Ε Μ Π
Σ Η Μ Μ Υ
Τ Τ Π Υ

Formally Verified Tag-Based Enforcement of Control Flow Integrity

Δ Ε

Ν Γ

Ε : Ν Π
Α . Κ Ε.Μ.Π.

Ε Τ Λ
Α , Σ 2014

Ε Μ Π
Σ Η Μ Μ Υ
Τ Τ Π Υ
Ε Τ Λ

Formally Verified Tag-Based Enforcement of Control Flow Integrity

Δ Ε

Ν Γ

Ε : Ν Π
Α . Κ Ε.Μ.Π.

Ε 11 Σ , 2014.

....	....	....
Ν Π	Κ Σ	Ι Σ
Α . Κ Ε.Μ.Π.	Α . Κ Ε.Μ.Π.	Α . Κ Ε.Μ.Π.

Α , Σ 2014

....

Ν Γ

Δ Η Μ

Μ Υ Ε.Μ.Π.

The purpose of this diploma thesis is to present a novel, hardware-assisted, formally verified implementation of low-level security policies, such as Control-Flow Integrity and Call Stack Protection. Contrary to existing (**TODO:** write an abstract)

Keywords

control-flow, security, verification, tagged architectures

Ε

Θ Κ Σ , . Σ
, .

Ε Ν Π , ,
Π . Η
.

Ε .

Τ , Μ . Σ !

Contents

List of Figures

List of Listings

List of theorems and definitions

Chapter 1

Introduction

1.1 Motivation

Computer hardware and software continuously grow in size and complexity and as a result ensuring the absence of exploitable behaviors is becoming increasingly difficult. In the era when *(NG: where?)*computer systems are used extensively to carry important information (e.g. credit card numbers, national security documents), it has been widely accepted that security of these systems is a priority. Researchers have identified a number of potential vulnerabilities which arise from the violation of known but in-practice unenforceable safety and security policies.

So far, computer security has been delegated mostly to software, while the hardware is being almost completely controlled by the software. High-level languages are becoming more widely used, due to features such as strong type systems with type inference and automatic memory management, making programming less error prone and reducing the number of exploitable bugs. Furthermore, in order to strengthen the security of computing systems a variety of low-level mitigation techniques [10, 24, 16]

(**TODO:** reference some? stack canaries, ASLR, $W \oplus X$) *(NG: done)* have been proposed, however these are mostly ad-hoc solutions designed to prevent specific known attacks, rather than enforcing a security policy by preventing a well-defined class of attacks, thus making it hard to reason about their effectiveness. In fact most of these mitigation techniques can be circumvented by attackers [27], which has lead to a continuous "chase" between attackers and security researchers.

One common attack technique is to exploit some low-level vulnerability such as a buffer overflow to redirect the control flow to attacker injected code. This attack can be stopped by a simple protection scheme known as $W \oplus X$, which enforces that a memory page is either executable or writable but not both. Unfortunately, clever attack techniques can bypass $W \oplus X$. In particular, attackers have been using code-reuse attacks (e.g. return/jump - oriented programming) that allows them to chain together existing pieces of code to achieve malicious behavior without directly introducing new code. Abadi *et al.* [1] introduced a property called Control Flow Integrity (CFI), which provides effective protection against control-flow hijacking attacks. CFI enforces that any execution of a program will respect a statically computed control flow graph (CFG). *(CH: missing references throughout)*

The main contribution of this thesis is the formalization and verification of a dynamic monitor for CFI, based on a generic hardware-software security mechanism. We provide a precise attacker model and prove in Coq that the monitor enforces a variant of the CFI property proposed by Abadi *et al.* [2]. To obtain this result we prove refinement between a concrete machine running a monitor satisfying our Coq specification and an abstract machine having CFI by construction. We conclude the proof using a novel generic result stating that under certain assumptions CFI is preserved by refinement. *(CH: Is there anything missing here?)*

1.2 Thesis Outline

Map 1. Intro 2a. Safety and Security Policies 2b. Micropolicies 3. CFI description 4. CFI formalization 5. Conclusions and Future work 6. Related work

Chapter 2 of this thesis briefly describes the motivation for effective and efficient security policies, the desired properties a robust security policy must satisfy and puts into context the framework we utilize in order to formalize the Control-Flow Integrity policy and reason about the effectiveness of the enforcement mechanism we used.

Chapter 3 discusses the current state of research on enforcing and formalizing Control-Flow Integrity and clarifies the design choices of our approach regarding enforcement of *CFI*.

Chapter 4 explains how we used the framework of chapter 2 in order to formally reason about the security properties of the *CFI* policy and our approach to enforcing it.

Chapter 5.. conclusions, future work? Appendix with code and/or step relations etc.?

1.3 What needs to be done

1. Re-read and polish the whole thing
2. Optimize figure placement - once comments are removed and content is settled
3. mention types on tags and DATA tag on registers
4. More things on concrete preservation?
5. A summary on the conclusions?
6. Call-stack protection in future work
7. Have a look at latest related work
8. think about appendix if we need one
9. think about diagrams, do we want more (e.g., stopping for concrete machine)
10. Unified numbering? Theorems,figures,table all having one counter. Last time I tried to do this it failed. And it seems strange!
11. Take care of first parts (abstract, thanksgiving, etc.)

Chapter 2

Micro-policies: Verified, Hardware-Assisted Monitors

Currently the hardware provides very limited security mechanisms (**TODO:** name some; 4 protection rings, page-level memory protection via virtual memory), leaving most of the work to the software. This requires that the software performs various sanity-checks during an execution and that it carefully maintains various safety and security invariants, a tedious and error-prone task that results in high runtime performance overheads.

Many potentially effective mitigation techniques are not deployed because of the performance overhead they incur. Another requirement for deployment of a protection mechanism is the compatibility with existing executables and the degree of intervention required by a human. Usually even making slight changes to a code and redistributing has high cost and the protection mechanism is likely to see very low adoption.

The lack of efficient and effective generic ways to enforce security policies, forces programmers to protect their own code, a task which is not trivial even for the small and simply programs. As a result most, if not all, software carries weaknesses which can be exploited by an attacker. "Safe" languages, automate some of the checks required and eases the work of the programmer, for example by implementing array bounds checking or by disallowing pointer-arithmetic. However these solutions only reduce the chance of introducing exploitable bugs in a program and do not enforce stricter, more effective policies such as Control Flow Integrity or complete Memory Safety (spatial/temporal protection for heap and stack). In addition, we still need effective and efficient protection mechanisms for a plethora of software written in unsafe languages such as C.

2.1 Micro-Policies

(CH: Can the main idea of the CFI micro-policy be introduced here already? See grant proposal.)

A wide range of security policies can be enforced by associating metadata to the data being processed (e.g., this is an instruction, this is from the network, this is private, etc.), propagating the metadata as instructions are executed and using a set of rules on the metadata to check whether a policy is violated and how the tags should be propagated.

Abstractly, these rules form a partial function from a set of input tags to a set of output tags

$$(opcode, tag_{pc}, tag_{instr}, tag_{arg1}, tag_{arg2}, tag_{arg3}) \rightarrow (tag_{pc'}, tag_{result})$$

informally read as, "if the next instruction to be executed is opcode, the current tag of the program counter is pc_{tag}, the current tag on the instruction location is tag_{instr} and the tags on the operands of the instruction are tag_{arg1}, tag_{arg2} and tag_{arg3} then if execution of the instruction is allowed the tag on the program counter should be set to $tag_{pc'}$ and any new data created by the instruction should be tagged tag_{result}".

More specific, a micro-policy is made up of the following elements:

1. a set of *metadata tags* used to tag the contents of the memory and all the registers as well as the pc.
2. a *transfer function* that implements the checks on the tags and the tag propagation as described above.
3. a tagging scheme for the initial state of the machine.
4. for some micro-policies, a set of *monitor services* (i.e., privileged code) that can be invoked by user code.

Furthermore, as we will see in section 2.4, a software-hardware mechanism that allows for implementation of micro-policies without sacrificing flexibility (in terms of the policies that can be enforced) has already been designed. Simulations and benchmarks show that the runtime overhead is very low compared to dedicated software solutions thus making it a realistic and appealing way to deploy a wide range of security policies in future computing systems.

2.2 Example: Non-Writable Code & Non-Executable Data

(CH: Make it clearer that this is informal and you will return to the formalization later on) (NG: I think I did now)

(CH: Symbolic vs concrete rules ... should introduce symbolic rules first, although this is a quite trivial example; ALT: write these as pseudo-Coq functions?)

(NG: Saying something explicitly about concrete rules here seems hard because it comes out of the blue without the PUMP. Writing a function would omit introduction of syntax for rules and I am not sure if this is what we want.)

In order to demonstrate the mechanism explained above we sketch a simply micro-policy that enforces *NWC* and *NXD*, omitting the formalization to which we will return in chapter 4.

Consider the set of tags $\mathcal{T} = \{Data, Code\}$. If we initially tag all executable regions in memory as *Code* and all non-executable as *Data* then we can enforce *NWC* and *NXD* by two rules of the form

$$\frac{}{Store : \{CI{=}Code, MR{=}Data\} \rightarrow \{PC'{=}-, RES{=}Data\}} \quad (\textsc{Store/Data})$$

$$\frac{opcode \notin \{Store\}}{opcode : \{CI{=}Code\} \rightarrow \{PC'{=}-, RES{=}-\}} \quad (\textsc{Rest})$$

Figure 2.1: Rules enforcing *NWC* and *NXD*

The dashes in the result vector, represent *don't care* values, meaning we will not use their values for anything, so any tag (usually a default tag set by the policy designer) can be used. Furthermore, we are omitting from the input vector the fields that are unused by the transfer function. For this simply policy, the transfer function only uses the tag on the current instruction (CI) and in the case of a Store instruction the tag on the memory (MR), i.e., the tag on the memory location we are trying to write. Additionally, if no rule applies, execution of the instruction is disallowed. Informally the above rules can be read as "If the tag on the current instruction is *Code*, then if the opcode of the instruction is Store, execution should be allowed only if the tag of the third operand is *Data*. In that case the tag on the new data on the memory should remain *Data*. If the opcode of the instruction is not Store, then it's allowed and the result tags are indifferent for the enforced policy."

2.3 Generic Verification Framework for Micro-Policies

Unsurprisingly, designing a security policy, reasoning about its effectiveness against potential attackers and encoding it as a micro-policy can become a complex task. Azevedo *et al.* [12] built a generic framework for defining and verifying micro-policies on top of a machine modeling a tagged RISC processor (referred to as concrete machine), formalized this framework in Coq and used it to define and formally verify micro-policies for dynamic sealing, control-flow integrity, memory safety, compartmentalization and protecting the enforcement mechanism (referred to as policy monitor) itself.

The framework offers a high-level machine, called the symbolic machine, that abstracts away from various - insignificant to security policies - implementation details, that can be used as an interface to the concrete machine, simplifying the work of the micro-policy designer and allowing him to define and reason about a micro-policy at a higher-level using structured mathematical objects than low-level machine words.

In order to implement the micro-policy at the concrete machine level, one needs to additionally provide machine code that implements the transfer function, an encoding of tags to words and machine code for any monitor services that the micro-policy may use. The relation between the symbolic and the concrete machine is formally defined as a two-way refinement (forward and backward). This is a generic refinement proof, parameterized by the encoding of the symbolic tags to words and a proof of correctness of the monitor code for a micro-policy. The designer of a micro-policy can use this two-way refinement simply by providing these two parameters.

2.3.1 Correctness of micro-policies

For each micro-policy an abstract machine, which serves as a specification to the invariants the policy designer wants to enforce, is defined. The abstract machine is "correct" by construction, meaning that it's designed to respect those invariants. Using the symbolic machine as an intermediate step to simplify the proofs, by proving a refinement between the symbolic and the abstract machine and by utilizing the generic refinement between the symbolic and the concrete machine, we can prove a refinement between the abstract and the concrete machine, thus showing that every step of the concrete machine adheres to the specification that is set by the abstract machine.

All the machines introduced in the original paper by Azevedo *et al.* [12], as well as this thesis, have a similar structure. In particular, they share a common RISC-based instruction set (with a few - uninteresting for the scope of this thesis - exceptions) and they have a fixed number of general-purpose registers, along with a pc register. Of course the abstract machine defined by the policy designer can differ in various ways, but more similarities with the symbolic machine implies easier proofs of correctness.

(CH: Introduce the (names of the) various machines and how they relate to each-other. Nice diagram?)

(**TODO:** Write instruction set? maybe not)

2.3.2 Symbolic Machine

As mentioned above, the symbolic machine enables us to abstract away from various low-level details of the concrete machine. We can express and reason about policies in terms of mathematical objects written in Gallina rather than machine code and the corresponding proofs for the concrete machine comes for free under some assumptions. In essence, the symbolic machine is parameterized by a micro-policy as it was defined in 2.1, with the addition of an internal state that can be used by monitor services.

The states of the symbolic machine consists of the memory, the registers, the *pc* register and the internal state. The memory and register contents, as well as the *pc*, are all tagged with a symbolic tag t. We name their contents *symbolic atoms* referred to with the notation $w@t$, where w is the value (word) and t is the tag.

At each step, a record named *mvector* is formed. It consists of the current opcode, the tag on the *pc*, the tag on the current instruction and optionally up to three tags depending on the opcode of the instruction. The *mvector* is passed to the transfer function which decides whether the step violated the policy enforced by the *transfer* function and in this case halts the machine, or if no violation occurred returns a tag for the new *pc* and a tag for any results the instruction execution produced.

In fig. 2.2 we give, in form of inference rules, the stepping relation for the Symbolic machine, demonstrating how the transfer function and the tag propagation works at each step.

Notice for example, that when a store instruction executed, the tag on the memory location to be overwritten is fetched, allowing the *transfer* function to know what kind of data we are trying to overwrite. Returning to the example micro-policy in 2.2 we can define the transfer function that is used by the symbolic machine as a Coq function.

```
Definition transfer ivec : option ovec :=
  match ivec with
  | mkIVec   Store  _ Code  [_ ; _ ; Data] ⇒
    Some (mkOVec _ Data)
  | mkIVec   Store  _  _  _  ⇒ None
  | mkIVec   _         _   Code _   ⇒
    Some (mkOVec _ _)
  | mkIVec _ _ _ _ ⇒ None
  end.
```

Listing 2.1: Transfer function for NWC and NXD in pseudo-code

(NG: heavily abusing notation with _ on result vectors)

2.4 A Programmable Unit for Metadata Processing

(CH: Could consider moving this one level up (turn it into chapter))

2.4.1 Hardware Architecture

The Programmable Unit for Metadata Processing (PUMP) architecture [14] allows us to efficiently implement a wide range of micro-policies [13], using software to define the rules enforcing the policy, while the hardware provides efficiency by undertaking the propagation of the tags and a rules cache.

On the hardware level, the PUMP is an extension to a conventional RISC architecture. Every word of data in the machine - whether in memory or a register, is extended with a word-sized metadata tag. These tags are not interpreted by hardware, instead the interpretation of the tags is left to the software, thus making it easy to implement new policies on the metadata. Since tags are word-sized, they can be pointers to complex data-structures of tags, such as tuples of tags, allowing for complex policies to be expressed and multiple orthogonal policies to be enforced in parallel.

The hardware undertakes the correct propagation of tags from operands to results according to the rules defined by the software. A hardware rule cache mapping sets of input tags to sets of output tags is used for common case efficiency. On each instruction dispatch, in parallel with the usual behavior of an instruction (e.g., execution of an addition in the ALU), the hardware forms the set of input tags and a lookup is performed on the rule cache. If the lookup is successful a set of output tags is returned and combined with the results of the normal execution of the instruction a new state is produced. On the other hand, if the lookup failed, the hardware invokes a trusted piece of system software - the fault handler - which checks the input tags and decides whether the execution should be allowed or not. In the first case, the fault handler returns a set of result tags, a pair of set of input and output tags is formed and inserted into the rules cache, while the faulting instruction is restarted and will now hit the cache. Otherwise, execution of this instruction violated some rules of the enforced policy and execution should not continue normally (e.g., should be halted).

As described in the original PUMP paper by Dehon *et al.* [14] and in [13] a rich set of effective security policies can be efficiently implemented using the architecture mentioned above. In particular, implementations of dynamic typing, memory safety for heap-based data, control flow integrity and taint tracking are described, evaluated against a specific threat model and benchmarked. The benchmarks are done using a simulation of the described hardware and the two papers claim low overhead (3% on average) for each of the policies named above.

Compared to other software solutions for enforcing security policies, the PUMP offers significantly lower overhead, thanks to dedicated hardware assistance, while the fact that interpretation of the metadata is done by software offers flexibility with regard to the policies that can be implemented, compared to hardware solutions implementing a specific policy.

While the PUMP offers flexibility at a low runtime performance overhead, there are more overheads associated to such a mechanism. For example adding metadata to all the data in the machine, would result in a 100% memory overhead. In addition, the extra hardware and the rule cache along with potentially larger memories could result into a 400% overhead on energy usage. [13] The authors claim that a careful and well-optimized implementation can reduce these numbers, resulting in a 50% energy overhead. *(CH: use optimized numbers)*

2.4.2 Concrete Machine Modeling PUMP Architecture

The concrete machine is a model of the PUMP architecture, modeling a RISC machine with a rules *cache* and the software *miss handler*. The instruction set has been extended with four additional instructions that are meant to be used by monitor code only, a restriction that is enforced by the monitor self-protection micro-policy.

