only propagate information flow tracking labels by assigning the constant labels of the CSR read to the respectively targeted register. We decided to model the labels of CSRs as constants because they control the state of the architecture and, as such, might be the target of an attack. We labelled all CSRs as trusted and mstatus as confidential. We decided to not label pmacfg or pmpcfg as confidential since it can be assumed that user-mode must be able to know where memory regions are to operate correctly.

Coming back to the information flow semantics of CSRRS and CSRRC, notice that in the respective definition of the semantical functions the value, the CSRs are written with, is not used. This must necessarily be the case since the labels of the CSRs are constant and as such cannot change based on the labels of the value to write them with. However, this leads us to phrase a property about the CSR labels that ensures their integrity:

II. Whenever a CSR is written, the value it is written with must be trusted.

```
1 LTLSPEC NAME CSR_INTEGRITY :=
2     assumptions -> G (
3         priv & op in { CSRRS, CSRRC }
4         -> regs_integrity[rs2] = 0h_FF
5     );
```

Snippet 4.2: Implementation of property II.

Both of these properties are founded in intuition as well. It is not safe to have an attacker control the load and store instructions of machine-mode since this could lead to unwanted behaviour of code. One might object that this is not problematic since, if a malicious value was to be loaded by machine-mode, information flow tracking should catch this. However, this must not necessarily be the case. An attacker could trick machine-mode into using a value that is labelled as trusted but coincidentally diverges the control flow of machine-mode in a way that is beneficial to an attacker. Additionally, it is obvious that CSRs should only be written with trusted data.

From all the variables our implementation of the MINRV8 architecture comprises we now considered regs and memory as well as cache in regards to the security of the

architecture by looking at the gaps of the information flow semantic functions. We also considered the variable `csrs` by assigning constant information flow labels to it and using it as an information flow target, i.e. we phrased properties ensuring that certain flows of information *to* `csrs` do not occur. However, up to this point, the `priv` variable, which holds the current privilege mode, was only handled implicitly. It was used as a source of labels since it determines the labels of machine words loaded as immediates by LOADI.

It is reasonable to assume, though, that `priv` also can serve as an information flow target - similar to `csrs`. We already did this partially for property I. where we assumed the architecture to be in privileged mode. These properties, however, only dealt with the integrity labels of information. The domain of confidentiality has yet been left out so we will introduce one final property for verification that covers the interplay between privilege mode and confidentiality labels:

III. Whenever the architecture is in user-mode, there is no confidential data in any register.

```
1 LTLSPEC NAME NO_LEAK :=
2     assumptions -> G (priv | (
3         regs_conf[0] = 0h_00
4         & regs_conf[1] = 0h_00
5         & regs_conf[2] = 0h_00
6         & regs_conf[3] = 0h_00
7     ));
```

Snippet 4.3: Implementation of property III.

4.4. Summary

In this chapter, information flow semantics was given for each instruction of the MINRV8 architecture and it was described how these were implemented in nuXmv. It was shown that the domains of confidentiality and integrity of the information flow policy can be tacked independently as bits where the $\sqcup$ operation on confidentiality labels is given by the logical disjunction and on integrity labels by the logical conjunction.

Labels of arguments to instructions not being used by their respective information flow semantics were used to constitute the first information flow properties regarding integrity of instructions. Additionally, the `priv` variable was considered when defining a property touching the confidentiality of data. In the end, all variables of the main module and all labels of arguments to instructions were considered such that they either led to an information flow property or are used to determine information flow in the model.

The properties that were established are:

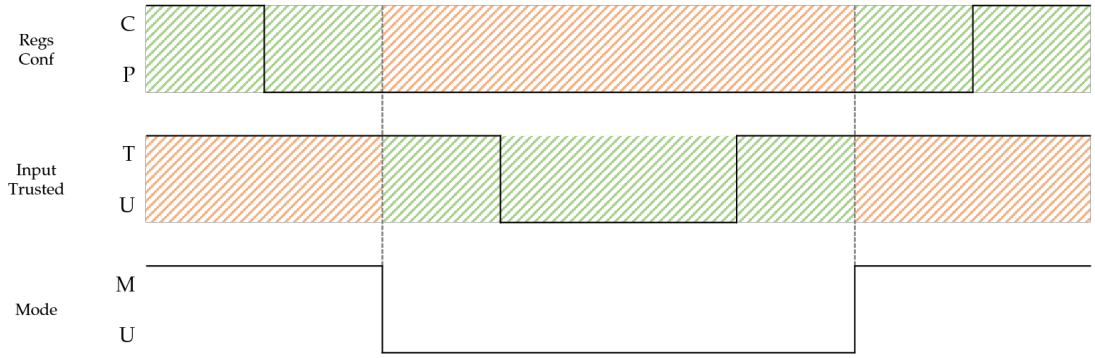


Figure 4.4.: Visualization of the Information Flow Properties

I. `MEMORY_OP_INTEGRITY`: whenever a memory operation is performed in privileged mode, the target address must be trusted.

II. `CSR_INTEGRTY`: whenever a CSR is written, the value it is written with must be trusted.

III. `NO_LEAK`: whenever the architecture is in user-mode, there is no confidential data in any register.

These properties are visualized in figure 4.4, inspired by digital timing diagrams. The last row visualizes the privilege mode of the architecture over the course of executing a sequence of instructions. There are two points in which the privilege mode changes: first, it changes from machine- to user-mode, then it changes back from user- to machine-mode. The `NO_LEAK` property (III.) mandates that the confidentiality labels of the registers always resemble the first row in the diagram, i.e. they are only high when the current privilege mode also is high. The green areas highlight where the registers could contain confidential data, i.e. the confidentiality labels *could* be high, but the orange areas highlight the point in time where the confidentiality labels *must* be low. The properties `MEMORY_OP_INTEGRITY` (I.) and `CSR_INTEGRITY` (II.) follow the same visualization. In the second row of the figure, it is visualized whether inputs to memory or CSR related instructions are trusted. The integrity labels of the input of these instructions must be high, whenever the architecture is in machine-mode and can be low if the architecture is on user-mode. Analogously, the green areas mark where the integrity labels *could* be low, but the orange areas indicate where it *must* be high.

Chapter 5.

Results

This section will cover the results of the verification process as it was described in section 2.6. The main finding is that there are eight assumptions which in total grant that the properties as described in section 4.3 cannot be violated, in other words: the MINRV8 architecture models the information flow properties introduced in this thesis, namely `MEMORY_OP_INTEGRITY`, `CSR_INTEGRITY` and `NO_LEAK`, if and only if the eight assumptions that will be introduced in subsequent sections are in place.

Additionally, roughly 23 bug fix patches[1] were applied to the model and some **`INIT`** constraints were set up. Taken together with aforementioned assumptions, these patches mark a fixpoint of the prove-refine loop discussed in section 2.6 and depicted in figure 2.6.

This section is grouped as follows: section 5.1 will introduce the aforementioned assumptions and **`INIT`** constraints.

Since only assumptions and no architectural refinements were introduced to the model[2], it was also tested how the results would change when canaries are introduced to the architecture. The term "canary" refers to birds historically used in coal mines to indicate a loss of oxygen. In the field of software development, this term refers to bugs that are added to code bases deliberately to test detection mechanisms. These canaries mimic known vulnerabilities to real-world architectures and will therefore also give an overview of the capabilities of the verification approach of this thesis. The results of these tests will finally be presented in section 5.2.

5.1. Assumptions

Technically, **`INIT`** constraints and assumptions serve the same purpose: They limit the state spaces that will be searched for property violations. Conceptually, however, these two concepts should be distinguished. In the model, **`INIT`** constraints are used to limit

[1] The number of 23 bug fix patches was measured by counting the amount of patches in the git graph containing the word "fix" in its description and being applied to the model file.

[2] It was tested whether two assumptions could be replaced by one architectural mitigation. It turned out, that this architectural refinement did not manage to rule out the respective vulnerability. This is discussed at the end of appendix A.1.

the search of the state space to *valid* counter-examples only. Validity in this context means, that properties are not false to begin with and that the initial state of the architecture complies with the specification. Consequently, these constraints were introduced during the prove-refine loop as fixes to the model. Assumptions, on the other hand, are not part of the model in a technical sense but mark a result of the verification of the architecture: Each assumption constitutes a rule, e.g. for OS or compiler engineers to obey to write or generate non-vulnerable code.

5.1.1. Initial Constraints

Validity of the counter-examples was guaranteed by 13 **`INIT`** constraints that can be grouped into four categories. Firstly, two groups of constraints guarantee that counter-examples are not false upfront. By constraining the contents of registers to only contain confidential data when the architecture starts in machine-mode and to only contain malicious data when the architecture starts in user-mode. Furthermore, cache and memory can only contain confidential data if the respective memory region is set to *not* be readable for user-mode and can only contain malicious data if the respective memory region *is* set to be writable by user-mode. For example, assume, the model would not be constrained to initially only contain confidential data in some register when in machine-mode. In this case, nuXmv returns a trivial counter-example violating the `NO_LEAK` property (III.) where the architecture starts in user-mode and has confidential data in some register.

Secondly, two more groups of constraints ensure the initial state to comply with the specification is guaranteed. The first is rather simple, given by a single constraint and depicted in the following snippet:

```
1  INIT MIE = 0b_1 -> MPP = 0b_0;
```

Snippet 5.1: mstatus **`INIT`** constraint for the model

This constraint ensures that the interrupt handling mechanisms are set up correctly as mandated by the RISC-V specification (cf. section 3.1.2 and figure 3.3). As `MPP` is cleared on return from a trap handler and `MIE` being high implies that no interrupt is being handled at the moment, `MPP` must be low.

The last group of constraints sets up caching errorless, i.e. if the cache is valid, it can only point to an address of a region that is cacheable and must match that addresses content unless the respective region is set to be write-back-cacheable.

5.1.2. Property Assumptions

During the prove-refine loop, whenever it was found that a counter-example to a property reported by nuXmv was valid and not an actual vulnerability of the architecture but a risk to be controlled by software, an assumption was phrased resembling a mitigation of the respective vulnerability. This led to a total number of eight assumptions which

		MEMORY_OP_INTEGRITY (I.)	CSR_INTEGRITY (II.)	NO_LEAK (III.)
Mode-boundary	SAN_ON_CALL	✓	✓	
	CLR_ON_RET			✓
Memory	NO_PUBLIC_READS	✓	✓	
	NO_PUBLIC_WRITES			✓
Memory privilege	SAN_ON_CLASSIFICATION	✓	✓	
	CLR_ON_DECLASSIFICATION			✓
	SAN_CACHE_ON_CLASSIFICATION	✓	✓	
	CLR_CACHE_ON_DECLASSIFICATION			✓

Table 5.1.: Assumptions Compared to Properties

in total grant the absence of any information flow control violation. These assumptions can be grouped into three categories:

- Mode-boundary crossing related assumptions
- Memory-related assumptions
- Memory privilege related assumptions

Find an overview of all these assumptions in table 5.1. Each row represents one assumption and each column one property. A checkmark denotes that the respective assumption is critical for the respective property, i.e. there exists a counter-example proving the property to be false if the assumption at hand is not assumed. These counter-examples are given in detail in appendix A.1 and will be skipped here.

Note that each of the assumptions is critical for at least one property. This shows that "assumptions $\Leftrightarrow$ properties" as opposed to only "assumptions $\Rightarrow$ properties" since "$\neg$assumptions $\Rightarrow$ $\neg$properties".

Mode-boundary crossing related assumptions As can be seen in table 5.1, there are two properties related to mode-boundary crossing. Mode-boundary crossing related here refers to vulnerabilities that violate a property at execution of the `Ecall` or `Mret` instruction.

The first of these is called `SAN_ON_CALL` and mandates from machine-mode to not use *dangerous* instructions before having cleared the registers from any untrusted

```
1 G (!priv & X priv -> X (
2     (priv -> !(op in { LOAD, STORE, CSRRS, CSRRC }))
3     U (regs_integrity[0] = 0h_FF
4         & regs_integrity[1] = 0h_FF
5         & regs_integrity[2] = 0h_FF
6         & regs_integrity[3] = 0h_FF)
7 ))
```

Snippet 5.2: Assumption SAN_ON_CALL

```
1 G (
2     priv & X !priv -> regs_conf[0] = 0h_00
3         & regs_conf[1] = 0h_00
4         & regs_conf[2] = 0h_00
5         & regs_conf[3] = 0h_00
6 )
```

Snippet 5.3: Assumption CLR_ON_RET

data when entering machine- from user-mode. Dangerous instructions here are LOAD, STORE, CSRRS and CSRRC and directly stem from the instructions used in properties MEMORY_OP_INTEGRITY (I.) and CSR_INTEGRITY (II.). The next assumption to be introduced is called CLR_ON_RET; it states: Whenever machine-mode hands back control to user-mode, the registers must be cleared of any confidential information. The implementation of SAN_ON_CALL and CLR_ON_RET can be found in snippet 5.2 and 5.3 respectively.

Taken together, these two assumptions ensure that machine-mode neither directly hands secrets to nor directly is handed malicious data by user-mode.

Both assumptions SAN_ON_CALL and CLR_ON_RET are similar to each other. Both were introduced under the label of being *mode-boundary crossing related*, which means that both assumptions are about information flow tracking labels when changing privilege mode - more precisely: both are about the labels of register contents. The relation of these two assumptions, however, also characterizes a pattern that can also be recognized for subsequent groups of assumptions. It turned out that for every integrity-related assumption, such as SAN_ON_CALL, there was a similar confidentiality-related assumption such as CLR_ON_RET. This can be seen easily in table 5.1. In alternating rows, two groups of assumptions can be distinguished: those which are relevant to the properties MEMORY_OP_INTEGRITY and CSR_INTEGRITY and those which are relevant to the property NO_LEAK. It is noteworthy that these two groups of assumptions can be characterized by the way they constrain information flow which aligns perfectly

```
1 G (priv & op = LOAD -> (
2     mem_addr < $REGION0_SIZE
3         ? !pmpcfg0.write
4         : !pmpcfg1.write
5 ))
```

Snippet 5.4: Assumption NO_PUBLIC_READS

```
1 G (priv & op = STORE -> (
2     mem_addr < $REGION0_SIZE
3         ? !pmpcfg0.read
4         : !pmpcfg1.read
5 ))
```

Snippet 5.5: Assumption NO_PUBLIC_WRITES

with the intuition of the two types of labels: Confidentiality is about machine-mode not handing out secrets to user-mode, integrity is about user-mode not gaining control over machine-mode. Consequently, whereas confidentiality related assumptions always constrain flow of information going *out of* machine-mode, integrity related assumptions always constrain flow of information going *into* machine-mode.

