Architectural Extensions for a RISC-V Security Processor

Tariq Kurd
CPU Architect
Huawei, UK
Tariq.Kurd@Huawei.com
What is a Security CPU?

A security CPU defends against...

- **Hardware attacks, e.g. fault injection**
  - Try to get the CPU to behave in an incorrect manner, such as to take an incorrect branch, typically to enable a software attack

- **Software attacks, e.g. Code Reuse Attacks (CRAs)**
  - Maintain CFI (control flow integrity) by preventing the attacker to run their own Turing complete code by reusing code already in the memory or ROM

- **Side channel attacks, e.g. power analysis**
  - Try to gain secret information such as algorithmic details or even encryption/decryption keys
What is a Security CPU?

<table>
<thead>
<tr>
<th>It defends against hardware attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inject a fault into the CPU</td>
</tr>
<tr>
<td>• EMFI (electromagnetic fault injection)</td>
</tr>
<tr>
<td>• Shoot flip-flops with a laser</td>
</tr>
<tr>
<td>• Glitch the power supply</td>
</tr>
<tr>
<td>• Unpredictable behaviour at low or high voltage (out of spec)</td>
</tr>
<tr>
<td>• Glitch the reset line – partially reset the CPU</td>
</tr>
<tr>
<td>• Modify the hardware</td>
</tr>
<tr>
<td>• Bridge wires together to force logic values</td>
</tr>
<tr>
<td>Why do this?</td>
</tr>
<tr>
<td>• Maybe I can flip the CPU from User mode into Machine mode by changing the state of one flip-flop…</td>
</tr>
<tr>
<td>• Or maybe I can make the CPU take an incorrect branch, such allowing a signature check to pass which should have failed</td>
</tr>
</tbody>
</table>
What is a Security CPU?

It defends against code reuse software attacks (CRAs)

- Return Oriented Programming (ROP) and Jump Oriented Programming (JOP).
  - Scan the memory for gadgets
  - Call gadgets instead of legitimate code by corrupting the execution state and taking control
  - Jumping to code sections which end in a return in ROP, and controlling the return address to jump to the next gadget
  - Jumping to code sections which end in an indirect branch in JOP
- Implement a Control Flow Integrity (CFI) scheme to detect the program flow has been subverted
  - No further details in this presentation
What is a Security CPU?

- Monitor the power dissipation to gain secret information

- This is a simplistic example, but it shows the basic problem.
- The encryption/decryption key can be read from the power trace

(image from slideplayer.com)
**Why choose RISC-V?**

**RISC-V vs “off the shelf” solution**

- The standard solutions for fault detection is lock-step execution
  - Run two copies of the CPU in lock-step
    - possibly with delayed execution between them
  - Fault detection is based on comparing the output
  - If there is any difference then **kill** the system

```
Core #1  =  Core #2

0 to 2 cycle fixed delay

kill the system on output mismatch
```
### Why choose RISC-V?

<table>
<thead>
<tr>
<th>RISC-V vs “off the shelf” solution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lock-step checking limitations</strong></td>
</tr>
<tr>
<td>- What if the attacker locates the Machine/User mode flip-flop?</td>
</tr>
<tr>
<td>- Then he can shoot the same flip-flop in both copies</td>
</tr>
<tr>
<td>- The constant delay makes it much easier for this attack to succeed</td>
</tr>
<tr>
<td>- Also the power dissipation peaks / troughs all double</td>
</tr>
<tr>
<td>- …it’s better than a single copy but not so secure…</td>
</tr>
</tbody>
</table>

![Diagram showing Core #1 and Core #2 with M/U mode flip-flop](image)
Why choose RISC-V?