The state of the concrete machine consists of the memory, the registers, the *pc* register, the *epc* register a special purpose register that holds the address of the faulting instruction after a cache miss and a rules cache. The cache works as a key-value store where a key is an *input vector* that contains an instruction opcode, the tag of the current instruction, the tag of the pc and up to three operand tags, and a value is an *output vector* which contain a tag for the new pc and a tag for any results from the execution of the instruction. In the context of the concrete machine a tag is the encoding into a word of a symbolic tag. Lifting this encoding relation to vectors, we get that a concrete vector is the encoding of a symbolic vector (*mvector*). Similar to the symbolic machine the contents of the memory, the registers, the pc and the epc are concrete atoms w@t where w is a word and t is the encoding of a tag into a word.

The stepping relation for the concrete machine is a bit more complicated than the one for the symbolic machine. In particular, on each step the machine forms the *input vector* and looks it up in the cache. If the lookup succeeds then the instruction is allowed, an *output vector* is returned by the cache and the next state is tagged according to it. If the lookup fails, then the *input vector* is saved in memory, the current *pc* is stored in *epc* and the machine traps to the *miss handler*. The above are demonstrated in the two example rules in fig. 2.3.

Addresses 0 to 5 are used to store the *input vector* and 6 to 7 are used by the miss handler to store the *output vector*. As a side-note, cache eviction is not modeled (an infinite cache is assumed).

2.4.3 Concrete Policy Monitor

Unlike the symbolic machine, where the user cannot cannot change the *transfer* function, enforcing a micro-policy on the concrete machine requires that we are able to protect the memory of the policy monitor and that privileged instructions are not executed by user code. This self-protection policy can be easily composed with another micro-policy and enforced by the infrastructure described above.

Using tags of the form, *User st*, *Entry st*, *Monitor* we can distinguish between user memory, monitor memory and monitor services. In particular *User st* is used to tag a user-level atom, where *st* is the word-encoding of a symbolic tag. *Monitor* is used to tag the monitor memory and a few reserved registers. The *pc* is tagged with *Monitor* when a monitor execution takes place and *User st* when user-code is executed. The tag *Entry st* is used to tag the first instruction of a monitor service and serves as an indication that execution will continue under the privileged *Monitor* mode.

The miss handler is a composed policy monitor that protects itself from *User* code and that enforces a desired micro-policy. One important thing to note is that the miss handler for the concrete machine can take an arbitrary number of steps before deciding that no violation occurred and returning to *User* mode, unlike the symbolic *transfer* function that does not need to take any steps.

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Nop \\ Nop : \{PC{=}t_{pc}, CI{=}t_i\} \rightarrow \{PC'{=}t'_{pc}, RES{=}-\} \end{array}}{(mem, reg, pc@t_{pc}, int) \rightarrow (mem, reg, pc + 1@t'_{pc}, int)} \quad (\textsc{Nop})$$

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Const\ n\ r \qquad reg[r]{=}w_{old}@t_{old} \\ Const : \{PC{=}t_{pc}, CI{=}t_i, OP1{=}t_{old}\} \rightarrow \{PC'{=}t'_{pc}, RES{=}t_{res}\} \\ reg' = reg[r{\leftarrow}n@t_{res}] \end{array}}{(mem, reg, pc@t_{pc}, int) \rightarrow (mem, reg', pc + 1@t'_{pc}, int)} \quad (\textsc{Const})$$

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Mov\ r_p\ r_s \\ reg[r_p]{=}w@t_p \qquad reg[r_s]{=}w_{old}@t_{old} \\ Mov : \{PC{=}t_{pc}, CI{=}t_i, OP1{=}t_p, OP2{=}t_{old}\} \rightarrow \{PC'{=}t'_{pc}, RES{=}t_{res}\} \\ reg' = reg[r_s{\leftarrow}w@t_{res}] \end{array}}{(mem, reg, pc@t_{pc}, int) \rightarrow (mem, reg', pc + 1@t'_{pc}, int)} \quad (\textsc{Mov})$$

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Binop\ op\ r_p\ r_s\ r_t \\ reg[r_p]{=}w_p@t_p \qquad reg[r_s]{=}w_s@t_s \qquad reg[r_t]{=}w_{old}@t_{old} \\ Binop\ op : \{PC{=}t_{pc}, CI{=}t_i, OP1{=}t_p, OP2{=}t_s, MR{=}t_{old}\} \rightarrow \{PC'{=}t'_{pc}, RES{=}t_{res}\} \\ reg' = reg[r_t{\leftarrow}(w_p op w_s@t_{res}] \end{array}}{(mem, reg, pc@t_{pc}, int) \rightarrow (mem, reg', pc + 1@t'_{pc}, int)} \quad (\textsc{Binop})$$

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Load\ r_p\ r_s \\ reg[r_p]{=}w_p@t_p \qquad mem[w_p]{=}w@t_{mem} \qquad reg[r_s]{=}w_{old}@t_{old} \\ Load : \{PC{=}t_{pc}, CI{=}t_i, OP1{=}t_p, OP2{=}t_{mem}, MR{=}t_{old}\} \rightarrow \{PC'{=}t'_{pc}, RES{=}t_{res}\} \\ reg' = reg[r_s{\leftarrow}(w@t_{res}] \end{array}}{(mem, reg, pc@t_{pc}, int) \rightarrow (mem, reg', pc + 1@t'_{pc}, int)} \quad (\textsc{Load})$$

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Store\ r_p\ r_s \\ reg[r_p]{=}w_p@t_p \qquad reg[r_s]{=}w_s@t_s \qquad mem[w_p]{=}w_{old}@t_{old} \\ Store : \{PC{=}t_{pc}, CI{=}t_i, OP1{=}t_p, OP2{=}t_s, MR{=}t_{old}\} \rightarrow \{PC'{=}t'_{pc}, RES{=}t'_d\} \\ mem' = mem[w_p{\leftarrow}w_s@t'_d] \end{array}}{(mem, reg, pc@t_{pc}, int) \rightarrow (mem', reg, pc + 1@t'_{pc}, int)} \quad (\textsc{Store})$$

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Jump\ r \qquad reg[r]{=}w@t_w \\ Jump : \{PC{=}t_{pc}, CI{=}t_i, OP1{=}t_w\} \rightarrow \{PC'{=}t'_{pc}, RES{=}-\} \end{array}}{(mem, reg, pc@t_{pc}, int) \rightarrow (mem, reg, w@t'_{pc}, int)} \quad (\textsc{Jump})$$

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Bnz\ r\ n \qquad reg[r]{=}w@t_w \\ Bnz : \{PC{=}t_{pc}, CI{=}t_i, OP1{=}t_w\} \rightarrow \{PC'{=}t'_{pc}, RES{=}-\} \end{array}}{(mem, reg, pc@t_{pc}, int) \rightarrow (mem, reg, w@t'_{pc}, int)} \quad (\textsc{Bnz})$$

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Jal\ r \\ reg[r]{=}w@t_w \qquad reg[ra]{=}w_{old}@t_{old} \\ Jal : \{PC{=}t_{pc}, CI{=}t_i, OP1{=}t_w, OP2{=}t_{old}\} \rightarrow \{PC'{=}t'_{pc}, RES{=}-\} \\ reg' = reg[ra{\leftarrow}pc + 1@t_{res}] \end{array}}{(mem, reg, pc@t_{pc}, int) \rightarrow (mem, reg', w@t'_{pc}, int)} \quad (\textsc{Jal})$$

$$\frac{\begin{array}{c} mem[pc] = \varnothing \qquad get\_service\ pc = (t_i, f) \\ Service : \{PC{=}t_{pc}, CI{=}t_i\} \rightarrow \{PC'{=}t'_{pc}, RES{=}-\} \\ f\ (mem, reg, pc, int,) = (mem', reg', pc', int,) \end{array}}{(mem, reg, pc@t_{pc}, int) \rightarrow (mem', reg', pc'@t'_{pc}, int')} \quad (\textsc{Service})$$

Figure 2.2: Stepping relation for the symbolic machine

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Store\ r_p\ r_s \\ reg[r_p]{=}w_p@t_p \qquad reg[r_s]{=}w_s@t_s \qquad mem[w_p]{=}w_{old}@t_{old} \\ cache \vdash (Store, t_{pc}, t_i, t_p, t_s, t_{old}) \mapsto (t'_{pc}, t'_d) \\ mem' = mem[w_p{\leftarrow}w_s@t'_d] \end{array}}{(mem, reg, pc@t_{pc}, epc, cache) \rightarrow (mem', reg, (pc{+}1)@t'_{pc}, epc, cache)} \quad (\textsc{Store})$$

$$\frac{\begin{array}{c} mem[pc] = i@t_i \qquad decode\ i = Store\ r_p\ r_s \\ reg[r_p]{=}w_p@t_p \qquad reg[r_s]{=}w_s@t_s \qquad mem[w_p]{=}w_{old}@t_{old} \\ cache \vdash (Store, t_{pc}, t_i, t_p, t_s, t_{old}) \not\mapsto \\ mem' = mem[0..5 \leftarrow (Store, t_{pc}, t_i, t_p, t_s, t_{old})] \end{array}}{(mem, reg, pc@t_{pc}, epc, cache) \rightarrow (mem', reg, trapaddr@Monitor, pc@t_{pc}, cache)} \quad (\textsc{Store-Miss})$$

Figure 2.3: Concrete step rules for Store instruction

Chapter 3

Control-Flow Integrity

Restricting the control-flow of a program in some way is a technique widely spread among security researchers. For example non-executable data (NXD) can be considered as a form of (very) coarse-grained *CFI* where control-flow is not allowed to reach any memory region that holds non-executable data. Another popular mitigation technique is to protect return addresses on the stack, thus restricting the control-flow on returns.

Moreover it is common that security properties are enforced dynamically by code that is statically injected to the program (e.g., Inlined Reference Monitors (IRM) [15] follow that approach), thus some form of *CFI* is required in order to ensure that these checks are not circumvented.

3.1 Related Work

3.1.1 Balancing between performance and security

Abadi *et al.* first proposed a technique to enforce *CFI* based on IRMs. In particular, they proposed to mark all valid targets of *indirect* control transfers with a unique identifier and inject checks before all indirect jumps (including return instructions). However they assume that any two destinations are equivalent, in the sense that they share the same identifier, if the CFG contains edges from the same set of sources, which may significantly reduce the precision of the CFG. The authors also note that a 2-ID approach where one identifier is used for calls and another for returns could provide adequate security in many cases.

The work of Abadi *et al.* sparked interest of researchers who tried to improve some of the weaknesses of the initial implementation, usually by choosing between performance against precision and vice-versa.

Bletsch *et al.* [6] followed the work of Abadi *et al.*, but changed their checking mechanism to perform the check after the control flow transfer has occurred which, as the authors claim, reduced the cache pressure and resulted in better performance. Precision remains the same with the implementation of Abadi *et al.*.

Zhang *et al.* [28] proposed *Compact Control Flow Integrity and Randomization* (CCFIR), a new efficient way to enforce coarse-grained *CFI*. CCFIR collects all valid targets of indirect control-transfers and stores them in a random order, in a protected section called "Springboard section". Indirect control-transfers are only allowed to addresses that are in the Springboard. Their implementation uses a 3-ID approach where one identifier is used for calls and the two other identifiers are for returns, separating them between returns to sensitive and non-sensitive functions. Their implementation also supports interaction between protected and un-protected modules, which makes it an attractive solution to coarse-grained *CFI*.

The above techniques are evaluated in [17] where the authors demonstrate code-reuse attacks against binaries protected by coarse-grained *CFI*. These attacks illustrate the need for fine-grained *CFI* which however incurs a high runtime-overhead penalty making deployment of such a mechanism unlikely.

Standard assumptions for effective *CFI* Most -if not all- *CFI* implementations also come with a set of assumptions under which *CFI* holds. Two standard assumptions for all mechanisms that attempt to enforce *CFI* are:

- Non-Executable Data (*NXD*), a security mechanism that disallows execution of data.
- Non-Writable Code (*NWC*). Changing the code of a program would allow an attacker to circumvent dynamic checks.

Both assumptions are fairly standard for modern computers and are enforced through hardware or software. In some cases *NXD* can be lifted, but additional security risks and complexity is not worth the minor advantages offered by such an action.

Many implementations that attempt to do fine-grained *CFI* also require that identifiers used to mark nodes in the CFG are unique.

3.1.2 Coarse-grained CFI Micro-Policy

(CH: consider moving to appendix, or related work section)

We can use the PUMP to implement the coarse-grained *CFI* mechanisms described earlier. Suppose we want to implement 1-ID *CFI*, we tag all indirect flow destinations and sources with a tag *Marked* and the rest of the instructions as *Unmarked*. Executing instructions that are sources of indirect flows, propagates their instruction tag to the *pc*. We then have to check that the tag on the destination matches the tag on the tag on the *pc*.

$$\frac{op \in \{Jump, Jal\}}{op : \{CI{=}Marked\} \rightarrow \{PC'{=}Marked, RES{=}-\}} \quad \text{(Mark)}$$

$$\frac{op \notin \{Jump, Jal\}}{op : \{PC{=}Marked, CI{=}Marked\} \rightarrow \{PC'{=}Unmarked, RES{=}-\}} \quad \text{(Check)}$$

$$\frac{op \notin \{Jump, Jal\}}{op : \{PC{=}Unmarked, CI{=}Unmarked\} \rightarrow \{PC'{=}Unmarked, RES{=}-\}} \quad \text{(NoCheck)}$$

Figure 3.1: Rules enforcing coarse-grained *CFI*, *NXD* and *NWC*

Rule *Mark* is used in the case the opcode is Jump or Jal (the only indirect jumps in the RISC machine we examine) and propagates the *Marked* tag on the tag of the new *pc*. Rule *Check* applies when the tag on the *pc* is set to *Marked* and corresponds to a legal destination and rule *NoCheck* corresponds to any instruction that is not a jump source or target.

We do not further study this coarse-grained approach as we consider it ineffective since attacks against it has already been demonstrated in [17]. Instead we are going to focus on implementing and formalizing a fine-grained *CFI* micro-policy.

3.1.3 Formal verification of Control-Flow Integrity

In [2] Abadi *et al.* extended their original paper, with -among other things- a more detailed formal study of *CFI*. Their formalization regarded a much simpler machine than the x86 omitting all the complexity of modern systems. The machine has a few instructions, a separate data memory and instruction memory which by the operational semantics of the machine are non-executable and non-writable respectively (enforcing *NXD* and *NWC* by construction), and a small set of registers. Moreover, their attacker model permits arbitrary changes to the data memory, arbitrary changes to all the registers but a few distinguished ones that are used during the dynamic checks and no changes to the instruction memory. The authors proof that under some assumptions every step respects the control-flow graph even in the presence of an attacker as powerful as the one described above. Their formal study served as a guideline for the implementation, but as it is done on paper their proofs cannot be machine checked. Furthermore, their formalization omits less interesting but important details such as instruction encoding and decoding which as shown in [21] are far from trivial for the x86.

Machine-checked formal verification efforts include [29], which is a SFI formalization for the ARM architecture that also enforces *CFI*. Their formalization was developed using the HOL theorem prover and a program logic framework they created. However their benchmarks report a 240% runtime overhead. The authors of [11] claim partial proofs for a *CFI* enforcement mechanism focused on the kernel of an operating system. Their runtime overhead can also reach 100%.

3.2 Fine-Grained Control-Flow Integrity Micro-Policy

The PUMP hardware allows us to avoid taking the difficult decision between performance and security. As shown in [13], we can enforce a *fine-grained CFI* policy with an average runtime overhead of less than 3% (maximum overhead of less than 10%), on the SPEC2006 benchmarks.

(CH: Shrink and polish this)(NG: done) We follow the standard approach, by designing a composed micro-policy that enforces *NXD*, *NWC* and *CFI*. We considered designs that lifted the *NXD* and *NWC* restrictions but we rejected them, as there did not seem to be any considerable advantages (i.e., compatibility with self-modifying programs, JIT compilers, etc.). Moreover unlike other *CFI* enforcement mechanisms we do not have to rely on the CPU or the operating system to enforce *NXD* and *NWC*, therefore lifting these restrictions would not reduce our assumptions and consequently would not increase our confidence in the robustness of our approach.

Our approach uses unique identifiers to tag the contents of the memory that correspond to sources and potential destinations of indirect flows according to a binary relation (on the identifiers) $\mathcal{CFG}$.

Consider the set of tags $\mathcal{T} = \{Data, Code\ id, Code\ \bot\}$ where *id* is a unique identifier (i.e., used to tag the contents of only one location in the memory). One simply way to achieve this is to use the address of the instruction as it's *id*, for example an instruction stored at address 100 would be tagged *Code* 100. This is the approach we take in our development. Adapting the rules from 2.2, we shall use *Data* to tag all contents in memory that are considered non-executable data, *Code id* to tag all contents in memory that are considered executable instructions and are sources or targets of indirect control flows and *Code* $\bot$ to tag all other instructions. The rules to enforce *NWC* and *NXD* are intuitively the same and only change to account for the splitting of the *Code* tag.

