A case against indirect jumps for secure programs

Alexandre Gonzalvez
IMT Atlantique, INRIA/TAMIS
alexandre.gonzalvez@imt-atlantique.fr

Ronan Lashermes
INRIA/SED&LHS
ronan.lashermes@inria.fr

ABSTRACT

A desired property of secure programs is control flow integrity (CFI): an attacker must not be able to alter how instructions are chained as specified in the program. Numerous techniques try to achieve this property with various trade-offs. But to achieve fine-grained CFI, one is required to extract a precise control flow graph (CFG), describing how instructions are chained together. Unfortunately it is not achievable in general. In this paper, we propose a way to overcome this impossibility result by restricting the instruction set architecture (ISA) semantics. We show that forbidding indirect jumps unlocks a precise CFG extraction for all acceptable programs. We discuss the implications and limitations of the new semantics and argue for the adoption of restricted ISAs for security-related computation.

KEYWORDS

control flow integrity, control flow graph, instruction set architecture

1 INTRODUCTION

In the wake of the publication of numerous microarchitectural attacks such as Spectre [11], Meltdown [12] and their variants, the need for the secure execution of programs is blatant. Secure execution in this context refers to the additional information security-related properties that must be guaranteed by the execution model.

One such property is the control flow integrity (CFI). Informally, this property should guarantee that an attacker is not able to redirect the control flow, *i.e.* the order of execution of the instructions in the program. But as we will see in section 4, this property is actually quite difficult to properly enforce. Moreover, guaranteeing CFI requires the defense mechanism to obtain the valid control flow in the form of the control flow graph (CFG). The mechanism is then responsible for comparing the actual control flow with the valid paths in the CFG. Unfortunately, in the real computers that we use every day, extracting the CFG from a binary is a problem impossible to do precisely in general as shown in subsection 2.2. It leaves few possibilities for the defender:

(1) Provides an escaping mechanism to the CFI in order to deal with the holes in our knowledge (but therefore creating a weakness).
(2) Enrich the binary with the necessary metadata to complete the CFG. In some cases, it requires annotation from the developer in the source language.
(3) Restrict the valid programs to the ones where the CFG is easily extractable.

If we are interested in providing CFI at the hardware level, we must be able to run securely all the programs fed to the chip: solution 1 is not acceptable. To comply with the solution 2, the metadata must either be filled manually by the developer or derived automatically by the compiler when possible. A consequence is that to be executed on a secure processor enforcing CFI, the sequence of instructions is not self-sufficient anymore. In addition to the instructions, the processor needs the metadata defining a complete CFG. Some solutions such as HAFIX [5] or CHERI [18] propose to enrich the instruction set architecture (ISA) semantic with new instructions, embedding the metadata in the instructions themselves.

Instead, in this paper, we argue for the third choice: the impossibility to extract efficiently the CFG is due to the presence of indirect jumps in the ISA. If we remove the indirect jumps, the binary now contains all the information to extract the CFG without requiring metadata.

The problem caused by indirect jumps is well known, and numerous works try to circumvent it (see [10, 16] for CFG extraction in the presence of indirect jumps). Yet removing indirect jumps has multiple benefits: first it rejects a class of program (the ones that cannot be converted to direct jumps only) that is insecure by nature. Then the CFG extraction becomes trivial, the binary program holds straightforwardly the necessary information for the CFG reconstruction.

A non-goal is to convert automatically programs written in an ISA with indirect jumps to our new ISA proposal without them. Indeed, as we show in subsection 2.2 and section 5, such a conversion is not possible in general: there are some programs that cannot be converted efficiently to an ISA without indirect jumps. But this impossibility result is actually a feature. Our new ISA rejects programs which contain jumps with an impredictable destination: unsecure programs if we want to enforce CFI.

Motivation. In this work, we want to provide an ISA allowing easy and precise CFG extraction for all valid programs. As proven in subsection 2.2, this is not possible if the ISA has indirect jumps. This property would allow to easily enforce CFI but also to forbid some forms of obfuscation: a malware packer would have a hard time to hide its payload. Additionally, it would improve the reach and the power of static analysis tools.

Contribution. To allow easy and precise CFG extraction, we propose to remove indirect jumps in the ISA with two new ISAs proposals.

In this paper, we start by defining our abstract machine in section 2, followed by the definitions of CFG and CFI. In particular we show, in subsection 2.2, that a particular program with an indirect jump can hide the jump destination.

The 3 ISAs with varying semantics to express the control flow are described in section 3. We compare the resulting ISAs with a simple benchmark to evaluate the performances and the ability to extract a precise CFG with basic CFG extraction algorithms. Our ISAs are very simple ones, we do not compare them to existing

ones; instead we evaluate the impact of indirect jumps removal in this simple setup.

A theoretical comparison is also performed on a dispatcher program since it is the main pattern requiring an indirect jump for efficient computations. We propose a new ISA modification to mitigate the latency penalty.

We critically analyze the reach of our proposals in section 4. We demonstrate in particular that the ability of extracting a CFG is not transferable in a virtual machine. There is a parallel between indirect jumps and indirect memory operations that would require to remove the latter operations in order to precisely and easily extract the CFG in any virtual machine (VM). Alas, this would seriously limit the machine computational capability.