Memory-related assumptions Two assumptions about memory reads and writes join the ranks of these groups, namely NO_PUBLIC_READS and NO_PUBLIC_WRITES, which are formally given in snippet 5.4 and 5.5 respectively. These two assumptions express exactly what the title says; they require machine-mode to never read from or write to public memory. As such, they directly extend the mode-boundary related assumptions. At first, it was ensured that machine-mode neither hands out secrets nor accepts malicious data directly. These two assumptions now ensure that the same must not happen by a detour into memory.

Memory privilege related assumptions The four assumptions that have been introduced up to this point seem to cover all relevant channels of information transmission: registers and memory. However, these are not enough to ensure that the MINRV8 architecture implements all information flow properties subject to this thesis. Whereas the two memory-related properties covered the group of actions by machine-mode where information directly is given to or taken from user-mode, it is also possible to transmit information indirectly via memory by writing it to or reading it from safe memory regions but then changing the attributes of respective memory regions. The two assumptions SANITIZE_ON_DECLASSIFICATION and CLR_ON_DECLASSIFICATION counter is-

```
1 G (pmpcfg0.write & X !pmpcfg0.write
2     -> memory_integrity[0] = 0h_FF
3      & memory_integrity[1] = 0h_FF)
4 & G (pmpcfg1.write & X !pmpcfg1.write
5     -> memory_integrity[2] = 0h_FF
6      & memory_integrity[3] = 0h_FF)
```

Snippet 5.6: Assumption SAN_ON_CLASSIFICATION

```
1 G (!pmpcfg0.read & X pmpcfg0.read
2     -> memory_conf[0] = 0h_00
3      & memory_conf[1] = 0h_00)
4 & G (!pmpcfg1.read & X pmpcfg1.read
5     -> memory_conf[2] = 0h_00
6      & memory_conf[3] = 0h_00)
```

Snippet 5.7: Assumption CLR_ON_DECLASSIFICATION

sues arising from these vectors. The former assumption demands from machine-mode to ensure that a memory region does not contain malicious information when it is set to be publicly inaccessible. This assumption is formalized in snippet 5.6. The latter demands from machine-mode to always ensure that a memory region does not contain confidential information when its made publicly accessible. This assumption is formalized in snippet 5.7.

Yet, these two assumptions still not suffice to grant the absence of memory privilege related property counter-examples. It turns out that only ensuring memory regions to not contain confidential/malicious words when (de-)classifying respective regions allows problematic data to remain in cache which bypasses SAN_ON_CLASSIFICATION and CLR_ON_DECLASSIFICATION. This brings the need to also assume that the cache is cleared and sanitized by the architecture whenever a memory region is (de-)classified. This is expressed in the assumptions SAN_CACHE_ON_CLASSIFICATION and CLR_CACHE_ON_DECLASSFICATION (cf. snippets 5.8, 5.9).

To summarize all assumptions: At first, SAN_ON_CALL and CLR_ON_RET assumed that no direct flow of sensitive or malicious data is possible between user- and machine-mode (snippets 5.2 and 5.3). Then, these assumptions were generalized to rule out detours into memory by NO_PUBLIC_WRITES and NO_PUBLIC_READS (snippets 5.5 and 5.4). Finally, it was assumed that machine-mode must not freely alter access privileges of memory regions. It must clear or sanitize regions accordingly, depending on the previous access privileges, e.g. machine-mode must ensure not to forget secrets in memory that is made publicly accessible.

```
1 G (pmpcfg0.write & X !pmpcfg0.write
2         & cache.valid & cache.addr < $REGION0_SIZE
3     -> cache.integrity = 0h_FF) &
4 G (pmpcfg1.write & X !pmpcfg1.write
5         & cache.valid & $REGION0_SIZE <= cache.addr
6     -> cache.integrity = 0h_FF)
```

Snippet 5.8: Assumption `SAN_CACHE_ON_CLASSIFICATION`

```
1 G (!pmpcfg0.read & X pmpcfg0.read
2         & cache.valid & cache.addr < $REGION0_SIZE
3     -> cache.conf = 0h_00) &
4 G (!pmpcfg1.read & X pmpcfg1.read
5         & cache.valid & $REGION0_SIZE <= cache.addr
6     -> cache.conf = 0h_00)
```

Snippet 5.9: Assumption `CLR_CACHE_ON_DECLASSIFICATION`

5.2. Canaries

In this section, it will be presented how the model of the MINRV8 architecture was altered in order to implement real-world attacks to the Intel x86 architecture. By default, the MINRV8 architecture is not vulnerable to these attacks. However, making the architecture vulnerable will put the approach of this thesis to formally verify an instruction set architecture to a test.

5.2.1. Cache Poisoning Attack on x86

The cache poisoning attack[3] was discovered and presented by Rafal Wojtczuk and Joanna Rutkowska in their publication *Attacking SMM Memory via Intel CPU Cache Poisoning* [WR09]. "SMM memory" refers to a specific region of memory that is designed to be only accessible by the mode of highest privilege in x86 architectures: System Management Mode. This memory is intended to be written only on start-up and then locked down for later write-accesses. Wojtczuk and Rutkowska, however, managed to find a vulnerability to the x86 architecture which allowed them to effectively write to SMM memory without SMM privileges by marking respective memory region as write-back-cacheable. The attack comprises the following steps and requires administrator privileges on the machine to be attacked:

1. Mark SMM memory as write-back-cacheable using administrator privileges

[3]The name "cache poisoning attack" is not official.

2. Generate write accesses to the memory. Since the region is marked as write-back-cacheable, the writes will not be propagated to the memory controller which would drop these accesses but will be cached.

3. Trigger a System Management Interrupt to transfer control to System Management Mode. Depending on the specific addresses written System Management Mode will now execute code written with administrator privileges in step 2.

Steps 2 and 3 can also be swapped to read from SMM memory. Then, triggering the interrupt will make SMM mode execute parts of SMM memory which will leave some parts of the handler being cached. After SMM returns from handling the interrupt, administrator mode can read these words from cache.

This attack does not fully apply to the MINRV8 architecture since it does not support as fine-grained privilege modes as the x86 architecture. In x86, administrator privileges are needed to set SMM memory cacheable and attack SMM-mode, which is of higher privilege than administrator-mode. To apply this attack to the MINRV8 would bring the need of user-mode being able to set cacheability properties of memory regions. It is not reasonable to assume that anyone would give user-mode this power. However, the basic idea can still be implemented by following changes to the model:

- For all cache transition relations, do not check whether current privilege mode suffices for the memory operation at hand, i.e. user-mode can write to cache of memory regions not set as writable to it.

- For load accesses to memory, do not check whether current privilege mode suffices for the memory read if the address is being cached, i.e. user-mode can read cached memory regions even if they're not set as readable to it.

These changes manage to introduce a vulnerability to the MINRV8 architecture. If assuming all assumptions introduced in section 5.1, nuXmv manages to find a counter-example for all three information flow properties. These, again, are discussed in the appendix (cf. A.2.1).

One distinction from the original cache-poisoning attack to the adaption that has been implemented is that the attacker cannot set the attacked memory region to be cacheable in the first place since user-mode lacks the capabilities to do so. This difference does not play a major role since the attack can take place nonetheless if some memory region happens to be locked down to user-mode and is set to be write-back-cacheable, this still weakens the attack to some degree.

What can be done to mitigate this attack? Obviously, the vulnerability purposefully introduced to the architecture could be dropped. At this point, though, it is not clear how realistic this characteristic of the MINRV8 architecture is since at least in the case of the x86 architecture, the findings of [WR09] suggest hardware does not perform privilege checks when writing to the cache. Another natural conclusion would be to invalidate the cache on mode-transitions. However, it turned out, that even without the cache poisoning attack in place, this mitigation does not suffice to prevent integrity-related property violations (cf. sec. A.1.1).

It not likely, however, that RISC-V is vulnerable to this attack. One the one hand, the specification states that cacheability settings can be modified by machine-mode only [WA17b, p.43]. One the other hand, it also states that PMP attributes (including access privileges) are checked in *parallel* to PMAs (defining cacheability), i.e. regardless of cacheability settings, and always trap at privilege violations [WA17b, pp. 44-45]. This indicates that neither would a privilege mode other than machine-mode be able to set any memory region cacheable nor would it be able to read from cache of memory regions it otherwise does not have access to. It should be part of future work to implement a complete model of the RISC-V specification which would verify whether RISC-V is actually not susceptible to this attack vector.

5.2.2. The SYSRET Vulnerability

The so-called *SYSRET vulnerability* to the Intel x86 architecture is listed as vulnerability *CVE-2012-0217* by the *Common Vulnerabilities and Exposures* database (CVE®) [12a] and allegedly has also been found by Rafal Wojtczuk [12c; 12b; 12e]. To our knowledge, the most concise and freely available explanation of the vulnerability can be found in the blog post *The Intel SYSRET Privilege Escalation* by George Dunlap published on the page of Xen Project, a community for virtualization-related open source projects which in part also have been affected by said vulnerability [Dun12].

The x86 architecture provides multiple ways to switch between privilege modes. In principle, these ways work analogously to how the MINRV8 architecture changes privilege modes: either an interrupt/exception leads to the privilege escalation or a dedicated instruction is executed that changes privilege mode, e.g. `ECALL` in case of the MINRV8 architecture. In general, when changing the privilege mode, the `RIP` register, which is the PC equivalent for x86 architectures, and the stack-pointer are set to the respective version of the targeted privilege-mode. The designers of the architecture found that parts of the normal privilege mode changing routines could be left out in some scenarios to improve performance. One of these routines to be left out was changing the stack-pointer by hardware. These performance improvements are implemented in the `SYSCALL` instruction, which makes a cheap transition to high-privilege mode by, for instance, not changing the stack-pointer by hardware. In consequence, there is the analogous `SYSRET` instruction to return control back to user-mode, which also does not set the stack-pointer. However, if an interrupt handler invoked by the `SYSCALL` instruction intends to use the stack-pointer, it must first set the stack-pointer up accordingly and afterwards set it back to user-mode[4] before executing the `SYSRET` instruction.

When `SYSRET` restores the user-mode's context, it reads the `RIP` value from the `RCX` register, which is used to store the original value of the `RIP` register during interrupt handling. Furthermore, the x86 architecture requires the contents of the `RIP` register to

[4]There are more than two privilege modes provided by the x86 architecture and they also have different names than the privilege modes of the MINRV8 architecture, however, the SYSRET vulnerability only touches two privilege modes which have the same purpose as user- and machine-mode of the MINRV8 architecture. Therefore, in this context, the terms user-mode and machine-mode can also be used to talk about the x86 architecture.

be in a special form. The same requirements are, though, not imposed on the contents of the RCX register. If on executing a SYSRET instruction on an Intel x86 system, the RCX register contains a malformed value in regards to the requirements of the RIP register, a general protection fault is generated by the architecture which aborts the transition to user-mode. It is, however, very likely that at this point the interrupt handler would have already set the stack-pointer back to the user-mode's context leaving the machine-mode with a user-mode controlled stack-pointer. At the time of disclosure of the SYSRET vulnerability, several major operating systems were affected by it, such as Debian [12b], FreeBSD [12c] and Microsoft Windows 7 [12d].

The core of the SYSRET vulnerability is the exception that is being generated during the executing of the SYSRET instruction and handled before privilege is set back to user-mode but after the stack-pointer has been set to user-mode.

It is noteworthy that the SYSRET vulnerability did not affect the x86 specification of Intel but OSes incorrectly using the specification. One could argue that the SYSRET vulnerability was only made possible by poor design decisions by Intel, however, at its core, the SYSRET vulnerability only came to effect because OS engineers did not have this corner case of the specification in mind when writing interrupt handlers. This stresses the benefits of this thesis's approach since not only the specification but also software running on it will be verified. It will turn out that the core of the SYSRET vulnerability will be detected by our approach and that both a mitigation in the architecture and in software can be verified to be successful.

Modification of the MINRV8 Architecture This very core of the SYSRET vulnerability can optionally be added to the MINRV8 architecture. This extension adds three new registers and four instructions to the architecture:

- Two stack-pointer registers, `sp[0]` and `sp[1]`
- A register to select the active stack-pointer `sp_sel`
- An instruction to set the currently active stack-pointer to `rs1 % 2`, `SPSEL rs1` (stack-pointer select)
- An instruction to set the value of the currently active stack-pointer to the address stored in register `rs1`, `SPSET rs1` (stack-pointer set)
- An instruction to push the value in register `rs1` onto the stack, `PUSH rs1`
- An instruction to pop a value from the stack and store it into register `rd`, `POP rd`

As can be guessed from the available instructions that deal with the stack, not much automation by hardware is supported. This lack of automation allows us to implement the SP, although the implementation does not support a PC, etc. Using the stack here effectively equals writing to or loading from a predetermined address. There are no stack-pointers dedicated to specific privilege-modes. Consequently, stack-pointers are not automatically switched by hardware on privilege-mode changes. However, when

pushing to or popping from the stack, the stack-pointer automatically is in- or decremented. Additionally, whenever the stack-pointer is in- or decremented and thereby crosses the boundary of a memory region or the memory as a whole, the architecture also ensures that it wraps around inside its memory region. It might be a threat to some architectures having the stack-pointer move into unsafe memory regions by repeatedly pushing to or popping from it, yet, this aspect is not relevant to the SYSRET vulnerability which is why it was decided to rule these vulnerabilities out by definition.

In the case of the original SYSRET vulnerability, a specially crafted value must have been placed into a special register to trigger an exception during the `SYSRET` instruction. In the case of the MINRV8 architecture, no such special mechanisms are given. Therefore, the vulnerability can only be triggered by an external interrupt but this does not conflict with the core idea of the SYSRET vulnerability.

An **`INIT`** constraint was added to the model ensuring that when the architecture starts in machine-mode, the currently active stack-pointer points to a memory region that is neither read- nor writable to user-mode. This, as usual, ensures that the information flow properties are not violated by some obvious initial condition.

Furthermore, with the extension to the architecture comes an assumption that emulates all the major steps that software vulnerable to the SYSRET attack already follows but which do not suffice to implement a secure system. This assumption is called `SP_BANK` and is depicted in snippet 5.10. It comprises four parts. Firstly, in lines 1-6, it is assumed that machine-mode must not use the `PUSH` or `POP` instructions before the stack points to a safe memory region. Alternatively, machine-mode also might not use these instructions at all until it returns control back to user-mode. Secondly, in lines 7-9, whenever machine-mode hands control back to user-mode the prior instruction must have selected a stack-pointer that is user-mode controllable. This assumption is the most critical to the SYSRET vulnerability. These two assumptions closely resemble the assumptions `SAN_ON_CALL` and `CLR_ON_RET` as shown in section 5.1 and snippets 5.2 and 5.3 respectively.

