Recommendations for a radically secure ISA

Mathieu Escouteloup, Ronan Lashermes, Jean-Louis Lanet, Jacques Jean-Alain Fournier

Recommendations for a radically secure ISA

Mathieu Escouteloup
Inria, Univ Rennes, CNRS, IRISA

Jacques Fournier
Univ. Grenoble Alpes, CEA Leti, DSYS/LSOSP

Jean-Louis Lanet
LHS-PEC

Ronan Lashermes
Inria/SED&LHS

ABSTRACT

The rising number of attacks targeting processors at micro-architecture level encourages more research on hardware level solutions. In this position paper, we specify a new `RV32S` "secure" instruction set architecture (ISA) derived from the `RV32I` RISC-V ISA. We propose modifications in the ISA to prevent timing side-channels, strengthen control flow integrity and ensure micro-architectural state isolation. The goal is to provide a new minimal hardware/software approach through which software attacks exploiting hardware vulnerabilities can be circumvented.

KEYWORDS

Instruction set architecture, hardware security context, side-channels

Mathieu Escouteloup, Jacques Fournier, Jean-Louis Lanet, and Ronan Lashermes. 2021. Recommendations for a radically secure ISA. In *Proceedings of Fourth Workshop on Computer Architecture Research with RISC-V (CARRV2020).* Valencia, Spain, 7 pages.

1 INTRODUCTION

In 2018, the Spectre [11] and Meltdown [15] attacks received a wide attention by showing how unsecured our modern central processing units (CPUs) are. But it has been known for a long time that the hardware was a source of vulnerabilities (*e.g.* as shown in 2005 by Bernstein [4] and Percival [19]).

What makes these new attacks so important is that, in the meantime, software security as gained a lot of maturity. Hardware is becoming the weakest security link in our systems.

Motivation. To improve the security of our CPUs we must rethink how to design them by taking into account the security issues from the beginning. Numerous solutions have been proposed.

(1) Adding more privilege levels (trusted execution environments (TEEs) such as Keystone [12], Hex-Five's MultiZone).
(2) Ensuring stronger access control policies to access memory (physical memory protection (PMP), Morello/CHERI [24]).
(3) Hardware hardening: build the processor around stronger primitives (CEA's IntrinSec project).
(4) Circumventing the problem by modifying the software: for example the retpoline countermeasure is an instruction pattern intended to prevent hardware speculation (and failing in some edge cases).

All these solutions are needed, but there is one that we think is not explored enough: modifying the instruction set architecture (ISA) to enable the software/hardware interface to "talk" about security, even in the abscence of any memory management unit (MMU). The most notable works in this area are, as far as we know, from Yu *et al.* [25] and Ge *et al.* [9] that propose to modify the ISA to fight timing side-channels.

Contribution. In this paper, we propose recommendations to design an ISA for CPUs supporting critical systems. These recommendations are centred around 3 themes. After giving our design guidelines in section 2, we show how to limit the ISA's power to mitigate some side-channels in section 3. We propose to restrict the ISA even more to have stronger control flow guarantees in section 4. Finally, we decribe a RISC-V extension to allow the ISA to express security boundaries in section 5. We conclude in section 6 by showing the links between our recommendations.

2 DESIGN GUIDELINES

We acknowledge that a trade-off is required between performance, energy consumption and security. To achieve a high CPU security, worthy of a critical system, we must compromise on both performance and energy consumption.

2.1 Principles

In order to guide our proposals, we identified a list of generic principles that help us decide if a feature should or should not be included in our secure system. We are following here the example given by Patterson and Waterman [17] who give the following principles as RISC-V design guidelines: *cost, simplicity, performance, isolation of architecture from implementation, room for growth, code size and programmability.* Our own principles are listed below.

Simplicity. To ensure security, we have to limit the attack surface. Simple systems can be modelled in an easier way, allowing the blue team to predict the behaviour of the system and therefore identify weaknesses. The simplicity principle opposes the guarantee of retro-compatibility: old interfaces must be tossed.

Principle of least privilege. Any entity in the system must only be able to do what it is designed to do and nothing more.

Habeas Corpus. The principle of least privilege requires compartmentalization, the *habeas corpus* principle is about the interaction between these compartments. A compartment management (creation, pause, data modification, ...) can only be performed from inside a compartment of higher power.

Transparency. The security of a system must not rely on its opacity (Kerckhoffs's principle). The secure system designer and user are usually not the same entities, and the first must prove the system security to the second.