We discuss the classes of accepted or rejected programs in section 5 to point the limitations of totally removing indirect jumps.

On these mixed results, we draw a conclusion in section 6

2 THE MACHINE, CONTROL FLOW GRAPH AND CONTROL FLOW INTEGRITY

2.1 Machine

Our analyzes are performed on an abstract machine representative of the real chips powering most computation today. The elementary instructions that our machine is able to execute are inspired (and simplified) from the RISC-V ISA. Apart from the common architecture described in this section, we will define 3 different ISAs in section 3 for this same machine.

Our machine is a 64-bit RISC processor: addresses, data memory words and registers are 64-bit integers.

The machine has a mutable state composed of registers and data memory. The registers are:

- generic (read/write) registers $x1, \cdots, x16$.
- Stack pointer (SP) (read/write) register for the data stack.
- Constant zero and full (read-only) registers. Full is defined such that $x \oplus full = \neg x$ (full is the value with all its bits to 1).
- A special register (neither readable nor writable directly), the program counter (PC), that contains the address of the next instruction.

The data memory is finite, as in a real machine, and is word-addressed. Since the memory is finite, the machine is not Turing-complete. But as in real machines, it has no practical implications.

All registers and memory are initialized with a default zero value. A program is a dictionary of instructions: the address (64-bit unsigned integer) is used to uniquely select one instruction in the dictionary. As such, our architecture is a Harvard architecture, instructions and data are separated.

The execution of our machine works as follows:

- First provide a program that is put in a dedicated (read-only) memory.
- Set the PC to the program entry address.
- Then in an infinite loop:
  - Fetch the next instruction at the address in PC
  - Execute it by changing the state according to the instruction semantic. Or terminate the machine on an `halt` instruction.
  - If the PC has not been changed by the instruction, increment the address.

We will explore 3 ISAs presenting slight variations on how to handle the control flow in section 3.

2.2 Control flow graph

In our machine definition, we have seen that instructions are stored linearly in memory, as a dictionary where the addresses are the entry keys. Yet at runtime, the control flow is not linear. We want to be able to go forward or back. As a result, in our machine, the control flow is represented as a graph (potentially with cycles): the control flow graph (CFG). In it, nodes are instructions and edges are the legal transitions between instructions. For example, a conditional branch has two possible successors: the node has two outgoing edges in the CFG.

All invalid program addresses, in our machine, are considered aliases for the program entry address (simplification hypothesis for our analysis).

A CFG is always defined for any ISA since the complete digraph (*i.e.* where all nodes are connected with all the others) is a valid CFG: meaning that all legal control flows follow connected paths in the graph. Therefore, any valid CFG is an overapproximation. The best CFG is the smallest that is still valid for all possible program executions.

CFG extraction is useful on several counts. If CFG extraction is easy and precise, it allows to obtain more efficient static analyzis tools (*e.g.* dead code elimination can be performed safely and precisely). Additionally, it allows powerful CFI schemes such as instruction set randomization (ISR) [15].

On another hand, it renders obfuscation harder (but possible as seen in section 4): it would be more difficult for malware packers to hide their payload.

The question that naturally arises is: **can the best CFG be extracted from the instructions in general?**

Beyond the fact that the difficulty of precise CFG extraction is related to the reachability problem in programs (undecidable generally), we can simply prove that it is possible to hide the destination of an indirect jumps by using cryptographic hash functions.

Trapdoor predicates.

Definition 2.1. **Trapdoor predicate** Let $p : \mathbb{F}_2^n \to \mathbb{F}_2$ be a function working on bit vectors of size n (for a large enough n). We call p a trapdoor predicate if $\exists k \in \mathbb{F}_2^n$ such that:

- $\forall x \neq k \in \mathbb{F}_2^n, p(x) = 0$.
- $p(k) = 1$, where k is a secret value. Meaning that knowing p, there is no better way to find k than brute-forcing every possible input without additional information (the adversary has negligible advantage otherwise).

Trapdoor predicates can be built from a cryptographic hash function (other constructions are possible, for example by exploiting the SAT problem). Let $h : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a cryptographic hash function.

Then we can define p for a given secret value $k \in \mathbb{F}_2^n$ as:

$$p(x) = \begin{cases} 1 & \text{if } h(x) = h(k) \\ 0 & \text{in the other cases} \end{cases}$$

By construction, $p(k) = 1$ but k cannot be deduced from the knowledge of p (of course $h(k)$ is precomputed).

How to use these predicates to hide the control flow. Trapdoor predicates allow hiding the control flow in any given program as shown in Listing 1.

Listing 1: Hiding the control flow with a trapdoor predicate.

```
x     <- user input
delta <- p(x)*(h(x + 1) xor constant)
jump 0x1000 xor delta
```

In Listing 1, for most input values $p(x) = 0$, the program jump to the $0x1000$ value. But if the attacker inputs the secret value k, the program jump to another secret address (hidden, equal to $0x1000 \oplus h(k+1) \oplus constant$ where constant is chosen according to the desired destination).

In this program, without the knowledge of k, it is impossible (without brute-forcing) to

- find x such that we jump to an address different than $0x1000$,
- find the destination address when we jump to somewhere different than $0x1000$. We use $h(x+1)$ since $h(k)$ may be reverse engineered from $p(x)$ program.