Thirdly, in lines 10-18, machine-mode is also not allowed to use the `PUSH` or `POP` instructions when it *selects* a stack-pointer pointing to a user-mode-controlled region. This ensures that machine-mode does not deliberately decide to use a bad stack-pointer. And lastly, in lines 19-27, for the same reasons it is also assumed that machine-mode does not set its stack-pointer to point to a memory region controlled by user-mode. These two parts follow the same structure as the first part of the `SP_BANK` assumption but trigger on different conditions.

All these assumptions, however, do not ensure that the architecture models all information flow properties. Similar to the SYSRET vulnerability, there is a stream of instructions that violates each of these properties which will be discussed in the appendices' section A.2.2.

Mitigations To explore the SYSRET vulnerability in more detail, two approaches to mitigate said vulnerability were tested and added to the model. One approach lies upon the burden of ensuring security on the programs running on the architecture and is

```
1  G (!priv & X priv -> X (
2      (priv -> !(op in { PUSH, POP }))
3      U ((op = MRET & MPP = 0b_0) | sp[sp_sel] < $REGION0_SIZE
4          ? !pmpcfg0.write & !pmpcfg0.read
5          : !pmpcfg1.write & !pmpcfg1.read)
6  )) &
7  G (op = MRET -> (sp[sp_sel] < $REGION0_SIZE
8      ? pmpcfg0.read & pmpcfg0.write
9      : pmpcfg1.read & pmpcfg1.write)) &
10 G (priv & op = SPSEL & (sp[rs1 mod 2] < $REGION0_SIZE
11         ? pmpcfg0.write | pmpcfg0.read
12         : pmpcfg1.write | pmpcfg1.read)
13     -> X (
14         (priv -> !(op in { PUSH, POP }))
15         U ((op = MRET & MPP = 0b_0) | sp[sp_sel] <
               ↪$REGION0_SIZE
16             ? !pmpcfg0.write & !pmpcfg0.read
17             : !pmpcfg1.write & !pmpcfg1.read)
18     )) &
19 G (priv & op = SPSET & (mem_addr < $REGION0_SIZE
20         ? pmpcfg0.write | pmpcfg0.read
21         : pmpcfg1.write | pmpcfg1.read)
22     -> X (
23         (priv -> !(op in { PUSH, POP }))
24         U ((op = MRET & MPP = 0b_0) | sp[sp_sel] <
               ↪$REGION0_SIZE
25             ? !pmpcfg0.write & !pmpcfg0.read
26             : !pmpcfg1.write & !pmpcfg1.read)
27     )) &
```

Snippet 5.10: Assumption SP_BANK

```
1 G (priv & op = MRET & MPP = 0b_0 -> MIE = 0b_0) &
```

Snippet 5.11: Assumption SP_SAFETY

expressed in the form of an assumption, the other modifies the architecture and does not rely on programs to implement certain guidelines.

Aforementioned assumption is depicted in snippet 5.11. It states that interrupts must be disabled, whenever machine-mode hands back control to user-mode. This rules out any attack vector where an external interrupt aborts the execution of the MRET instruction leaving machine-mode with a user-mode-controlled stack pointer. Some pending external interrupt can then only be handled after the MRET instruction has been executed. This resembles AMD's version of the x86 architecture, which is not susceptible to the SYSRET vulnerability [Dun12]. In AMD's implementation of the x86 architecture, when setting the RIP with a bad value, the general protection fault is thrown after the privilege mode has been lowered, rendering the SYSRET attack vector ineffective.

Consider Fedora's mitigation to the SYSRET vulnerability [Jon] as a real-world example. The interrupt handlers were modified such that if the user could have changed the RIP value, i.e. might contain an uncanonical value, the IRET instead of the SYSRET instruction is used. This would also be verifiable by our approach: the condition under which IRET is chosen over SYSRET would form the antecedent and op = IRET the consequent of the implication in snippet 5.11. For the MINRV8, however, there is no other choice than disabling interrupts as the source of the *external* interrupt cannot be controlled.

The other mitigation modifies the architecture. The stack-pointer extension of the MINRV8 architecture was altered such that the two stack-pointers are assigned to privilege-modes and switched by hardware. Whenever the architecture changes into machine-mode, sp_sel is set to 1 and whenever it changes into user-mode sp_sel is set to 0. Furthermore, the SPSEL instruction is disabled. Therefore, with this modification to the architecture, the assumptions SP_BANK and SP_SAFETY can be dropped. However, new assumptions and **INIT** constraints must be introduced. With the hardware automatically switching stack-pointers, it must be ensured that the memory regions these stack-pointers point to always are assigned with the respective privileges, i.e. user-mode-stack-pointer is user-mode-read- and -writeable, etc. This is assumed initially and made stable by the assumption given in snippet 5.12. It cannot be taken for granted that machine-mode actually is capable of setting up the memory regions correctly and guaranteeing that user-mode does not interfere with these settings. However, verifying such mechanisms does not serve any purpose at this point and was skipped as a consequence. We therefore make the aforementioned assumptions to show that *given* such mechanisms, the herein proposed hardware mitigation to the SYSRET vulnerability can be verified by our approach.

With both stack-pointer related assumptions, SP_BANK and SP_SAFETY, nuXmv

```
1 G (op != SPSET & (sp[0] < $REGION0_SIZE
2          ? pmpcfg0.write & pmpcfg0.read
3          : pmpcfg1.write & pmpcfg1.read)
4      & (sp[1] < $REGION0_SIZE
5          ? !pmpcfg0.write & !pmpcfg0.read
6          : !pmpcfg1.write & !pmpcfg1.read)) &
```

Snippet 5.12: Additional assumptions for hardware changes

manages to verify that the architecture does not violate integrity information flow properties. The same is true for the modification of the architecture where the stack-pointer is selected automatically by the architecture and the assumptions `SP_BANK` and `SP_SAFETY` are dropped.

5.3. Summary

In this chapter, the results of the prove-refine loop were presented:

- 23 fixes were applied to the model of the MINRV8 architecture,
- four categories of **`INIT`** constraints were set up, and most importantly
- eight assumptions were phrased that were verified to grant the absence of information flow property violations.

These assumptions are visualized in figure 5.1 that again takes inspiration from digital timing diagrams. The eight assumptions introduced in section 5.1 constrain streams of instructions to always resemble the following properties: Whenever machine-mode is called from user-mode by the `ECALL` instruction or there is an external interrupt, machine-mode must not use memory or CSR related instructions until some form of a sanitization procedure (`SAN`) has been completed. This windows is highlighted as the area coloured in yellow. This is implemented by the `SAN_ON_CALL` assumptions and leads to the integrity labels of inputs to these instructions being high after the sanitization procedure. The effect of this assumption is indicated by an arrow: it forces the confidentiality labels of the registers to high. This property is made stable by the `NO_PULIC_READS` and `SAN_(CACHE)_ON_CLASSIFICATION` assumptions which ensure that no malicious data is loaded from memory or cache as long as machine-mode has control. If any of these are not assumed, the inputs to memory or CSR related instructions might become low, e.g. somewhere in the orange-coloured area of the second row. The arrow again illustrates the effect of these two assumptions: they prevent the confidentiality labels from changing to low.

In general, during execution of a trap handler in machine-mode, confidential data will be handled and therefore, the confidentiality labels of the registers become high at some

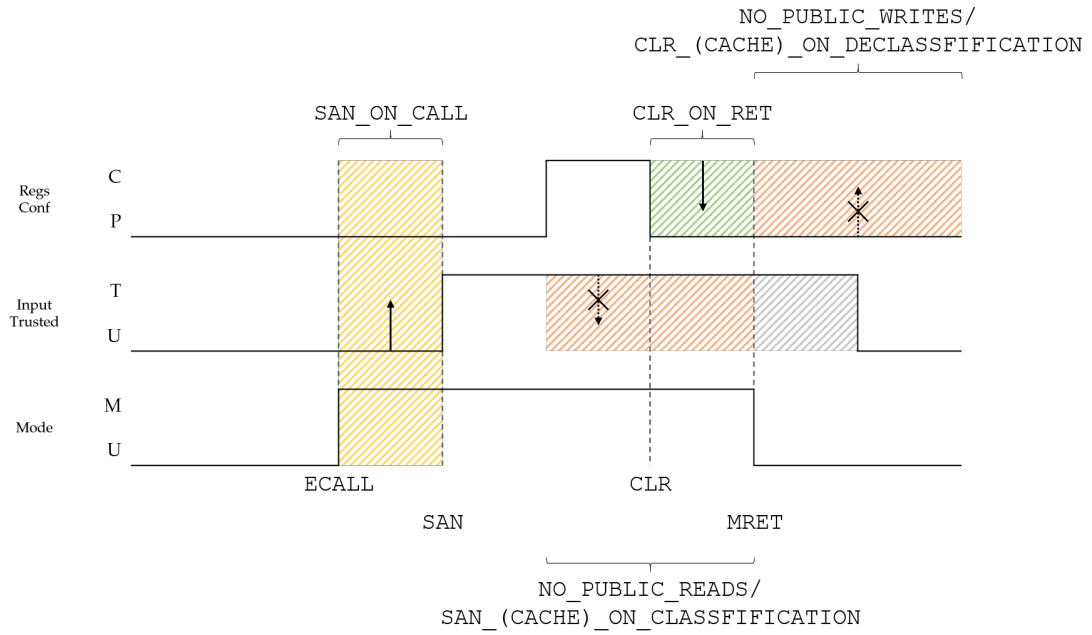


Figure 5.1.: Assumptions visualized

point. The CLR_ON_RET assumption then enforces that before the MRET instruction is executed by machine-mode to hand back control to user-mode, a procedure to clear the contents of the registers must be executed (CLR) after which no other confidential data may be loaded into registers. This window is coloured in green. The assumptions NO_PUBLIC_WRITES and CLR_(CACHE)_ON_DECLASSIFICATION again make this property stable after the registers have been cleared and control has been returned to user-mode, leaving it no room to circumvent the clearing of registers. Again, the orange-coloured area of the first row shows where user-mode might be able to load secrets from memory or cache if these assumptions were not in place.

Two aspects were examined to strengthen these assumptions: two canaries, the deliberate insertion of vulnerabilities into the model, were tested and it was shown that "assumptions $\Leftrightarrow$ properties" as not assuming a single one of these leads to the failure of at least one information flow property. It was discussed that it is possible to find both the cache poisoning [WR09] and the SYSRET vulnerabilities [12a; Dun12] to the x86 architecture without any manual intervention. The information flow properties and the assumptions did not need to be adjusted such that nuXmv would return a trace illustrating the respective vulnerability.

Part III.

Reflection

Chapter 6.

Discussion

In section 4.3, a set of IFC properties based on the work in [Fer+17] has been introduced; the set of these properties will be referred to as P. These properties were model checked, the main result of which was a set of rules R as presented in the previous chapter 5. What is the consequence of this? The formal achievement of this thesis can be summarized as follows:

> It can be guaranteed that programs running on the MINRV8 architecture, e.g. OSes, are *secure* (in the sense of not violating the IFC properties P) if they obey to the rules R.

But what does "secure" mean here besides not violating P? What is the *actual* sense of security that can be guaranteed?

At this point, the class of vulnerabilities covered by these properties cannot be defined strictly. However, the canaries these properties have been tested against in section 5.2, exemplify this class of vulnerabilities. Assessing the results of the approach to formally verify higher-level properties of the MINRV8 architecture shall be subject of this section. We will cover five aspects of our work:

- Scope, i.e. what are the limits of our results,
- trustworthiness, i.e. whether our results are sound,
- whether the model could have been implemented more abstractly,
- if there are alternatives to tracking information on a per-bit basis, and finally
- whether the results meet the self-imposed requirements as set up in chapter 1 and section 2.6.

Whereas the topics of the scope and trustworthiness of our work touch more general aspects of our work, the two questions following up on this reflect on the methodology of this thesis and will elaborate other possible approaches to our subject.

6.1. Scope

6.1.1. Architecture

MINRV8 is meant to be a reasonable abstraction of a real-world RISC-V architecture from a security standpoint. However, up to this point, nothing has been said about the limits of this abstraction. In this section, we will reflect on the limits of the MINRV8 architecture and its implementation.

In summary, the MINRV8 architecture knows three groups of instructions (cf. table 3.2):

- Computational instructions such as MOV, AND, ADD, etc.
- Memory instructions LOAD and STORE
- System instructions ECALL, MRET, CSRRS, and CSRRC

A reader, experienced in the field of microcontrollers or computer-architecture in general, might wonder why our model does not include:

1. Executable memory and a PC
2. Jump or branch instructions

The reason for both of these points lies in the implementation of the model. In the introduction of the MINRV8 architecture in section 3.2, we mentioned that the idea of the architecture was tightly coupled to its implementation in nuXmv. The design of the model had to answer the question: How can a stream of instructions be implemented? nuXmv allows thinking of the following options:

1. Use **FROZENVAR**s to model executable memory - a **FROZENVAR** (frozen variable) is something like a constant in other programming languages but without a fixed value. nuXmv chooses the value on the first simulation step but does not change it afterwards. This is more efficient than using a plain **VAR** and constraining it to not change, e.g. **TRANS** `next(x)= x;`.
2. Use **VAR**s to model executable memory. In practice, this would mean that for memory to hold all instructions executed, it would need to be much larger than the current 4 bytes.
3. Finally, use **IVAR**s to model the stream of instruction. This is the option we decided to go for, as described in section 3.3. Using input variables means that there is no model of executable memory. Instead, the input variables provided to the implementation model a stream of a fully decoded instruction on each transition of the simulated model. As such, the architecture does not need to worry *where* these instructions come from.

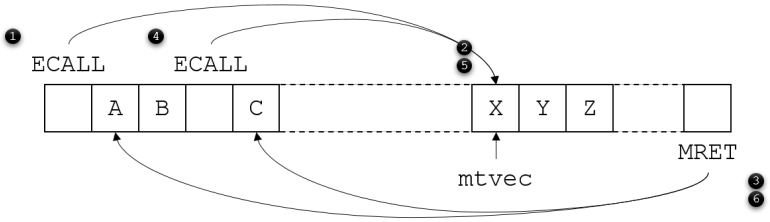


Figure 6.1.: Program flow involving interrupts

The decision towards using input variables as a model of the stream of instructions was made because of two reasons: a) in the prototyping phase of the model, it was quickly found that using input variables led to significant performance improvements, and b) using input variables simplified the implementation notably. Having used purely either frozen variables or variables would have increased the size of the state space to be traversed by nuXmv and would have introduced the necessity to implement instruction decoding increasing the complexity of the transition relation. Furthermore, it would not have allowed for unbounded model checking of the architecture since the size of programs to be taken into consideration would have been bounded by the size of executable memory.