Prove yourself. Human developers are error-prone. No amount of dedication can prevent the one mistake that will break a system

security. Therefore, secure systems must push for and ease the use of proof-oriented software. A lot of properties can and must be proven: memory safety, absence of data races, no data leakage *etc.* Proofs must be everywhere, in hardware, in software and in the tools used for development.

Defence in depth. If a given property holds on a program, then we surely do not need additional hardware isolation. Mathematical proofs are only valid under a set of hypotheses. When the program runs on hardware and thus get a physical incarnation (power consumption, memory access timings *etc.*), then the hypotheses may change and as a consequence, proofs do not hold any more. As a result, several protection mechanisms, sometimes redundant, must be put in place to counteract the collapse of the abstractions.

2.2 Designing a new secure ISA

In the rest of this paper, we make recommendations to build a new secure ISA by relying on the RISC-V specification. As a consequence, the recommendations should be read as a modification of the `RV32I` base ISA to form a new one: `RV32S` (S as in Secure).

3 THE SEMANTICS OF SCA HARDENED INSTRUCTIONS

Since the cache timing attack on AES [4], software and hardware designers try to devise techniques to avoid such vulnerabilities. The principle is to avoid any timing dependency on a secret value. Sometimes, it can be achieved in software by reordering the instructions.

But there are several difficulties with this technique: the application developer does not have perfect control on how the compiler generates instructions corresponding to its code. In particular, widespread compilers do not conserve timing semantics: today, timing sensitive applications must be written in assembly. Yet new research has shown that timing semantics preserving compilers are possible [3].

Unfortunately, it is not enough if there is no guaranteed timing semantics at the ISA level. A classic example is an hardware multiplication whose duration depends on the data fed. In this case, the timing leakage reappears.

The conclusion is that we cannot let the sole responsibility of preventing side-channels falling on the developers' shoulders. Some confidentiality properties must be guaranteed by the hardware and the software ecosystem (compilers, libc, operating systems (OSs), ...) must take advantage of it. This idea has been explored by Yu *et al.* [25], with a RISC-V extension to ensure that confidential data is only handled by safe instructions, by tracking and propagating confidentiality labels on data.

Here we suggest simpler solutions: we rely on the compiler to know what is confidential or not, and we enable it to map confidential data to confidential registers, no hardware tracking needed.

3.1 Confidential registers

In order to communicate the desire to enforce stronger confidentiality with respect to side-channels, we can tag some registers as confidential. The micro-architecture must try to enforce this guarantee (with an effort related to its security profile) and the semantics of the instructions is modified in consequence.

Recommendation 1 *Confidential registers*
Tag some registers (*e.g.* `x8` to `x15`) as confidential, with the following semantic consequences.

- It is not possible to branch depending on a confidential register.
- Instructions are authorized only if their timing is not data dependant (integer multiplication and division in particular may be forbidden depending on their hardware implementation).
- They cannot be used as the source address in `load` and `store` instructions.

Any violation of these rules should raise an hardware security fault.

In addition, the hardware must try to prevent any leakage on these registers' values. Depending on the security profile, these registers can be hardened: the values are masked to avoid power consumption leakage, the micro-architecture uses hardened execution units (*e.g.* masked computation) for instructions involving at least a confidential register, ...

Recommendation 2 *Automatic memory encryption*
An additional behaviour can be attached to the use of confidential registers as destination registers for a `load` or as the second source register for a `store`. In this case, we can encrypt the confidential data in memory, as soon as it leaves the pipeline. Devising a sound encryption scheme for that purpose would require a paper of its own.
If the hardware security contexts (HSCs) were used (concept developped in section 5), we could use the automatic encryption feature to bind a data to an HSC: the encryption key would depend on the current context, therefore preventing another context to read the data.

3.2 Consequence of confidential registers

Instructions at the confidentiality boundary, that have a confidential input and a public output or vice-versa, can be trivially detected. There is no need to forbid such instructions, but they should be carefully audited to ensure that the developer intention is respected.

On the software side, confidential registers allow the compiler to have a clear semantics for confidentiality: if a data is confidential, it must only be put into confidential registers. Doing an unsafe operation such as branching on a confidential register is now a compile time error (it cannot even be compiled).

Static analysis can now trivially stress out all the confidentiality boundaries (converting a confidential data into public data, ...), making it easier to detect flaws in the confidentiality policy. It becomes easier to prove such policies automatically, with *e.g.* a theorem prover.