In conclusion, because of the existence of trapdoor predicates, a precise CFG is not extractable without additional information in general. Specifically, the program is not enough: in Listing 1, the CFG extraction must assume that the jump instruction can reach the whole address space (where it can, in reality, jump to only two destinations).

Please note that we did not use opaque predicates for our demonstration since they are not theoretically sound as shown by Zobernig *et al.* [19].

2.3 Control flow integrity

A CFI mechanism is responsible for ensuring that only a valid control flow is taken. For every instruction, it must verify that the transition to the next instruction is an edge in the CFG. CFI has an extensive literature. A recent review has been published by Burow *et al.* [4]. Most CFI solutions try to verify that jumps can only reach legitimate addresses (forward edges). A special case is the `return` instruction to return from a routine call (backward edges).

Abadi *et al.* [1] show software CFI implementations: they propose code snippets to replace dangerous instructions (indirect jumps) in order to guarantee CFI.

Tice *et al.* [17] demonstrate a software solution that leverages the compiler to automatically insert the appropriate protections at jump sites (forward edges only). In particular, they tackle the problem caused by virtual method tables, necessary in some programming languages (*e.g.* C++) to enforce runtime polymorphism: a dispatcher is necessary there.

Backward edges (*e.g.* `return` instruction) are traditionally protected with a shadow stack [8]: the call stack is duplicated. On a `return` instruction, the return address on both stacks are read then compared. If they differ, an alarm is triggered. Another possibility explored by Davi *et al.* [5, 6] is to add instructions to the ISA for the sole purpose of validating function calls and returns. On any indirect function call, the processor switch to a particular state. The next instruction must be a special `CFIBR label` instruction in order to continue execution. The label is used to keep track of which functions are currently executing.

Another common solution proposed in the context of CFI is the use of a shadow stack, a second stack that duplicates the return addresses of the main stack. Upon a function return, both return addresses from the two stacks are compared before the backward indirect jump. This countermeasure is actually a duplication countermeasure, there is no added semantic in the program and there is no way to validate the return address against a truth value. As such it is only useful in some specific attack scenarios and is not a general solution for CFI.

To resist stronger attacker models, hardware CFI becomes necessary. SOFIA [15] proposes such a solution with an instruction set randomization (ISR) scheme: instructions are encrypted with a value representing the corresponding edge in the CFG. More precisely, instructions are encrypted at program startup using the previous and current PC values as shown on Equation 1.

$$i' = E_k(PC_{prev}||PC||...) \oplus i \tag{1}$$

Effectively, the CFG edge is encoded in the encrypted instruction and must be valid for decryption. The difficulty arises when two predecessors (thus two edges) are possible for one instruction. Then a hack is proposed to overcome this difficulty.

Hiscock *et al.* [9] explore how to properly implement an ISR scheme to ensure CFI by discussing and proposing encryption strategies.

For all these propositions, the system must be able to check the control flow with respect to a predetermined CFG. Since trapdoor predicates do not allow this in general, in the next section we propose to modify the ISA semantic to allow easy and precise CFG extraction for all programs.

3 SEMANTICS

The key point that forbid easy and precise CFG extraction is the presence of indirect jumps whose destinations are not statically predictable. Indirect jumps are mostly present in two forms: forward edges (`jump x0`) where we jump to the address present in a register, and backward edges (`return`) where we jump to the address following the last procedure call.

A radical solution to our problem is therefore to forbid indirect jumps (both forward and backward). Yet it is possible to handle indirect backward jumps in a statically predictable way, allowing as a consequence a more versatile ISA.

In this section, we will describe and test 3 ISAs:

- ISAv1 allows indirect jumps.
- ISAv2: similar to ISAv1 without indirect jumps.
- ISAv3: building on ISAv2, a restricted indirect jump is authorized in the form of a call/return semantic with a dedicated return stack, also called "call stack".

The source code for our implementation and our results can be found on the git repository at https://gitlab.com/Artefaritaj/simple_risc.

Here is a short description of the instructions:

- Integer arithmetic instructions: perform an arithmetic operation on two operands (in registers) and place the result in a register. Operations are: $+, -, \oplus, \wedge, \vee, \cdot, /,$ mod , <<, >>.

- Conditional branches: evaluate the condition among $=, \neq, \leq, <, \geq, >$ between two registers. If the condition is valid, jump to the specified address if not follow the usual control flow.
- Direct jump: goto the specified address for the next instruction.
- Call: direct jump to specified address and store the value of PC in a register (x14 for ISAv1) or a dedicated stack (ISAv3).
- Return: jump to the address popped from the dedicated return stack (ISAv3 only).
- Indirect jump: go to the address present in the given register (ISAv1 only).
- Direct load/direct store: load or store a data in memory to/from a register.
- Indirect load/indirect store: load or store a data in memory at address in a register to/from another register.
- Load immediate: set the value of the given register (source of constants).
- Register move: copy a register into another.
- Non-deterministic: load a non-deterministic value into a register. The semantic does not specify the source: it can be either user input, result of a true random number generator, etc.
- Halt: terminate the computation.

The `push` and `pop` instructions are also used and are pseudo-instructions developed as several machine instructions to do stack operations.

In this paper, we will change the ISA semantic by adding or removing instructions in order to enforce security properties at the ISA level.