Yet, using input variables also has downsides. First and foremost, using input variables means that it is pointless to model :

- a PC or anything address related like return addresses,
- a dedicated LR,
- or CSRs such as machine trap-vector base-address (mtvec), the CSR holding the trap-vector base address.

Without a model of executable memory there is nothing addresses could point to. Instead, instructions will simply magically appear on each cycle of the processor model. This means that the simulation is inaccurate in this regard which in itself is not a problem because the model is supposed to be an abstraction.

Since we decided to use input variables over a model of the mtvec register and a PC, the question now arises: How accurately is the implementation of the MINRV8 architecture capable of simulating real architectures?

The use of input variables over (frozen) variables introduces a sense of nondeterminism to the model of the MINRV8 architecture when it comes to handling interrupts and exceptions which is illustrated by the following example: Consider some memory layout for the RISC-V architecture as depicted in figure 6.1 where code is executed from left to right starting with an `ECALL` instruction. Executing this instruction triggers a shift in control to machine-mode, which jumps to address `X` pointed to by the mtvec register. Then, the instructions at addresses `X` to `Z` and subsequent ones are executed until finally, a `MRET` instruction is read an executed which triggers a jump to address `A`. Two instructions later, another `ECALL` instruction is executed, etc. We would expect the order of instructions to be executed to look something like:

1. First environment call: ECALL, LOAD, ADD, STORE, ..., MRET
2. Return: LOAD, MOV
3. Second environment call: ECALL, LOAD, ADD, STORE, ..., MRET
4. Second return: SLT, ...

Where LOAD, ADD, and STORE are the instructions located at addresses X to Z and LOAD, MOV, and SLT the instructions located at addresses A, B and C respectively.

However, as input variables cannot be constrained in nuXmv, there is no way to model that the instructions being executed on a trap always are equal, i.e. a valid stream of instructions as a sequence of input variable valuations to the model might be:

1. First environment call: ECALL, LOAD, ADD, STORE, ..., MRET
2. Return: LOAD, MOV
3. Second environment call: ECALL, AND, SRA, ..., MRET
4. Second return: SLT, ...

Such a stream of instructions is not realistic without some program modifying the code located at mtvec and as such should not be considered when checking the IFC properties. One would expect to find the same code after an ECALL instruction being executed since in both cases, an architecture would jump to the respective interrupt handler[1]. However, there also exists a sequence of input variable valuations that exactly matches a real stream of instructions where the instructions executed after an environment call coincide.

This illustrates one example of why the implementation of the MINRV8 *abstracts* from real architectures. In regard to program flow on traps, it can be expected that the model accurately comprises all expected streams of instructions but also includes many more one would naturally consider to be unrealistic.

The second concept, the abstract implementation of the MINRV8 architecture is missing are jump and branch instructions. They are missing for the same reasons as described in the previous paragraphs. Since the implementation does not support executable memory and as such does not support addresses, jumps and branches are pointless instructions. This, however, does not turn out to be a fundamental problem. The implementation might not be able to model programs including jumps and branches but the set of programs considered by nuXmv does include traces that include the same instructions that would have been executed when running a program with jumps and branches but without the jumps and branches. We call such a trace a linearization[2] of a program with jumps and branches. Consider figure 6.2 where in the upper half depicts an ordinary program, being executed from left to right and including jump instructions.

[1] To be precise, one would also need to take vectored interrupts into account (c.f. sec. 3.1.2) but even then, the issue applies.

[2] Linearization is similar to function inlining supported by many compilers.

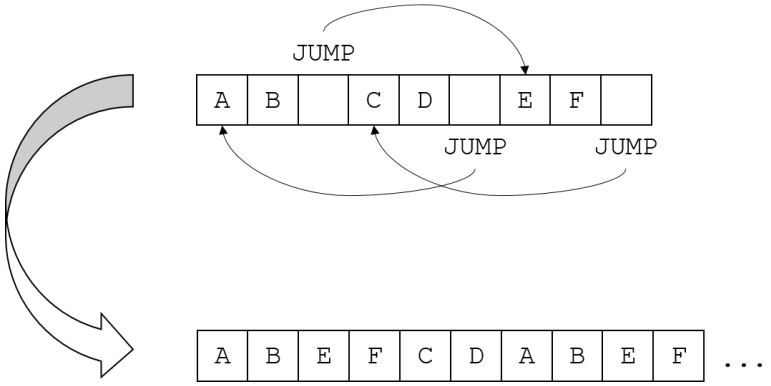


Figure 6.2.: Jump linearization

The respective counterpart to the program in the upper half is depicted in the lower half of figure 6.2. This linearized counterpart is considered by nuXmv and the sequence of states generated by the linearized programs is equal to the sequence of states of the non-linearized program modulo PC values - which is not modelled in nuXmv anyways.

Summary Starting with the premise of this thesis, it was always clear that the MINRV8 architecture is a simplification. This is not a problem since by focusing on the core, security-related instructions allowed to implement a tractable and small model in nuXmv. If one added jump and branch instructions to the MINRV8 architecture, it would be fully comparable to actual RISC-V architectures. From this point of view, it is not a problem to apply the results of the verification process, i.e. the set of assumptions, to the RISC-V architecture.

The implementation of the MINRV8 architecture abstracted from real-world architectures in two main points:

1. nuXmv does not guarantee stable behaviour on ECALL instructions; this introduces a sense of non-determinism where nuXmv considers more programs than could actually be run on the MINRV8 architecture. This is not problematic because overabstracting the space of all programs P still leaves nuXmv considering all programs in P.

2. The implementation does not support jump or branch instructions. This is not problematic because programs including jumps can be linearized; such programs *are* considered by nuXmv.

Therefore, the set of programs nuXmv quantifies over when running proofs includes the set of "real" programs of the MINRV8 architecture, including jumps and branches, by abstracting from it.

6.1.2. Assumptions

Recall the requirements of the assumptions that have been laid out in section 2.6:

```
1 G (!priv & X priv -> X (
2      (priv -> !(op in { PUSH, POP }))
3      U ((op = MRET & MPP = 0b_0) | sp[sp_sel] <
          ↪$REGION0_SIZE
4           ? !pmpcfg0.write & !pmpcfg0.read
5           : !pmpcfg1.write & !pmpcfg1.read)
6 )) &
```

Snippet 6.1: First part of the assumption `SP_BANK`

1. The assumptions must be non-trivial,
2. the assumptions must only constrain machine-mode, and
3. the assumptions must be practical and verifiable.

In this section, we will investigate whether the assumptions that ultimately established the correctness of the information flow properties meet all these requirements.

Regarding point 1, it must be checked that neither are the assumptions contradictory nor do they trivially entail the information flow properties. Recall that we were able to prove that "assumptions $\Leftrightarrow$ properties". In this case, the assumptions could only be contradictory if the properties are contradictory as well, which they are obviously not as there are valid counter-examples to the properties when no assumptions are given. On the other hand, recall that in section 5.2, we were able to make changes to the architecture such that "$\neg$(assumptions $\Rightarrow$ properties)" holds. This indicates that the assumptions actually express something meaningful and do not trivially entail the information flow properties.

Point 2 also is met by the assumptions introduced in section 5.1. Whenever it is stated that certain instructions must not be executed under certain circumstances, these formulas are guarded by the antecedent `priv ->`. Additionally, all other assumptions only constrain state transitions being under control of machine-mode only, such as changes to CSRs or state before machine-mode hands back control to user-mode.

For point 3, meeting point 2 is one step in the right direction. However, it must remain an open question here whether the assumptions introduced in section 5.1 can be applied to real-world systems in a verification scenario. The problems behind this are illustrated by the `SP_BANK` assumption introduced parallel to the SYSRET vulnerability in section 5.2.2. There are two aspects of the first part of this assumption which are noteworthy. The relevant part of the `SP_BANK` assumption is given in snippet 6.1.

Whilst working on this assumption, it was found that the specific implementation of two ideas was critical to the presence of the SYSRET vulnerability, both:

1. replacing the expression `!priv & X priv` in line 1 by `(op = ECALL)| (bool(MIE)& bool(MEIP))` which is the actual condition under which a trap is taken and

2. omitting the expression `(op = MRET & MPP = 0b_0)` in line 3

led to nuXmv not reporting the SYSRET vulnerability. Why is that?

The first part of the `SP_BANK` assumption states: "Whenever there is a switch from user- to machine-mode, latter can only use the `PUSH` or `POP` instructions after having returned to user-mode or having ensured that the stack-pointer is safe". If the antecedent of this assumption would have been replaced by: "Whenever a trap is taken, ...", i.e. by `(op = ECALL)| (bool(MIE)& bool(MEIP))`, the consequent would also have applied to cases where a trap was taken from machine-mode to machine-mode[3]. Such a trap is critical for the SYSRET vulnerability as it interrupts machine-mode from properly returning to user-mode while already having set the stack-pointer back to user-mode. The consequent also applying to traps from machine- to machine-mode in context of the MINRV8 architecture leads to machine-mode ensuring that the proper stack-pointer is set up correctly on *all* traps rendering the SYSRET vulnerability impossible. For example, the OSes being affected by the SYSRET vulnerability clearly did not set the stack-pointer on *all* execution paths of the trap handler for the general protection fault.

The condition `(op = MRET & MPP = 0b_0)` on the right-hand side of the **U**-expression in line 3 equals telling machine-mode: "You can forget about not using the `PUSH` and `POP` instructions after you have *attempted* to return control to user-mode". For the same reasons as stated above, this part is critical. Leaving this expression out would equal machine-mode having state persistent over user-mode-calls and -returns to and from machine-mode, by which machine-mode memorizes whether it has set the stack-pointer to user-mode or to machine-mode. With such a mechanism in place, machine-mode could not be fooled into using the user-mode stack-pointer on any trap handler since it would first check this state.

These two examples stress that the precise phrasing of the assumptions is critical for both how realistic the assumptions are and how well they might be suited to serve as a target for verification of actual code-bases. While it would be sufficient to find a stronger precondition for these assumptions in existing code-bases, it is non-trivial whether such preconditions are easy to find or whether the assumptions might be rephrased such that they still grant the absence of information flow related bugs but are easier to verify in programs. Thoroughly investigating how to phrase properties, however, would be out of the scope of this thesis and, as such, marks future work.

[3] Furthermore, it is noteworthy that the antecedent in the original assumption resembles a more result-driven approach to phrasing properties, i.e. talk about *what* is happening, whereas the alternative discussed here resembles a more architecture-driven approach, i.e. talk about *when* things are happening. The discussion, which of these approaches should be preferred is a topic on its own. In this thesis, we used both approaches but tried to stick to the result-driven approach whenever possible to keep the assumptions concise. In certain situations, however, it was more practical to go for the architecture-driven approach.

A similar issue was discussed by Reid when he elaborates how the properties he verified the ARM architecture against were phrased (cf. [Rei17, p.88:4]).

6.2. Trustworthiness of the Approach

Verification processes must not be arbitrary. The core of verification is to enhance the trust in a given system. For trust to arise, it is mandatory for the process itself to be trustworthy.

But what does *trustworthy* mean? Regehr expressed discussed sense of trustworthiness in his blog post *The Piano Test for Program Verification* as: "[I]t is extremely difficult to argue convincingly that a verification result is correct. By 'correct' I mean not only 'mathematically sound' but also 'the result is faithful to the intent of the verification effort.'" [Reg11] Here, we will skip the aspect of technical soundness of this thesis, i.e. the question of whether the work presented in this thesis is without errors. In consequence, the degree of trustworthiness of this thesis is determined by whether the results of the verification process actually enhance the confidence in the correctness of the MINRV8 architecture, i.e. rather than asking whether what we did was correct, we ask: did we do the *right* thing?

A possible issue with verification is to make circular arguments (an example of this will be given in chapter 7). Verification engineers do not start their work out of nowhere. They usually have a feel for what they attempt to verify and have some vulnerabilities in mind they try to tackle. If the actual work of modelling a prototype and implementing a verification procedure is influenced by this too strongly, one might object: "You were able to find vulnerability X only because you knew that it was there. Your procedure is nothing but an abstracted simulation of known issues."

This leads to another concern: When aiming to verify systems against certain bugs, these are usually generalized to classes of related vulnerabilities. But with such an approach, one can never be sure that there is no other class of vulnerabilities that was missed. In other words: Does the verification procedure have blind spots?

It will now be investigated whether this work is touched by the aforementioned two issues.

First, we will investigate whether a circular argument has been made in this thesis. Recall the steps that have been taken to verify the MINRV8 architecture against the information flow properties:

1. Specify the MINRV8 architecture based on RISC-V
2. Define information flow semantics for the instructions available on MINRV8
3. Use fields not being propagated in the information flow semantics and some intuition to state information flow properties
4. Enter a prove-refine loop until a fixpoint has been reached
5. Test the model by subjecting it to vulnerabilities

We find that steps 2 to 4 are well-founded by both theory and intuition and do obviously not face the risk of making a circular argument. This, however, is less obvious for

steps 1 and 5. In both cases, the selection of RISC-V features and instruction to include or vulnerabilities to test, respectively, can be doubted.

The work on this thesis was indeed influenced by the canaries the model was ultimately tested against, e.g. it was decided to implement a cache to be able to cover the cache poisoning attack, and vice versa: the selection of vulnerabilities to test the model against was based on the features implemented in it. Despite the circular character of these two steps, we find that this is not an issue. Steps 1 and 5 must be understood as a prototype of a verification procedure proposed in this thesis which is given by steps 2 to 4. Having the setup to test these steps being derived via a methodology that is circular to some degree does not make the procedure itself circular. The opposite is the case: We find that the definition of information flow semantic, information flow properties and the prove-refine loop are all well-founded and comprehensible, i.e. follow intuition. On the other hand, we must admit that it is unknown whether there are blind spots to the verification procedure or more precisely: to the information flow properties.

We argue that our results are indeed trustworthy as we managed to show that our properties are not meaningless since they are able to detect both the SYSRET and the cache poisoning vulnerability and well-founded in intuition. But it is unclear a) which class of bugs or vulnerabilities can be detected by this procedure and b) whether this class covers *all* vulnerabilities in the sense of Regehr, i.e. "the result is faithful to the intent of the verification effort." [Reg11] We mark this problem as future work.

