Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Control Flow Integrity

Διπλωματική Εργασία

του

Νίκου Γιανναράκη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Σεπτέμβριος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Control Flow Integrity

Διπλωματική Εργασία

του

Νίκου Γιανναράκη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την $11^{η}$ Σεπτεμβρίου, 2014.

....	....	....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Ιώαννης Σμαραγδάκης
Αν. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2014

....
Nick Giannarakis
Διπλωματούχος Ηλεκτρολόγος Μηχανικός
και Μηχανικός Υπολογιστών 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

Keywords

concolic testing, Erlang, software testing, dynamic symbolic execution, SMT solving

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον Κωστή Σαγώνα για την καθοδήγησή του, την υπομονή του και την πίστη του σε εμένα. Σε μια περίεργη περίοδο της ακαδημαϊκής μου πορείας, η εμπιστοσύνη που μου έδειξε αποτέλεσε το σημαντικότερο κίνητρό μου.

Επίσης θα ήθελα να ευχαριστήσω τον Νίκο Παπασπύρου, καταρχάς για την πολύτιμη βοήθεια του, αλλά πολύ περισσότερο για το ότι αποτελεί τον βασικό λόγο για τον οποίο ασχολήθηκα με την Πληροφορική. Η διδασκαλία του και το ήθος του ήταν και συνεχίζουν να είναι έμπνευση για εμένα.

Ευχαριστώ την οικογένειά μου που με έκανε τον άνθρωπο που είμαι σήμερα.

Τέλος, ευχαριστώ την Μαιρούλα μου για την στήριξη της όλα αυτά τα χρόνια. Σε ευχαριστώ που είσαι εσύ και που με κάνεις καλύτερο!

Άγγελος Γιάντσιος

Contents

List of Figures

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 (Feedback: 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. Programming languages have evolved, from low-level unmanaged languages, to high-level languages with features such as strong type systems 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 mitigation techniques (TODO: reference some?) have been proposed, however these are mostly ad-hoc solutions designed to prevent specific known attacks, rather than enforcing a security policy along with a well defined class of attacks that are prevented, thus making it hard to reason about their effectiveness. In fact most of these mitigation techniques can be circumvented by attackers, (TODO: reference) which has lead to a continuous "chase" between attackers and security researchers.

One common attack technique is to deploy new code in the memory of the vulnerable program and then exploit some low-level vulnerability such as a buffer overflow to redirect the control flow to this attacker code. This attack can be stopped by a simple protection scheme known as $W \oplus X$, which enforces that a memory region is either executable or writable but not both. Unfortunately, new attack techniques can easily 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.

The goal of this thesis is to describe and formalize in Coq a novel hardware-assisted implementation of an effective mitigation technique called Control Flow Integrity (CFI). CFI enforces that any execution of a program will respect a statically computed control flow graph (CFG), thus stopping a wide range of attacks that attempt to modify the control flow. As part of the formalization effort, following the work of Abadi *et al.* [1], we provide an attacker model and prove a variant of the CFI property described in [1].

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 basic requirments a security policy must satisfy and puts into context the framework we utilize in order to formalize and enforce a Control-Flow Integrity (CFI) policy. Chapter 3 discusses the current state of research on Control-Flow Integrity and clarifies our goals and contributions to it. Chapter 4 describes in detail the design of a fine-grained CFI policy and how we used the framework from Chapter 2 in order to enfore the policy and formally reason about it's security properties. Chapter 5.. conclusions, future work? Chapter 6.. related work and bibliography? Appendix with code and/or step relations etc.?

Chapter 2

Safety and Security Policies

Currently the hardware provides only a small number of security mechanisms (TODO: name some), 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 A Programmable Unit for Metadata Processing

The Programmable Unit for Metadata Processing (PUMP) architecture [6] allows us to efficiently implement a wide range of security policies [9] by associating metadata to the data being processed (e.g., this is an instruction, this is from the network, this is private), propagating the metadata as instructions are executed and using a rules-based system to check invariants on the metadata in parallel with the main computation. Abstractly, the tag propagation rules form a partial function from a set of input tags to a set of output tags

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

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}".

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.* [6] and in more detail in the follow-up [9] 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 (10% 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. [9] The authors claim that a careful and well-optimized implementation can reduce these numbers, resulting in a 50% energy overhead.