3.3 Random number generator

A lot of security-related applications require a strong and reliable random number generator. Without an hardware random number

generator, applications must harvest system-level entropy themselves (usually by timing peripherals). But system-level entropy cannot provide strong guarantee of randomness, and then a piece of software becomes responsible for converting this entropy into a suitable random number generator. All these steps may leak information through side-channels.

Recommendation 3 *RNG instructions*
An instruction for random number generation should be added.
`CSPRNG rd`: a cryptographically secure pseudo-random number generator, following NIST SP 800-90A, to use for most purposes (including generation of cryptographic keys).

Adding the `CSPRNG` instructions changes the nature of the ISA. Without them, we can call the ISA functional: the conformance of a device to the specification can be tested by testing its functionalities. But a `CSPRNG` cannot be tested to know if it works properly: vulnerable random number generators can successfully pass all the randomness statistical tests. A good `CSPRNG` implementation can only be recognized by its design and implementation, not (only) by its functionality. The transparency of the implementation becomes critical to build trust between designers and users. Now, the ISA has become a formal contract between software and hardware that is not only tied to functionality, that cannot be tested: it has become a formal ISA.

4 IMPROVING SECURITY GUARANTEES WITH STRICTER CONTROL FLOWS

Abusing the control flow is a common way to tamper with a system. Ensuring the static integrity of an application usually requires to compare the hash of the binary you are about to launch with a truth value stored somewhere safe. But the attacker can still abuse the legit instructions in your application, and reorder them to build their payload. This principle is behind return-oriented programming (ROP) [20] that uses a chain of gadgets (instructions patterns of interest to the attacker) that are glued together with `RETURN` instructions.

Preventing ROP (and its variants) is a widely studied problem. Control flow integrity (CFI) tries to ensure that the control flow of the initial application is respected and not abused. CFI solutions usually verify that (forward) jumps and returns (backward jumps) can only go to legitimate addresses. These techniques can be implemented in software [2], now with the compiler automatically adding the verification code [22].

But CFI can also be proposed in hardware. In [6], Davi *et al.* propose an efficient way to restrict where a jump can land to by adding new CFI dedicated instructions. The CHERI project proposes, among other things, to protect pointers with unforgeable capabilities [24]. SOFIA [7] is an instruction set randomization (ISR) solution that encodes the control flow graph in the encrypted binary. It enables the processor to verify the correct behaviour of the running application.

More generally, it seems that for all sufficiently large pieces of software, we observe the emergence of virtual machines, often called weird machines [8]. The problem with these virtual machines is that security properties usually do not transpose into them: the attacker gains an effective way to bypass countermeasures. To counteract this issue, the most effective technique seems to limit the size and complexity of all the programs.

Presenting some relevant RISC-V instructions. In this section recommendations, we propose to modify the `JAL` and `JALR` instructions. Here is a brief reminder of how these instructions work.

- `JAL rd, offset` (jump and link): jump to the address given by the offset (sign extended, relative to the program counter (PC)). Put the following instruction address into `rd`.
- `JALR rd, offset(rs1)` (jump and link register): jump to the address given by the offset (sign extended) applied to the value in `rs1`. Put the following instruction address into `rd`.

4.1 Same-length instructions only

Variable-length instructions help ROP attacks: the attacker can choose to interpret a part of a long instruction as a short one, giving him more opportunities to find useful instructions in a given piece of code. Additionally, variable-length instructions make possible a new class of errors in case of bad alignment with respect to a given instruction size.

The solution is simple, a secure ISA must only propose same-length instructions. In the case of RISC-V, the C extension must therefore be forbidden.

The cost is the missed opportunity of code size reduction due to the compressed representation (estimated to $25-30\%$ [18]), as well as the corresponding performance improvement due to increased effective instruction cache size.

Recommendation 4 *Same-length encoding*
Enforce same-length instruction encoding (forbid RISC-V C extension).

This Recommendation 4 allows us to recover two bits per instructions (previously used to encode instruction size).

Recommendation 5 *Jump alignment modification*
Change the `JAL` instruction alignment requirement: the J immediate now encodes for a multiple of 4 bytes.