6.3. Alternative Levels of Abstraction

One might ask, why it was decided to simulate all flows of information through the model of the MINRV8 architecture and indeed: In scope of this thesis, the actual content of registers is not mandatory, i.e. the actual contents of registers could be removed from the model such that only information flow tracking labels are tracked. Register values do play a role when memory is indexed, but this could likely be abstracted by non-deterministically choosing a memory location to be targeted by loads and stores.

There are two reasons why this decision was made. Firstly, since the work of Ferraiuolo et al. [Fer+17] was adopted where the authors did rely on precise values to track[4] and performance of nuXmv did not mandate to shrink the model, it initially was decided to model values in the MINRV8 architecture.

However, secondly, working with concrete values has two benefits which might become important in future work.

1. If the model ever was to be extended by jumps and branches, deterministic addresses are needed to model static code in executable memory. If not only loads and stores but also jumps and branches would use non-deterministic mechanisms to determine target addresses, modelling executable memory (cf. sec. 6.1.1) and for example interrupt handlers located in this very memory would be pointless.

[4] This is due to the fact that HDL code is annotated. Or in other words: the authors did not have a chance to not track precise values.

2. Concrete values lead to counter-examples being easier to interpret.

With these two reasons at hand and the overall goal of this thesis to implement a prototype for a methodology to verify ISAs, we find that having modelled register values as opposed to abstractly tracking information flow labels only improves on the significance of our results.

6.4. Per-Bit Information Flow Tracking

Another issue is that in chapter 5, it was mentioned that counter-examples never involved flows of information where the bit-wise labels of values in the architecture was critical. In other words: for this thesis, tracking only one bit for confidentiality and one bit for integrity for each register and memory address would suffice. Initially, it was decided to track labels bit-wise because a) we followed the approach of Ferraiuolo et al. [Fer+17] which tracked information flow bit-wise as well, b) again, the performance was sufficient to allow for it, and c) one could not have been sure prima facie non-bit-wise tracking of information would suffice.

Additionally, it can also be seen that computational instructions never played a major role in any counter-example. These results suggest that in future work, if the model ever was to be expanded to a more accurate model of RISC-V and/or to a higher machine-word width and performance problems then are encountered, one could abstract the information flow tracking labels from bit- to word-wise labels. This would decrease the size of the state space to be considered by nuXmv exponentially. Additionally, one could remove instructions that only alter data but do not add new ways for information to flow. E.g., the `ADD` instruction - from a pure standpoint of *where* information flows - adds nothing in comparison to the `MOV` instruction. This option only shrinks the state space by a linear factor - nevertheless, this might matter.

The gist of these findings is: Whereas in the work of Ferraiuolo et al. [Fer+17] tracking of each and every bit for a given system was crucial because the authors *could not choose where bugs occur*, it seems that in our more-high-level approach, for every type of bug, there is a very simple trace illustrating it. nuXmv is able to choose where a bug occurs since it quantifies over programs as opposed to the work of Ferraiuolo et al. where the program of interest is fixed.

6.5. Reflection on Requirements

In the introduction, three goals were set for this thesis: it was claimed that the approach presented as part of this thesis is *viable*, i.e. feasible, realistic and generalizable, *effective*, i.e. it is able to detect issues, and *supplemental*, i.e. it enhances on related work.

All proofs were run on a laptop equipped with an Intel Core i7-7500U CPU and 8GB of RAM. It is safe to say that these are very moderate hardware requirements. Proofs were conducted per property. None of these proofs took more than 60s. While we can give no clear count of working hours spent on this thesis, all of it was developed and

written by a single person in no more than nine months. During this time, this person did not solely work on this thesis. In a non-research scenario, one can assume that development would be more straight forward since it would not include the definition of an architecture.

Due to the exponential blow-up of binary encoded state spaces, there are no guarantees that this approach scales well to realistic and complete architectures. However, there are two reasons which make seem likely that this approach does indeed scale. Firstly, the proofs here ran on a small machine and did not take much time. Secondly, in sections 6.4, it was discussed that bit-wise tracking of information flow might be dropped, i.e. there is room for performance improvements. All of this gives good reasons to believe that our approach is indeed *viable* and could be applied to other architectures with reasonable effort.

To discuss the *relevance* of this work, recall that in section 5.1, it was shown that the implementation of the MINRV8 architecture models the property "assumptions $\Leftrightarrow$ properties". This was put to the test by exposing the implementation to the cache poisoning or SYSRET vulnerability where it was found that it then does violate the properties as presented in section 4.3. In other words, it was shown that the three properties `MEMORY_OP_INTEGRITY`, `CSR_INTEGRITY` and `NO_LEAK` cover at least vulnerabilities related to the cache poisoning or SYSRET vulnerability without any manual intervention besides changing the architecture itself. This marks the main result of this thesis and shows that our approach to verifying ISAs is *effective*. Furthermore, mitigations to these vulnerabilities can be verified using the same model. It must be decided on a case by case basis whether a given violation of the information flow properties marks a vulnerability of the architecture leaving it with the need to be altered or marks a feature the risks of which must be controlled in software.

The discussion of whether our work also is *supplemental* will be postponed to a later stage. In the following chapter 7, related work will be presented. This discussion will be picked up, following up on this.

In addition to these self-imposed goals, it was also decided in section 2.1 to adopt the four design goals that were initially introduced in [Rei17]. These were defined to apply to high-level properties for specifications and list as follows:

> "The central design challenge we face is to create a set of properties that:
>
> - express the major guarantees that programmers depend on;
> - are concise so that architects can easily review and remember the entire set of properties;
> - are stable so that architectural extensions do not invalidate large numbers of rules;
> - and that describe the architecture differently from existing specification to reduce the risk of common-mode failure." [Rei17, pp.88:2-3]

These objectives are met by our work:

- The assumptions give rules for programmers of OSes and compilers that guarantee programs adhering to these do not violate basic information flow properties.
- Both assumptions and properties are small in number and short and can be expressed in intuitive natural language, hence are concise.
- The properties itself are completely architecture-independent and, as such, stable. The assumptions are architecture-dependent and rely on both instructions available and concrete semantics of these instructions. However, we feel subjectively that only basic mechanics are used and predict that the assumptions in a practical environment of an architecture should be relatively stable as well.
- The premise of information flow tracking on instruction level is to describe the architecture from a different point of view. It is therefore obvious that there is very little risk of common-mode failure.

Chapter 7.

Related Work

As related work, we will consider seven papers - one of which we will investigate in detail. This work was founded both on "Who Guards the Guards? Formal Validation of the Arm V8-m Architecture Specification" [Rei17] and "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis" [Fer+17] but both papers have been introduced in chapter 2 extensively, we will not consider them again in this section. However, we will make an extensive comparison to the paper "Model checking to find vulnerabilities in an instruction set architecture" [BS16] by Bradfield and Sturton since this work is closest to ours.

Before this, we will first introduce work related to software and, after that, work related to hardware. All related work covers verification[1] of specifications or via information flow properties. We found that related work can be characterized by four dimensions:

1. It has either software or hardware as a subject.
2. It verifies either implementations or specifications.
3. Verification is performed either live or statically.
4. Information flow properties are either expressed directly or as non-interference properties.

Given these categories, our work has hardware as subject verifying a specification statically via direct information flow properties.

7.1. Software Verification

Two papers that do not consider hardware but software verification come from the Massachusetts Institute of Technology. Firstly, Gordon et al. present their tool DroidSafe for static analysis of information flow properties for android applications in "Information Flow Analysis of Android Applications in DroidSafe" [Gor+15]. Secondly, Suh et al.

[1] It was a deliberate decision to not label all related work as *formal* verification. E.g. [Zha+15; Suh+04] do verify either properties or live program execution but do not use formal methods to accomplish this.

propose an approach to live track information on OS level tacking by traps on security policy violations in "Secure program execution via dynamic information flow tracking" [Suh+04]. It is quite far-fetched to call [Suh+04] a paper about *verification*. Nonetheless, it is methodologically close to ours.

The authors of [Gor+15] consider information flow privacy-sensitive sources of information, e.g. unique device ID, SMS messages and other private files, to riskful sinks of information, e.g. the network, NFC and the local file system. Both sources and sinks are guarded by the android API. The authors track information flow from sensitive sources to sensitives sinks which both are given by specific calls to methods of the android API. DroidSafe manages to detect at least 90% of malicious information flows in selected benchmarks whilst reporting at most 13 false positives.

The approach in [Suh+04] aims to prevent the misuse of programming errors such as buffer overflows. As opposed to verifying the absence of such vulnerabilities, the authors track data from sensitive sources through instructions using per-byte bit labels to observe when such vulnerabilities are being targeted by attacker payloads from such sources. As opposed to [Gor+15], where information was tracked from the "user" to the "outside world", the flow of information here is the other way round: from the "outside world" to the "user". This is due to the fact that [Gor+15] considers the domain of confidentiality and [Suh+04] the domain of integrity when tracking information. Additionally, this in line with our work where information flow properties regarding confidentiality state that confidential data may not flow to user-mode (the "outside world") and malicious data from user-mode may not flow to sensitive places in machine-mode (the "user" in this analogy). Similarly to our work, the authors here also gave information flow semantics for each instruction available. The authors propose a simple security policy involving two rules:

> "*No instruction can be generated from I/O inputs*, and *No I/O inputs and spurious values based on propagated inputs can be used as pointers unless they are bound-checked and added to an authentic pointer*." [Suh+04]

Whereas the first cannot be compared to the properties of this work meaningfully as executable memory is not modelled, the second comes remarkably close to our integrity property I., which expresses that no malicious data might be used as argument to a memory operation or in words of the authors of [Suh+04] states that no "spurious" value can be used as pointer. The exception of bound-checks is implicitly modelled by the `SAN_ON_*` assumptions in chapter 5, which mandate from machine-mode to sanitize register of spurious values before using them. Such a sanitization might be implemented by boundary-checking. Here, machine-mode can only sanitize registers by clearing them, e.g. by using the shift left instruction `SLL`. This is not critical for our results since if an actual value ever was to be relevant for an attack, nuXmv could have chosen to let machine-mode load this exact value via a `LOADI` instruction. If, however, executable memory was part of the model, sanitization methods might need to be adopted.

The approach of [Suh+04] achieves a 100% success rate in a benchmark and the authors claim that their approach has a negligible impact on performance.

7.2. Hardware Verification

Zhang et al. consider the detection of hardware trojans in "VeriTrust: Verification for Hardware Trust" [Zha+15], i.e. parts of an integrated circuit that behave maliciously, can be activated by very few inputs and otherwise do not affect the functionality of the circuit. They analyzed HDL implementations of such integrated circuits searching for (mostly) unused parts in the design. The techniques employed in this work do not come close to what has been used in this thesis. However, one can view the detection of (mostly) unused parts of a design and information flow tracking as two sides of the same coin: whereas here, we strived for finding out where information can flow, the authors in [Zha+15] aim for finding out where information (mostly) does not flow.

Fox formally verifies the ARM architecture by redundancy in *Formal Verification of the ARM6 micro-architecture* [Fox02]. He implements a formal model of the ARM architecture in the theorem prover HOL, using general first-order logic. In this implementation, Fox models two views on the ARM architecture: the ISA itself and the micro-architecture implementing this ISA. Fox uses theorems expressing that there always must be some correspondence between runs of the ISA-world and runs of the micro-architecture-world. If for a given list of instructions or a given list of steps in the micro-architecture, no corresponding run of the opposite world can be found, either the micro-architecture or the ISA must be flawed. This work was picked up by Khakpour, Schwarz, and Dam in "Machine Assisted Proof of ARMv7 Instruction Level Isolation Properties" [KSD13]. The authors here extend the implementation of Fox proving three isolation properties of user- and machine-mode: User-mode processes must only depend on process-accessible-resources, processes must only modify those memory regions they can access by a given MMU configuration and user-mode processes must only execute code by calling machine-mode through a trap. The first of these is a non-interference property. Non-interference properties were introduced in [GM82] and can be best summarized as:

> "[O]ne group of users, using a certain set of commands, is noninterfering with another group of users if what the first group does with those has commands has no effect on what the second group of users can see." [GM82, p.11]

Here, think of these two groups of users as two user-mode processes. This is an indirect way of expressing information flow properties. As opposed to formalize: "information from source X must not flow to sink T", directly, non-interference properties state that "information from any illegitimate source must not flow to any sink", hence must not be able to interfere computation in any way.

In contrast to the work of this thesis, Khakpour, Schwarz, and Dam achieve to distinguish between multiple user-mode processes and, in this way, tackle a more broad problem. In our work, we only consider the scenario of user-mode attacking machine-mode and not user-mode attacking different user-mode processes. It would be an interesting line of research to add an MMU to the model and extend the information flow properties accordingly.

On the other hand, Khakpour, Schwarz, and Dam do not give any sense for what these properties actually achieve when it comes to security. It is not clear what kind of bugs concretely might be tackled by their properties; whilst they are intuitive and certainly should be met by the ARM architecture, it is unclear whether they actually guarantee any desirable characteristics of the architecture. These properties come out of nowhere. Whilst this critique to some extent also applies to our work, we think that the methodology of deriving the information flow properties as presented in section 4.3 gives some sense of completeness to our properties which is missing from the work of Khakpour, Schwarz, and Dam.

Lastly, we consider the work of Nienhuis et al. in "Rigorous engineering for hardware security: formal modelling and proof in the CHERI design and implementation process" [Nie+19]. This work comes from a completely other direction: as opposed to verifying information flow properties in existing architecture, the authors present a new architecture, CHERI, that guards memory accesses by *capabilities*. These capabilities allow processes to perform given actions on a given range of memory, specified per virtual address. The development of CHERI was accompanied by formal methods and proofs to enhance the trust in this system. The CHERI architecture is an attempt to equip an architecture with strong mechanisms of controlling information flow such that the trust in it is very high from the beginning, supported by formal proofs that certain desirable properties hold for this architecture.

7.3. Model Checking Instruction Set Architectures

The work of Bradfield and Sturton in "Model checking to find vulnerabilities in an instruction set architecture" [BS16] comes closest to what has been attempted in this thesis. Bradfield and Sturton also attempt to verify an ISA and also use model checking to do so. However, they do not use higher-level properties or information flow tracking to achieve this. Coincidentally, the authors focus on the SYSRET vulnerability to the x86 architecture. Their goal is to prove the absence of SYSRET-like vulnerabilities which is expressed as the property: never can the processor be in privileged mode when at the same time the stack pointer, base pointer or instruction pointer are controlled by user-mode.