(NG: if/when we move coarse-grained to appendix, then we don't follow the same idea...)
We follow the same idea as with coarse-grained *CFI*, propagating the instruction tag of instructions that are sources of indirect flows to the tag on the *pc* of the next state and upon execution of the next instruction, checking that the tag on the *pc* and on the instruction are in some relation. In the case of coarse-grained *CFI* we required that they match but for fine-grained *CFI* we require that they are in the $\mathcal{CFG}$ relation.

$$\frac{op \in \{Jump, Jal\} \qquad (src, dst) \in \mathcal{CFG}}{op : \{PC{=}Code\ src, CI{=}Code\ dst\} \to \{PC'{=}Code\ dst, RES{=}-\}}\ (\textsc{Flow/Check})$$

$$\frac{op \in \{Jump, Jal\}}{op : \{PC{=}Data, CI{=}Code\ dst\} \to \{PC'{=}Code\ dst, RES{=}-\}}\ (\textsc{Flow/NoCheck})$$

$$\frac{(src, dst) \in \mathcal{CFG}}{Store : \{PC{=}Code\ src, CI{=}Code\ dst, MR{=}Data\} \to \{PC'{=}Data, RES{=}Data\}}\ (\textsc{Store/Check})$$

$$\frac{ti \in \{Code\ dst, Code\ \bot\}}{Store : \{PC{=}Data, CI{=}ti, MR{=}Data\} \to \{PC'{=}Data, RES{=}Data\}}\ (\textsc{Store/NoCheck})$$

$$\frac{opcode \notin \{Jump, Jal, Store\} \qquad (src, dst) \in \mathcal{CFG}}{opcode : \{PC{=}Code\ src, CI{=}Code\ dst\} \to \{PC'{=}Data, RES{=}-\}}\ (\textsc{Rest/Check})$$

$$\frac{opcode \notin \{Jump, Jal, Store\} \qquad ti \in \{Code\ dst, Code\ \bot\}}{opcode : \{PC{=}Data, CI{=}ti\} \to \{PC'{=}Data, RES{=}-\}}\ (\textsc{Rest/NoCheck})$$

Figure 3.2: Rules enforcing fine-grained *CFI*, *NXD* and *NWC*

We note in the above rules that the tag on the *pc* is *Data* when no check for a control-flow violation is required and *Code src* where *src* is some id, when an indirect flow instruction was executed and a check for a control-flow violation is required. An important observation is that the rules above allow for one control-flow violation to occur, but disallow the next step and therefore the machine will certainly halt after a violation.

If the PUMP hardware fetched the tag on the memory address the machine is jumping to and passed it as an argument to input vector, as it does in the case of a Store instruction, we would be able to enforce *CFI* with no violations at all. (**TODO:** It can't do that for efficiency reasons?)

Chapter 4

Formally Verified Control-Flow Integrity Micro-Policy

Using the micro-policies framework described in section 2.3 we proved that the concrete machine instantiated with a fine-grained*CFI* micro-policy like the one described in section 3.2 *simulates* an abstract machine that has *CFI* by construction. We do this proof by using the symbolic machine as an intermediate step, first proving backward simulation between the symbolic and the abstract machine and afterwards by leveraging the framework of section section 2.3 we obtain a backward refinement between the concrete and the abstract machine.

In addition, we provide an attacker model for all the machines used and we prove that a property capturing the notion of *CFI* holds even when the attacker tampers with the machine, similarly to what is proposed in [2], but adapted to the setting of our machines. We do this by first proving this property for the abstract machine and then by using a generic preservation theorem we developed, we prove that this property is *preserved* by backward refinement and thus transferring the property to the symbolic and consequently to the concrete machine. This proof structure, allows us to build our proofs in a modular way and additionally reduced the proof effort, as it allowed us to reuse the preservation theorem for proving *CFI* for both the symbolic and the concrete machine and allowed us to do most of the reasoning at the symbolic level, even for proofs that concerned the concrete machine.

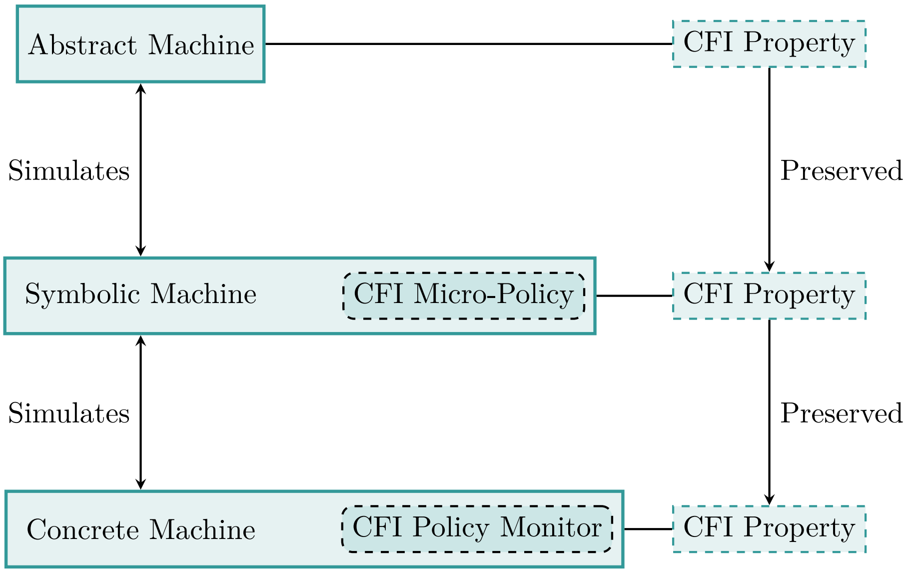

Figure 4.1: Diagram explaining proof structure

(**TODO:** Think about labels, what should be dashed and not and how to improve diagram.)

4.1 Representing control-flow graphs

Our approach for enforcing *CFI*, as explained in section 3.2, requires that we encode the nodes in the control-flow graph in terms of identifiers, which in turn are used to tag all sources and targets of indirect control-flows.

At this point we take a detour, to point out an important design point of the micro-policies framework and our *CFI* micro-policy. Throughout both developments, a heavily parametric and modular approach was taken. This parametric design is enabled by the use of the *Section* and *Type Classes* mechanisms of Coq. As an example, the node identifiers, along with a number of properties we require of them are expressed by the following interface (defined in terms of a type class):

```
Context {t : machine_types}.

Class cfi_id := {
  id          : eqType;

  word_to_id : word t → option id;
  id_to_word : id → word t;

  id_to_wordK : ∀ x, word_to_id (id_to_word x) = Some x;
  word_to_idK : ∀ w x, word_to_id w = Some x → id_to_word x = w

}.
```

Listing 4.1: Interface of node identifiers

The *Context* command on the top of the code above, allows us to *assume* that there exists an instance of this interface. In fact, the *machine_types* argument is just another type class, serving as a specification of the various types of the machine (e.g., the word size). This approach allowed us to abstract away from various details and structure our proofs in a clean way. In addition, we can easily instantiate a different machine with minimal changes in our proofs and definitions (e.g., instantiate the machine with a different word size).

However, one drawback is that one wrong specification in a type class would disallow us to instantiate it and would require that we go back and change all parts that used this wrong specification (e.g., in our case, the *cfi_id* class was widely used). Therefore one should be careful when doing heavy use of such mechanisms.

Returning to the identifiers, looking at the definition in listing 4.1, we require that the type of the identifiers *id* is an Eqtype (has decidable boolean equality) and that there exists conversion functions between elements of type *word* and *id*, satisfying some constraints.

(NG: Should I give some intuition, as to why word_to_id is partial? Is it obvious?)

As mentioned in section 3.2, we check for violations of the control-flow with respect to a binary relation (on the identifiers) $\mathcal{CFG}$ which represents the set of allowed (indirect) jumps. We can extend this relation to precisely describe the control-flow of a program, by lifting $\mathcal{CFG}$ to a relation $\mathcal{SUCC}_{\mathcal{CFG}}$ on machine states, that includes the set of allowed targets for the rest of the instructions. In our Coq development we assumed a translation of the allowed jumps in form of a function on two identifiers.

```
Variable cfg : id → id → bool.
```

Listing 4.2: Function on ids representing the set of allowed jumps

In addition we defined a function valid_jmp (referred to with the notation $\mathcal{J}$) that expresses the set of allowed jumps between words, by using the *word_to_id* function.

```
Definition valid_jmp w1 w2 :=
  match word_to_id w1, word_to_id w2 with
    | Some id1, Some id2 ⇒ cfg id1 id2
    | _, _ ⇒ false
  end.
```

Listing 4.3: Function on words representing the set of allowed jumps

4.2 Control-Flow Integrity Property

Our formalization includes a definition of *CFI*, similar to the one found in [2], which we prove to be true of all our machines. The need for a new definition arises from fundamental differences between our enforcement mechanism on the concrete mechanism and the one used by Abadi *et al.*. In particular, our enforcement-mechanism does not prevent a violation, instead it can detect it after it has occurred by taking an arbitrary number of “protected” (monitor mode) steps before eventually bringing the machine to a halt. This does not have any impact on the security effectiveness of our mechanism, it does however lead to a more complex definition and therefore more complex proofs.

The definition of *CFI* is further parameterized by an attacker model. We model the attacker as a step relation ($\rightarrow_a$). Intuitively the attacker is allowed to change any *user-level* data but not the code of the program and the *pc*, as well as the tags in the case of a tagged machine. This limitations ensures that an attacker cannot directly circumvent the monitor protection mechanism and our user-level policies (*NWC*, *NXD* and *CFI*). To account for attacker steps, the stepping relation is extended as the union of the normal step relation ($\rightarrow_n$), as defined by the machine semantics, and the attacker step relation ($\rightarrow_a$), as defined by the attacker model.

$$\frac{s \rightarrow_n s'}{s \rightarrow s'} \qquad\qquad \frac{s \rightarrow_a s'}{s \rightarrow s'}$$

Figure 4.2: Step relation definition

We define a predicate *initial s*, where s is a machine state, that states that s is an initial state. We use this predicate to express some invariants that are preserved through execution (e.g., the initial tagging scheme for the memory). Finally we define a stopping predicate on an execution trace that states that the machine is coming to a halt with respect to normal steps.

Since we want to instantiate the above parameters in a different way for each of our machines, it makes sense to wrap them in a type class which we will instantiate for each machine to get the corresponding definition of *CFI*.

```
Class cfi_machine := {
  state : Type;
  initial : state → Prop;

  step : state → state → Prop;
  step_a : state → state → Prop;

  succ : state → state → bool;
  stopping : list state → Prop
}.
```

Listing 4.4: Interface of a cfi_machine

For a machine of type cfi_machine we give the following definitions:

Definition 4.1 (Trace has CFI)**.** *We say that an execution trace* $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ has CFI *if for all* $i \in [0, \ldots, n)$ *if* $s_i \rightarrow_n s_{i+1}$ *then* $(s_i, s_{i+1}) \in \mathcal{SUCC}_{\mathcal{CFG}}$

(NG: The word relation for succ and cfg is strange since they are booleans, is it ok, or does it confuse you, making you believe they are Props?)

The above definition corresponds to the one found in [2], however it is stronger in the sense that it requires that steps that are in the intersection of normal and attacker steps respect the control-flow. If we did not allow for any violations then the above definition would be enough, but since our enforcement mechanism allows for one violation we have to resort to a weaker definition.

Definition 4.2 (CFI)**.** *We say that the machine* $(State, initial, \rightarrow_n, \rightarrow_a, \mathcal{SUCC}_{\mathcal{CFG}}, stopping)$ *has* CFI *with respect to the set of allowed indirect jumps* $\mathcal{CFG}$ *if, for any execution starting from initial state* s_0 *and producing a trace* $s_0 \rightarrow \ldots \rightarrow s_n$*, either*

1. *The whole trace has* CFI *according to definition 4.1, or else*

2. *There is some* i *such that* $s_i \rightarrow_n s_{i+1}$*, and* $(s_i, s_{i+1}) \notin \mathcal{SUCC}_{\mathcal{CFG}}$*, where the sub-traces* $s_0 \rightarrow \ldots \rightarrow s_i$ *and* $s_{i+1} \rightarrow \ldots \rightarrow s_n$ *both have CFI and the sub-trace* $s_{i+1} \rightarrow \ldots \rightarrow s_n$ *is stopping.*

4.3 The Abstract Machine

The abstract machine has *CFI*, *NXD* and *NWC* by construction and will serve as a specification for the symbolic and eventually the concrete machine that implement *CFI* through the tag-based system explained in the previous chapter.

Unlike the symbolic and the concrete machine, this abstract machine splits the memory into two disjoint memories, an instruction memory and a data memory. The instruction memory is fixed (non-writable) and the machine uses this memory to fetch instructions to execute, so *NWC* and *NXD* are enforced by construction.

In addition the state of the machine includes an *ok* bit, indicating whether a control-flow violation has occurred or not. The rest of the machine state is completed by a set of registers and a *pc* register. We use a 5-tuple notation for the state (im, dm, reg, pc, ok), where the first field is the instruction memory, the second the data memory, the third the registers, the fourth is the pc register and the fifth is the *ok* bit.

4.3.1 Operational semantics

Below is the step rule for the Store instruction, illustrating both *NWC* and *NXD*. Notice that the instruction is fetched by the instruction memory and the store is done on the data memory.

$$\frac{\begin{array}{c} im[pc] = i \qquad decode\ i = Store\ r_p\ r_s \qquad reg[r_p] = p \\ reg[r_s] = w \qquad dm' = dm[p \leftarrow w] \end{array}}{(im, dm, reg, pc, true) \rightarrow (im, dm', reg', pc + 1, true)} \quad (\textsc{Store})$$

Figure 4.3: Step rule for Store instruction of abstract machine

In the above rule, the *ok* bit is true for both the starting and the resulting state. In fact, the machine can take a step only when the *ok* bit is set to true. In the above rule, the *ok* bit is set to true in the resulting state, indicating that no control-flow violation has happened, as expected by the execution of a Store instruction. Control-flow violations in the *NWC* setting our machine is executing, can only occur from *indirect* jump instructions, in our case the Jump and Jal instructions. Upon execution of a Jump or Jal instruction, we consult $\mathcal{J}$ (see listing 4.3) to check whether the change of control-flow is legal. If the jump is not allowed according to $\mathcal{J}$ then the jump is taken but the *ok* bit is set to false, which will halt the machine in the next step, as it is only allowed to step when the *ok* bit is set to true. Otherwise the *ok* bit will remain true.

As the abstract machine serves as a specification to a machine with *CFI*, a more intuitive definition of it would not include the *ok* bit and would only allow the Jump and Jal instructions to step if they do not violate the control-flow graph. However, this abstract

$$\frac{im[pc] = i \qquad decode\ i = Jal\ r \qquad reg[r] = pc' \qquad reg' = reg[ra \leftarrow pc + 1] \qquad ok = (pc, pc') \in \mathcal{J}}{(im, dm, reg, pc, true) \rightarrow (im, dm, reg', pc', ok)} \quad (\textsc{Jal})$$

$$\frac{im[pc] = i \qquad decode\ i = Jump\ r \qquad reg[r] = pc' \qquad ok = (pc, pc') \in \mathcal{J}}{(im, dm, reg, pc, true) \rightarrow (im, dm, reg', pc', ok)} \quad (\textsc{Jump})$$

Figure 4.4: Step rule for Jump and Jal instruction of abstract machine

machine would not allow for any violations to occur unlike our enforcement mechanism for the symbolic and the concrete machine and would lead to more complex simulation proofs, therefore we do not favor it.

The abstract machine also allows for monitor services to be included, although the *CFI* enforcement mechanism does not require any. We assume that a monitor service is a privileged action and that it's execution does not violate the control-flow of the program. Execution of a monitor service is done simply by jumping to it's address, there is no separate instruction. As with all other instructions, execution of the monitor service is only allowed if the *ok* bit is set to true.

$$\frac{pc \notin dom(im) \qquad pc \notin dom(dm) \qquad get\_service\ pc = (addr, f) \qquad f\ (im, dm, reg, pc, true) = (im, dm', reg', pc', true)}{(im, dm, reg, pc, true) \rightarrow_n (im, dm', reg', pc', true)} \quad (\textsc{Service})$$

Figure 4.5: Step rule for monitor services of abstract machine

(**TODO:** Put all rules in appendix?)

4.3.2 Attacker model

The attacker for the abstract machine is allowed to change the contents of the data memory and the registers but not the rest of the state.

$$\frac{dom\ dm = dom\ dm' \qquad dom\ reg = dom\ reg'}{(im, dm, reg, pc, ok) \rightarrow_a^A (im, dm', reg', pc, ok)}$$

Figure 4.6: Attacker model for the abstract machine

4.3.3 Allowed control-flows for the abstract machine

We can construct a function $\mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$ for the abstract machine that represents the set of allowed control-flows for all instructions, by extending the set of allowed jumps $\mathcal{CFG}$ we introduced earlier.

Below we give a specification of the $\mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$ function for the abstract machine, in form of inference rules. A function is defined in the actual Coq development.