4.2 Removing forward indirect jumps

Indirect jumps are at the origin of the biggest defender challenge: decide between valid and invalid control flows. Since an indirect jump can jump to any address depending on a data value, knowing the jump destinations requires to find all possible values. As shown in [10], this task is (provably) not possible. In order to make control flow statically decidable and to make runtime countermeasures easier to deploy, we must forbid forward indirect jumps.

This is a simple way to promote a structured control flow at the ISA level. Notably, the recent WebAssembly virtual machine has

similar (even stronger) constraints, a controversial feature and the origin of inefficient compiler integration [1].

Recommendation 6 *Forbidding forward indirect jumps*
Remove the `JALR` instruction to forbid forward indirect jumps. A mechanism is presented in section 5 to reintroduce safer indirect jumps.

Removing the JALR instructions has several consequences that must be mitigated.

- Dispatcher patterns become costly: we need to add a new DISPATCH instruction as shown in subsection 4.3.
- It becomes impossible to transfer the control flow to a different program. This point is dealt with in section 5.
- It is no longer possible to return to the instruction following a JAL instruction even if we stored the correct address in a register. In particular, it prevents to simply return from a procedure. Apply Recommendation 7 to enable a RETURN instruction.
- There is no possibility to perform a direct long jump. Apply the trick detailed below if you really want to enable long jumps.

Recommendation 7 *Returns*
For efficient returns from procedures, add a new `RETURN` instruction. Semantically, the instruction should jump to the instruction following the last executed JAL whose bit 7 was set to 1.
In other words, the CPU should implement a call stack. The JAL destination register becomes useless and its freed bits (all but the least significant, *i.e.* 4 bits) can be used to extend the jump reach. A call to JAL now pushes the return address to the call stack if bit 7 (the least significant bit of rd) is 1. Executing RETURN pops the last address in the call stack and jumps to it.

Implementing the call stack. Recommendation 7 asks for a call stack, and its integrity must be guaranteed. Yet, there are several possibilities to implement it. Here is a proposal.

- Introduce a new register csp (call stack pointer) which is implicitely used by JAL and RETURN.
- Introduce a hidden register csp_depth which hold the depth of the call stack pointer, independantly of csp's value.
- csp can be read from and written to with a dedicated CSR (control and status register) mapping.
- To ensure integrity, a push to the call stack (with JAL) or a pop (with RETURN) must protect and verify the addresses in the stack with cryptography.

In this case, if we execute `JAL 1, offset` at the 32-bit address a_0, the following sequence occurs.

- Compute return address: $a_1 = a_0 + 4$.
- We store the encrypted address in memory, taking 64 bits: $M[csp] \leftarrow E_k(a_1||csp\_depth)$, with a suitable encryption algorithm E with key k. This binds the return address to the csp's depth.
- csp ← csp+8 (two words increment).
- csp_depth ← csp_depth+1.

To RETURN:

- csp ← csp−8.
- csp_depth ← csp_depth−1.
- Decrypt the return address, consumming 64 bits on the call stack: $a_1||csp\_depth_{verif} \leftarrow E_k^{-1}(M[csp])$.
- Verify the csp match: $csp\_depth_{verif} ==? csp\_depth$.
- If we have a match, proceed with the jump to a_1.

Long jumps. By removing JALR, we prevent the possibility to do long jumps, outside of the reach a JAL instruction. The new JAL jump range is ±32 MiB (4 bits taken from the rd register in Recommendation 7 and 1 bit gained from Recommendation 5).

Long jumps can be obtained by, for example, modifying the `LUI x0, imm` instruction to store the immediate value into a temporary xLJ register. At the next JAL instruction, the jump destination is the combination of the xLJ value and the JAL own immediate value. At the same time, the xLJ value is reset to 0.

But do we really need long jumps? If we implement Recommendation 10, the impossibility to do long jumps is a soft limit on the size of a compartment, which can be a good security property.

4.3 Adding a dispatch instruction

Removing the JALR instruction can really impact the performances of dispatchers: when the program must branch to numerous different instructions depending on a data value. Previously, the JALR instruction was used for this, but if we follow Recommendation 6 that is not possible any more. The alternative is the inefficient pattern of cascading branching instructions.

To recover the lost performances, we can, as in [10], introduce a new DISPATCH instruction.

Recommendation 8 *Dispatch instruction*
Introduce a `DISPATCH rs1, imm` instruction. Where rs1 is the decision register and imm is an unsigned immediate value that gives a bound to the dispatch.
If the DISPATCH instruction is at address a_0 then the program branches to $a_0 + 4 + 4 * rs1$ if $rs1 < imm$. If $imm \leq rs1$, then the program branches to the "error" address $a_0 + 4 + 4 * imm$.