3.1 Benchmarks

After defining our three ISAs, we want to verify that our new proposals achieve our goal of easy and precise static CFG extraction. At the same time, we measure the performance differences for two benchmarks:

- An Advanced Encryption Standard (AES) encryption is a simple program with an interesting instruction mix. The execution requires a lot of memory accesses and many arithmetic operations (*cf.* Table 1).
- A theoretical analysis of a dispatcher pattern. Dispatchers are the main argument to keep indirect jumps: we will analyze the penalty incurred by ISAv2 and ISAv3.

3.1.1 AES.

Implementation. This benchmark is an AES encryption (symmetric cryptography algorithm), encoded by hand in an assembly language. The instruction mix for ISAv1, ISAv2 and ISAv3 can be seen on Table 1.

The (naive) implementation is done in our own assembly language corresponding to the 3 ISAs. This assembly language is also used for listings 2, 3, 4, 5 and 7. Instructions map quite directly to their defining semantic. The @ symbol is used as an optional hint that we are using the value at the address present in the register. # is used to denote literal values. Finally, ! is used to denote global symbols (by default, symbols are defined only in their file).

Listing 2: Xtimes assembly implementation (AES subfonction) with ISAv1.

```
1  //x1 byte value to modify
2  !xtimes:
3      // save context
4      push    x2, x4
5
6      // x1=x1*2
7      load    x4, #0x1
8      sla     x1, x1, x4
9
10     // test modulo
11     load    x4, #0x100
12     and     x2, x1, x4
13     beq     end, x2, zero
14
15 sub_modulo:
16     load    x4, #0x11B
17     xor     x1, x1, x4
18
19 end:
20     // restore context
21     pop     x4, x2
22     //return
23     jump    x14
```

Results. For the three ISAs, we evaluate their performance and the size of the CFG that can be extracted without annotation, shown on Table 2. For the ISAv2, all the `return` instructions have been replaced by dispatchers (*cf.* subsubsection 3.1.2), another possibility would be to inline everything but this latter solution cannot be scaled to any program.

The performance is evaluated as the mean and the standard deviation for 10,000 AES executions. The CFG extraction is done with 2 different algorithms:

- Structural: the successors to an instruction are all the ones that can be reached according to the instruction semantic. The CFG is deduced from the structure of the program, not the data content. In particular, the successors to an indirect jump are all the instructions in the program. The extraction algorithm is described in algorithm 1.
- Tracking: it is possible to extract precisely the CFG with the ISAv3 call/stack semantic. In addition to the structural information, we track the return stack and compute all edges for all stack states. In other words, if the same procedure is called from 2 different locations, the `return` instruction will jump to 2 different addresses that can be determined if we track the whole return stack in our CFG analysis. The extraction algorithm is described in algorithm 2 and algorithm 3.

It is often possible to have a better CFG extraction algorithm in the presence of indirect jumps, for example Kinder *et al.* in [10] propose a (complex) solution to the problem. Balakrishnan *et al.* [2] propose quite a complete toolsuite for static analysis in general and CFG extraction in particular. Yet these solutions are, of course, non-general: they cannot deal with the snippet presented in subsection 2.2. The lesson being that the presented snippet should be rejected in the context of secure execution.

Our CFG extraction algorithms are simpler than the one proposed by Kinder *et al.*: only a simple control flow analysis is performed. A data-flow analysis may improve a bit the accuracy of the extracted CFG if some branch conditions are always true or always

Table 1: Instruction mix for one AES run for each ISA. *A return instruction is an indirect jump.

	ISAv1		ISAv2		ISAv3	
Type	Count	Ratio	Count	Ratio	Count	Ratio
Arithmetic	10011	0.44	9978	0.41	9825	0.44
LoadImmediate	5232	0.23	6767	0.27	5218	0.23
IndirectLoad	2650	0.12	2628	0.11	2554	0.11
IndirectStore	2464	0.11	2441	0.10	2368	0.11
DirectJump	668	0.03	719	0.03	668	0.03
IndirectJump*	616	0.03	X	X	616	0.03
ConditionalBranch	601	0.03	1703	0.07	601	0.03
RegisterMove	370	0.02	370	0.02	370	0.02
NonDeterministic	2	0.00	2	0.00	2	0.00
Halt	1	0.00	1	0.00	1	0.00
Total	22615	1.00	24609	1.00	22223	1.00

false. But such a case would represent a failure of the compiler: it should have detected this invariant and simplified the program accordingly.

Data: E: Address (entry address), P: Dictionary<Address, Instruction> (program)
Result: CFG: Graph<Address> (control flow graph)
`/* list of addresses to analyze */`
addresses_buffer ← [E];
`/* addresses already analyzed */`
analyzed ← {};
while *addresses_buffer is not empty* **do**
  `/* we now analyze instruction at new address */`
  current_address ← pop from addresses_buffer;
  push current_address to analyzed;
  `/* find successors, the set of addresses that can follow this address. Here the successors are chosen according to the nature of the instruction only (branches have 2 successors, etc.). */`
  successors ← successor_analysis(program P, current_address);
  for *successor* ∈ *successors* **do**
    `/* The add edge method adds nodes too if necessary */`
    add edge to CFG: current_address → successor;
  end
  successors_to_analyze ← filter successors to keep only addresses not in analyzed;
  append successors_to_analyze to addresses_buffer;
end
return CFG;

Algorithm 1: Structural CFG extraction algorithm.