Let’s find a better solution… for an execution checker…

- **What are the requirements?**
  - Check the CPU execution is correct, small attack “window”
    - Don’t wait for a core output mismatch, detect faults much more quickly
    - Kill the system as soon as a mismatch is found
  - Different execution model to check the core is correct
    - For example, work backwards from divide results using the multiplier
      - Execute:  \( A / B = D \mod R \) (variable latency)
      - Check:  \( D \times B + R = A \) (no iteration required)
  - Different attack “surface”
    - a successful attack on one core should not succeed on the checker core
  - Different power profile
    - Don’t double all the peaks and troughs
    - Obfuscate the power profile instead
  - Low area
    - We don’t need to duplicate all functionality just for checking, do we?
    - For example, do we really need all the MPU CSRs in both cores?
Why choose RISC-V?

RISC-V offers implementation freedom to build the Shadow Core

- RISC-V offers complete freedom to implement *whatever we need*
  - The Shadow Core is the execution checker core
  - It meets all the requirements from the previous slide
  - The need for the shadow core drove us to move rapidly to RISC-V
  - Because we can’t build the shadow core using a licensed architecture…
  - …because it’s not a full implementation

![](image)

- M/U mode flip-flop
- Completed insns, variable delay
- **kill** the system on mismatch
Why choose RISC-V?

RISC-V offers implementation freedom

- The shadow core monitors the main core
  - It probes input signals and also completed instructions
  - It re-executes the instruction stream differently and checks the results
  - The M/U mode CPU state is implemented differently complicating the attack
  - Variable delay makes the attack even harder
  - Check execution state after every instruction
  - E.g. if the main core is in M-mode but the shadow core is in U-mode then kill the system

The shadow core is covered by European Patent PCT/EP2019/080874
Architectural extensions - PMP

RISC-V offers freedom to modify the architecture based on our requirements

• The standard PMP doesn’t meet our customer requirements
  • The customer requires fully separate permissions for M-mode and U-mode
  • Therefore U-mode code, which is not fully trusted cannot accidentally, or maliciously, be executed in M-mode
  • Therefore U-mode code/data must be able to exclude M-mode privileges
    • The standard PMP spec
      • can only remove M-mode permissions by locking the page
      • Does not allow U-mode permissions without also allowing M-mode
  • Also, 16 PMP entries aren’t enough for the application
  • And… memory must not be fully accessible if the PMP is disabled
  • So we need to change the architecture…and we can…
<table>
<thead>
<tr>
<th>Architectural extensions - PMP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RISC-V offers freedom to modify the architecture based on our requirements</strong></td>
</tr>
<tr>
<td><strong>• Add PMPMRWX CSRs – for M-mode RWX permissions</strong></td>
</tr>
<tr>
<td>• Same format as PMPCFG – one 8-bit field per MPU entry</td>
</tr>
<tr>
<td>• Each 8-bit field has only 3 bits – R, W and X</td>
</tr>
<tr>
<td>• M-mode permissions come from PMPMRWX</td>
</tr>
<tr>
<td>• U-mode permissions come from PMPCFG</td>
</tr>
<tr>
<td>• If the region is locked in PMPCFG then the same field is locked in PMPMRWX</td>
</tr>
<tr>
<td>• Locking the entry freezes the permissions – <em>it doesn’t modify them</em>….</td>
</tr>
<tr>
<td>• <em>Software has freedom to program the permissions independently</em></td>
</tr>
</tbody>
</table>
Architectural extensions - PMP

RISC-V offers freedom to modify the architecture based on our requirements

- PMP CSR fields

<table>
<thead>
<tr>
<th>CSR field</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmpcfg.L</td>
<td>Lock config</td>
<td>doesn’t effect M-mode permissions</td>
</tr>
<tr>
<td>pmpcfg.A</td>
<td>Address mode</td>
<td>Standard behaviour</td>
</tr>
<tr>
<td>pmpcfg.X</td>
<td>U-mode X</td>
<td>doesn’t effect M-mode permissions</td>
</tr>
<tr>
<td>pmpcfg.W</td>
<td>U-mode W</td>
<td></td>
</tr>
<tr>
<td>pmpcfg.R</td>
<td>U-mode R</td>
<td></td>
</tr>
<tr>
<td>pmpmrwx.X</td>
<td>M-mode X</td>
<td>doesn’t effect U-mode permissions</td>
</tr>
<tr>
<td>pmpmrwx.W</td>
<td>M-mode W</td>
<td></td>
</tr>
<tr>
<td>pmpmrwx.R</td>
<td>M-mode R</td>
<td></td>
</tr>
</tbody>
</table>
Architectural extensions - PMP