2.2 Micro-policies: A Framework for Verified, Hardware-Assisted Security Monitors

The software components that can be changed to enforce a security policy are collectively called a micro-policy. Unsurprisingly, designing a security policy, reasoning about it's effectiveness against potential attackers and encoding it as a micro-policy can become a complex task. Azevedo *et al.* [5] built a generic framework for defining micro-policies on top of a simple machine modeling a RISC processor augmented with the PUMP hardware (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 monitor code itself. (Feedback: maybe I should mention the word monitor at some point earlier)

The framework offers a high-level machine, called the symbolic machine, that abstracts away from unnecessary implementation details and can be used as an interface to the concrete machine, simplifying the work of the micro-policy designer. Additionally the symbolic machine is used to simplify correctness proofs. To instantiate the symbolic machine, the micro-policy designer needs to provide a set of symbolic tags which will be used to tag the various values of the machine, a transfer function that monitors program execution and determines how tags are propagated in each step and optionally a set of monitor services that are partial functions from machine states to machine states and can be used to control the monitor's behavior dynamically.

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.

2.2.1 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 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 valid step for the concrete machine is also a valid step for the abstract machine. (Feedback: say smth about steps and refinement earlier..)

2.2.2 Basic Machine

All the machines introduced in the original paper by Azevedo *et al.* [5], 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.

(TODO: write down a few rules?)

2.2.3 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.The symbolic machine follows the structure of the basic machine but it's augmented to better match a PUMP architecture. Specifically the symbolic machine is parameterized by the following:

- A set of symbolic tags, used to tag the contents of the memory, the registers and the pc.
- A partial function *transfer*, that on every step checks whether the step is allowed according to opcode of the instruction executed and the tags on it. In the case it's allowed it returns a tag for the new pc and a tag for any resulting data from executing the instruction.
- A partial function *get_service*, mapping addresses to *symbolic monitor services*. In the symbolic machine, monitor services are represented as a tuple of a partial function on machine states and a symbolic tag.
- An internal machine state with an initial value, that can be used by monitor services.

The states of the symbolic machine consists of a memory, registers, a *pc* register and an 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.

We write, in form of inference rules, the stepping relation for the Store and Jump instructions, in order to demonstrate the above mechanism. The complete definition of the stepping relation can be found at (TODO: cite appendix)

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

$$\frac{\begin{array}{c} \mathit{mem}[\mathit{pc}] = i@t_i \qquad \mathit{decode}\ i = \mathit{Jump}\ r \qquad \mathit{reg}[r]{=}w@t_w \\ \mathit{transfer}(\mathit{Jump}, t_{pc}, t_i, t_w, -, -) = (t'_{pc}, -) \end{array}}{(\mathit{mem}, \mathit{reg}, \mathit{pc}@t_{pc}, \mathit{int}) \to (\mathit{mem}, \mathit{reg}, w@t'_{pc}, \mathit{int})} \quad (\textsc{Jump})$$

Figure 2.1: Symbolic stepping relation for Store and Jump

Notice 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.

2.2.4 Concrete Machine

The concrete machine is a model of the basic machine with PUMP hardware, in particular a rules *cache*and a 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 enforced by the monitor self-protection mechanism.

The states of the concrete machine consists of a memory, registers, a *pc* register, an *epc*register a special purpose register that holds the address of the faulting instruction after a cache miss and a cache. The cache works as a key-value store where a key is an *input vector* that contains an instruction opcode, the concrete tag of the current instruction, the concrete 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. Intuitively a concrete 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*). In accordance (Feedback: em this sucks? does this word even exist? think about smth else?) 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, a *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 below:

(TODO: put example rules)

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).

2.2.5 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.

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. Other mitigation techniques such as protecting return addresses on the stack enforce a form of coarse-grained *CFI*.

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) [7] follow that approach), thus some form of *CFI* is required in order to ensure that these checks are not circumvented.

(TODO: Think about title)

3.1 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.* [3] 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.* [11] 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 [8] 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.

3.1.1 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:

- *NXD* is an abbreviation for Non-Executable Data, a security mechanism that disallows execution of data.

- *NWC* stands for Non-Writable Code. 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.

3.2 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 in 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 (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 *CFI* is preserved for every step 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 [10] are far from trivial for the x86.