On Table 2, we show the implementations performances and the CFG nodes and edges coverage. The CFG nodes (respectively edges) coverage is the ratio of nodes (resp. edges) in the CFG that are touched during one execution. E.g. an edges coverage of 6.4% means that only 6.4% of the edges have effectively been followed during one execution: the CFG is imprecise in this case. In our case, it means that numerous instruction transitions are considered valid where they should not.

On the performance side, we observe that ISAv2 is slower. Indeed, for all returns from procedures we have a small dispatcher to jump back to the correct location. ISAv1 is a bit slower than ISAv3 since the stack handling (push and pop) is explicit instead of implicit in ISAv3.

For CFG extraction, we note that the structural extraction is imprecise for ISAv1 and ISAv3. In the latter case, the problem comes from the `return` instructions: the algorithm cannot infer where they jump and assume that all instructions are potential successors. In these two cases, we can say that static CFG extraction is impractical. But for ISAv2 with structural extraction and ISAv3 with tracking extraction, the CFG is extremely precise. For ISAv2 the coverage is not perfect since, because of the semantic of the ISA, unreachable `halt` instructions must be added to deal with error cases.

This benchmark showcases the advantages for our ISAv2 and ISAv3 proposals: the CFG is extractable with a 9% loss of performance for ISAv2 or a 2% gain of performances for ISAv3.

3.1.2 Dispatcher. A dispatcher is an instruction pattern where a particular control flow is chosen depending on a data value. This pattern is used to call functions in a library, as a result of a system call or as a way to implement runtime polymorphism in object-oriented languages. This pattern is particularly dangerous with respect to information security as we will see in section 4. It often relies on indirect jumps for an efficient implementation, therefore we expect that our ISA proposals will have varying dispatcher implementations.

Listing 3: ISAv1 dispatcher pattern

```
1   //x1 contains the address to branch to
2   //(may be the result of pointer arithmetic)
3   call   x1
4 dispatcher_end:
5   //continue
6
7   //...
8 procX:
9   push   x14
```

Data: E: Address (entry address), P: Dictionary<Address, Instruction> (program)
Result: CFG: Graph<Address> (control flow graph)
`/* list of pairs (addresse to analyze, corresponding ReturnStack) */`
addresses_buffer ← [(E, empty ReturnStack)];
`/* the data contained in the pair (Address, ReturnStack) is hashed for easier handling and for dealing with recursion */`
`/* hashes already analyzed */`
analyzed ← {};
while *addresses_buffer is not empty* **do**
  `/* we now analyze instruction at new address */`
  (current_address, current_stack) ← pop from addresses_buffer;
  push (hash(current_address) ⊕ hash(current_stack)) to analyzed;
  `/* find successors, the set of pairs (addresses, corresponding return stack) that can follow this (address, return stack) pair. Here the successors are chosen according to the nature of the instruction and the values in the return stack. */`
  successors ← successor_analysis(program P, current_address, current_stack);
  for *(succ_address, succ_stack)* ∈ *successors* **do**
    `/* The add edge method adds nodes too if necessary */`
    add edge to CFG: current_address → successor;
  end
  successors_to_analyze ← filter successors to keep only pairs (address, stack) such that (hash(add) ⊕ hash(stack)) is not in analyzed;
  append successors_to_analyze to addresses_buffer;
end
return CFG;

Algorithm 2: Tracking CFG extraction algorithm, uses algorithm 3 for ReturnStack hashing.

```
10    //...
11    pop    x14
12    jump   x14
```

Listing 4: ISAv2 dispatcher pattern

```
1     //x1 contains the value that decides where to branch
2     load   x15, #0
3     beq    proc1, x1, x15
4     jump   proc2
5
6   dispatcher_end:
7     //continue
8
9     //...
10  procX:
11    //...
```

Data: RS: Vec<Address> (return stack)
Result: H: integer (hash value)
`/* The objective is to transparently deal with recursion. If a repeated pattern is found at the end, one repetiton is not taken into account. hash(A) = hash(AA), hash(AB) = hash(ABAB), hash(ABC) = hash(ABCABC), ... */`
max ← length(RS);
if *max > 1* **then**
  stack_size ← length(RS);
  pattern_size ← 1;
  while *pattern_size * 2 <= stack_size* **do**
    `/* detect pattern repetition of size pattern_size */`
    if *RS[(stack_size − pattern_size) .. stack_size] == RS[(stack_size − 2 ∗ pattern_size) .. (stack_size − pattern_size)]* **then**
      `/* recursion found */`
      max ← max - pattern_size;
      break;
    else
      pattern_size ← pattern_size + 1;
    end
  end
end
H ← 0;
for *i ← 0* **to** *max* − 1 **do**
  H ← hash(H | RS[i]);
end
return H;

Algorithm 3: ReturnStack hash algorithm.

```
12    jump   dispatcher_end
```

Listing 5: ISAv3 dispatcher pattern

```
1     //x1 contains the value that decides where to branch
2     load   x15, #0
3     beq    call_proc1, x1, x15
4     jump   call_proc2
5   call_proc2:
6     call   proc2
7     jump   dispatcher_end
8   call_proc1:
9     call   proc1
10    jump   dispatcher_end
11    //...
12  dispatcher_end:
13    //continue
14
15    //...
16  procX:
17    //...
18    return
```

For the 3 proposed patterns, we evaluate the number of instructions necessary for the dispatcher logic as a function of the number p of procedures that must be reachable by the dispatcher, shown on Table 3.