Notice that a monitor service is allowed to return anywhere. As we mentioned before, monitor services, execute in a protected by the monitor environment where we assume that an attacker who can only tamper the machine at the user level cannot interfere.

$$\frac{im[pc] = i \qquad decode\ i \in \{Jal\ r, Jump\ r\} \qquad (pc, pc') \in \mathcal{J}}{((im, dm, reg, pc, ok), (im, dm', reg', pc', ok)) \in \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}}\ (\textsc{IndirectFlows})$$

$$\frac{\begin{array}{c} im[pc] = i \qquad decode\ i = Bnz\ r\ imm \\ (pc' = pc + 1) \vee (pc' = pc + imm) \end{array}}{((im, dm, reg, pc, ok), (im, dm', reg', pc', ok)) \in \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}}\ (\textsc{ConditionalFlows})$$

$$\frac{\begin{array}{c} im[pc] = i \qquad decode\ i \notin \{Jal\ r, Jump\ r, Bnz\ r\ imm, \varnothing\} \\ pc' = pc + 1 \end{array}}{((im, dm, reg, pc, ok), (im, dm', reg', pc', ok)) \in \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}}\ (\textsc{NormalFlows})$$

$$\frac{\begin{array}{c} im[pc] = \varnothing \qquad dm[pc] = \varnothing \\ get\_service\ pc = (addr, f) \end{array}}{((im, dm, reg, pc, ok), (im, dm', reg', pc', ok)) \in \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}}\ (\textsc{ServiceFlows})$$

Figure 4.7: Allowed control-flows for instructions of the abstract machine

4.3.4 Stopping predicate for the abstract machine

Finally, we define what it means for the abstract machine to be "stopping" by defining a predicate on execution traces:

Definition 4.3 (Abstract Stopping Predicate)**.**

1. *All states in the trace are stuck with respect to normal steps* ($\rightarrow_n$)
2. *All steps in the trace are attacker steps* ($\rightarrow_a$)

4.3.5 CFI proof for the Abstract Machine

Regarding initial states, we only require that the *ok* bit is set to true. We can now instantiate the class of the machines defined in listing 4.4, with the abstract machine and that the abstract machine has *CFI* according to definition 4.2. We first prove a helpful lemma.

Lemma 4.4 (Step Intersection)**.** *For all states st st' such that* $st \rightarrow^{A}_{a} st'$ *and* $st \rightarrow_n st'$*,* $(st, st') \in \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$.

Proof.

- By the relation $st \rightarrow_n st'$ we know that the *ok* bit of *st* is set to true.
- The relation $st \rightarrow^{A}_{a} st'$ retains the *ok* bit of *st*, therefore *st'* has the *ok* bit set to true.
- It trivially follows from the definition of $\mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$ that $(st, st') \in \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$.

□

Theorem 4.5 (Abstract CFI)**.** *The abstract machine has the* CFI *property stated by definition 4.2.*

Proof. The proof proceeds by induction on the execution trace.

- **Base Case** In this case the execution trace is made up of a single step $st \rightarrow st'$. We proceed with case analysis on the step.
  - **Attacker Step** By lemma 4.4 we note that an attacker step cannot be a normal step outside the $\mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$ relation. Thus in this case the whole trace has *CFI* according to definition 4.1.
  - **Normal Step** By case analysis, if $(st, st') \in \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$ then trivially the whole trace has *CFI*. Otherwise $(st, st') \notin \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$ and the sub-traces *st* and *st'* vacuously have *CFI*. In addition the sub-trace *st'* is stopping, as the *ok* bit of *st'* is set to false and the state is stuck with respect to normal steps.
- **Inductive Case** In this case the execution trace is extended by an additional step at it's beginning $\mathbf{s_0} \rightarrow \mathbf{s_1} \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$. By the induction hypothesis either:
  - The trace $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$ has *CFI*, by case analysis if $(s_0, s_1) \in \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$ the whole trace has *CFI*. Otherwise $(s_0, s_1) \notin \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$, the sub-trace s_0 vacuously has *CFI* and the sub-trace $s_1 \rightarrow \ldots \rightarrow s_n$ has *CFI* by the induction hypothesis. Additionally, the sub-trace $s_1 \rightarrow \ldots \rightarrow s_n$ is stopping because:
    * The whole trace is made up of attacker steps. Since $(s_0, s_1) \notin \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$ the *ok* bit of s_1 will be set to false and a normal step is not allowed by the operational semantics, while attacker steps retain the *ok* bit.
    * The whole trace is stuck with respect to normal steps. Trivial from the above.
  - There exists a step $s_{v1} \rightarrow_n s_{v2}$ such that $(s_{v1}, s_{v2}) \notin \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$ and the sub-traces $s_1 \rightarrow \ldots \rightarrow s_{v1}$ and $s_{v2} \rightarrow \ldots \rightarrow s_n$ both have *CFI* and the later is also a stopping trace.
    * If $(s_0, s_1) \in \mathcal{SUCC}^{\mathcal{A}}_{\mathcal{CFG}}$ then definition 4.2 still holds and the sub-trace $s_1 \rightarrow \ldots \rightarrow s_{v1}$ is extended by one step to $s_0 \rightarrow \ldots \rightarrow s_{v1}$.
    * Otherwise the *ok* bit for s_1 is set to false and the rest of the trace is stuck with respect to normal steps. However from the induction hypothesis we know that $s_{v1} \rightarrow_n s_{v2}$, which is a contradiction.

□

4.4 The Symbolic Machine

The symbolic machine was described in section 2.3.2. Unlike the abstract machine, the symbolic machine has one memory and the distinction between data and executable instructions is made through tags, in a fashion similar to what was shown in sections 2.2 and 3.2. We instantiate the symbolic machine, according to the aforementioned sections, with a set of tags $\mathcal{T} = \{Data, Code\ id, Code\ \bot\}$ where id is drawn from the class of identifiers listing 4.1.

Although enforcement of *CFI* does not require any monitor services we expose the monitor services mechanism and we check whether calls to each monitor service are allowed or not according to the control-flow graph. This is done by assuming a lookup-table of monitor services where each entry has a tag that is used to check for control-flow violations and a semantic function from symbolic state to symbolic state which produces the new machine state after execution of the system call, as shown in fig. 2.2.

We do not need any internal state for this micro-policy therefore, only the transfer function is left to implement.

```
Context {ids : @cfi_id t}.

Inductive cfi_tag : Type :=
| INSTR : option id → cfi_tag
| DATA  : cfi_tag.
```

Listing 4.5: Coq definition of Symbolic tags

4.4.1 Transfer Function

We implement the *transfer* function based on the rules found in 3.2, using Gallina to define a function mapping input vectors (mvector) to output vectors (rvector).

```
Definition cfi_handler (ivec : Symbolic.IVec cfi_tags) :
           option (Symbolic.OVec cfi_tags (Symbolic.op ivec)) :=
  match ivec with
  | mkIVec   (Jump as op) (Code (Some n))   (Code (Some m))   _
  | mkIVec   (Jal  as op) (Code (Some n))   (Code (Some m))   _  ⇒
    if cfg n m then
      Some (mkOVec (Code (Some m)) (default_rtag op))
    else
      None
  | mkIVec   (Jump as op)  Data  (Code (Some n))   _
  | mkIVec   (Jal  as op)  Data  (Code (Some n))   _   ⇒
    Some (mkOVec (Code (Some n)) (default_rtag op))
  | mkIVec   Jump   Data  (Code None)  _
  | mkIVec   Jal    Data  (Code None)  _  ⇒
    None
  | mkIVec   Store  (Code (Some n))  (Code (Some m))  [_ ; _ ; Data] ⇒
    if cfg n m then Some (mkOVec Data Data) else None
  | mkIVec   Store  Data  (Code _)  [_ ; _ ; Data]  ⇒
    Some (mkOVec Data Data)
  | mkIVec   Store  _  _  _  ⇒ None
  | mkIVec   op     (Code (Some n))  (Code (Some m))  _  ⇒
    (* this includes op = Service *)
    if cfg n m then
      Some (mkOVec Data (default_rtag op))
    else
      None
  | mkIVec   op     Data  (Code _)  _  ⇒
    (* this includes op = Service, fall-throughs checked statically *)
    Some (mkOVec Data (default_rtag op))
  | mkIVec _ _ _ _ ⇒ None
  end.
```

Listing 4.6: Transfer function for symbolic machine in Coq pseudo-code

(**TODO:** Should I remove the aggressive capitilization above? It may make it less painful on the eye... Thanks to dependent types it also looks super ugly too, probably make it pseudo-code at some point) *(NG: simplified the above to be closer to the rest of the document and avoid all the dependent type magic/hackery)*

Although, the rules in section 3.2 were fairly simply, expressing them using Gallina's pattern matching increased their size. We also experimented, with different ways of writing the transfer function but we decided to stick with the definition above as it's the most straightforward. It's worth to note that bugs in the above definition were easily made apparent when proving theorems involving the transfer function. In fact, an "interesting" experiment was to re-define the above function in a different way and prove the two equivalent. It took two iterations before getting both functions to agree and although for small definitions like the one above, testing or manually reviewing the code will reveal most if not all bugs, the importance of formal verification in software engineering and critical software is made obvious even for definitions that may seem trivial at first. The correctness of the transfer function will come from simulation proofs between the abstract and the symbolic machine.

4.4.2 Attacker model

Similar to the abstract attacker, the symbolic attacker can change all words tagged as *Data* but not the ones tagged as *Code*. This is expressed by the following relations:

$$\frac{}{w_1@Data \rightarrow^S_a w_2@Data} \quad (\textsc{AttackData})$$

$$\frac{}{w_1@Code\ id \rightarrow^S_a w_1@Code\ id} \quad (\textsc{AttackInstr})$$

Figure 4.8: Attacker capabilities

These attacker capabilities on symbolic atoms are lifted to the memory and registers by a pointwise extension. (**TODO:** be more specific about pointwise?)

$$\frac{mem \rightarrow^S_a mem' \qquad reg \rightarrow^S_a reg'}{(mem, reg, pc@t_{pc}, int) \rightarrow^S_a (mem', reg', pc@t_{pc}, int)}$$

Figure 4.9: Attacker model for the Symbolic machine

4.4.3 Allowed control-flows for the Symbolic Machine

Similar to the abstract machine of section 4.3.3, we construct $\mathcal{SUCC}^S_{\mathcal{CFG}}$ for the symbolic machine (fig. 4.10) by extending the set of allowed jumps $\mathcal{CFG}$.

4.4.4 Initial states of the Symbolic Machine

For the symbolic machine, we do require that certain tagging conventions are respected initially. Additionally we prove that these initial conditions are invariants of the machine and they are preserved at every (normal or attacker) step.

These invariants are required for backward simulation between the symbolic and the abstract machine.

Definition 4.6 (Instructions Tagged). *For all addresses addr in the memory such that*

$$mem[addr] = i@Code\ id$$

$$\frac{\begin{array}{c} mem[pc] = i@(Code\ src) \qquad decode\ i \in \{Jal\ r, Jump\ r\} \\ mem[pc'] = i'@(Code\ dst) \\ (src, dst) \in \mathcal{CFG} \end{array}}{((mem, reg, pc, int,), (mem, reg, pc', int,)) \in \mathcal{SUCC}^{\mathcal{S}}_{\mathcal{CFG}}} \quad (\textsc{IndirectFlows})$$

$$\frac{\begin{array}{c} mem[pc] = i@(Code\ src) \qquad decode\ i \in \{Jal\ r, Jump\ r\} \\ mem[pc'] = \varnothing \qquad get\_service\ pc = (Code\ dst, f) \\ (src, dst) \in \mathcal{CFG} \end{array}}{((mem, reg, pc, int,), (mem, reg, pc', int,)) \in \mathcal{SUCC}^{\mathcal{S}}_{\mathcal{CFG}}} \quad (\textsc{IndirectFlows2})$$

$$\frac{\begin{array}{c} mem[pc] = i@(Code\ _) \qquad decode\ i = Bnz\ r\ imm \\ (pc' = pc + 1) \vee (pc' = pc + imm) \end{array}}{((mem, reg, pc, int,), (mem, reg, pc', int,)) \in \mathcal{SUCC}^{\mathcal{S}}_{\mathcal{CFG}}} \quad (\textsc{ConditionalFlows})$$

$$\frac{\begin{array}{c} mem[pc] = i@(Code\ _) \qquad decode\ i \notin \{Jal\ r, Jump\ r, Bnz\ r\ imm, \varnothing\} \\ pc' = pc + 1 \end{array}}{((mem, reg, pc, int,), (mem', reg', pc', int,)) \in \mathcal{SUCC}^{\mathcal{S}}_{\mathcal{CFG}}} \quad (\textsc{NormalFlows})$$

$$\frac{mem[pc] = \varnothing \qquad get\_service\ pc = (t'_i, f)}{((mem, reg, pc, int,), (mem', reg', pc', int',)) \in \mathcal{SUCC}^{\mathcal{S}}_{\mathcal{CFG}}} \quad (\textsc{ServiceFlows})$$

Figure 4.10: Allowed control-flows for instructions of the symbolic machine

addr is in the domain of word_to_id *and additionally*

$$word\_to\_id\ addr = id$$

Definition 4.7 (Entry Points Tagged)**.** *For all addresses addr such that*

$$\begin{aligned} &mem[addr] = \varnothing \\ &get\_service\ addr = (it, f) \\ &it = Code\ id \end{aligned}$$

addr is in the domain of word_to_id *and additionally*

$$word\_to\_id\ addr = id$$

Definition 4.8 (Valid Jumps Tagged)**.** *For all addresses saddr, taddr such that*

$$(saddr, taddr) \in \mathcal{J}$$

it holds that

$$\exists i, mem[saddr] = i@Code\ (word\_to\_id\ saddr)$$

and either

$$\exists i', mem[taddr] = i'@Code\ word\_to\_id\ taddr$$

or

$$\begin{aligned} &mem[taddr] = \varnothing \\ &\exists (it, f), get\_service\ addr = (it, f) \\ &it = Code\ (word\_to\_id\ taddr) \end{aligned}$$

Additionally we need two ((**TODO:** Or three)) more invariants for forward simulation. These two invariants enforce that all Jump and Jal instructions are tagged with a unique identifier.

Definition 4.9 (Jumps Tagged). *For all addresses addr and instructions i such that* $mem[addr] = i@Code\ x$ *and decode* $i = Jumpr$*, it holds that*

$$\exists id, word\_to\_idaddr = id \land x = id$$

Definition 4.10 (Jals Tagged). *For all addresses addr and instructions i such that* $mem[addr] = i@Code\ x$ *and decode* $i = Jalr$*, it holds that*

$$\exists id, word\_to\_idaddr = id \land x = id$$

We define a predicate *initial* that determines whether a symbolic state is an initial state.

Definition 4.11 (Symbolic Initial States). *A symbolic state* s^S *is an initial state (*$initial^S\ s^S$*) if definitions 4.6 to 4.10 hold for* s^S *and additionally the tag on the pc is set to Data.*

It's straightforward by the semantics of the step relations to prove that both normal and attacker steps preserve each of the invariants. We only need to assume that this holds for monitor services (i.e., if we were to provide some monitor services they would have to preserve these invariants).

Lemma 4.12 (Symbolic Invariants preserved by normal steps). *For all symbolic states (st, st'),*

$$\begin{aligned}&invariants\ st \implies\\&st \rightarrow_n st' \implies\\&invariants\ st'\end{aligned}$$

Lemma 4.13 (Symbolic Invariants preserved by attacker steps). *For all symbolic states (st, st'),*

$$\begin{aligned}&invariants\ st \implies\\&st \rightarrow_a^S st' \implies\\&invariants\ st'\end{aligned}$$

4.4.5 Stopping predicate for the Symbolic Machine

Similar to the abstract machine, we say that an execution trace of the symbolic machine is stopping if:

Definition 4.14 (Symbolic Stopping Predicate).

- *All states in the trace are stuck with respect to normal steps* ($\rightarrow_n$)
- *All steps in the trace are attacker steps* ($\rightarrow_a$)

4.4.6 Symbolic-Abstract simulation

The Symbolic-Abstract simulation formally defines the connection between the two machines. We prove a 1-backward simulation theorem for both normal and attacker steps. This means that every step of the symbolic machine can be matched by one step of the abstract machine. Additionally we prove a 1-forward simulation for normal steps, which means that every step of the abstract machine can be matched by one on the symbolic machine.

(**TODO:** Could use some help on improving the text above)

Definition 4.15 (1-Backward Simulation)**.** *A low-level machine simulates a high-level machine with respect to a simulation relation* $\sim$ *between low-level machine states and high-level machine states, if* $s_1^H \sim s_1^L$ *and* $s_1^L \rightarrow_n s_2^L$ *implies that there exists* s_2^H *such that,* $s_2^H \sim s_2^L$ *and* $s_1^H \rightarrow_n s_2^H$.

We visualize the above definition with the following diagram:

$$\begin{tikzcd}
s_1^L \arrow[r] \arrow[d, squiggly, no head] & s_2^L \arrow[d, squiggly, dashed, no head] \\
s_1^H \arrow[r, dashed] & s_2^H
\end{tikzcd}$$

(Plain lines denote premises, dashed ones conclusions.)

Definition 4.16 (1-Forward Simulation)**.** *A high-level machine simulates a low-level machine with respect to a simulation relation* $\sim$ *between low-level machine states and high-level machine states, if* $s_1^H \sim s_1^L$ *and* $s_1^H \rightarrow_n s_2^H$ *implies that there exists* s_2^L *such that,* $s_2^H \sim s_2^L$ *and* $s_1^L \rightarrow_n s_2^L$.