Machine-checked formal verification efforts include [12], 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 [4] claim partial proofs for a *CFI* enforcement mechanism focused on the kernel of an operating system. Their runtime overhead can also reach 100%.

3.3 Control-Flow Integrity over PUMP

The PUMP hardware allows us to avoid taking the difficult decision between performance and security. As shown in [9], we can enforce a *fine-grained CFI* policy with an average overhead of 8%. (TODO: Is this number right?)

In our design, we take the standard approach and claim *CFI* under *NXD* and *NWC*. We considered designs that lifted these assumptions but we rejected them, for the time being, as there did not seem to be any considerable advantage i.e.,compatibility with self-modifying programs, JIT compilers, etc. Allowing the code of the program to change, would in practice require for the CFG to change as well, which unless done in a controlled, "safe" way, would invalidate the enforcement of *CFI*. However, we do not have to rely on special hardware or software to enforce *NXD* and *NWC*. We can achieve this easily and efficiently by creating a separate micro-policy.

3.3.1 Enforcing Non-Writable Code & Non-Executable Data

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

$$
\begin{aligned}
&(Store, \_, \_, \_, \_, Instr\ ) \rightarrow \varnothing \\
&(Store, \_, \_, \_, \_, Data) \rightarrow (\_, Data)
\end{aligned}
$$

Figure 3.1: Rules enforcing *NWC*

The _in the vectors, represent *don't care* values. In the context of the input vector their behavior is the same as *don't care* values in match expressions in ML languages. In the context of the output vector it just captures the intuition that we will not really use the result tags, so anything could be returned as a result tag (i.e., *Data* or we can copy-through tags from the input vector). Informally the above rules reads as "If the current opcode is Store and the content of the memory location we are trying to write is tagged *Instr* then the memory write is not allowed. Otherwise if it is tagged *Data* then the write is permitted and the new value will also be tagged *Data*."

We can enforce *NXD* in a similar fashion

$$
\begin{aligned}
&(-, -, Data, -, -, -) \rightarrow \varnothing \\
&(-, -, Instr\ , -, -, -) \rightarrow (-, -)
\end{aligned}
$$

Figure 3.2: Rules enforcing *NXD*

Informally the above rules reads as "If the tag on the current instruction is *Data* then execution is not allowed. Otherwise if it is *Instr*then execution is allowed".

(Feedback: Used _and -, I think the second one looks better, opinions?)
(TODO: Perhaps explain what each tag means for each opcode earlier – or maybe just in appendix?)

(Feedback: These tuple-vectors make it hard for people not familiar with them to remember what each field is, any better ways to represent them?)

3.3.2 Enforcing Control-Flow Integrity

Coarse-grained Control-Flow Integrity

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*.

$$(Jump/Jal, -, Marked, -, -, -) \rightarrow (Marked, -) \tag{1}$$
$$(-, Marked, Marked, -, -, -) \rightarrow (Unmarked, -) \tag{2}$$
$$(-, Marked, Unmarked, -, -, -) \rightarrow \varnothing \tag{3}$$

Figure 3.3: Rules enforcing coarse-grained *CFI*

(TODO: align all elements of the rules above)

Rule 1 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 2 applies when the tag on the *pc* is set to *Marked* and corresponds to a legal destination and rule 3 corresponds to an illegal destination (i.e., one that is tagged *Unmarked*) and is not allowed.

We do not further study this coarse-grained approach as we consider it ineffective since attacks against it has already been demonstrated in [8]. Instead we are going to focus on implementing and formalizing a fine-grained *CFI* micro-policy.

Fine-grained Control-Flow Integrity

The micro-policy we implemented and studied is a composition of a fine-grained *CFI* micro-policy and the *NWC*, *NXD* micro-policies explained above.

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 $\mathcal{T} = \{Data, Instr\ id, Instr\ \perp\}$ where *id* is a unique identifier (i.e., used to tag the contents of only one location in the memory). Adapting the rules from 3.3.1, we shall use *Data* to tag all contents in memory that are considered non-executable data, *Instr id* to tag all contents in memory that are considered executable instructions and are sources or targets of indirect control flows and $Instr\ \perp$ 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 *Instr*tag.