In their methodology, the authors first identify a group of vulnerable instructions which might lead to such a state by being interrupted. Eight instructions were identified as such - all of them being ECALL/MRET equivalents. The authors then model these instructions alongside with all state touched by these instructions. By sparsely constraining the initial state of the model they finally prove: for all initial states, there is no direct successor state where a general protection fault has been thrown and either the stack, base or instruction pointer is controlled by user-mode. The authors claim that by the constraints chosen for the initial states of the model, these states actually embody an over-approximation of all reachable states of the x86 architecture. With this methodology, the authors manage to correctly find that Intel's version of the x86 architecture is vulnerable to the SYSRET vulnerability while AMD's version is not.

On the one hand, we find it inadequate to compare this work on all aspects to ours since the work of Bradfield and Sturton does not have the same scope as ours - their paper only comprises five pages. On the other hand, we find that the work in [BS16] lacks in two, key aspects. First and foremost, the problem of a circular argument in verification, as discussed in section 6.2 fully applies to this work. The authors do not use the model checker to verify the x86 architecture but to simulate the SYSRET vulnerability: they restrict their model to instructions of which they know that these are relevant to the SYSRET vulnerability or at least highly related to instructions affected by the SYSRET vulnerability with no clear reasons for why others were excluded or not included and they specifically encoded the gist of the SYSRET vulnerability in the property to be verified[2].

This poses a big problem for a generalization of this work. Whilst it might be an adequate showcase for instances of problems where model checking might be helpful, their work only shows how to check that an architecture is *specifically* not susceptible to the SYSRET vulnerability.

Secondly, the property proposed by Bradfield and Sturton is not in line with Intel's x86 specification (cf. [Dun12]). The purpose of the SYSENTER and SYSEXIT instructions *is* to execute privileged code with the stack pointer being controlled by user-mode to reduce the overhead of the mode transition. This means that taking a general protection fault while handling a call to machine-mode via SYSENTER, leaving the general protection fault handler with a stack pointer controlled by user-mode is intended behaviour. This might be badly designed behaviour - but meets the specification. In practice, operating systems opted to ensure that no general protection fault could occur when executing the SYSRET instruction. However, to verify Intel's x86 architecture to not be susceptible to the SYSRET vulnerability asks from it to violate the specification.

In contrast, regarding the first issue, in our work, we not only proposed more general properties that were not rooted in specific vulnerabilities but in a structured model of information flow and also would apply to other architectures with few adjustments required, we also took clear and justifiable steps when minimizing the RISC-V architecture such that it would be feasible to verify it in scope of this thesis. Regarding the second issue of verifying the x86 architecture to not meet the specification, in our work we took a more moderate approach of verifying mitigations to the SYSRET vulnerability. Instead of proving the absence of the SYSRET vulnerability, we opted to prove that the mitigations of either disabling interrupts when executing the SYSRET instruction or changing the stack-pointer by hardware are effective (cf. sec. 5.2.2).

7.4. Summary

Seven papers were considered as related work. Find an overview of all papers in table 7.1. Each column classifies related work in regard to the categories introduced at the

[2]With the property's encoding, the authors miss the original intent of the work: to verify the absence of SYSRET-*like* vulnerabilities. By only considering general protection fault, a potentially huge class of other vulnerabilities is ruled out by assumption.

	Subject		Level		Context		Information Flow	
	SW	HW	Impl.	Spec.	Live	Static	Direct	Non-I.
This thesis		✓		✓		✓	✓	
[Gor+15]	✓		✓			✓	✓	
[Suh+04]	✓		✓		✓		✓	
[Zha+15]		✓	✓			✓	(✓)	
[Fox02]		✓		✓		✓	N/A	
[KSD13]		✓	✓			✓		✓
[Nie+19]		✓		✓		✓	N/A	
[BS16]		✓		✓		✓	N/A	

Table 7.1.: An overview of related work

beginning of this section: the subject of verification is software/hardware and an implementation/specification, verification is performed live/statically, and information flow properties are expressed directly/as non-interference properties (if at all).

The work coming closest to our methodology of verification via information flow control was [Gor+15; Suh+04] and considered software. The work coming closest to our goal of verifying specifications against higher-level properties was [Fox02; KSD13; BS16]. Additionally, the work of [Suh+04; KSD13] motivates to augment our model by an implementation of an MMU to prove isolation properties on user-process level.

With this overview at hand, we can come back to the discussion postponed in section 6.5 whether our work is *supplemental* to related work. In this discussion, we will consider three references: the work of Reid [Rei17], of Fox [Fox02] and of Bradfield and Sturton [BS16]. Only the work of these authors is in direct competition to ours as only they consider the verification of ISAs in a static context against higher-level properties. The work of Nienhuis et al. [Nie+19] was excluded from this discussion since this work did not consider the verification but the development of an ISA.

The most critical addition to the work of [Rei17] and [Fox02] is that we consider non-redundant verification of specifications. [Rei17] verified the machine-readable specification of the Arm architecture against higher-level properties that stem from the natural language specification of the Arm architecture. He restricted his work to the verification of the most critical parts of the natural language specification when it comes to security, thereby striving to give programmers a *concise* set of rules. Nevertheless, this aspect relies on redundancies between the machine-readable and natural language specification of the Arm architecture. The same applies to [Fox02]. He verified that the micro-architecture, i.e. the actual implementation of the ISA, agrees with the ISA itself. In comparison to both [Rei17] and [Fox02], our work verifies the MINRV8 architecture against truly independent properties, i.e. information flow properties. These do not stem from the specification itself but from the information flow semantics and common sense. Yet, by our methodology we still managed to achieve a sense of completeness, i.e. we can argue that no properties were missed as every part of the architecture was either covered by the information flow semantics or the properties. Additionally, this

makes our approach more architecture-independent then those of [Rei17] and [Fox02]. Our work could not be carried over to other architectures without adaptions. However, many parts of our work should be relatively stable - especially the properties themselves.

In addition to this, we enhance on the work of [Rei17] by being able to constrain unbounded number of transitions as opposed to a single transition only (cf. 2.1) and on the work of [Fox02] by not relying on an interactive theorem prover but nuXmv where the proofs do not require manual intervention.

Finally, we extensively compared our work to [BS16], the only case where model checking was applied to architectural specifications. Unfortunately, this approach was flawed, did not use model checking to its full potential and implemented a very limited model. In [BS16], the properties again only took one transition in the state spaces into consideration whereas in this work we proved properties touching traces of possibly infinite length. Also, their model only comprised instructions related to mode-transitioning, whereas we implemented a more or less complete set of basic RISC instructions. And finally, the approach of [BS16] was not only limited to the SYSRET vulnerability, it also attempted to verify that x86 architecture violates its own specification as the authors did not take into account that verifying specifications is a trade-off between changing the architecture and defining requirements that must be met by software running on it. All in all, this shows that our work is *supplemental*.

Chapter 8.

Conclusion

Find an overview of this thesis as a whole in figure 8.1. As a first step, the MINRV8 architecture inspired by the RISC-V architecture was defined and implemented in nuXmv in section 3.2. In chapter 4, information flow semantics and three information flow properties forming an information flow control in spirit of the work of Ferraiuolo et al. [Fer+17] were developed. The information flow semantics was used to augment the model of the MINRV8 by information flow tracking. In chapter 5, eight assumptions that, when implemented software running in machine-mode, guarantee the absence of vulnerabilities covered by the aforementioned information flow properties were presented. Our model, the properties and the assumptions were evaluated by showing that taken together, they manage to detect both the cache poisoning [WR09] and the SYSRET vulnerability [Dun12]. Finally, in chapter 6, the limitations and the scope of our work were discussed and it was reflected whether our methodology is trustworthy.

Before a final summary of what has been achieved in this thesis will be given, we first recap all directions future work might take that were hinted towards throughout this thesis:

Executable memory First and foremost, the model could be enhanced by a model of executable memory to resemble modern architectures more closely. This was discussed extensively in section 6.1.1.

MMU In light of [KSD13] a model of an MMU could be implemented to make use of information flow tracking on user-mode level. This was discussed in chapter 7.

Machine-generated model The model of the architecture that was implemented by hand in this thesis could be generated from existing machine-readable architectural specifications. For example, there are machine-readable versions of the ARM architecture; both [Rei17; Fox02] either used or developed machine-readable specifications that might be used for this endeavour. For RISC-V, there also is a formal specification available [SiF]. Not implementing the model by hand would a) possibly enhance the trust in the model itself, depending on the trust in the source and the translation procedure, and b) make the approach even more viable and stable towards architectural changes.

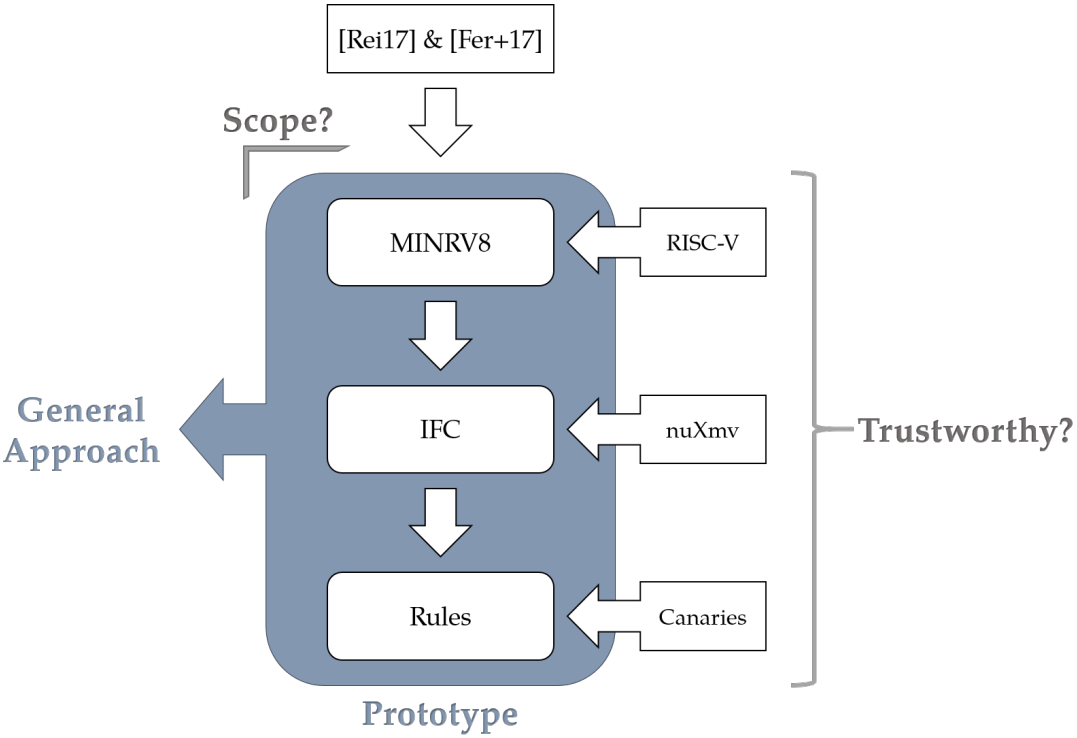


Figure 8.1.: Thesis Overview

Complete model An obvious improvement of this work would be to enhance it by more instructions or a bigger word-size. This would bring the model closer to real-world architectures but might also introduce performance issues.

Verification of assumptions Finally, it could be investigated whether the assumptions introduced in the context of the verification process serve as practical and sensible target of verification for compilers and/or OSes. It could be investigated whether code generated from compilers or OSes itself adheres to these assumptions. Applying the assumptions to the next level of abstraction in modern computing might lead to simpler and mainstream usage of formal verification in programming. Since these properties are stable for a given architecture, there could be off the shelf tools verifying programs for high-level correctness removing the need to tailor verification efforts per system to be verified.

Finally, the main contribution of this thesis is a new approach to verifying ISAs by higher-level information flow properties using a model checker. This approach promises to take into account unbounded numbers of transitions, i.e. instructions, is non-redundant and architecture-independent and results in either architectural changes to combat vulnerabilities or rules for, e.g. OS or compiler engineers that are: *practical* and *verifiable themselves*, *concise*, and *stable*.

This approach was applied to the MINRV8 architecture to implement a prototype. During this, three properties were found that lead to eight rules being stated. These properties were broad enough to cover the cache poisoning and SYSRET vulnerabilities which both applied to the x86 architecture.

Bibliography

[12a] *CVE-2012-0217*. 2012. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0217 (visited on 09/16/2019).

[12b] *Debian Security Advisory DSA-2508-1 kfreebsd-8 – privilege escalation*. debian. July 2012. URL: https://www.debian.org/security/2012/dsa-2508 (visited on 09/17/2019).

[12c] *FreeBSD-SA-12:04.sysret Security Advisory*. The FreeBSD Project. July 2012. URL: https://www.freebsd.org/security/advisories/FreeBSD-SA-12:04.sysret.asc (visited on 09/17/2019).

[12d] *Microsoft Security Bulletin MS12-042 - Important. Vulnerabilities in Windows Kernel Could Allow Elevation of Privilege (2711167)*. Microsoft Corporation. July 2012. URL: https://docs.microsoft.com/en-us/security-updates/SecurityBulletins/2012/ms12-042 (visited on 09/17/2019).

[12e] *SYSRET 64-bit operating system privilege escalation vulnerability on Intel CPU hardware. Vulnerability Note VU#649219*. CERT Coordination Center. July 2012. URL: https://www.kb.cert.org/vuls/id/649219/ (visited on 09/17/2019).

[14] *MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS64 Architecture*. Version Revision 6.01. MIPS. 2014.

[16] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*. Order Number: 325384-060US. Intel. Sept. 2016.

[17] *pyexpander documentation*. Helmholtz-Zentrum Berlin GmbH. 2017. URL: http://pyexpander.sourceforge.net/ (visited on 12/18/2019).

[19] *Arm Architecture Reference Manual. Armv8, for Armv8-A architecture profile*. Version E.a. Arm Limited. 2019.

[BB94] D. L. Beatty and R. E. Bryant. "Formally Verifying a Microprocessor Using a Simulation Methodology". In: *31st Design Automation Conference*. June 1994, pp. 596–602. DOI: 10.1145/196244.196575.

[BD94] Jerry R. Burch and David L. Dill. "Automatic verification of pipelined microprocessor control". In: *Computer Aided Verification*. Ed. by David L. Dill. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 68–80. ISBN: 978-3-540-48469-1.