Intuitively, backward simulation is enough to capture the desired security property. Our intuition is further strengthened later, when we prove that the *CFI* property given by definition 4.2 is preserved by backward refinement. However, a trivial machine that cannot take any step also enjoys *CFI* vacuously. Forward simulation guarantees that this is not the case for our symbolic machine and proves that it is a meaningful implementation of the abstract machine.

Simulation Relation

We define the state simulation relation between the symbolic and abstract machine by defining the simulation relation for each component of the state.

Definition 4.17 (Data Memory Simulation)**.** *An abstract data memory dm is in simulation with a symbolic memory mem, if for all words w, x it holds that*

$$mem[w] = x@Data \iff dm[w] = x$$

Definition 4.18 (Instruction Memory Simulation)**.** *An abstract instruction memory im is in simulation with a symbolic memory mem, if for all words w, x it holds that*

$$(\exists\ it,\ mem[w] = x@(Code\ it)) \iff im[w] = x$$

Definition 4.19 (Registers Simulation). *An abstract register set areg is in simulation with a symbolic register set sreg, if for all registers r and words x it holds that*

$$sreg[r] = x@Data) \iff areg[r] = x$$

Definition 4.20 (PC simulation). *The abstract pc (apc) is in simulation with the symbolic pc ($spc@t_{pc}$), if it holds that*

$$apc = spc \land (t_{pc} = Data \lor \exists n \in id,\ t_{pc} = Code\ n)$$

Definitions 4.17 to 4.20 relate the basic components of the state. What is left to do, is relate the *ok* bit of the abstract machine with the state of the symbolic machine.

Definition 4.21 (Correctness). *The statement of correctness, states that for the symbolic memory (smem), the symbolic pc ($spc@t_{pc}$) and the ok bit of the abstract machine, it holds that for all words i and tags ti,*

$$\begin{aligned}
&smem[spc] = i@ti \implies \\
&ok = true \iff \\
&(\forall src \in id,\ t_{pc} = Code\ src \implies \\
&\quad \exists dst \in id, \\
&\qquad ti = Code\ dst \land (src, dst) \in \mathcal{CFG})
\end{aligned}$$

Informally definition 4.21 states that if the tag on the current instruction is *ti*, then if the tag on the pc is set to *Code* src (which means an indirect flow occurred in the previous step), there exists an *id* dst which is used to tag the current instruction and additionally the flow from an instruction with *id* src to one with *id* dst is allowed according to $\mathcal{CFG}$, if and only if the *ok* bit of the abstract machine is set to true. This definition captures the notion that a violation in the abstract machine is also a violation in the symbolic machine and vice-versa.

We give one more definition of correctness, for the case of monitor services. The intuition is the same, but because monitor services live outside the addressable memory of the machines, it's statement needs to be adapted a bit.

Definition 4.22 (Monitor Service Correctness). *Correctness for monitor services, states that for the symbolic memory (smem), the symbolic pc ($spc@t_{pc}$) and the ok bit of the abstract machine, it holds that for all monitor services sc,*

$$\begin{aligned}
&smem[spc] = \varnothing \implies \\
&get_service\ spc = (ti, f) \implies \\
&ok = true \iff \\
&(\forall src \in id,\ t_{pc} = Code\ src \implies \\
&\quad \exists dst \in id, \\
&\qquad ti = Code\ dst \land (src, dst) \in \mathcal{CFG})
\end{aligned}$$

The simulation relation ($\sim$) is defined as the conjunction of definitions 4.17 to 4.22 and the invariants 4.6 to 4.8.

Proving 1-backward simulation for normal steps

Proving a 1-backward simulation for normal steps is relatively straight-forward, mostly thanks to the fact that the symbolic machine abstracts away many details of the concrete machine that would make the proofs more tedious. Additionally we do not have to provide such proofs for any monitor service as we did not use any. Therefore we will only have to reason about the small set of instructions that the symbolic and the abstract machine share.

We start with some helpful lemmas about registers and memory updates. These lemmas serve as the basis for proving simulation for instructions that change the registers or the memory. The corresponding Coq definitions and proofs can be found. (**TODO:** cite appendix)

Lemma 4.23 (Registers Update Backward Simulation)**.** *For all symbolic register sets (sreg,sreg'), abstract register sets (areg), registers (r), words (v,v'),*

$$\begin{aligned}
&areg \sim_{regs} sreg \implies \\
&sreg[r] = v@Data \implies \\
&sreg[r \leftarrow v'@Data] = sreg' \implies \\
&\quad \exists areg', \\
&\qquad areg[r \leftarrow v'] = areg' \wedge \\
&\qquad areg' \sim_{regs} sreg'
\end{aligned}$$

Lemma 4.24 (Memory Update Backward Simulation)**.** *For all symbolic memories (smem,smem'), abstract data memories (amem) and words (addr,v,v'),*

$$\begin{aligned}
&amem \sim_{dmem} smem \implies \\
&smem[addr] = v@Data \implies \\
&smem[addr \leftarrow v'@Data] = smem' \implies \\
&\quad \exists amem', \\
&\qquad amem[addr \leftarrow v'] = amem' \wedge \\
&\qquad amem' \sim_{dmem} smem'
\end{aligned}$$

With these definitions and lemmas we are able to prove 1-backward simulation for normal steps between the Symbolic and the Abstract machine as defined by definition 4.15, where the low-level machine is the Symbolic machine and the high-level machine is the Abstract machine.

Theorem 4.25 (1-Backward Simulation Symbolic-Abstract)**.** *Definition 4.15 holds for the Symbolic (low-level) and the Abstract (high-level) machines.*

Proving 1-backward simulation for attacker steps

The same definition as 4.15 of 1-backward simulation is used for the attacker, with the sole difference being that steps now refer to attacker steps.

Definition 4.26 (1-Backward Simulation Attacker)**.** *A low-level machine simulates a high-level machine with respect to a simulation relation $\sim$ between low-level and high-level machine states, if $s_1^H \sim s_1^L$ and $s_1^L \rightarrow_a^L s_2^L$ implies that there exists s_2^H such that, $s_2^H \sim s_2^L$ and $s_1^H \rightarrow_a^H s_2^H$.*

We prove that 1-backward simulation for attacker steps hold, by first showing how we can construct attacker steps at the abstract level from symbolic attacker steps and then showing that this way of building attacker steps preserves the simulation relation ($\sim$).

A step of the symbolic attacker, as mandated by the semantics of the attacker model, can only change the memory and register contents tagged *Data*, formally $mem \rightarrow_a^S mem'$ and $reg \rightarrow_a^S reg'$.

Intuitively, we can construct *areg* by *mapping* a function on the set of registers, that changes a symbolic atom to a word by removing it's tag.

```
Definition untag_atom (a : atom (word t) cfi_tag) := common.val a.
```

Listing 4.7: Untag symbolic atom function

We can trivially prove that the abstract attacker can take a step by *mapping* untag_atom over a symbolic register set. This is trivial because the attacker can arbitrarily change all registers.

Lemma 4.27 (Abstract attacker registers)**.**

$$\begin{aligned}&sreg \rightarrow_a^S sreg' \implies \\ &areg \rightarrow_a^A map\ untag\_atom\ sreg'\end{aligned}$$

However, we still need to prove that the simulation relation between the two machines does not break when attacker steps are taken. We can proof that simulation of registers is preserved by attacker steps. The proof proceeds by using the correctness theorem for the map function.

Theorem 4.28 (Map Correctness instance)**.**

$$(map\ untag\_atom\ sreg')[r] = option\_map\ untag\_atom\ (sreg'[r])$$

where option_map is defined as

```
Definition option_map f x :=
  match x with
  | Some y ⇒ Some (f y)
  | None ⇒ None
  end.
```

Listing 4.8: Option Map function

Lemma 4.29 (Attacker preserves register simulation)**.** *For all abstract register sets (areg) and symbolic register sets (sreg, sreg'),*

$$\begin{aligned}&areg \sim_{regs} reg \implies \\ &sreg \rightarrow_a^S sreg' \implies \\ &map\ untag\_atom\ sreg' \sim_{regs} sreg'\end{aligned}$$

In order to complete the proof of 1-backward simulation for attacker steps, we also need to construct an abstract memory and to show that the $\sim_{mem}$ relation is preserved by attacker steps. Due to the fact that the abstract machine has split data and instruction memories, in order to follow the same methodology as with registers, we will need to split the symbolic memory. We achieve this, using a filter function.

Firstly we proof that attacker steps do not break simulation of instruction memories. Intuitively this is trivial, as the symbolic attacker can only change memory contents tagged *Data*.

Lemma 4.30 (Attacker preserves instruction memory simulation)**.** *For all abstract instruction memories (imem) and symbolic memories (smem, smem'),*

$$\begin{aligned} &imem \sim_{imem} smem \implies \\ &smem \rightarrow_a^S smem' \implies \\ &imem \sim_{imem} smem' \end{aligned}$$

Constructing a data memory is more complicated than in the previous cases. Our approach, uses the filter function to create a subset of the symbolic memory that only contains atoms tagged *Data* and then applies the same methodology with registers, mapping the *untag_atom* function over this subset to obtain an abstract data memory.

```
Definition is_data (a : atom (word t) cfi_tag) :=
  match common.tag a with
    | DATA ⇒ true
    | INSTR _ ⇒ false
  end.
```

Listing 4.9: Function that checks if atom is tagged *Data*

Again we can prove a few helpful lemmas that ease the final proof.

Lemma 4.31 (Attacker preserves data memory simulation)**.** *For all abstract data memories (dmem) and symbolic memories (smem, smem'),*

$$\begin{aligned} &dmem \sim_{dmem} smem \implies \\ &smem \rightarrow_a^S smem' \implies \\ &map\ untag\_atom\ (filter\ is\_data\ sreg' \sim_{dmem} dmem' \end{aligned}$$

The proof of lemma 4.31 is slightly more complex than the one for registers, as we now have to invoke the filter correctness theorem as well.

Theorem 4.32 (Filter Correctness instance)**.**

$$(filter\ is\_data\ smem')[addr] = option\_filter\ is\_data\ (smem'[addr])$$

where option_map is defined as

In all cases, we have to show that the domains of the abstract memories and registers are also preserved. We include here the corresponding lemma for the data memory. It's proof was again more complicated, due to the fact that we had to split the symbolic memory.

```
Definition option_filter f x :=
  match x with
  | Some x0 ⇒ if f x0 then Some x0 else None
  | None ⇒ None
  end.
```

Listing 4.10: Option Filter function

Lemma 4.33 (Attacker preserves data memory domains)**.** *For all abstract data memories (dmem, dmem') and symbolic memories (smem, smem'),*

$$\begin{aligned}
&dmem \sim_{dmem} smem \implies \\
&smem \to_a^S smem' \implies \\
&dmem' \sim_{dmem} smem' \implies \\
&\mathcal{D}(dmem) = \mathcal{D}(dmem')
\end{aligned}$$

Likewise with normal steps, we can now prove a 1-backward simulation for attacker steps as defined by definition 4.26.

Theorem 4.34 (1-Backward Simulation Symbolic-Abstract for Attacker)**.** *Definition 4.26 holds for the Symbolic (low-level) and the Abstract (high-level) machines when the two machines are related by $\sim$.*

Proving 1-forward simulation for normal steps

The 1-forward simulation proof between the abstract and the symbolic machine is similar to the 1-backward simulation proof. Again, we take the same approach and prove some auxiliary lemmas about memory and registers updates.

Lemma 4.35 (Registers Update Forward Simulation)**.** *For all abstract register sets (areg,areg'), symbolic register sets (sreg), registers (r) and words (v'),*

$$\begin{aligned}
&areg \sim_{regs} sreg \implies \\
&areg[r \leftarrow v'] = areg' \implies \\
&\quad \exists sreg', \\
&\qquad sreg[r \leftarrow v'@Data] = sreg' \wedge \\
&\qquad areg' \sim_{regs} sreg'
\end{aligned}$$

Lemma 4.36 (Memory Update Forward Simulation)**.** *For all abstract data memories (dmem,dmem'), symbolic memories (smem) and words (addr,v'),*

$$\begin{aligned}
&dmem \sim_{dmem} smem \implies \\
&dmem[addr \leftarrow v'] = dmem' \implies \\
&\quad \exists smem', \\
&\qquad smem[addr \leftarrow v'@Data] = smem' \wedge \\
&\qquad dmem' \sim_{dmem} smem'
\end{aligned}$$

Lemma 4.37 (Outside Memory)**.** *For all abstract data memories (dmem), abstract instruction memories (imem), symbolic memories (smem) and words (addr),*

$$\begin{aligned} & dmem \sim_{dmem} smem \implies \\ & imem \sim_{imem} smem \implies \\ & imem[addr] = \varnothing \implies \\ & dmem[addr] = \varnothing \implies \\ & smem[addr] = \varnothing \end{aligned}$$

For proving forward simulation between the abstract and the symbolic machine it is required that all indirect jumps are tagged with a unique identifier, which we enforce by the invariants 4.9 and 4.10.

Theorem 4.38 (1-Forward Simulation Abstract-Symbolic)**.** *Definition 4.16 holds for the Symbolic (low-level) and the Abstract (high-level) machines.*

4.5 The Concrete Machine

Assuming the existence of correct code that implements the *CFI* monitor, we can utilize the framework of section 2.3 to instantiate the concrete machine and obtain a refinement between the concrete and the symbolic machines, we need to provide the encoding of symbolic tags. For the concrete machine we only considered a 32-bit architecture, but as already mentioned, we could very easily instantiate the concrete machine with 64-bit words with minimal changes to our proofs.

4.5.1 Concrete tags

In order to obtain the concrete tags, we need to wrap the symbolic tags with the monitor self-protection tags (*User*, *Entry*, *Monitor*) and provide an encoding to words of these tags.

We instantiate the *id* type of cfi_id class (listing 4.1) as bit-fields of size 28-bits. That means, that we can uniquely identify up to 2^{28} instructions. Trying to tag more instructions than this, would break the symbolic invariant 4.6, because by the simulation relation between the concrete and symbolic machines, the two machines follow the same tagging scheme for *User* and *Entry* tags.

Defining the conversion functions [1] between *words* and *ids* is straight forward. We make the simply choice, to convert *words* to *ids* only if they are equal or less than the maximum word our 28-bit *ids* can fit. Note that this does not mean we reduce the addressable space to 28-bits. You can use addresses higher than 2^{28} to place contents tagged as *Data* or *Monitor* or even *Code* $\perp$ but not instructions with an identifier.

The conversion from *ids* to *words* is trivial by expanding the id to 32-bit words by adding zeros to the high bits.

(NG: Do the above make sense to you?)

Finally we prove the two properties required by cfi_id, *id_to_wordK* and *word_to_idK*. We can now instantiate cfi_id with 28-bit sized *ids*.

When using identifiers of 28-bits, we can encode the symbolic tags using 30 bits, with an encoding like the one in table 4.1, where the two least-significant bits are used to

[1] Numbers in the Coq definitions are off by one (e.g., 27 means 28), for reasons relating to the underlying words library (**TODO:** cite library)

```
Definition id_size := Word.int 27.
Definition id := [eqType of id_size].
Definition bound : word t :=
  Word.repr ((Word.max_unsigned 27) + 1)%Z. (*29 bits*)

Definition word_to_id (w : word t) : option id_size :=
  if (Word.ltu w bound) then Some (Word.castu w) else None.

Definition id_to_word (x : id) : word t :=
  Word.castu x.
```

Listing 4.11: Coq definitions of conversion functions for ids and words

```
Instance ids : cfi_id := {|
 id := id;
 word_to_id := word_to_id;
 id_to_word := id_to_word
|}.
Proof.
  - by apply id_to_wordK.
  - by apply word_to_idK.
Defined.
```

Listing 4.12: cfi_id instance with 28-bit sized *ids*

distinguish between *Data*, *Code* $\perp$ and *Code id*, and the 28 higher-bits are the *id* in the last case and zero otherwise.

Symbolic Tag	Encoding
Data	0
Code $\perp$	1
Code id	$4x + 2$

Table 4.1: Encoding of Symbolic Tags

Having an encoding into 30-bits of symbolic tags, we can use the 2-bits left, to wrap the symbolic tags with the monitor self-protection tags. We use the two least-significant bits to distinguish between *User* (01), *Entry* (10) and *Monitor* (00). Only the *User* and *Entry* wrap around symbolic tags. The policy monitor does not use symbolic tags and the corresponding tag *Monitor* does not need to wrap around them. Thus the encoding of the *Monitor* tag has all its bits set to zero.

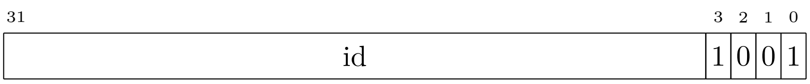

Figure 4.11: Encoding of an instruction with a unique identifier id

With the above encoding, we can easily define a *decode* function and prove that the *decode* function is the *left inverse* of the *encode* function ($decode(encode\ t) = t$) and *right* *inverse* for all elements in the domain of *decode* ($decode\ w = t \implies encode\ t = w$).