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.

$$\frac{opcode \in \{Jump, Jal\} \qquad (src, dst) \in \mathcal{CFG}}{(opcode, \mathit{Instr}\ src, \mathit{Instr}\ dst, -, -, -) \rightarrow (\mathit{Instr}\ dst, -)} \ (\textsc{Flow/Check})$$

$$\frac{opcode \in \{Jump, Jal\}}{(opcode, \mathit{Data}, \mathit{Instr}\ dst, -, -, -) \rightarrow (\mathit{Instr}\ dst, -)} \ (\textsc{Flow/NoCheck})$$

$$\frac{(src, dst) \in \mathcal{CFG}}{(Store, \mathit{Instr}\ src, \mathit{Instr}\ dst, -, -, \mathit{Data}) \rightarrow (\mathit{Data}, \mathit{Data})} \ (\textsc{Store/Check})$$

$$\frac{ti \in \{\mathit{Instr}\ dst, \mathit{Instr}\ \bot\}}{(Store, \mathit{Data}, ti, -, -, \mathit{Data}) \rightarrow (\mathit{Data}, \mathit{Data})} \ (\textsc{Store/NoCheck})$$

$$\frac{opcode \notin \{Jump, Jal, Store\} \qquad (src, dst) \in \mathcal{CFG}}{(opcode, \mathit{Instr}\ src, \mathit{Instr}\ dst, -, -, -) \rightarrow (\mathit{Data}, -)} \ (\textsc{Rest/Check})$$

$$\frac{opcode \notin \{Jump, Jal, Store\} ti \in \{\mathit{Instr}\ dst, \mathit{Instr}\ \bot\}}{(opcode, \mathit{Data}, ti, -, -, -) \rightarrow (\mathit{Data}, -)} \ (\textsc{Rest/NoCheck})$$

Figure 3.4: 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 *Instr 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 2.2 we proved that the concrete machine instantiated with a *CFI* micro-policy like the one described in 3.3.2 *simulates* an abstract machine that has *CFI* by construction.

Additionaly, 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 [1], but adapted to the setting of our machines.

4.1 Expressing the control-flow through tags

Our approach for enforcing *CFI*, as explained in 3.3.2, 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 := {
  $id : eqType$;

  $word\_to\_id : word\ t \to option\ id$;
  $id\_to\_word : id \to word\ t$;

  $id\_to\_wordK : \forall\ x,\ word\_to\_id\ (id\_to\_word\ x) = Some\ x$;
  $word\_to\_idK : \forall\ w\ x,\ word\_to\_id\ w = Some\ x \to id\_to\_word\ x = w$

}.

Figure 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 insignificant 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 *machine_types* 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 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.

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

4.2 A Theorem About Control-Flow Integrity

Our formalization includes a definition of *CFI*, similar to the one found in [1], 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.

As mentioned in 3.3.2, 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.

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
}.

Figure 4.3: Interface of a cfi_machine

(**Question:** Unsure about the caption on the above figure)

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

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

(**Question:** 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 [1], 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 2 (CFI)**.** We say that the machine $(State, initial, \to_n, \to_a, cfg, stopping)$ 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 $s_0 \to \ldots \to s_n$, either

1. The whole trace has *CFI* according to 1, or else
2. There is some i such that $s_i \to_n s_{i+1}$, and $(s_i, s_{i+1}) \notin \mathcal{SUCC}_{\mathcal{CFG}}$, where the sub-traces $s_0 \to \ldots \to s_i$ and $s_{i+1} \to \ldots \to s_n$ both have CFI and the sub-trace $s_{i+1} \to \ldots \to s_n$ is stopping.

4.3 The Abstract Machine

The abstract machine is based on the basic machine explained in 2.2.2, 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 4.4: 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, a function $\mathcal{J}$, which represents the set of allowed jumps, checks 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.

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

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

Figure 4.5: Step rule for Jump and Jal instruction of abstract machine

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 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. Th As with all other instructions, execution of the monitor service is only allowed if the *ok* bit is set to true.

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

Figure 4.6: Step rule for monitor services of abstract machine

4.3.2 Proving *CFI* for the abstract machine

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 4.7: Attacker model for the abstract machine

Legal control-flows for the abstract machine

Assuming a function $\mathcal{CFG}$ that represents the set of allowed (indirect) jumps, we can construct a function $\mathcal{SUCC}_{\mathcal{CFG}}$ for the abstract machine that represents the set of allowed control-flows for all instructions.