5 HARDWARE SECURITY CONTEXTS

The microarchitecture has been shown to be the source of numerous information leakages. In 2005, Bernstein [4] proposed a technique to exploit cache timings to recover an AES cryptographic key. He showed how the key can be deduced from the timings of array lookups. Later, Percival showed in [19] that since different hardware threads share the same cache memory at some level, it becomes possible to spy on one thread from another one. They were the first to reveal the isolation issue at the hardware level. Unfortunately, this problem was not taken seriously and was even amplified by the addition of multiple performance-oriented mechanisms directly exploiting shared information (speculation, branch prediction ...). Finally, this led to the publication in 2018 of the famous Spectre [11] and Meltdown [15] attacks, but also of their multiple variants [5, 16, 21, 23] showing that the whole modern microarchitecture is impacted.

Vulnerabilities on devices of multiple manufacturers [11, 13, 14] have shown that the isolation issue is not just the result of poor implementation choices, but is a deeper problem that requires rethinking the architecture. Depending on the executed application, we need to make the software able to send specific security informatio to the hardware. But it also has to preserve the basic abstraction role of the ISA, by keeping a clean and simple interface.

5.1 The problem of leaky abstractions

Considering security, from the software point of view, the abstraction provided by current ISAs can be a real burden. If it greatly simplifies the functional vision of the system, it does not take into account the multiple physical phenomena or other artificial mechanisms exploited by attackers.

Timing variations, fault injections or speculation mechanisms are all existing possibilities on the hardware side but totally hidden on the software side. Current software is completely blind to threats targeting the microarchitecture since they are out of the scope of the ISA's semantics.

One way to solve this issue is to radically change that by adapting the ISA to give it the full responsibility of hardware micromanagement as proposed in [9]. Dedicated instructions can be added to be able to flush independently cache memories or prediction tables, create speculation barriers *etc.* However, if this represents an interesting approach in the short-term to be able to patch the system at any moment (the way that the software manages the hardware can be easily changed by modifying the code), it represents a dangerous approach in mid to long-term.

Modern CPU microarchitectures can be extremely heterogeneous, from a very simple pipeline to multithreaded cores with many different performance-oriented mechanisms. First, managing hardware security at the software-level involves being able to support all these different cases with the ISA. For that purpose, each mechanism must have dedicated instructions, which totally breaks the abstraction role of the interface. The final result of this approach is that software must be designed depending on the targeted microarchitecture. Each future target change will require more or less significant modifications to be secure: portability of secure application becomes impossible. Second, considering the whole system security (both hardware and software issues) only at the software-level directly impacts the code size. Increasing the software complexity opens the way to more mistakes by developers. In addition, the new micromanagement instructions offer new possibilities for the attackers, increasing the attack surface.

Finally, a clean interface between hardware and software helps to simplify security problems, potentially allowing automatic verification of security properties. From that point, we can deduce the following recommendation.

Recommendation 9 *No micro-architecture management*
Forbid all micro-architecture management instructions, cache management in particular.

However, it is clear that we must have a tool to guide these micro-architectural operations. We introduce now a new way to talk about security in the ISA by using an implementation-independent concept.

5.2 Managing security domains with HSCs

We call hardware security context (HSC) a domain where all executed operations respect the rules associated with this domain. These rules define a security policy which deals with different properties: isolation, confidentiality, integrity *etc.*

The notion of security policy is essential to be able to efficiently design secure applications: it gives an implementation-independent information to the hardware. And since a security policy is an abstract concept usable by the software independently of how it is really implemented, a liberty of implementation is granted to the hardware developer.

At the ISA level, these security policies are defined with instructions. From this analysis and the previous leaky abstraction issue, we can deduce one more recommendation about micro-architectural guarantees:

Recommendation 10 *HSC instructions*
Micro-architectural security guarantees must be provided through the HSC instructions described in this section.

5.3 RISC-V HSC extension proposal

This proposal can be seen as an extension over our base RV32S ISA.

5.3.1 Single hart. We consider the simplest case where only one hart (also named hardware thread) is available for the whole system. Therefore, for the software, there is only one HSC at a time.