4.5.2 Concrete-Symbolic backward refinement

We can now instantiate the backward refinement between the concrete and the symbolic machine that is provided by the micro-policies framework [12]. For the concrete to symbolic backward refinement we no longer get a 1-backward simulation, due to the fact that the steps the concrete policy monitor takes are not matched by any steps of the symbolic machine. For user mode steps (i.e., when the tag of the *pc* is *User*) the framework does provide a proof of 1-backward simulation as defined by definition 4.15, with respect to a simulation relation ($\sim_U$), where the low-level machine is now the concrete machine and the high-level machine is the symbolic machine.

For *Monitor* steps a weaker simulation relation ($\sim_M$) is used. Eventually we obtain a $\{0,1\}$-backward simulation between the concrete and the symbolic machine.

Definition 4.39 (Weak simulation relation for Monitor steps)**.** *A concrete state s^C is in weak simulation with a symbolic state s^S ($s^S \sim_M s^C$), if the tag of the pc of state s^C is Monitor and there exists a concrete user state s_0^C such that $s^S \sim_U s_0^C$ and there is an execution trace $s_0^C \to_n \ldots \to_n s^C$ formed only by monitor steps (all states have Monitor tag on the pc).*

We visualize the above definition with the following diagram:

$$\begin{tikzcd}
& s^S \arrow[dl, squiggly, dash, "U"'] \arrow[d, squiggly, dash, "M"'] \arrow[dr, squiggly, dash, "M"] & \\
s_0^C \arrow[r, "UM"'] & s_1^C \arrow[r, "M^*"'] & s^C
\end{tikzcd}$$

We define the simulation relation $\sim_{CS}$ between the concrete and symbolic machines inductively.

$$\frac{s^S \sim_U s^C}{s^S \sim_{CS} s^C} \qquad\qquad \frac{s^S \sim_M s^C}{s^S \sim_{CS} s^C}$$

Figure 4.12: Concrete-Symbolic simulation relation

Theorem 4.40 ($\{0,1\}$-Backward simulation between Concrete and Symbolic Machines)**.** *For all concrete states s_1^C, s_2^C and symbolic states s_1^S such that, $s_1^S \sim_{CS} s_1^C$ and $s_1^C \to_n s_2^C$ it holds that $s_1^S \sim_{CS} s_2^C$ or there exists s_2^S such that $s_1^S \to_n s_2^S$ and $s_2^S \sim_U s_2^C$.*

Using the 1-backward simulation between the symbolic and abstract machines (theorem 4.25) and the $\{0,1\}$-backward simulation between the concrete and the symbolic machine (theorem 4.40), we can obtain our first result, which is the backward refinement between the concrete machine running a policy monitor that enforces *CFI* and the abstract machine with respect to a refinement relation ($\sim_{CA}$) between concrete and abstract states. We define $\sim_{CA}$ in terms of the simulation relation between the concrete and the symbolic machine ($\sim_{CS}$) and the simulation relation between the symbolic and the abstract machine ($\sim_{SA}$).

$$\frac{s^S \sim_{CS} s^C \qquad s^A \sim_{SA} s^S}{s^A \sim_{CA} s^C}$$

Figure 4.13: Refinement relation between Concrete and Abstract machines

Theorem 4.41 (Concrete-Abstract backward refinement)**.** *For all abstract machine states (s_1^A), concrete machine states (s_1^C, s_2^C), if $s_1^A \sim_{CA} s_1^C$ and $s_1^C \to_n^* s_2^C$ and s_2^C is in user mode, then there exists an abstract machine state s_2^A such that $s_1^A \to_n^* s_2^A$ and $s_2^A \sim_{CA} s_2^C$.*

In order to obtain our second result, which is a proof that the property stated by definition 4.2 holds for the concrete machine, we will need to make the concrete machine an instance of the 4.4, by defining all it's parameters, similar to what we did for the abstract and symbolic machines.

4.5.3 Attacker model

The attacker model for the concrete machine, models an attacker that can tamper with the machine only when it's in user mode. The capabilities of the concrete attacker when the machine is in user mode, directly matches the capabilities of the symbolic attacker, which means that the attacker can only change the values of atoms that have a *User* tag. This prevents the attacker from changing monitor data in memory or registers, as well as the tags.

$$\frac{w_1@ut_1 \to_a^S w_2@ut_2}{w_1@\mathit{User}\ ut_1 \to_a^C w_2@\mathit{User}\ ut_2} \quad (\textsc{AttackUser})$$

Figure 4.14: Concrete attacker capabilities on atoms

$$\frac{mem \to_a^C mem' \qquad reg \to_a^C reg'}{(mem, reg, cache, pc@\mathit{User}\ ut, epc) \to_a^C (mem', reg', cache, pc@\mathit{User}\ ut, epc)}$$

Figure 4.15: Attacker model for the Concrete machine

4.5.4 Concrete-Symbolic 1-backward simulation for Attacker

For attacker steps we can prove a 1-backward simulation, instantiating definition 4.15, with the concrete machine as the low level machine, the symbolic machine as the high machine and using $\sim_U$ as a simulation relation.

In order to prove the simulation, we apply the same technique as in the case of Symbolic-Abstract backward simulation for attacker steps, constructing attacker steps at the symbolic level from attacker steps in the concrete level and additionally showing that the way we build the steps preserve the simulation relation.

We can construct a symbolic memory and a symbolic set of registers from their concrete counterparts by filtering all non-user data of the concrete memory and registers and then decoding all the concrete tags to symbolic ones. We can achieve this using the filter and map functions as seen in section 4.4.6.

We can now prove lemmas 4.42 and 4.43 the two lemmas that will allows us to easily proof the 1-backward simulation for attacker steps.

```
Definition is_user (x : atom (word mt) (word mt)) :=
  rules.word_lift (fun t ⇒ rules.is_user t) (common.tag x).
```

Listing 4.13: Function that returns true if atom has a *User* tag

```
Definition coerce (x : atom (word mt) (word mt))
   : atom (word mt) (cfi_tag) :=
   match rules.decode (common.tag x) with
   | Some (rules.USER tg) ⇒ (common.val x)@tg
   | _ ⇒ (common.val x)@DATA (*this is unreachable in our case*)
   end.
```

Listing 4.14: Function that converts a concrete atom to a symbolic one

Lemma 4.42 (Concrete-Symbolic attacker registers 1-backward simulation)**.** *For all symbolic register sets (sreg) and concrete register sets (creg, creg'),*

$$\begin{aligned}&sreg \sim_{regs} creg \implies\\ &creg \to^{C}_{a} creg' \implies\\ &sreg \to^{S}_{a} map\ coerce\ (filter\ is\_user\ creg')\end{aligned}$$

Lemma 4.43 (Concrete-Symbolic attacker memory 1-backward simulation)**.** *For all symbolic memories (smem) and concrete memories (cmem, cmem'),*

$$\begin{aligned}&smem \sim_{mem} cmem \implies\\ &cmem \to^{C}_{a} cmem' \implies\\ &map\ coerce\ (filter\ is\_user\ cmem') \sim_{mem} cmem'\\ &smem \to^{S}_{a} map\ coerce\ (filter\ is\_user\ cmem')\end{aligned}$$

We additionally have to prove that attacker steps preserve some low-level invariants of the concrete machine that are required by the framework we use, but the proofs are mostly trivial as the invariants regard pieces of state the attacker cannot tamper with e.g., monitor data.

Theorem 4.44 (1-Backward Simulation Concrete-Symbolic for Attacker)**.** *Definition 4.26 holds for the Concrete (low-level) and the Symbolic (high-level) machines when the two machines are related by $\sim_U$.*

4.5.5 Allowed control-flows for the Concrete Machine

Once again we construct a function that decides the validity of all control-flows $\mathcal{SUCC}^{\mathcal{C}}_{\mathcal{CFG}}$, this time for the concrete machine. $\mathcal{SUCC}^{\mathcal{C}}_{\mathcal{CFG}}$ allows all flows involving monitor mode and only restricts the control-flow for user mode execution.

4.5.6 Initial states of the Concrete Machine

For the concrete machine, we require that it's initial states matches the initial states of the symbolic machine under the simulation relation $\sim_U$. This ensures that concrete initial

$$\frac{\mathit{in\_monitor}\ s_1 \mid\mid \mathit{in\_monitor}\ s_2}{(s_1, s_2) \in \mathcal{SUCC}^{\mathcal{C}}_{\mathcal{CFG}}} \quad (\textsc{MonitorFlows})$$

$$\frac{\begin{array}{c} \mathit{mem}[pc] = i@\mathit{User}\ (\mathit{Code}\ \mathit{src}) \qquad \mathit{decode}\ i \in \{\mathit{Jal}\ r, \mathit{Jump}\ r\} \\ \mathit{mem}[pc'] = i'@\mathit{User}\ (\mathit{Code}\ \mathit{dst}) \\ t_{pc} = \mathit{User}\ \mathit{ut} \qquad t'_{pc} = \mathit{User}\ \mathit{ut}' \qquad (\mathit{src}, \mathit{dst}) \in \mathcal{CFG} \end{array}}{((\mathit{mem}, \mathit{reg}, \mathit{cache}, pc@t_{pc}, \mathit{epc}), (\mathit{mem}, \mathit{reg}, \mathit{cache}, pc'@t'_{pc}, \mathit{epc})) \in \mathcal{SUCC}^{\mathcal{C}}_{\mathcal{CFG}}} \quad (\textsc{IndirectFlows})$$

$$\frac{\begin{array}{c} \mathit{mem}[pc] = i@\mathit{User}\ (\mathit{Code}\ \mathit{src}) \qquad \mathit{decode}\ i \in \{\mathit{Jal}\ r, \mathit{Jump}\ r\} \\ \mathit{mem}[pc'] = i'@\mathit{Entry}\ (\mathit{Code}\ \mathit{dst}) \\ t_{pc} = \mathit{User}\ \mathit{ut} \qquad t'_{pc} = \mathit{User}\ \mathit{ut}' \\ \mathit{decode}\ i' = \mathit{Nop} \qquad (\mathit{src}, \mathit{dst}) \in \mathcal{CFG} \end{array}}{((\mathit{mem}, \mathit{reg}, \mathit{cache}, pc@t_{pc}, \mathit{epc}), (\mathit{mem}, \mathit{reg}, \mathit{cache}, pc'@t'_{pc}, \mathit{epc})) \in \mathcal{SUCC}^{\mathcal{C}}_{\mathcal{CFG}}} \quad (\textsc{IndirectFlows2})$$

$$\frac{\begin{array}{c} \mathit{mem}[pc] = i@\mathit{User}\ (\mathit{Code}\ \_) \qquad \mathit{decode}\ i = \mathit{Bnz}\ r\ \mathit{imm} \\ t_{pc} = \mathit{User}\ \mathit{ut} \qquad t'_{pc} = \mathit{User}\ \mathit{ut}' \\ (pc' = pc + 1) \vee (pc' = pc + \mathit{imm}) \end{array}}{((\mathit{mem}, \mathit{reg}, \mathit{cache}, pc@t_{pc}, \mathit{epc}), (\mathit{mem}, \mathit{reg}, \mathit{cache}, pc'@t'_{pc}, \mathit{epc})) \in \mathcal{SUCC}^{\mathcal{C}}_{\mathcal{CFG}}} \quad (\textsc{ConditionalFlows})$$

$$\frac{\begin{array}{c} \mathit{mem}[pc] = i@\mathit{User}\ (\mathit{Code}\ \_) \qquad \mathit{decode}\ i \notin \{\mathit{Jal}\ r, \mathit{Jump}\ r, \mathit{Bnz}\ r\ \mathit{imm}, \varnothing\} \\ t_{pc} = \mathit{User}\ \mathit{ut} \qquad t'_{pc} = \mathit{User}\ \mathit{ut}' \\ (pc' = pc + 1) \vee (pc' = pc + \mathit{imm}) \end{array}}{((\mathit{mem}, \mathit{reg}, \mathit{cache}, pc@t_{pc}, \mathit{epc}), (\mathit{mem}', \mathit{reg}', \mathit{cache}, pc'@t'_{pc}, \mathit{epc})) \in \mathcal{SUCC}^{\mathcal{C}}_{\mathcal{CFG}}} \quad (\textsc{NormalFlows})$$

Figure 4.16: Allowed control-flows for instructions of the concrete machine

states satisfy both the invariants we enforced on symbolic initial states and any low-level invariants enforced by $\sim_U$.

Definition 4.45 (Concrete Initial States)**.** *A concrete state s^C is an initial state if there exists a symbolic state s^S such that initial^S s^S and $s^S \sim_U s^C$.*

4.5.7 Stopping predicate for the Concrete Machine

The stopping predicate for the concrete machine is more complex than the one for the symbolic or the abstract machine, due to the monitor steps. In particular, on the next step after a violation the machine will enter monitor mode to determine whether the step is allowed or not. The miss handler will take an arbitrary number of steps to determine that execution should be disallowed because a violation of the policy occurred. This is modeled by disallowing the concrete machine to return to user mode. However, note that it could be the case that the machine cannot step at all after a control-flow violation, for example if the *pc* is outside the memory of the machine.

In addition to the above, there may be attacker steps. These can only come immediately after the violating step and before the machine enters monitor mode. Attacker is not allowed to take steps during monitor mode and as mentioned above the machine will not return to user mode.

We can summarize the conditions that hold for an execution trace to be stopping.

Definition 4.46 (Concrete Stopping Predicate)**.**

- *There is an optional prefix of attacker steps ($\to^C_a$) and all states in the prefix are user states.*

- *There is an optional suffix of monitor steps ($\rightarrow_n$) and all states in the suffix are monitor steps.*

(**TODO:** Diagram showing stopping?)

4.6 Generic Preservation Theorem

In this section, we discuss the preservation theorem that we used, along with the simulation proofs of sections 4.4.6 and 4.5.2, in order to prove *CFI* (definition 4.2) for the concrete machine.

The statement of the theorem is parameterized by two machines that are instances of cfi_machine (listing 4.4). Moreover, we require that a $\{0, 1\}$-backward simulation between the two machines, holds for normal steps and a 1-backward simulation for attacker steps. The $\{0, 1\}$ simulation for normal steps, stems from the fact that the steps of the concrete machine in monitor mode are not matched by any steps on the symbolic (or the abstract) level. We generalize this, by a notion of *checked steps* on the steps of the low-level machine. Intuitively we only check for control-flow violations when a checked step is taken.

We require a strong 1-backward simulation for checked steps and a $\{0, 1\}$-backward simulation for the rest.

$$\begin{tikzcd} s_1^L \arrow[r, "n"] \arrow[d, squiggly, dash] & s_2^L \arrow[d, squiggly, dash, dashed] \\ s_1^H \arrow[r, dashed, "n"] & s_2^H \end{tikzcd}$$

Figure 4.17: 1-backward simulation

$$\begin{tikzcd} s_0^L \arrow[r, "n"] \arrow[d, squiggly, dash] & s_1^L \arrow[dl, squiggly, dash, dashed] \\ s^H & \end{tikzcd}$$

Figure 4.18: 0-backward simulation

$$\begin{tikzcd} s_1^L \arrow[r, "a"] \arrow[d, squiggly, dash] & s_2^L \arrow[d, squiggly, dash, dashed] \\ s_1^H \arrow[r, dashed, "a"] & s_2^H \end{tikzcd}$$

Figure 4.19: 1-backward simulation for attacker

The class machine_refinement captures the above specifications.

From these relations on single steps, we can build a refinement relation on execution traces. We define this trace refinement relation inductively and we say that two traces are in refinement if they are built this way.

In listing 4.16 we distinguish between three separate cases, from which we may build two traces that are in refinement.

Zero Step. If the low-level machine takes an unchecked step, $s_1^L \rightarrow_n s_2^L$ and for a high-level machine state s^H it holds that $s^H \sim s_1^L$ and $s^H \sim s_2^L$ then if traces

```
Variable amachine : cfi_machine. (*high-level machine*)
Variable cmachine : cfi_machine. (*low-level machine*)

(* General notion of refinement between two machines*)
Class machine_refinement
  (amachine : cfi_machine) (cmachine : cfi_machine) := {
    refine_state : (@state amachine) → (@state cmachine) → Prop;

    check : (@state cmachine) → (@state cmachine) → bool;

    backward_refinement_normal :
      ∀ ast cst cst'
        (REF: refine_state ast cst)
        (STEP: step cst cst'),
        (check cst cst' = true →
        ∃ ast', step ast ast' ∧ refine_state ast' cst')
        ∧ (check cst cst' = false →
             refine_state ast cst' ∨
             ∃ ast', step ast ast' ∧ refine_state ast' cst');

    backward_refinement_attacker :
      ∀ ast cst cst'
        (REF: refine_state ast cst)
        (STEPA: step_a cst cst'),
        ∃ ast', step_a ast ast' ∧ refine_state ast' cst'
}.
```

Listing 4.15: Interface of machine_refinement

$s^H :: tr^H$ and $s_2^L :: tr^L$ are in refinement, the traces $s^H :: tr^H$ and $s_1^L :: s_2^L :: tr^L$ are also in refinement.

Normal Step. If the low-level machine takes a checked step, $s_1^L \to_n s_2^L$ and the high-level machine takes a step $s_1^H \to_n s_2^H$ and $s_1^H \sim s_1^L$ and $s_2^H \sim s_2^L$ then if traces $s_2^H :: tr^H$ and $s_2^L :: tr^L$ are in refinement, the traces $s_1^H :: s_2^H :: tr^H$ and $s_1^L :: s_2^L :: tr^L$ are also in refinement.