Table 2: Evaluating ISAs performances and CFG coverage for an AES implementation

	CFG extraction	Mean duration	Std deviation	CFG nodes count	nodes coverage	CFG edges	edges coverage
ISAv1	structural	22635.0	24.1	778	99.6%	12437	6.4%
ISAv2	structural	24605.2	24.1	816	99.1%	836	99.0%
ISAv3	structural	22230.7	23.8	747	99.7%	11942	6.4%
ISAv3	tracking	22230.8	24.0	745	100%	764	100%

Table 3: Dispatcher instruction count (for $p > 3$)

	Branch logic (red)	Call logic (green)	Return logic (blue)	Total
ISAv1	0	1	$3 \cdot p$	$3 \cdot p + 1$
ISAv2	$2 \cdot p$	0	p	$3 \cdot p$
ISAv3	$2 \cdot p$	$2 \cdot p$	p	$5 \cdot p$

Table 4: Dispatcher latency

	Branch latency (red)	Call latency (green)	Return latency (blue)	Total
ISAv1	0	1	3	4
ISAv2	$2 \cdot \lceil \log_2(p) \rceil + 1$	0	1	$2 + 2 \cdot \lceil \log_2(p) \rceil$
ISAv3	$2 \cdot \lceil \log_2(p) \rceil + 1$	2	1	$4 + 2 \cdot \lceil \log_2(p) \rceil$

The branch latency, a measure of the time taken to call a particular procedure, is shown in Table 4. The value for ISAv2 and ISAv3 can be proven by recurrence. Indeed, for a given p the branching can be done by using a branch (2 instructions) and the 2 patterns for $\lceil \frac{p}{2} \rceil$ and $\lfloor \frac{p}{2} \rfloor$. As a consequence $Latency(p) = 2 + Latency\left(\lceil \frac{p}{2} \rceil\right)$. If we admit that $\forall l < p$, the relation $Latency(l) = 2 \cdot \lceil \log_2(l) \rceil + 1$ holds (manually verified for the firsts l), then

$$\begin{aligned} Latency(p) &= 2 + 1 + 2 \cdot \lceil \log_2(\lceil \frac{p}{2} \rceil) \rceil, \\ Latency(p) &= 1 + 2 \cdot \left(1 + \lceil \log_2(p) - 1 \rceil\right), \\ Latency(p) &= 1 + 2 \cdot \lceil \log_2(p) \rceil \quad . \end{aligned}$$

In the case where all procedures called by the dispatcher return to the same location, then ISAv2 is actually slightly better than ISAv3 for the two metrics (instruction count and latency). Yet the true reason why indirect jumps are preeminent is because of the low dispatching latency, **independent from the number of procedures to call**. Moreover, the high number of branches for ISAv2 and ISAv3 does not play well with speculative execution: dispatchers are highly penalized by the deletion of indirect jumps. Indirect jumps allow a high fanout control flow: an instruction can have a lot of possible successors. Whereas with ISAv2 and ISAv3, an instruction has 2 successors at most.

An additional problem with the dispatchers with ISAv2 and ISAv3 is that the latency of a procedure call depends on what procedure is called. Depending on the application, this timing side-channel can be a vulnerability.

Therefore, the usage (or not) of indirect jumps is a trade-off between the performance of a high fanout control flow and the possibility (or ease) of knowing the CFG statically.

3.2 Branching

In order to meet a better trade-off between performances and CFG discoverability, a new instruction may be added: the *n-fanout conditional branch*.

Listing 6: n-fanout conditional branch illustration

```
1  nbranch x1, #3
2  jump    proc1
3  jump    proc2
4  jump    proc3
5  jump    error_handling
```

The deciding register is expected to hold an integer value i in $[\![0, n-1]\!]$ (n being the constant litteral present in the instruction). An implicit jump is performed to the address: $AddressOf(branch)+1+i$. If the register holds an illegal value, we jump to the error handling address at $AddressOf(branch) + 1 + n$. A usage example can be found on Listing 6.

With this new instruction, the latency is now $O\left(\log_n(p)\right)$, where n can be chosen equal to p to minimize latency.

4 LIMITS OF CONTROL FLOW INTEGRITY

The ability to precisely extract any CFG limits the reach of virtualization as a defense mechanism: the virtual machine inner mechanisms cannot be hidden. But as demonstrated in this section, it is still possible to hide the control flow in the data domain.

4.1 Virtual machines

Guaranteeing CFI is particularly important to mitigate Return-Oriented Programming (ROP) attacks [14] where an attacker modifies the control flow from a valid program to obtain a malicious behavior. The trouble is CFI is not enough. Indeed, the existence of emerging virtual machines, in particular *weird machines* [7], limits the guarantees that CFI can offer. An illustration, valid with all our ISAs, can be seen on Listing 7.