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

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

$$\frac{im[pc] = i \qquad decode\ i = Bnz\ r\ imm \quad (pc' = pc + 1) \lor (pc' = pc + imm)}{((im, dm, reg, pc, ok), (im, dm', reg', pc', ok)) \in \mathcal{SUCC}_{\mathcal{CFG}}}\ (\textsc{ConditionalFlows})$$

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

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

Figure 4.8: Legal control-flows for instructions of the abstract machine

Notice that a monitor service is allowed to return anywhere. As we mentioned before, monitor services, execute in a protected environment and we assume them to be secure.

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:

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

2. All steps in the trace are attacker steps ($\rightarrow_a$)

Abstract machine as a *CFI* 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 4.3, with the abstract machine and then prove the following theorem.

Theorem 1 (Abstract CFI)**.** The abstract machine has the *CFI* property defined by 2.

Proof. The proof procceeds by induction on the execution trace. □

(**Question:** Should I write some proofs informally?)

4.4 The Symbolic Machine

The symbolic machine was described in 2.2.3. Unlike the abstract machine, the symbolic machine has one memory and distinction between data and executable instructions is made through tags, in a similar fashion to what was shown in **??** and **??**. We instantiate the symbolic machine, according to the aforementioned sections, with a set of tags $\mathcal{T} = \{Data, Instr\ id, Instr\ \bot\}$, where id now is drawn from the class of identifiers **??**. We do not have any monitor services and we do not need any internal state for this micro-policy therefore, only the transfer function is left to implement.

4.4.1 Transfer Function

We implement the *transfer* function based on the rules found in 3.3.2, using Gallina to define a function mapping input vectors (mvector) to output vectors (rvector).

```
Definition cfi_handler umvec :=
  match umvec with
  | mkMVec JUMP (INSTR (Some n)) (INSTR (Some m)) _
  | mkMVec JAL (INSTR (Some n)) (INSTR (Some m)) _ ⇒
    if cfg n m then Some (mkRVec (INSTR (Some m)) DATA)
    else None
  | mkMVec JUMP DATA (INSTR (Some n)) _
  | mkMVec JAL DATA (INSTR (Some n)) _ ⇒
    Some (mkRVec (INSTR (Some n)) DATA)
  | mkMVec JUMP DATA (INSTR None) _
  | mkMVec JAL DATA (INSTR None) _ ⇒
    None
  | mkMVec STORE (INSTR (Some n)) (INSTR (Some m)) [_ ; _ ; DATA] ⇒
    if cfg n m then Some (mkRVec DATA DATA) else None
  | mkMVec STORE DATA (INSTR _) [_ ; _ ; DATA] ⇒
    Some (mkRVec DATA DATA)
  | mkMVec STORE _ _ _ ⇒ None
  | mkMVec _ (INSTR (Some n)) (INSTR (Some m)) _ ⇒
    if cfg n m then Some (mkRVec DATA DATA) else None
  | mkMVec _ DATA (INSTR _) _ ⇒
    Some (mkRVec DATA DATA)
  | mkMVec _ _ _ _ ⇒ None
  end.
```

Figure 4.9: Transfer function for symbolic machine in Gallina

Although, the rules in 3.3.2 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 *Instr*. This is expressed by the following relations:

$$\frac{w_1 @ \mathit{Data}}{w_2 @ \mathit{Data}}\ (\textsc{AttackData}) \qquad \frac{w_1 @ \mathit{Instr}\ id}{w_1 @ \mathit{Instr}\ id}\ (\textsc{AttackInstr})$$

Figure 4.10: Attacker capabilities

These attacker relations on symbolic atoms are extended to a relation on the memory and the registers.

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

Figure 4.11: Attacker model for the abstract machine

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] 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.

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

[5] 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.

[6] 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.

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

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

[9] C. Hriţcu, U. Dhawan, N. Vasilakis, S. Chiricescu, J. M. Smith, B. C. Pierce, and A. DeHon. Programming the PUMP: Hardware-assisted micro-policies for security. Under Review, May, May 2014.

[10] 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.

[11] 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.

[12] 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.