Attacker Step. If the low-level machine takes an attacker step $s_1^L \to_a^L s_2^L$ and additionally $s_1^L \not\to_n s^L 2$ and the high-level machine takes an attacker step $s_1^H \to_a^H s_2^H$ and $s_1^H \sim s_1^L$ and $s_2^H \sim s_2^L$ then if traces $s_2^H :: tr^H$ and $s_2^L :: tr^L$ are in refinement, the traces $s_1^H :: s_2^H :: tr^H$ and $s_1^L :: s_2^L :: tr^L$ are also in refinement.

Notice in the last case that we require that the step from s_1^L to s_2^L cannot be a normal step. Intuitively this is used to enforce that if a step is in the intersection of the normal and attacker step relations, one should prefer the normal step to build the trace.

We can now extend the backward refinements of listing 4.15 to whole execution traces which we relate with refine_traces.

Theorem 4.47 (Trace Backward Refinement)**.** *If $s_1^H \sim s_1^L$ and $s_1^L \to \ldots \to s_n^L$ where $n > 0$ then, there exists an execution trace such that $s_1^H \to \ldots s_m^H$ where $m \geq 0$ and additionally the traces $s_1^H \ldots s_m^H$ and $s_1^L \ldots s_n^L$ are in refinement.*

```
Inductive refine_traces :
  list (@state amachine) → list (@state cmachine) → Prop :=
| TRNil : ∀ ast cst,
          refine_state ast cst →
          refine_traces [ast] [cst]
| TRNormal0 : ∀ ast cst cst' axs cxs,
    step cst cst' →
    check cst cst' = false →
    refine_state ast cst →
    refine_state ast cst' →
    refine_traces (ast :: axs) (cst' :: cxs) →
    refine_traces (ast :: axs) (cst :: cst' :: cxs)
| TRNormal1 : ∀ ast ast' cst cst' axs cxs,
    step cst cst' →
    step ast ast' →
    refine_state ast cst →
    refine_state ast' cst' →
    refine_traces (ast' :: axs) (cst' :: cxs) →
    refine_traces (ast :: ast' :: axs) (cst :: cst' :: cxs)
| TRAttacker : ∀ ast ast' cst cst' axs cxs,
     step cst cst' →
    step_a cst cst' →
    step_a ast ast' →
    refine_state ast cst →
    refine_state ast' cst' →
    refine_traces (ast' :: axs) (cst' :: cxs) →
    refine_traces (ast :: ast' :: axs) (cst :: cst' :: cxs).
```

Listing 4.16: Inductive definition of trace refinement

In order to prove that *CFI* is preserved by backwards refinement, we make some additional assumptions about the two machines.

Definition 4.48 (Step Decidability). *The normal step relation of the low-level machine is decidable.*

Definition 4.49 (Initial States). *For all initial states of the low-level machine, there exists an initial state of the high-level machine so that the two are in simulation.*

Definition 4.50 (Unchecked Steps). *All unchecked steps are allowed according to the* $\mathcal{SUCC}_{\mathcal{CFG}}$ *function.*

Definition 4.51 (Successor Functions). *For the states* $s_1^H, s_2^H, s_1^L, s_2^L$ *such that* $s_1^H \sim s_1^L$ *and* $s_2^H \sim s_2^L$ *and* $s_1^H \to_n s_2^H$ *and there is a checked step* $s_1^L \to_n s_2^L$*, the functions* $\mathcal{SUCC}_{\mathcal{CFG}}^{\mathcal{H}}$ *and* $\mathcal{SUCC}_{\mathcal{CFG}}^{\mathcal{L}}$ *agree on their results.*

Definition 4.52 (No Attacker Steps on Violation). *For a high-level machine step* $s_1^H \to_n s_2^H$ *such that* $(s_1^H, s_2^H) \notin \mathcal{SUCC}_{\mathcal{CFG}}^{\mathcal{H}}$ *it holds that* $s_1^H \not\to_a^H s_2^H$*.*

Definition 4.53 (Stopping Predicates). *If there is a step in the high-level machine* $s_1^H \to_n s_2^H$ *such that* $(s_1^H, s_2^H) \notin \mathcal{SUCC}_{\mathcal{CFG}}^{\mathcal{H}}$ *and if the traces* $s_2^H :: tr^H$ *and* $s_2^L :: tr^L$ *are in refinement and* $s_2^H :: tr^H$ *is a stopping trace for the high-level machine then* $s_2^L :: tr^L$ *is a stopping trace for the low-level machine.*

```
Class machine_refinement_specs := {

  step_classic : ∀ (cst cst': @state cmachine),
    (step cst cst') ∨ ( step cst cst');

  initial_refine : ∀ (cst : @state cmachine),
    initial cst →
    ∃ (ast : @state amachine), initial ast ∧ refine_state ast cst;

  cfg_nocheck : ∀ asi csi csj,
    refine_state asi csi →
    step csi csj →
    check csi csj = false →
    succ csi csj = true;

  cfg_equiv : ∀ (asi asj : @state amachine) csi csj,
    refine_state asi csi →
    refine_state asj csj →
    step asi asj →
    check csi csj = true →
    step csi csj →
    succ csi csj = succ asi asj;

  av_no_attacker : ∀ (asi asj : @state amachine) csi,
    refine_state asi csi →
    succ asi asj = false →
    step asi asj →
      step_a asi asj;

  as_implies_cs : ∀ axs cxs asi asj csi csj,
    check csi csj = true →
    succ asi asj = false →
    step asi asj →
    refine_state asi csi →
    refine_traces (asj :: axs) (csj :: cxs) →
    stopping (asj :: axs) →
    stopping (csj :: cxs)
}.
```

Listing 4.17: Assumptions under which CFI preservation holds

Again we create an interface for these assumptions using type-classes.

Under these assumptions we can now obtain a preliminary result about our *CFI* definitions.

Theorem 4.54 (Trace Refinement preserves Trace Has CFI)**.** *For all execution traces $s_0^H \to \dots s_n^H$ and $s_0^L \to \dots s_m^L$ that are in refinement (listing 4.16), if the high-level trace $s_0^H \to \dots s_n^H$ has* CFI *(definition 4.1) then the low-level trace $s_0^L \to \dots s_m^L$ also has* CFI.

Proof. The proof proceeds by induction on the trace refinement.

- **Base Case** In this case the two traces are singletons and the low-level trace vacuously has *CFI*.

- **Zero Step** By the induction hypothesis the trace $s_0^L \to \ldots \to s_m^L$ has *CFI*. In order to prove that the augmented with an unchecked step $s^L \to_n s_0^L$ trace ($s^L \to_n s_0^L \to \ldots \to s_m^L$) also has *CFI* we need to prove that $(s^L, s_0^L) \in \mathcal{SUCC}^{\mathcal{L}}_{\mathcal{CFG}}$. We know that $s^L \sim s_0^H$ (by construction of the trace refinement relation), our goal is immediately provable by the assumption on unchecked steps (definition 4.50).

- **One Step** Again by the induction hypothesis we easily obtain that $s_0^L \to \ldots \to s_m^L$ has *CFI*, therefore it's left to prove that for the checked step $s^L \to_n s_0^L$ at the beginning of the trace it holds that $(s^L, s_0^L) \in \mathcal{SUCC}^{\mathcal{L}}_{\mathcal{CFG}}$. We know by the trace refinement that $s^H \sim s^L$, $s_0^H \sim s_0^L$ and that $s^H \to_n s_0^H$.
  - If the step $s^L \to_n s_0^L$ is checked, then by the assumption on the $\mathcal{SUCC}_{\mathcal{CFG}}$ functions (definition 4.51) $(s^H, s_0^H) \in \mathcal{SUCC}^{\mathcal{H}}_{\mathcal{CFG}} \iff (s^L, s_0^L) \in \mathcal{SUCC}^{\mathcal{L}}_{\mathcal{CFG}}$. But by the second premise we know that the trace $s^H \to s_0^H \to \ldots \to s_n^H$ has *CFI* and therefore $(s^H, s_0^H) \in \mathcal{SUCC}^{\mathcal{H}}_{\mathcal{CFG}}$. Thus we conclude that $(s^L, s_0^L) \in \mathcal{SUCC}^{\mathcal{L}}_{\mathcal{CFG}}$.
  - If the step $s^L \to_n s_0^L$ is unchecked, again it is immediately provable by definition 4.50.

- **Attacker Step** By the induction hypothesis we easily obtain that $s_0^L \to \ldots \to s_m^L$ has *CFI*. The step $s^L \to_a s_0^L$ is an attacker step and additionally $s^L \not\to_n s_0^L$ by the trace refinement definition. Therefore it vacuously holds that $(s^L, s_0^L) \in \mathcal{SUCC}^{\mathcal{L}}_{\mathcal{CFG}}$ and the whole trace has *CFI*.

□

We have now proved that the $\{0,1\}$-backward simulation of listing 4.15 preserves the *CFI* property of execution traces. We will use this preliminary result to prove that this backward simulation also preserves the *CFI* property of machines described by definition 4.2.

We start with an auxiliary lemma that states that if there is a trace refinement between a high-level trace and a low-level trace and then we split the high-level trace to sub-traces in a certain way, then there exists low-level sub-traces such that trace refinement holds between the sub-traces. Naturally, with definition 4.2 in mind, we choose to split the high-level trace at the step that violates the control-flow.

(NG: this became kind of heavy with all the subtrace lines. Could lose the whole trace (trH and trL) lines as a first step and use two boxes that enclose the refined subtraces.)

Lemma 4.55 (Refine Traces Split)**.** *If the traces $s_0^H \to \ldots \to s_n^H$ (referred to as tr^H) and $s_0^L \to \ldots \to s_m^L$ (referred to as tr^L) are in refinement and there is a splitting of the high-level trace such that $tr^H = tr^H_{hd} \mathbin{++} s^H_{u1} :: s^H_{u2} \mathbin{++} tr^H_{tl}$ and additionally $s^H_{u1} \to_n s^H_{u2}$ and $(s^H_{u1}, s^H_{u2}) \notin \mathcal{SUCC}^{\mathcal{H}}_{\mathcal{CFG}}$, then there exists a splitting of the low-level trace such that $tr^L = tr^L_{hd} \mathbin{++} s^L_{u1} :: s^L_{u2} \mathbin{++} tr^L_{tl}$, the traces $tr^H_{hd} \mathbin{++} [s^H_{u1}]$ and $tr^L_{hd} \mathbin{++} [s^L_{u1}]$ are in refinement, the traces $s^H_{u2} :: tr^H_{tl}$ and $s^L_{u2} :: tr^L_{tl}$ are in refinement, $s^H_{u1} \sim s^L_{u1}$, $s^H_{u2} \sim s^L_{u2}$ and $s^L_{u1} \to_n s^L_{u2}$.*

Combining theorem 4.54 and lemma 4.55 we can now prove that $\{0,1\}$-backward simulation preserves *CFI* as defined by definition 4.2 under certain assumptions.

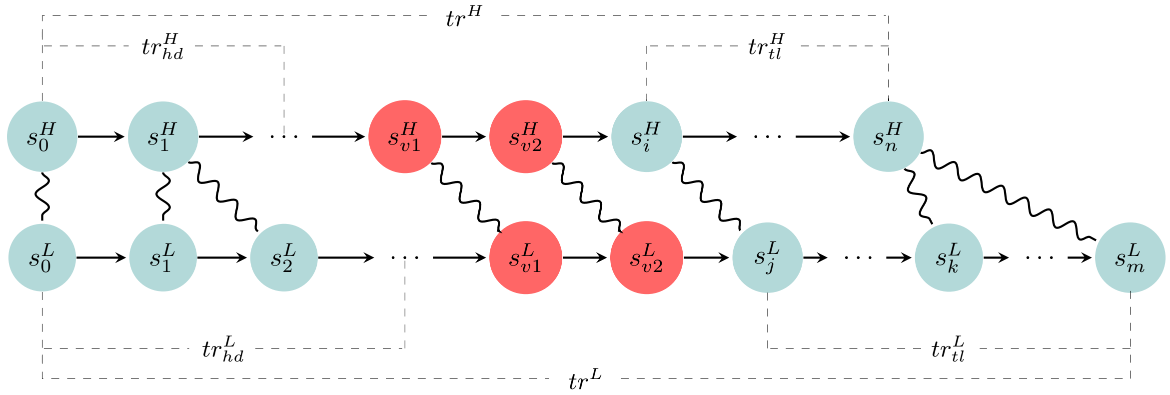

Figure 4.20: Splitting trace refinement on violation

Theorem 4.56 (CFI Preservation)**.** *If a low-level machine simulates (as defined by listing 4.15) a high-level machine and the high-level machine has* CFI *then the low-level machine also has* CFI *under the assumptions 4.17.*

(**TODO:** informal proof?)

4.6.1 CFI proof for the Symbolic Machine

To prove *CFI* for the symbolic machine, we instantiate the preservation theorem of section 4.6 with the abstract machine as the high-level machine and the symbolic machine as the low-level machine. For the symbolic machines all steps are considered checked. Instantiating class 4.15 with the symbolic and abstract machines is trivial by using the 1-backward simulation for both normal and attacker steps from section 4.4.6.

The only thing left to prove before being able to use the *CFI* preservation theorem is that the assumptions 4.17 hold when instantiated with the symbolic and abstract machines.

Lifting preservation assumptions for Symbolic-Abstract machines

Lemma 4.57 (Symbolic Step Decidable)**.** *Definition 4.48 holds for the Symbolic machine.*

Proof. Decidability for Symbolic normal steps In order to decide whether $s^S_0 \to_n s^S_1$ or $s^S_0 \not\to_n s^S_1$ we resort to the computational interpretation of the step relation. If $step^S_n s^S_0 = s^S$ then if $s^S_1 = s^S$ we obtain $s^S_0 \to_n s^S_1$ otherwise we conclude that $s^S_0 \not\to_n s^S_1$. □

Lemma 4.58 (Symbolic-Abstract Initial States)**.** *Definition 4.49 holds for Symbolic-Abstract machines.*

Proof. To prove that there exists an abstract state that is initial and simulates an initial symbolic state, we use a technique similar to the one we used when building attacker steps in sections 4.4.6 and 4.5.4. We build the abstract registers set by mapping the untag atom function (listing 4.7) over the symbolic registers set and the instruction and data memories by first using the filter function on the symbolic memory to remove all data tagged *Data* (respectively *Code*) and then mapping the untag atom function. The pc is the same as the one for the symbolic state and the *ok* bit is set to true. Proving simulation between the two states is trivial. □

Lemma 4.59 (Unchecked steps of Symbolic machine)**.** *Definition 4.50 holds for the Symbolic machine.*

Proof. Vacuously true in the case the low-level machine is the symbolic machine as all steps are checked. □

Lemma 4.60 (Successor Functions)**.** *Definition 4.51 holds for the Symbolic-Abstract machines.*

Proof. The proof is mostly straight-forward by case analysis on the opcode of the instruction. □

Lemma 4.61 (No Abstract Attacker Steps on Violation)**.** *Definition 4.52 holds for the Abstract machine.*

Proof. The proof proceeds by contradiction. Suppose $s_1^H \to_a^H s_2^H$ then by lemma 4.4 we obtain that $(s_1^H, s_2^H) \in \mathcal{SUCC}_{\mathcal{CFG}}^{\mathcal{H}}$. But we know by the second premise that $(s_1^H, s_2^H) \notin \mathcal{SUCC}_{\mathcal{CFG}}^{\mathcal{H}}$, therefore we reached a contradiction and it must be that $s_1^H \not\to_a^H s_2^H$. □

Lemma 4.62 (Abstract stopping implies Symbolic stopping)**.** *Definition 4.53 holds for the Symbolic-Abstract machines.*

Proof. According to definition 4.14 we have to prove that all steps in the symbolic trace are attacker steps and all states in the symbolic trace are stuck with respect to normal steps. The proof proceeds by induction on the trace refinement.

- **Base Case** In this case the two traces are singletons. It vacuously holds that all steps of the symbolic machine are attacker steps. To show that the state forming the singleton trace is stuck we resort to a contradiction.

  Suppose that the state (s^S) is not stuck, therefore there exists some state s_c^S such that $s^S \to_n s_c^S$. Additionally we know by trace refinement that $s^H \sim s^S$. By 1-backward simulation (checked step) we conclude that there exists some state s_c^H such that $s^H \to_n s_c^H$. But the abstract trace is stopping and by definition 4.3 all states in it are stuck with respect to normal steps. Therefore we reached a contradiction, thus it must be that s^S is a stuck state.

- **Zero Step** In this case there is an unchecked step in the trace. But all steps of the symbolic machine are checked, so we immediately reach a contradiction.

- **One Step** In this case, the trace refinement relation gives us that there is a normal step at the abstract level, which contradicts with the fact that the abstract machine is stuck with respect to normal steps by definition 4.3.

- **Attacker Step** The two traces are now augmented by an attacker step at their beginning ($s^H \to_a s_0^H \to_a \ldots \to_a s_n^H$ and $s^L \to_a s_0^L \to \ldots \to s_m^L$). By the induction hypothesis we easily obtain that the tail of the symbolic trace is stopping. We need to prove that new step is an attacker step and that the new state is stuck with respect to normal steps. The former is trivial as we are in the case an attacker step is taken. To show that s^L is stuck with respect to normal steps, we once again resort to a contradiction.