Listing 7: subleq virtual machine

```
// data memory is filled with user-defined values

// initialization
// virtual program counter
// VPC(x13) = 0
  load  x13, #0

//exec one subleq instruction
subleq:
//read operands from data memory
  move  x1, x13
  load  x15, #1
  add   x13, x13, x15
  move  x2, x13
  add   x13, x13, x15
  move  x3, x13
//increment for next instruction
//if no jump
  add   x13, x13, x15

//subleq execution
  load  x4, @x1
  load  x5, @x2
  sub   x6, x5, x4
  store @x2, x6
  ble   ijump, x6, x0

//start next instruction
  jump  subleq

//virtual indirect jump
ijump:
  move  x13, x3
//start next instruction
  jump  subleq
```

Subleq machines have been proven to be universal computers [13]. Our subleq VM has a simple control flow with only direct jumps (it can be written with any of the three ISA). Yet the instructions to the VM are in fact in data memory and indirect jumps are performed in the data domain. This program is an example where even if CFI is guaranteed, we cannot enforce any security property as a result. In other words, security properties such as CFI are not automatically transposed in the virtual machine: if an attacker replace a word in data memory, she can hijack the VM control flow.

A possible conclusion would be that since Harvard architectures can be converted to von Neumann architectures (and reciprocally) via virtual machines, then properly removing indirect jumps requires removing indirect (data) memory accesses. Non-coincidentally, for safety-critical application the use of the dynamic heap allocation is forbidden (*e.g.* the rule 20.4 of MISRA-C:2004) since that generates lots of indirect memory accesses making the behavior of the program harder to statically predict. But removing entirely indirect memory accesses is a lot more constraining.

To respect our design principle that the ISA must intrinsically contain the semantic information about the control flow, pointers used for direct memory accesses must be augmented with metadata inside the ISA. The instructions below should replace indirect memory accesses to restrict their reach:

- `BoundedIndirectLoad Rd, Rs, B1, B2`: load the value present in memory at address contained `Rs` in register `Rd` if the address is between the bounds `B1` and `B2` (being literal values).
- `BoundedIndirectWrite Rd, Rs, B1, B2`: same principle but store at address `Rd` the value register `Rs`.

This is a proposition present in CHERI [18]. Unfortunately, these new instructions may mitigate the power of emerging virtual machines but do not allow to strictly enforce CFI.

4.2 Pointing our model simplifications

The machines presented in sections 2 and 3 are simplified versions of actual machines. In particular, real machines implement mechanisms where the control flow is not defined by instructions: interrupts and traps. An interrupt is a change of control flow due to an external event: *e.g.* when a new byte has arrived on the serial bus, an interrupt is raised that divert the control flow to a dedicated subroutine, *whatever the previous instruction was*. A trap is a special interrupt caused by an instruction: the `trap` instruction divert the control flow to an implementation-defined location. Traps are notably used to transfer control from the user land to the kernel.

These mechanisms are not captured by our models but neither are they by most CFI mechanisms presented in subsection 2.3. Modeling the security of these control transfers is a major hurdle that has yet to be achieved. For example, how can we ensure CFI in the presence of the *page-fault weird machine* [3]?

Another limitation is our use of a Harvard architecture. Even if it is well known that the distinction between von Neumann and Harvard architectures has little relevance theoretically, we have no ways to modify the instructions in our machine after the initial program write. As such it is not possible to install new programs after the initial programming. Quite limiting for a general-purpose computer. The reason being that any instruction modification has the potential to break CFI and is harder to model.

5 ACCEPTED AND REJECTED PROGRAMS

In order to express a program with ISAv2 or ISAv3, CFG extraction has to be performed at compilation (or conversion) time. In our new ISAs, the new program will reflect the quality of the prior CFG extraction. But any subsequent CFG extraction becomes trivial (*cf.* algorithms 1 and 2 in the appendix) and equivalent to the prior one: the structure of our new program reflects it. Additionally, if the prior CFG extraction is not precise, the new program becomes extremely inefficient (both in size and execution time) due to the presence of big dispatchers. For all these elements, we can say that a program without indirect jumps structurally mirrors its control flow graph.

But generally, the possibility of expressing a program in our new ISAs is equivalent to the possibility of extracting a complete CFG at compilation time (no invalid edge is ever taken at runtime).

Accepted programs. As seen in subsection 2.2, there are programs that cannot be expressed efficiently in our ISAv2 or ISAv3. Here is how to write the snippet shown in Listing 1 in an ISA without indirect jumps. If the program has size n instructions, we can write the new version by first computing the destination address as in the initial version, then dispatching on this address. As shown in Table 3, the new program will have at least size $3 \cdot n$ instructions with ISAv2 and ISAv3 (n for the initial version, $2 \cdot n$ to be able to

branch to all initial instructions). We still do not know the secret value that provokes a different branching destination or what that new destination is. But in this new version, the structure of the program makes explicit all possible branches.

All programs can therefore be converted to the new ISAs, if and only if their size is finite and the memory layout is known (we consider only jumps to valid instruction addresses). But not all conversions are efficient. Equivalently for a new program, it can be expressed in the new ISAs if all its instructions are known at compile time.

In particular, virtual method tables can be written with ISAv2 and ISAv3. These tables are used in Object-Oriented languages to deal with dynamic dispatch: depending on the caller's type, a different function (at a dedicated address) is called. Virtual method tables can be replaced with dispatchers since all destinations are known at compile time.