HSC data structure. Each HSC is composed of at least two pieces of data: a cryptographic key and an entry point. The key is generated, managed and used by the hardware and never directly accessible by the software. It is used as the HSC identifier: two HSCs with different keys are different and must be isolated. The entry point represents the only program counter (PC) allowed to enter the corresponding HSC.

Additional data can be added to this structure. To illustrate such a use case, we add a capability register where each bit corresponds to the ability to perform or not some operations in the given context, as shown on Table 1. On the same model, other data could be tied to the HSC: more capabilities, memory ranges *etc.*

Table 1: Operations on HSC configuration data structures

Configuration	`hscconf0`	`hscconf{1..+}`
`Key`	Return hash	Return hash | Load | Generate
`Entry point`	Read-only	Read | Load | Write new
`Capabilities`	Read-only	Read | Load | Write less

During execution, these data structures are locally stored in dedicated registers: we call them HSC configurations (or `hscconf`). As an example, we arbitrarily consider that we have only two `hscconf` registers available. Each `hscconf` is composed of the same three pieces of data described previously: key, entry point and capabilities. The first register, `hscconf0`, corresponds to the properties of the currently executed HSC. The properties of an HSC cannot be changed during its execution: `hscconf0` is read-only. The other `hscconf1` register defines an HSC that can be modified with the adequate rules and switched to.

Accessing and modifying a `hscconf` *register.*
First, `HSCREAD rd, hscconfs1, offset` is used to read a value inside configuration register `hscconfs1` with `offset` defining the specific element of the data structure (*e.g.* 1 for key, 2 for entry point or 3 for capabilities). The key must not be directly accessible by the software. To still be able to identify it, a 32-bit hash of the key is returned instead of the key itself.

Different instructions are then used to modify the configurations. `HSCGENKEY hscconfd` is used to generate a new key, with `hscconfd` different from 0. After that operation, the `hscconfd` register is also tagged as *new*: it is writable. Its capabilities are set with the same value as the current capabilities in `hscconf0` and can only be reduced.

`HSCWENTRY hscconfd, rs1` is used to change the entry point of a configuration tagged as *new*: `rs1` contains the corresponding address. Lastly, `HSCWCAP hscconfd, rs1` is used to modify the capabilities of a configuration tagged as *new*: `rs1` contains the capability values that are verified against the current context capabilities. A register is no longer *new* when it is stored or a switch occurs.

Storing and loading. The number of different existing HSCs in the whole system is not fixed and can be very important: all the HSC data configurations cannot be simultaneously contained in registers. We need to be able to perform specific memory accesses to store and load HSC data. `HSCSTORE hscconfs1, offset(rs1)` is used to store an HSC in memory, at address `rs1+offset`.

In the same way, `HSCLOAD hscconfd, offset(rs1)` is used to load an HSC from memory. Because the memory alone is not a safe place to store sensitive data, some measures must be implemented to ensure that loaded configurations from memory are valid and have not been modified. Different strategies are possible: dedicated memory range to store HSCs, memory access control, stored data encryption using the HSC key …

Context switching. Finally, we need to change the current executed HSC: we can introduce `HSCMV hscconfd, hscconfs1`. This instruction is used to transfer all values contained in `hscconfs1` to `hscconfd`. Both `HSCLOAD` and `HSCMV` are able to switch the context by modifying the `hscconf0` register when `hscconfd` = 0. During an HSC switch, the different `hscconf` registers are not cleared, so they can be used to transfer HSC data to the next HSC. Another new register, `prevhsc`, can be used to automatically store the 32-bit hash of the HSC caller key.

Now, assume we want to create a new HSC and switch to it. `hscconf1` is used to configure it, `x1` contains its entry point and `x2` its capabilities. A possible code sequence is

- `HSCGENKEY hscconf1` to generate a new key.
- `HSCWENTRY hscconf1, x1` to write the entry point.
- `HSCWCAP hscconf1, x2` to set the capabilities.
- `HSCMV hscconf0, hscconf1` to switch to the new HSC.

5.3.2 Multiple harts and cores. Modern systems are a bit more complex than a simple hart running: they can be numerous, distributed within several cores. We consider here a static model where an HSC, to safely execute more instructions simultaneously, have to explicitly request it. Then, the hardware has to verify that this request is possible, depending on the security policy. We need to introduce a notion of parallelism to correctly use all the available resources. For that purpose, we introduce two new instructions for HSC management which assume that all resources are interconnected and support HSC.

