Recommendations for a radically secure ISA

Mathieu Escouteloup [1] Jacques Fournier [2] Jean-Louis Lanet [1]
Ronan Lashermes [1]

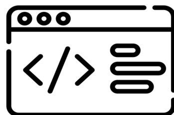
[1]INRIA [2]CEA Leti

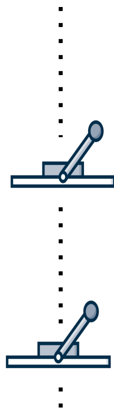
May 30th, 2020

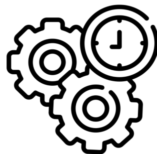
CARRV

The problem

SOFTWARE **HARDWARE**







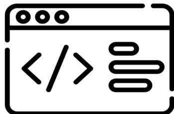
[1]

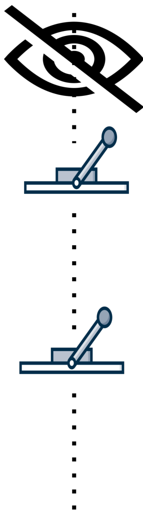
[1] Icon made by Freepik from www.flaticon.com

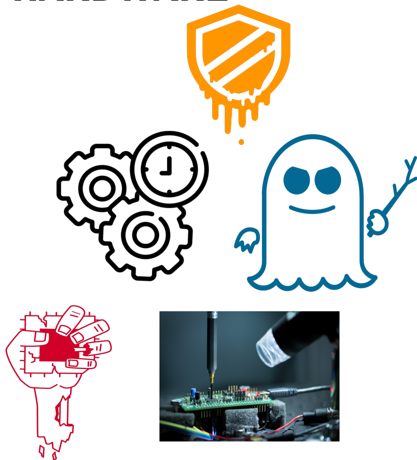
The problem

SOFTWARE

HARDWARE







Content

1. Confidential registers
2. Stricter control flows
3. Hardware security contexts
4. Wrap-up

Table of contents

1. Confidential registers
2. Stricter control flows
3. Hardware security contexts
4. Wrap-up

Ensuring confidentiality at the ISA level

Writing programs handling confidential data is hard

- As a developer, we must prevent any timing dependency on the confidential data.
- Compilers are not our friends.
- Timing dependency is not the only possible leakage depending on the context (power consumption, EM radiation, . . .).

There is no way for the developer to tell the machine: **"Take care, this value must be kept confidential !"**.

This issue should be addressed by the ISA

Confidential registers

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 a hardware security fault.

Consequences of confidential registers

Implementation liberty

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...

Confidentiality boundary

Confidential registers enable compilers to map confidential variables to confidential registers with a guarantee that the hardware won't mess things up.
In particular, the confidentiality boundaries, when a confidential data is made public, become trivial to detect.

Table of contents

1. Confidential registers
2. Stricter control flows
3. Hardware security contexts
4. Wrap-up

Control flow integrity

Abusing the control flow

The ability to abuse the control flow while keeping static integrity is a well-known threat from return-oriented programming (ROP) and its variants.
Numerous control flow integrity (CFI) solutions have been proposed both in software and hardware. Only hardware solution can withstand the most powerful attackers, but to be effective **the hardware must be able to discriminate legal control flows from malicious ones**.

For effective hardware CFI, the control flow graph must be static. Therefore forward indirect jumps should be forbidden.

Removing forward indirect jumps

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

Consequences

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

- Dispatcher patterns become costly: we need to add a new `DISPATCH` instruction.
- It becomes impossible to transfer the control flow to a different program.
- It prevents to simply return from a procedure.
- There is no possibility to perform a direct long jump.

Returns

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.

How to implement this call stack and how to mitigate the other consequences of removing the indirect jumps are in the paper.

Table of contents

1 Confidential registers

2 Stricter control flows

3 Hardware security contexts

4 Wrap-up

Split or flush

Definition: security domain

A security domain is the set of elements respecting the same security policy (isolation, confidentiality, integrity, etc.).

Lesson from past attacks

The big lesson of the various recent attacks is that it is not possible to share data from different security domains into the same micro-architectural resource.
Either the resource must be **flushed** before accepting another input, or the resource must support being **split** into isolated sub-resources.

The hardware has no support for security domains. The software cannot tell to the hardware where the domain boundaries are.

Ensuring portability

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

Since each CPU has its own resources that may be split or must be flushed, adding instructions for the management of these elements creates a portability issue.

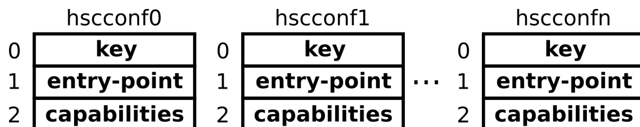
Recommendation 10 *Hardware security context (HSC) instructions*
Micro-architectural security guarantees must be provided through the HSC instructions described in this section.

Instead, add instructions to delimit domains and let the hardware handle the consequences.

HSC extension - single hart

Single hart

In a single hart system, there is always only one HSC at a time.



HSC data structure

Everything is store in dedicated registers called HSC configurations (`hscconf`) which have 3 fields:

- Key: a unique cryptographically secure random value that can be derived to identify the HSC.
- Entry point: the entry address for this HSC.
- Capabilities: registers to store what can be done by this HSC.

Configuration handling

Configuration addressing

`hscconf0` corresponds to the current HSC: its fields cannot be modified.
Other configurations have dedicated registers `hscconf1`, `hscconf2`, ...

Accessing and modifying

- `HSCREAD rd, hscconfs1, offset` to read a value.
- `HSCGENKEY hscconfd` to generate a new key and set an HSC as *new*.
- `HSCWENTRY hscconfd, rs1` to modify an entry-point of a *new* HSC.
- `HSCWCAP hscconfd, rs1` to modify capabilities of a *new* HSC.

Loading and storing

- `HSCSTORE hscconfs1, offset(rs1)` to store an HSC in memory.
- `HSCLOAD hsccond, offset(rs1)` to load an HSC from memory.

Context switching

Move

- `HSCMV hscconfd, hscconfs1` to transfer all values from a configuration to another one.

Switch

Both `HSCMV` and `HSCLOAD` can overwrite `hscconf0` and therefore trigger a context switch.

- Upon a switch, the microarchitecture can detect the security domain change with the key register and act accordingly: split or flush.
- The switch is also a forward indirect jump since the program counter is set to the new domain entry point: forward indirect jumps are only authorized when the security domain changes.

Multiple harts

Multiple harts sharing the same micro-architectural resources must be statically allocated to avoid port contention attacks in particular.

No more always-on hart

A hart must now be initiated as a fork of another one and it must be ended to definitively free the corresponding resources

Hart start/stop

- `HSCSTART rd, hscconfs1` is used to create a hart with context `hscconf1`. `rd` is set to 0 if the static allocation was successful and the new hart started.
- `HSCEND rd` is used to end the current hart and possibly releasing all corresponding resources.

The other alternative is to forbid resource sharing inside a core: all the harts of the same core must be in the same HSC.

Table of contents

1 Confidential registers

2 Stricter control flows

3 Hardware security contexts

4 Wrap-up

Wrap-up

1. We need a new class of CPUs, the secure one,
2. around a new ISA with modified semantics.
3. The new ISA must enable the software to communicate security properties to the hardware (confidentiality, HSCs, . . .).
4. Indirect jumps imply to switch to a new security domain.
5. We can combine the ISA changes for new guarantees: *e.g.* automatic memory encryption.

Thank you for listening!

What is your take on this matter ?