<table>
<thead>
<tr>
<th>PMP example which is not possible with the standard specification</th>
</tr>
</thead>
</table>

- User mode untrusted software can execute without M-mode permissions
  - Code which can only execute in U-mode
    - \texttt{pmpmrwx = RWX:0;} // no M-mode permissions
    - \texttt{pmpcfg = L:0, RX:1, W:0;} // read / execute U-mode permissions
- User mode untrusted software data area
  - Data which can only be accessed in U-mode
    - \texttt{pmpmrwx = RWX:0;} // no M-mode permissions
    - \texttt{pmpcfg = L:0, RW:1, X:0;} // read/write U-mode permissions
- We considered a limited set of U/M-mode permission possibilities
  - We decided that it is safer (and simpler) to allow the full set of combinations, to cover scenarios we hadn’t considered
Architectural extensions - PMP

RISC-V offers freedom to modify the architecture based on our requirements

- PMP Memory access permission defaults to being closed
  - The reset PMP state is to open a small window of executable memory
  - Boot software is required to reprogram the PMP registers first
  - This prevents attackers from booting from other locations
- We also increased the PMP entries up to 20 for finer control
  - *So we changed the architecture! Ideally these features would be options in the standard…*
## Architectural extensions

<table>
<thead>
<tr>
<th>RISC-V offers implementation freedom to add features</th>
</tr>
</thead>
</table>

- **STOP custom instruction**
  - If software detects an error condition execute STOP to kill the CPU
  - *We also added a lot of custom instructions to reduce code size (no further details here)*

- **Custom bus interfaces**
  - Enhanced bus interfaces with extra signals for security
Architectural removals

RISC-V offers implementation freedom to remove features

- **Debug**
  - Blow an OTP fuse to cause the Security CPU to be in *production* mode
  - In production mode any debug access *kills* the CPU
  - Standard licensed architectures require full debug support at the core level
    - Typically lock the JTAG / debug port at the SoC level
    - It’s *much* safer to disable debug mode throughout the core
    - Secrets don’t stay secret if the attacker can access the core through the debug port....
Architectural removals

RISC-V offers implementation freedom to remove features

• Code Profiling
  • We don’t want an attacker to be able to profile code that’s running on the core
  • This would give away secrets so we don’t implement any performance monitors

• NMI
  • Dangerous…… don’t let the CPU be interrupted when executing secret code
  • We remove this completely….. Don’t let customers try to connect it!
## Preventing side-channels

<table>
<thead>
<tr>
<th>Low susceptibility to side-channel attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Main core and shadow core execute code differently</td>
</tr>
<tr>
<td>• Both have a different power profile</td>
</tr>
<tr>
<td>• For example, one could have out-of-order writeback and one could have in-order writeback</td>
</tr>
<tr>
<td>• Variable execution power</td>
</tr>
<tr>
<td>• Dynamically modify the execution power so that the power trace is not consistent between loop iterations</td>
</tr>
<tr>
<td>• Randomise the CPU behaviour to dynamically change the power profile</td>
</tr>
<tr>
<td>• Consistent instruction timing</td>
</tr>
<tr>
<td>• Avoid timing attacks</td>
</tr>
</tbody>
</table>
Summary

Using RISC-V has let us be successful against:

<table>
<thead>
<tr>
<th>Hardware attacks</th>
<th>Software attacks</th>
<th>Side-channel (e.g. power analysis) attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Not forced to use lockstep execution, so get better error detection with lower area</td>
<td>• Hardware CFI scheme implemented (our custom design, not available in standard solutions)</td>
<td>• Not using lockstep execution makes power analysis resistance much better, because we have freedom to have different hardware implementations for the two cores with different power profiles</td>
</tr>
<tr>
<td>• Freedom to add parity / CRC checks as required throughout the implementation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thank you.