  Suppose that there exists some s_c^L such that $s^L \to_n s_c^L$. We additionally know that $s^H \sim s^L$ by the trace refinement relation. By backward simulation we get that there exists some state s^H_c such that $s^H \to_n s^H_c$. But we know that the abstract trace is stopping, therefore all states in it are stuck with respect to normal steps, thus we reached a contradiction.

□

We can now utilize the preservation theorem for the first time and obtain that the Symbolic machine has *CFI*.

Theorem 4.63 (Symbolic CFI). *The Symbolic machine has the* CFI *property stated by definition 4.2.*

Proof. Follows immediately by theorem 4.56. □

4.6.2 CFI proof for the Concrete Machine

We will now leverage the preservation theorem for a second time, to transfer the *CFI* property from the symbolic to the concrete machine.

For this we instantiate the preservation theorem with symbolic machine as the high-level machine and the concrete as the low-level machine. A step is considered checked only if both states forming the step are in user mode. Instantiating the machine_refinement class (4.15) in this case is not as straight-forward as before due to the fact that we have unchecked steps as well, but we can still take advantage of the $\{0, 1\}$-backward simulation (theorem 4.40) provided by the micro-policies framework. We use $\sim_{CS}$ as the refinement relation.

Theorem 4.64 (Backward Refinement Normal). *Backward refinement holds for the concrete-symbolic instance of 4.15.*

Proof. For a normal step $(s^C_1 \to_n s^C_2)$ of the concrete machine and for some symbolic state s^S_1 such that $s^S_1 \sim_{CS} s^C_1$, we distinguish between three cases.

1. s^C_1 and s^C_2 are user states. In this case the step is checked and by the second case of theorem 4.40 we obtain the 1-backward simulation required by 4.15.

2. s^C_1 is a user state and s^C_2 is a monitor state. In this case the step is unchecked and the symbolic machine does not take a step. We prove that the simulation relation (sim_{CS}) is preserved by proving the weak simulation relation. The state s^C_2 is in monitor mode and there exists a concrete state (s^C_1) such that $s^S_1 \sim_U s^C_1$ and additionally $s^C_1 \to_n s^C_2$ therefore by 4.39 we obtain that $s^S_1 \sim_M s^C_2$ and consequently $s^S_1 \sim_{CS} s^C_2$.

3. s^C_1 is a monitor state. In this case the step is unchecked and theorem 4.40 proves our goal.

□

For simulation of attacker steps the theorem 4.44 applies directly.

We now have to show that the assumptions 4.48 to 4.53 hold for this instantiation of the preservation theorem.

Lifting preservation assumptions for Concrete-Symbolic machines

Lemma 4.65 (Concrete Step Decidable)**.** *Definition 4.48 holds for the Concrete machine.*

Proof. We apply the same technique, we used for Symbolic steps in lemma 4.57. □

Lemma 4.66 (Concrete-Symbolic Initial States)**.** *Definition 4.49 holds for Concrete-Symbolic machines.*

Proof. The proof of this is trivial by the way we defined initial states of the concrete machine in definition 4.45. □

Lemma 4.67 (Unchecked steps of Concrete machine)**.** *Definition 4.50 holds for the Concrete machine.*

Proof. An unchecked step $s_1^C \to_n s_2^C$ implies that either *in_monitor* s_1^C or *in_monitor* s_2^C. By rule *MonitorFlows* of 4.16 $(s_1^C, s_2^C) \in \mathcal{SUCC}^{\mathcal{C}}_{\mathcal{CFG}}$. □

Lemma 4.68 (Successor Functions)**.** *Definition 4.51 holds for the Concrete-Symbolic machines.*

Proof. The proof proceeds by case analysis on the type of instruction. *(NG: this one is not too hard, long and ugly to list..)* □

Lemma 4.69 (No Symbolic Attacker Steps on Violation)**.** *Definition 4.52 holds for the Abstract machine.*

Proof. We sketch the intuition behind the proof. Suppose $s_1^S \to_n s_2^S$. For all instructions other than Jump and Jal there is a clear contradiction, as $(s_1^S, s_2^S) \notin \mathcal{SUCC}^{\mathcal{S}}_{\mathcal{CFG}}$ implies that the *pc* of the new state is not the one mandated by the operational semantics which cannot be because $s_1^S \to_n s_2^S$. *(NG: crappy at explaining the obvious, well done)*

In the case of a jump or jal instruction, it must be that the instruction is a self-loop, because $s_1^S \to_a^S s_2^S$ implies that $s_1^S.pc = s_2^S.pc$. If the tag of the instruction at *pc* is *Code* x where $x \in id$, we distinguish two cases:

1. If the tag on the *pc* of s_1^S is different than *Code* x, according to the semantics of normal steps for Jump/Jal instructions the tag on the instruction executed is propagated to the tag on *pc* of s_2^S, therefore the tag on the *pc* of s_2^S should be *Code* x. But by the semantics of the symbolic attacker, the tag on the *pc* of s_1^S and s_2^S remains the same. Contradiction.

2. If the tag on the pc of s_1^s is *Code* x, by $(s_1^S, s_2^S) \notin SUCCmS$ we know that $(x, x) \notin \mathcal{CFG}$. Therefore by the transfer function (4.6) $s_1^S \not\to_n s_2^S$. Contradiction.

□

Lemma 4.70 (Symbolic stopping implies Concrete stopping)**.** *Definition 4.53 holds for the Concrete-Symbolic machines.*

Proof. According to definition 4.46 we have to prove that the trace is made up of some optional attacker steps at first and then by some optional monitor steps. By 4.53, we know that for some s_1^S, s_2^S it holds that there is step step $s_1^S \to_n s_2^S$ and additionally $(s_1^S, s_2^S) \notin SUCCmS$. The proof proceeds by inversion on the construction of trace refinement.

- **Base Case** In this case both the symbolic and the concrete traces are singletons made up of s_2^S and s_2^C respectively. The stopping condition holds vacuously since the trace is a singleton.
- **Zero Step** In this case an unchecked step $s_2^C \to_n s_3^C$ is taken and the trace is of the form $s_2^C \to_n s_3^C \to \ldots \to s_n^C$. The prefix of the trace is made up of one state that is in user mode (s_2^C) and it vacuously holds that it is made up of attacker steps. For the suffix of the trace $s_3^C \to \ldots \to s_n^C$ we distinguish between two cases.
  - In case the mvector for s_2^S exists, as there was a violation, intuitively the transfer function will not allow any steps from this state. At the concrete level, the policy monitor will take a number of monitor steps and eventually halt the machine.
  - In case the mvector for s_2^S, since $s_2^C \to_n s_3^C$ it must be that the step $s_1^S \to_n s_2^S$ tried to access monitor data (e.g., jumped to monitor code). Again the policy monitor takes a number of monitor steps and eventually halts the machine.
- **One step** In this case the trace refinement relation gives us that $s_2^S \to_n s_3^S$ for some s_3^S. But we know that s_2^S is in the stopping trace of the symbolic machine and all states in that trace are stuck with respect to normal steps, therefore we reach a contradiction.
- **Attacker step** In this case an attacker step $s_2^C \to_a^C s_3^C$ is taken and the trace is of the form $s_2^C \to_a^C s_3^C \to \ldots \to s_n^C$. We distinguish between two sub-cases.
  - The whole trace $s_2^C \to \ldots \to s_n^C$ is made of attacker steps and there is suffix of monitor steps in it.
  - At some point in the trace there is a normal step $s_i^C \to_n s_j^C$. Intuitively because attacker steps cannot change tags we know that $s_i^C \to_n s_j^C$ will be a step from user to monitor mode. The monitor will detect the violation and take a series of steps before eventually halting the machine.

  (NG: this is super simplified, but the details are so gory.. in general the proof of stopping for the concrete machine is very watered down. Perhaps I will return to explain a few things about forming mvectors and the difference between symbolic mvectors and concrete ones)

□

We now invoke the preservation theorem for a second time, to transfer the *CFI* property from the Symbolic to the Concrete machine.

Theorem 4.71 (Concrete CFI)**.** *The Concrete machine has the* CFI *property stated by definition 4.2.*

Proof. Follows immediately by theorem 4.56. □

Chapter 5

Conclusion

5.1 Future Work

There are many directions still left to explore before we can consider our work done. Some of them include writing the *CFI* monitor code and verifying it, increasing precision by enforcing call-stack protection, scaling to more complex architectures and looking for ways to enforce *CFI*-like policies on self modifying programs.

5.1.1 Writing and Verifying Monitor Code

In this thesis, we described the *CFI* micro-policy and reasoned about its security properties by using a high-level specification of the policy monitor, expressed in terms of a *transfer* function written in Coq. In reality, when we leveraged the micro-policies framework we *assumed* the existence of machine code implementing the *CFI* policy monitor and its correctness as specified by the high-level *transfer* function.

Although we have not written the machine code for the policy monitor - and consequently not verified it - we consider the existence of correct code implementing the policy monitor as a realistic assumption. Azevedo *et al.* provided code for a dynamic sealing micro-policy in [**?**], although they did not verify it. Furthermore in [5], that can be considered as a predecessor to the micro-policies project, machine code for an IFC monitor was obtained using structured code generators and a verified DSL compiler. *(NG: shrink references to IFC and sealing? I want them as a witness to my claims about the possibility of writing/verifying the cfi monitor code)*

Arguably the code for a dynamic sealing monitor is simpler than the code for a *CFI* monitor, but even an efficient implementation of a *CFI* monitor would probably resemble a compiled switch statement/match expression, for which there are plenty of resources on efficient compilation strategies. One could even write the *CFI* policy monitor by hand, however we decided not to attempt this, as it seemed that without verifying it, there was little added value considering the amount of effort required. Furthermore, in order to be able to at least test the correctness of the implementation, we would be required to provide machine code for programs and to also compute their control-flow graph, which would be tedious and time consuming without the appropriate tools.

As noted in [**?**] it would make more sense to go through the effort of writing and verifying machine code for a more realistic architecture. In a standard RISC architecture setting (e.g., ARM) we could write the policy monitor in a higher-level language (even C) and use a (verified) compiler (e.g., CompCert [19]) to obtain the machine code. Furthermore, we could leverage existing verification frameworks, either for low-level code [9, 18] or for the high-level language we used to code the policy monitor (e.g., [4] in the case of C code), in order to verify the correctness of our implementation.

(NG: probably rephrase that and somehow unify the citation to VST and compcert?)

5.1.2 Call-Stack Protection/ XFI

Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13(1), 2009.

[3] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), November 1996.

[4] A. W. Appel. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *Proceedings of the 41st Symposium on Principles of Programming Languages (POPL)*, POPL, pages 165–178. ACM, Jan. 2014.

[6] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 353–362, New York, NY, USA, 2011. ACM.

[7] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proc. ACM CCS*, pages 27–38, Oct. 2008.

[8] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming without Returns. In A. Keromytis and V. Shmatikov, editors, *Proceedings of CCS 2010*, pages 559–72. ACM Press, Oct. 2010.

[9] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 391–402. ACM, 2013.

[10] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[11] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. 2014.

[12] A. A. de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: A framework for verified, hardware-assisted security monitors. Under Review, July, July 2014.

[13] U. Dhawan, C. Hriţcu, N. Vasilakis, S. Chiricescu, J. M. Smith, B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. Under Review, August, Aug. 2014.

[14] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon. PUMP – A Programmable Unit for Metadata Processing. In *Proceedings of the 3rd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '14, New York, NY, USA, June 2014. ACM.

[15] Ú. Erlingsson. *The inlined reference monitor approach to security policy enforcement.* PhD thesis, Cornell University, Jan. 2004.

[16] U. Erlingsson. Foundations of security analysis and design iv. chapter Low-level Software Security: Attacks and Defenses, pages 92–134. Springer-Verlag, Berlin, Heidelberg, 2007.

[17] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

[18] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. In *40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 301–314. ACM, 2013.

[19] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[20] J. McDonald. Bugtraq: Defeating Solaris/SPARC Non-Executable Stack Protection, Mar. 1999.

[21] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 395–404. ACM, 2012.

[22] T. Newsham. Bugtraq: Re: Smashing the Stack: prevention?, Apr. 1997.

[23] B. Niu and G. Tan. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 58. ACM, 2014.

[24] PaX Team. Pax address space layout randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`.

[25] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[26] Solar Designer. Bugtraq: Getting around non-executable stack (and fix), Aug. 1997.

[27] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62. IEEE Computer Society, 2013.

[28] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy*, 2013.

[29] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. ARMor: fully verified software fault isolation. In *11th International Conference on Embedded Software*, pages 289–298. ACM, 2011.

Appendix A

Stuff

A.1 Control-Flow Integrity Micro-Policy

We begin with a micro-policy targeting control-flow hijacking attacks, in which an attacker exploits a low-level vulnerability (e.g. a buffer or integer overflow) to gain full control of a target program [**?**, 27, 3, **?**, **?**, **?**, **?**, **?**, **?**]. As a first line of defense, we can use tags to make code non-writable (NWC) and data non-executable (NXD), preventing the injection and execution of an attacker payload. This useful defense appears in various forms in existing systems. However, it does not prevent code-reuse attacks [22, 26, 20, 25, 8, 7, **?**, 17] such as return- or jump-oriented programming [25, 8], where the attacker chains together existing code snippets ("gadgets") to induce arbitrary malicious behavior. We therefore use tags to enforce fine-grained *control-flow integrity (CFI)* [2, 29, 28, 11, 23, 29, 11] on top of basic NWC and NXD protection. This ensures that all indirect control flows (computed jumps) adhere to a fixed control flow graph (CFG).

We use tags to distinguish between code and data. Tags on memory and the PC are drawn from the set $Data \mid Code\ addr \mid Code\ \bot$ (registers are always tagged *Data*). To simplify the CFG conformance checks, instructions that are the source or target of indirect control flows are tagged with *Code addr*, where *addr* is the address of that instruction in memory. For example, a *Jump* instruction stored at address 500 is tagged *Code* 500. All other instructions are tagged $Code\ \bot$. *(AAA: Actually, we can't use the instruction's address on the tag if we are to have the same number of bits on words and tags. Maybe change to "id"?)*

We write transfer functions as a collection of *symbolic rules* [12, 14]. (The PUMP hardware uses a lower-level *concrete rule* format, described in **??**.) Each symbolic rule has the form "$opcode : (PC, CI, OP_1, OP_2, OP_3) \rightarrow (PC', R')$," which says that the rule matches on the given *opcode* together with the metadata tags on the program counter (PC), the current instruction (CI), and on up to three operands (OP_1 to OP_3). If the rule applies, the right-hand side determines how to update the tags on the PC (PC') and on the result of the operation (R'). We write "$-$" to indicate input or output fields that are ignored ("wildcard"). All instructions that are not explicitly allowed by the symbolic rules are disallowed. *(AAA: We should choose only one of $-$ or _ for our wildcard and use it consistently (cf. the "Store" rule below))*

The CFI transfer function enforces that only memory locations tagged *Data* can be modified (NWC) and only instructions fetched from locations tagged *Code* can be executed (NXD). The symbolic rule for the *Store* instruction illustrates both these points:

$$Store : (Data, Code\ \_, -, -, Data) \rightarrow (Data, -)$$

It requires the fetched *Store* instruction to be tagged *Code* and the written location to be tagged *Data*. This rule only applies when the PC is also tagged *Data*, which is the case when the *Store* instruction was reached by direct control flow (not a computed jump). The rule preserves the *Data* tag on the PC, since *Store* is not a computed jump. Performing a computed jump (e.g., using *Jal*, a jump-and-link instruction) requires that the current instruction be tagged *Code src* for some address *src*.

$$Jal : (Data, Code\ src, -, -, -) \rightarrow (Code\ src, -)$$

This rule copies *Code src* to the PC tag to indicate that a jump from *src* has just occurred. Only on the next instruction do we have enough information about the destination in the tags to check that the jump is indeed allowed by the CFG. For this we add a second rule for *Store*, dealing with the case where it is the target of a jump and thus the PC is tagged *Code src*.

$$\frac{(src, tgt) \in CFG}{Store : (Code\ src, Code\ tgt, -, -, Data) \rightarrow (Data, -)}$$

*(AAA: Maybe we could discuss here a little bit why we verify the jump on the next instruction, as opposed to when the jump is performed. This might get some people confused, since this is not very natural and fundamentally driven by our current design of the PUMP. Even Nick wanted to know if we couldn't do it differently.)*The premise of this rule ensures that the source and target of the just-performed jump are allowed by the CFG. We add a similar rule for each instruction, including jumps (since the target of a computed jump can itself be another computed jump):

$$\frac{(src, tgt\text{-}src) \in CFG}{Jal : (Code\ src, Code\ tgt\text{-}src, -, -, -) \rightarrow (Code\ tgt\text{-}src, -)}$$

This micro-policy enforces fine-grained CFI [23, 17, 11], not coarse-grained approximations [2, 28] that are potentially vulnerable to attack [17]. Indeed, we recently proved [12] that this micro-policy enforces a variant of the CFI property introduced by Abadi *et al.* [2], ensuring that all indirect control flows adhere to a fixed CFG. Recent simulations of an optimized PUMP architecture [13] show that the CFI policy can be enforced with around 3% average runtime overhead.