[Ber+98] Sergey Berezin et al. "Combining Symbolic Model Checking with Uninterpreted Functions for Out-of-Order Processor Verification". In: *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*. FMCAD '98. London, UK: Springer-Verlag, 1998, pp. 369–386. ISBN: 3-540-65191-8. URL: http://dl.acm.org/citation.cfm?id=646185.683063.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. Cambridge, Massachusetts: The MIT Press, 2008. ISBN: 978-0-262-02649-9.

[Boz+16] Marco Bozzano et al. *nuXmv User Manual*. Version 1.1.1. Fondazione Bruno Kessler. 2016.

[Bra11] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87. ISBN: 978-3-642-18275-4.

[BS16] Chris Bradfield and Cynthia Sturton. "Model checking to find vulnerabilities in an instruction set architecture". In: *HOST*. IEEE Computer Society, 2016, pp. 109–113.

[Bur+92] J. R. Burch et al. "Symbolic model checking: 10^{20} States and beyond". In: *Information and Computation* 98.2 (1992), pp. 142–170. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90017-A. URL: http://www.sciencedirect.com/science/article/pii/089054019290017A.

[Cav+14] Roberto Cavada et al. "The nuXmv Symbolic Model Checker". In: *CAV*. 2014, pp. 334–342.

[Cla+00] Edmund Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-45047-4.

[CS12] K. Claessen and N. Sörensson. "A liveness checking algorithm that counts". In: *2012 Formal Methods in Computer-Aided Design (FMCAD)*. Oct. 2012, pp. 52–59.

[Dat00] Chris J. Date. *An Introduction to Database Systems*. 7th ed. Reading, Masschusetts: Addison-Wesley, 2000. ISBN: 0-201-38590-2.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. "On the Verification of Memory Management Mechanisms". In: *Correct Hardware Design and Verification Methods*. Ed. by Dominique Borrione and Wolfgang Paul. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 301–316. ISBN: 978-3-540-32030-2.

[Dij72] Edsger W. Dijkstra. "Structured Programming". In: ed. by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. London, UK, UK: Academic Press Ltd., 1972. Chap. Chapter I: Notes on Structured Programming, pp. 1–82. ISBN: 0-12-200550-3. URL: http://dl.acm.org/citation.cfm?id=1243380.1243381.

[Dun12] George Dunlap. *The Intel SYSRET Privilege Escalation*. Xen Project. June 2012. URL: https://xenproject.org/2012/06/13/the-intel-sysret-privilege-escalation/ (visited on 09/17/2019).

[Fer+17] Andrew Ferraiuolo et al. "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 555–568. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037739. URL: http://doi.acm.org/10.1145/3037697.3037739.

[FMW05] Harry Foster, Erich Marschner, and Yaron Wolfsthal. "IEEE 1850 PSL: The next generation". In: (2005).

[Fox02] A.C.J. Fox. *Formal Verification of the ARM6 micro-architecture*. Tech. rep. 548. University of Cambridge, 2002.

[GM82] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*. Apr. 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014.

[Gor+15] Michael I. Gordon et al. "Information Flow Analysis of Android Applications in DroidSafe". In: *NDSS*. The Internet Society, 2015.

[HBM11] Krystof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. "μZ - An Efficient Engine for Fixed Points with Constraints". In: *CAV*. 2011.

[Hol04] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Boston: Addison-Wesley, 2004. ISBN: 0-321-22862-6.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Amsterdam: Elsevier, Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[Irf+16] A. Irfan et al. "Verilog2SMV: A tool for word-level verification". In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2016, pp. 1156–1159.

[Jon] Dave Jones. *Fedora alert FEDORA-2006-423 (kernel)*. URL: https://lwn.net/Articles/180820/ (visited on 02/04/2020).

[Kle+09] 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. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[Koc+19] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P’19)*. 2019.

[Kom+13] Anvesh Komuravelli et al. “Automatic Abstraction in SMT-Based Unbounded Software Model Checking”. In: *CoRR* abs/1306.1945 (2013). arXiv: 1306.1945. URL: http://arxiv.org/abs/1306.1945.

[KRS14] C. Krieg, M. Rathmair, and F. Schupfer. “A Process for the Detection of Design-Level Hardware Trojans Using Verification Methods”. In: *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*. Aug. 2014, pp. 729–734. DOI: 10.1109/HPCC.2014.112.

[KSD13] Narges Khakpour, Oliver Schwarz, and Mads Dam. “Machine Assisted Proof of ARMv7 Instruction Level Isolation Properties”. In: *CPP*. Vol. 8307. Lecture Notes in Computer Science. Springer, 2013, pp. 276–291.

[Lip+18] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[MB08] Leonardo de Moura and Nikolaj Bjørner. “Z3: an efficient SMT solver”. In: vol. 4963. Apr. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[Muk+16] Rajdeep Mukherjee et al. “Unbounded Safety Verification for Hardware Using Software Analyzers”. In: *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. DATE ’16. Dresden, Germany: EDA Consortium, 2016, pp. 1152–1155. ISBN: 978-3-9815370-6-2. URL: http://dl.acm.org/citation.cfm?id=2971808.2972077.

[Nie+19] Kyndylan Nienhuis et al. “Rigorous engineering for hardware security: formal modelling and proof in the CHERI design and implementation process”. In: 2019.

[PH13] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 9780124077263.

[Reg11] John Regehr. *The Piano Test for Program Verification*. Jan. 27, 2011. URL: https://blog.regehr.org/archives/364 (visited on 11/27/2019).

[Rei+16] Alastair Reid et al. “End-to-End Verification of Processors with ISA-Formal”. In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 42–58. ISBN: 978-3-319-41540-6.

[Rei17] Alastair Reid. "Who Guards the Guards? Formal Validation of the Arm V8-m Architecture Specification". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), 88:1–88:24. ISSN: 2475-1421. DOI: 10.1145/3133912. URL: http://doi.acm.org/10.1145/3133912.

[SiF] *Formal Specification of RISC-V ISA in Kami*. SiFive. URL: https://github.com/sifive/RiscvSpecFormal (visited on 01/08/2020).

[Suh+04] G. Edward Suh et al. "Secure program execution via dynamic information flow tracking". In: *ASPLOS*. ACM, 2004, pp. 85–96.

[TLM19] C. Trippel, D. Lustig, and M. Martonosi. "Security Verification via Automatic Hardware-Aware Exploit Synthesis: The CheckMate Approach". In: *IEEE Micro* 39.3 (2019), pp. 84–93. DOI: 10.1109/MM.2019.2910010.

[VS12] Alexander Vaynberg and Zhong Shao. "Compositional Verification of a Baby Virtual Memory Manager". In: *Certified Programs and Proofs*. Ed. by Chris Hawblitzel and Dale Miller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 143–159. ISBN: 978-3-642-35308-6.

[WA17a] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Version 2.2. RISC-V Foundation. May 2017.

[WA17b] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Version 1.10. RISC-V Foundation. May 2017.

[Wal15] Ben Walshe. *A Brief History of Arm: Part 1*. Arm Ltd. Apr. 2015. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-brief-history-of-arm-part-1 (visited on 02/14/2020).

[Wat+11] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Tech. rep. UCB/EECS-2011-62. EECS Department, University of California at Berkeley, May 2011.

[Wol] Clifford Wolf. *RISC-V Formal Verification Framework*. URL: https://github.com/SymbioticEDA/riscv-formal (visited on 11/06/2019).

[WR09] Rafal Wojtczuk and Joanna Rutkowska. *Attacking SMM Memory via Intel CPU Cache Poisoning*. Online. Mar. 2009. URL: https://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf.

[Zha+15] J. Zhang et al. "VeriTrust: Verification for Hardware Trust". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.7 (July 2015), pp. 1148–1161. DOI: 10.1109/TCAD.2015.2422836.

Appendices

Appendix A.

Counter-Examples to Information Flow Properties

A.1. Assumption Related Counter-Examples

In this section, the counter-examples associated with the assumptions which resulted from the prove-refine loop, as presented in section 5.1, will be presented. The structure of section 5.1 will be adopted, i.e. counter-examples will be presented in order of the three groups of mode-boundary crossing related, memory-related and memory privilege related assumptions.

Mode-boundary crossing related assumptions There are two mode-boundary crossing related assumptions: `SAN_ON_CALL` and `CLR_ON_RET`, demanding from machine-mode to sanitize registers when being called from user-mode and clearing registers when returning control to user-mode.

Without assuming `SAN_ON_CALL` nuXmv manages to find two counter-examples. One for property `MEMORY_OP_INTEGRITY` (I.) and one for `CSR_INTEGRITY` (II.). The traces for both counter-examples are depicted in tables A.1a and A.1b respectively. Table A.1c illustrates a counter-example for the `NO_LEAK` (III.) property when not assuming `CLR_ON_RET`.

Example 13. The format of these tables illustrating counter-example traces will be used frequently to illustrate counter-examples. To introduce it, table A.1a will be explained in detail now. The first row of the table gives the processor's initial state. Each subsequent row of the table represents an instruction being executed and shows the values that have changed in result of this instruction. Whenever a cell is empty, the value represented by it did not change. Furthermore, the table is separated into three main columns: The first column shows the instructions being executed. In the case of table A.1a, three instructions were executed: `ECALL`, `LOAD` and `STORE`.

The second group of columns hold the information flow labels for the registers and memory on a per-word basis. During the verification process, it was found that no counter-example required information flow labels of a per-bit granularity. Therefore, for the sake of simplicity, the information flow labels of machine-words in memory or

registers were combined to a single label[1]. The two inner rows in the cells of the second major column denote the labels of the registers (r) and memory (m) respectively whereas the inner columns mark the index of the labels, e.g. in table A.1a all registers and the cells of memory region zero are labelled with PU, i.e. public and untrusted, whereas the cells of memory region one are labelled with PT, i.e. public and trusted, in the initial state.

The third major column shows the status of the processor. This column is separated into five inner columns as well. The first (labelled with p) shows the privilege mode of execution where U stands for user-mode and M stands for machine-mode. The other four columns show the current memory settings per region. Inner columns starting with 0 give the settings for first memory region and starting with 1 for the second memory region respectively. Inner columns ending with c hold the cacheability settings. Here, UN stands for uncacheable, WB for write-back-cacheable, WT for write-through-cacheable and WP for write-protected-cacheable (cf. section 3.2). Inner columns ending with p hold the memory privilege settings for the respective region. Memory privilege settings are indicated by a string of the form:

⟨*privelege-string*⟩ ::= (‘-’ | ‘L’) (‘-’ | ‘R’) (‘-’ | ‘W’)

The first letter in the privilege string denotes whether the region is locked (L) or not (-), the second whether user-mode can read the region (R) or not (-) and the third whether user-mode can write to the region (W) or not (-). In table A.1a, none of the regions is locked or cacheable and user-mode can write to region zero only and has not further permissions.

Traces generated by nuXmv are infinitely long but finitely representable by including a loop. In this setting, counter-examples generated by nuXmv always violated the property at hand after having executed the second last instruction, whereas the last instruction introduced the loop and led to a fixpoint in the state-space. For the sake of completeness, the last instruction of counter-example traces will be included nonetheless.

Note that the traces for property MEMORY_OP_INTEGRITY in table A.1a and for property CSR_INTEGRITY in table A.1b are very similar. In both traces, the processor starts in user-mode with some untrusted data in the registers. User-mode then calls machine-mode which decides to use the contents of the registers without sanitization to either write a CSR or to memory. The violation of property NO_LEAK (III.) depicted in table A.1c when not assuming CLR_ON_RET is analogously simple. Since machine-mode directly hands over confidential data to user-mode by returning control to user-mode without clearing its registers, user-mode trivially gains access to secrets. It is obvious that such flows of information are no fault of the architecture but must be fought in programming, i.e. in compilers or OS programming. As such, we express sufficient techniques to combat vulnerabilities of this kind in the aforementioned assumptions.

[1] The labels have been combined by using the $\sqcup$ operator, i.e. for a label w, the combined label is $\bigsqcup_i w_i$.

	0	1	2	3	p	0c	0p	1c	1p
	r PU m PU	PU PU	PU PT	PU PT	U	UN	--W	UN	---
ECALL	r m				M				
LOAD 3, 3	r m			PT					
STORE 1, 3	r m								

(a) MEMORY_OP_INTEGRITY (I.)

	0	1	2	3	p	0c	0p	1c	1p
	r PU m PU	PU PU	PU PT	PU PT	U	UN	--W	UN	---
CSRRC 1, 0, 1	r m				M				
CSRRS 0, 0, 3	r CT m								
	r m								

(b) CSR_INTEGRITY (II.)

	0	1	2	3	p	0c	0p	1c	1p
	r CT m PU	PT PU	PT PU	PT PU	M	UN	--W	UN	--W
MRET	r m				U				
LOAD 1, 3	r m								

(c) NO_LEAK (III.)

Table A.1.: Counter-examples for SAN_ON_CALL and CLR_ON_RET

Memory-related assumptions As for mode-boundary crossing related assumptions, there are two memory-related assumptions NO_PUBLIC_READS and NO_PUBLIC_WRITES that prohibit machine-mode to either read from or write to memory that is also accessible to user-mode.

The three counter-examples involved in these two assumptions are straight-forward as well. If machine-mode is allowed to load arbitrary words from public memory nuXmv gives a counter-example to the properties MEMORY_OP_INTEGRITY (I.) and CSR_INTEGRITY (II.) where some or all memory is untrusted in the initial state, machine-mode in the next step decides to load some word from an untrusted source in memory and then uses that as target address of a STORE instruction or to write a CSR with. These counter-examples are illustrated in table A.2a and A.2b respectively. On the other hand, if machine-mode is allowed to store arbitrary words into public memory, a simple counter-example to the NO_LEAK property (III.) is that machine-mode simply stores a secret into public memory which is then loaded by user-mode as illustrated by the trace in table A.2c.

Additionally, in this counter-example, the consequences of assuming CLR_ON_RET can be seen. Machine-mode tries to hide secrets from user-mode by writing a public value into register zero before returning, which is mandated by the aforementioned assumption. Yet, as shown in the last paragraph, this assumption taken alone does not ensure the absence of NO_LEAK related counter-examples.

Memory privilege related assumptions Lastly, the four memory privilege related assumptions will be considered now. In general, these express that machine-mode must ensure that no malicious data lies and no secrets remain in memory or cache memory or cache of a region to be locked down to user-mode. The assumptions SAN_ON_CLASSIFICATION and CLR_ON_DECLASSIFICATION ensure this for the actual memory and SAN_CACHE_ON_CLASSIFICATION and CLR_CACHE_ON_DECLASSIFICATION for the cache.