Rejected programs. They are programming patterns where extracting a complete CFG is impossible at compile time. An example would be an application able to dynamically load plugins for additional functionality. When the application is compiled, the plugin instructions are not known. If we forbid indirect jumps, the application cannot divert the control flow to the plugin. Similarly, the kernel cannot launch a new application not known when we compile it. We must recompile our kernel for each new application added. In fact, the question is how to execute a program after compilation, since the new program was not known before.

From a security standpoint, these are dangerous patterns. For example, executing a plugin program is equivalent to execute unknown instructions from the application context.

Indirect jumps are needed. Launching a new unknown application from the kernel seems to be more acceptable than launching a plugin from an application. This is because in the former case, we change the execution context: the application has fewer privileges than the kernel.

Finally, these examples give the solution to our problem. Indirect jumps are absolutely required to get non-trivial functionalities. But an indirect jump is also equivalent to switching to a new security domain. As a consequence in a secure ISA, if we want to remove indirect jumps as in our new ISAs, we must also introduce hardware security domains. What is instruction and what is data is tied to the security domain, so that switching domain allows transferring data into instruction memory. With this new feature, an indirect jump can be authorized when switching to a new context. The new jump would not be directly to a particular instruction, but to the new security domain entry point, able to verify the validity of the new control flow.

With this mechanism, an application can launch a plugin, but in a new security domain. Compilation is possible, but launching a new application is done in a new security domain.

6 CONCLUSION

In order to execute secure programs, our computation model must guarantee specific security properties. Control flow integrity (CFI) is a critical property preventing the attacker from altering the program control flow. Yet to ensure this property, we have shown that the system must be able to extract the CFG which is not possible in general because of the existence of trapdoor predicates.

As a consequence, in order to execute generic programs and still ensure CFI, we need to modify the computational model to allow easy and precise CFG extraction. We compare 3 ISAs: one with indirect jumps, one without and a last ISA allowing a restricted form of indirect jumps. We show that an AES encryption can be performed in the new ISAv3 without loss of performance but allowing CFG extraction. Yet we show that an efficient dispatcher pattern requires a forward indirect jump. Our new ISA proposals incur a performance penalty for high-fanout control flows. Finally, we have shown that because of the existence of emerging VMs, the CFI property cannot be properly guaranteed without greater constraints on indirect memory transfers. This negative result suggests that the CFI property, even if improving the system security, is no silver bullet.

Finding how to prevent the emergence of VMs would go a long way to ensure the security of our computers.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. [n.d.]. Control-flow integrity principles, implementations, and applications. 13, 1 ([n. d.]), 1–40. https://doi.org/10.1145/1609956.1609960

[2] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug. 2010), 84 pages. https://doi.org/10.1145/1749608.1749612

[3] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W Smith. 2013. The page-fault weird machine: lessons in instruction-less computation. In *Presented as part of the 7th {USENIX} Workshop on Offensive Technologies.*

[4] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. [n.d.]. Control-Flow Integrity: Precision, Security, and Performance. 50, 1 ([n. d.]), 1–33. https://doi.org/10.1145/3054924

[5] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. [n.d.]. HAFIX: hardware-assisted flow integrity extension. ACM Press, 1–6. https://doi.org/10.1145/2744769.2744847

[6] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. [n.d.]. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. IEEE, 1–6. https://doi.org/10.1109/DAC.2014.6881460

[7] T. F. Dullien. 2019. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing* (2019), 1–1. https://doi.org/10.1109/TETC.2017.2785299

[8] Mike Frantzen and Mike Shuey. [n.d.]. StackGhost: Hardware Facilitated Stack Protection. USENIX.

[9] T. Hiscock, O. Savry, and L. Goubin. 2017. Lightweight Software Encryption for Embedded Processors. In *2017 Euromicro Conference on Digital System Design (DSD).* 213–220. https://doi.org/10.1109/DSD.2017.25

[10] Johannes Kinder, Florian Zuleger, and Helmut Veith. 2009. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Verification, Model Checking, and Abstract Interpretation*, Neil D. Jones and Markus Müller-Olm (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 214–228.

[11] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. [n.d.]. Spectre Attacks: Exploiting Speculative Execution. ([n. d.]), 19.

[12] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. [n.d.]. Meltdown: Reading Kernel Memory from User Space. ([n. d.]), 18.

[13] Farhad Mavaddat and Behrooz Parhami. 1988. URISC: the ultimate reduced instruction set computer. *International Journal of Electrical Engineering Education* 25, 4 (1988), 327–334.

[14] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (March 2012), 34 pages. https://doi.org/10.1145/2133375.2133377

[15] Ruan de Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen de Bosschere, Bart Preneel, Bjorn de Sutter, and Ingrid Verbauwhede. [n.d.].

SOFIA: Software and Control Flow Integrity Architecture. IEEE.

[16] H. Theiling. 2000. Extracting safe and precise control flow from binaries. In *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*. 23–30. https://doi.org/10.1109/RTCSA.2000.896367

[17] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. [n.d.]. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. ([n. d.]), 15.

[18] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 457–468.

[19] Lukas Zobernig, Steven D. Galbraith, and Giovanni Russello. 2017. When Are Opaque Predicates Useful? Cryptology ePrint Archive, Report 2017/787. https://eprint.iacr.org/2017/787.