`HSCSTART rd, hscconfs1` is used to create a new instance of the HSC in the `hscconfs1` structure by launching it in a new concurrent hart. This instance can be launched anywhere at the hardware-level, depending on the implementation, as long as its security policies and capabilities can be used there. The necessary resources are then statically allocated and associated to the corresponding HSC. If the operation is possible and successful, then `rd` = 0 else `rd` = 1. As a complement, `HSCEND rd` is used to end the current hart. It allows to statically release resources owned by an HSC, making them available to execute a hart with another HSC. If the operation fails, then `rd` = 1 else nothing happens due to the HSC termination.

These two instructions deviate significantly from the traditional approach of having always-on harts. But in the post transient execution attacks world, that is not possible anymore. The only other possibility is to forbid resource sharing for different HSCs, simultaneous multithreading becomes impossible if it involves more than one HSC.

5.3.3 Secure property extension. Naturally, HSC support provides all the needed information to allow isolation: each executed instruction or data handling is performed in an HSC. The implementation must ensure that clear barriers between the HSC are respected at the hardware-level. But more importantly, HSC support provides interesting tools to ensure other security properties.

To offer privacy of an HSC, the key provided by each one can be used to encrypt all code and data stored in memory. It ensures that only an HSC, with its key, can access this information. In section 4, we talked about CFI issues and why we need to banish indirect jumps. HSC support offers a secure alternative with the possibility to create a new HSC with a particular entry point before switching.

6 CONCLUSION

In this position paper, we propose to modify the ISA in order to offer strong security guarantees. With the introduction of hardware security contexts (HSCs), we have an ISA isolation primitive that both hardware and software can rely on. Furthermore, we can use HSCs to build other protections such as restricting indirect jumps to HSC switches, or binding confidential data to an HSC using confidential registers.

Work is needed to validate the different proposals and to precisely understand the underlying trade-offs, in particular for the micro-architecture, for the compilers and operating systems.

ACKNOWLEDGMENTS

We thank the Direction générale de l'armement (DGA) for its financial support.

REFERENCES

[1] Accessed 2020-04-20. https://github.com/WebAssembly/design/blob/master/Rationale.md
[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (2009), 4:1–4:40. https://doi.org/10.1145/1609956.1609960

[3] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2019. Formal Verification of a Constant-Time Preserving C Compiler. *Proc. ACM Program. Lang.* 4, POPL, Article 7 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371075

[4] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).

[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2018. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *arXiv:1811.05441 [cs]*. http://arxiv.org/abs/1811.05441 arXiv: 1811.05441.

[6] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: hardware-assisted flow integrity extension. ACM Press, 1–6. https://doi.org/10.1145/2744769.2744847

[7] Ruan de Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen de Bosschere, Bart Preneel, Bjorn de Sutter, Ingrid Verbauwhede, and KU Leuven. 2016. SOFIA: Software and Control Flow Integrity Architecture. (2016), 6.

[8] T. F. Dullien. 2017. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing* (2017), 1–1.

[9] Qian Ge, Yuval Yarom, and Gernot Heiser. 2018. No Security Without Time Protection: We Need a New Hardware-Software Contract. In *Asia-Pacific Workshop on Systems (APSys)* (2018-8-27). ACM SIGOPS, Korea, 9. https://doi.org/10.1145/3265723.3265724

[10] Alexandre Gonzalvez and Ronan Lashermes. 2019. A Case against Indirect Jumps for Secure Programs. In *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering* (San Juan, Puerto Rico) *(SSPREW9 '19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/3371307.3371314

[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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1–19. https://doi.org/10.1109/SP.2019.00002

[12] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[13] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clementine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX, USA, 549–564.

[14] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. 13.

[15] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[16] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. 2019. Fallout: Reading Kernel Writes From User Space. 14.

[17] David Patterson and Andrew Waterman. 2017. *The RISC-V Reader: An Open Architecture Atlas.* Strawberry Canyon.

[18] David A Patterson and John L Hennessy. 2017. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface.* Morgan Kaufmann.

[19] Colin Percival. 2005. Cache missing for fun and profit. (2005).

[20] 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

[21] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 753–768.

[22] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. (2014), 15.

[23] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *40th IEEE Symposium on Security and Privacy (S&P'19)*. San Francisco, CA, USA, 18.

[24] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 457–468. https://doi.org/10.1109/ISCA.2014.6853201

[25] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/data-oblivious-isa-extensions-for-side-channel-resistant-and-high-performance-computing/