The counter-examples when not assuming SAN_ON_CLASSIFICATION for properties MEMORY_OP_INTEGRITY and CSR_INTEGRITY (cf. tables A.3b, A.3a) - as usual - work analogously and illustrate exactly what has been given as intuition for this assumption in the introduction of this paragraph. In each counter-example, at least one memory region initially is writable for user-mode and contains untrusted data. Machine-mode changes the privilege settings of that memory region, making it non-writable to user-mode and then loads and uses a machine-word which is untrusted violating the respective integrity-related property.

The counter-example given when not assuming CLR_ON_DECLASSIFICATION is given in table A.3c. The architecture starts with some confidential data in the memory of the first memory region, which is not publicly readable. In the first step, machine-mode, however, sets just that memory to be publicly readable. Machine-mode does clear its registers before giving control back to user-mode, yet this is irrelevant since user-mode can just load the contents from the now accessible memory region one which violates the NO_LEAK property (cf. III.).

		0	1	2	3	p	0c	0p	1c	1p
	r	CT	PT	PT	PT	M	UN	--W	UN	---
	m	CU	PU	PT	PT					
LOAD 3, 3	r				CU					
	m									
STORE 3, 3	r									
	m									
SRA 0, 0, 1	r									
	m									

(a) MEMORY_OP_INTEGRITY (I.)

		0	1	2	3	p	0c	0p	1c	1p
	r	CT	CT	CT	CT	M	WT	LRW	UN	L-W
	m	PU	PU	PU	PU					
LOAD 3, 3	r				PU					
	m									
CSRRC 0, 1, 3	r	PT								
	m									
SUB 0, 1, 2	r	CT								
	m									

(b) CSR_INTEGRITY (II.)

		0	1	2	3	p	0c	0p	1c	1p
	r	CT	PT	PT	PT	M	UN	LRW	UN	L--
	m	PU	PU	PT	PT					
STORE 0, 0	r									
	m		CT							
CSRRC 0, 3, 2	r	PT								
	m									
MRET	r					U				
	m									
LOAD 0, 1	r	CT								
	m									
LOAD 0, 3	r									
	m									

(c) NO_LEAK (III.)

Table A.2.: Counter-examples for NO_PUBLIC_WRITES and NO_PUBLIC_READS

		0	1	2	3	p	0c	0p	1c	1p
	r	PT	PT	PT	CT	M	UN	-RW	UN	--W
	m	PU	PU	PU	PU					
CSRRC 3, 3, 0	r				PT					
	m									---
LOAD 2, 0	r			PU						
	m									
LOAD 0, 2	r	PU								
	m									
STORE 2, 0	r									
	m									

(a) MEMORY_OP_INTEGRITY (I.)

		0	1	2	3	p	0c	0p	1c	1p
	r	CT	CT	PT	PT	M	UN	-R-	UN	--W
	m	PT	PT	CU	PU					
CSRRC 2, 1, 1	r									
	m									---
LOAD 3, 0	r				CU					
	m									
CSRRS 3, 1, 3	r									
	m									
MRET	r									
	m									

(b) CSR_INTEGRITY (II.)

		0	1	2	3	p	0c	0p	1c	1p
	r	PT	CT	PT	PT	M	UN	---	UN	L--
	m	CT	PT	PT	PT					
CSRRS 1, 1, 0	r		PT							
	m							-R-		
MRET	r									
	m					U				
LOAD 2, 1	r				CT					
	m									
MRET	r									
	m									

(c) NO_LEAK (III.)

Table A.3.: Counter-examples for CLR_ON_DECLASSIFICATION and SAN_ON_CLASSIFICATION

Coming to the assumptions that constrain interaction with cache, the counter-examples given by nuXmv are not completely straight forward. As a start, consider the counter-examples generated when not assuming SAN_CACHE_ON_DECLASSFICATION for the MEMORY_OP_INTEGRITY property in table A.4a. In the first instruction, machine-mode sets memory region one to be inaccessible by user-mode, which does not violate assumption SAN_ON_CLASSIFICATION since all memory provides trusted values. In the next instruction, machine-mode decides to load a word from memory, which happens to be untrusted. This must come from cache since no other memory cell matches this label. This is only possible because memory region one is set to be write-back-cacheable where writes not necessarily must be reflected in memory. This word is then used as an address for a LOAD instruction violating property I.. The counter-example for property CSR_INTEGRITY (II.) depicted in table A.4b follows the same pattern and therefore will not be handled separately in this context.

The trace generated to prove the NO_LEAK property (III.) wrong when not assuming CLR_CACHE_ON_DECLASSIFICATION flips the integrity-related counter-examples. It is given in table A.4c. Here, memory region zero is set to be write-back-cacheable. This time though, machine-mode declassifies it in the first instruction as opposed to the previous counter-examples where the respective region was being classified. Machine-mode then gives back control to user-mode, which performs a load. Again, a mysterious confidential word appears in register three which must have come from the cache of memory region one since only this region is accessible to user-mode.

A.1.1. Cache-Invalidation Mitigation

Consider the counter-example to the cache-relate assumptions CLR_CACHE_ON_DECLASSIFICATION depicted in table A.4c. It can be seen that it resulted in a trace where memory privileges are lowered but sensitive data remains in cache. One, therefore, might think that this vulnerability can be mitigated by changing the architecture such that the cache is invalidated on mode-transitions[2]. However, the critical part of the cache-related assumptions does not lie in the privilege-transition itself.

Consider the trace depicted in table A.5. This trace was generated *with* the adoption to the architecture that the cache is invalidated on any mode-transition and is straightforward: machine-mode first gives user-mode read access to memory region one which is fine since only public values are stored in that region. Then, machine-mode attempts to load some word from memory which, however, leads to a confidential value appearing in memory region one. How can that be? Note that memory region one is set to be write-back cacheable. There must have been some confidential value in the cache which was written back to memory because the cache was about to be filled with other data by the Load instruction. When machine-mode now returns control back to user-

[2]The traces A.4a and A.4b both do not involve mode-transitions. However, it could be safely assumed that initially, the confidentiality or integrity labels in the cache line are low or high respectively. This would rule out the initial state where malicious data is in cache and bring the need for user-mode to prep the cache with such data.

		0	1	2	3	p	0c	0p	1c	1p
	r	CT	PT	CT	PT	M	UN	---	WB	--W
	m	PT	PT	PT	PT					
CSRRC 0, 1, 0	r	PT								---
	m									
LOAD 3, 2	r				PU					
	m									
LOAD 2, 3	r			PT						
	m				PU					
STORE 3, 2	r									
	m									

(a) MEMORY_OP_INTEGRITY (I.)

		0	1	2	3	p	0c	0p	1c	1p
	r	PT	CT	PT	PT	M	UN	---	WB	--W
	m	PT	PT	PT	PT					
CSRRC 1, 1, 3	r		PT							---
	m									
LOAD 3, 0	r				CU					
	m									
CSRRS 2, 3, 3	r							-RW		
	m									
CSRRC 0, 1, 3	r									
	m									

(b) CSR_INTEGRITY (II.)

		0	1	2	3	p	0c	0p	1c	1p
	r	PT	CT	PT	PT	M	WB	---	UN	--W
	m	PT	PT	PU	PU					
CSRRS 1, 1, 3	r		PT					-RW		
	m									
MRET	r					U				
	m									
LOAD 1, 0	r		CT							
	m									
	r									
	m									

(c) NO_LEAK (III.)

Table A.4.: Counter-examples for CLR_CACHE_ON_DECLASSIFICATION and SAN_CACHE_ON_CLASSIFICATION

		0	1	2	3	p	0c	0p	1c	1p
	r	PT	PT	PT	PT	M	WB	---	WT	---
	m	PT	PT	PT	PT					
CSRRS 1, 1, 3	r							-R-		L-W
	m									
LOAD 1, 0	r									
	m		CT							
MRET	r					U				
	m									
LOAD 2, 3	r			CT						
	m									
LOAD 2, 2	r									
	m									

Table A.5.: NO_LEAK (III.) counter-example when invalidating the cache on privilege transitions

mode, having the cache invalidated still allows user-mode to load the confidential value from the memory region accessible to it. Therefore, this architectural mitigation does not suffice and cache needs to be taken into account when changing access privileges of memory regions.

A.2. Canary Related Counter-Examples

In section 5.2, two vulnerabilities were purposefully added to the model of the MINRV8 architecture in order to test both the information flow properties and the assumptions that resulted from the prove-refine loop as presented in chapter 5. In this section, the counter-examples that were found by nuXmv when the respective modifications were put into effect will be presented.

A.2.1. Cache Poisoning Attack on x86

When the cache poisoning attack discussed in section 5.2.1 was injected into the model, all information flow properties were violated. As usual, the traces for MEMORY_OP_INTEGRITY (I.) and CSR_INTEGRITY (II.) as depicted in tables A.6a and A.6b are highly similar. The counter-example to property MEMORY_OP_INTEGRITY (I.) will be introduced in detail now. Initially, user-mode executes a STORE instruction which is not reflected in memory. Then, it reads from a CSR to put a trusted value into register 0. Interestingly, user-mode thereby aids machine-mode in sanitizing the registers since now, all registers contain trusted values. In the next step, a pending external interrupt is taken, which leads to the transition to machine-mode. Machine-mode performs a load instruction targeting the same address as the user's store instruction which

leads to an untrusted value being stored in register two. This value is then used to perform a load/store operation.

The load in the fourth step resulting in an untrusted value means, the initial store must have been targeting the only region set cacheable which is region one in table A.6a and region zero in table A.6b respectively. Here, it can be seen that dropping memory privilege controls on cache-writes and -reads leads to states inconsistent with the MINRV8 specification. In table A.6a, memory region one is locked and as neither the read- nor the write-bit is set, both user- and machine-mode should not be able to read from or write to memory region one. Yet, the cache-access succeeds.

In the case of the original cache-poisoning attack, the authors wrote to executable memory directly influencing what would be executed by System Management Mode. Since the MINRV8 model does not support a model of executable memory, it is not possible to implement this exact vulnerability here. However, these counter-examples show that it is still possible to implement the very core of the cache poisoning attack into the MINRV8 model, which leads to a similar vulnerability of the architecture.

This carries over to the counter-example for property `NO_LEAK` (III.) which is depicted in table A.6c. This counter-example more closely matches the alternative version of the cache-poisoning attack used for reading SMM memory. nuXmv sets the initial state up such that the cache holds some confidential data for one of the memory regions, which both are set to be write-back-cacheable and non-readable to user-mode. Yet still, user-mode can gain access to the confidential word in cache as no access controls are performed when reading from cache.

A.2.2. The SYSRET Vulnerability

For the SYSRET vulnerability as presented in section 5.2.2, only the integrity-related properties were considered as only these are relevant to the SYSRET vulnerability. The counter-examples illustrating this are depicted in table A.7. A new feature is used to visualize the two stack-pointers. It has already been mentioned that the architecture supports two stack-pointers being indexed by `0` and `1`. These indices are visualized by a superscript in the cells that contain the memory labels. For example, in table A.7a the stack-pointer `sp[1]` points to address `1` and `sp[0]` to address `3`. The stack-pointer selected as currently active is highlighted by being underlined.

As usual, both counter-examples for the `MEMORY_OP_INTEGRITY` (I.) and `CSR_INTEGRTIY` (II.) properties are very similar. Therefore, only the first of these counter-examples will be introduced in detail. In the counter-example for the `MEMORY_OP_INTEGRITY` property, the architecture starts in machine-mode with stack-pointer one being active and set to memory region zero. which only contains trusted values[3]. In the first two instructions, machine-mode selects a user-mode controlled stack-pointer as the active one and attempts to give back control to user-mode. However, an external interrupt must have been set pending and taken exactly when machine-mode attempted to execute the `MRET` instruction since no privilege-mode transition occurs.

[3]Furthermore, memory region zero is locked and not set to be write- or readable which makes it inaccessible to machine-mode but this is not relevant to the counter-example.

		0	1	2	3	p	0c	0p	1c	1p
	r	PU	PT	PT	PT	U	UN	LR-	WB	L--
	m	PT	PT	PT	PT					
STORE 0, 0	r									
	m									
CSRRS 0, 3, 0	r	PT								
	m									
CSRRS 0, 1, 2	r					M				
	m									
LOAD 2, 2	r			PU						
	m									
STORE 2, 2	r									
	m									
LOAD 1, 3	r									
	m									

(a) MEMORY_OP_INTEGRITY (I.)

		0	1	2	3	p	0c	0p	1c	1p
	r	PT	PT	PU	PT	U	WT	L--	UN	---
	m	PT	PT	PT	PT					
STORE 1, 2	r									
	m									
ECALL	r					M				
	m									
SLT 2, 0, 3	r			PT						
	m									
LOAD 1, 1	r		PU							
	m									
CSRRS 0, 3, 1	r									
	m									
CSRRS 0, 1, 1	r									
	m									

(b) CSR_INTEGRITY (II.)

		0	1	2	3	p	0c	0p	1c	1p
	r	PU	PU	PU	PU	U	WB	--W	WB	--W
	m	PU	PU	PU	PU					
LOAD 1, 0	r		CU							
	m									
LOADI 0 IMM	r									
	m									

(c) NO_LEAK (III.)

Table A.6.: Counter-examples for the Cache Vulnerability

The memory region that is pointed towards by the now active stack-pointer contains untrusted values only. This alone marks the core of the SYSRET vulnerability but it only becomes a problem because machine-mode then decides to pop a value from the stack, which consequently is untrusted and then uses that value.

		0	1	2	3	p	0c	0p	1c	1p
	r m	CT PU^0	PT PU	PT PT	PT $PT^{\underline{1}}$	M	UN	-RW	WT	L--
SPSEL 0	r m	$\underline{0}$			1					
MRET	r m									
POP 2	r m		$\underline{0}$	PU						
STORE 2, 2	r m									
LOAD 2, 2	r m									

(a) MEMORY_OP_INTEGRITY (I.)

		0	1	2	3	p	0c	0p	1c	1p
	r m	CT $PT^{\underline{1}}$	CT PT	CT PU	PT PU^0	M	UN	---	UN	LRW
SPSEL 0	r m	1			$\underline{0}$					
MRET	r m									
POP 3	r m			$\underline{0}$	PU					
CSRRC 2, 1, 3	r m			PT						
	r m									

(b) CSR_INTEGRITY (II.)

Table A.7.: Counter-examples for the SYSRET vulnerability

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann. Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.

Ort: Datum: Unterschrift: