Integration of RISC-V Security Standards in the HIGHER project

Nick Kossifidis (FORTH) <mick@ics.forth.gr>

June 12, 2026

https://higher-project.eu

The story so far...

Filtering outgoing memory accesses

By execution context:

- From U-mode: MMU or sPMP (WiP
- From {H/V}S-mode: MMU and/or {e}PMP
- From M-mode: {e}PMP

By Supervisor domain ID (WiP):

- Using Smmpt extension

- Synchronous exceptions
- {e}PMP -> Mostly static / boot-time configuration
- MPT/MMU -> Dynamic configuration at runtime
- sPMP -> Mostly for embedded systems, mutually exclusive with MMU (somewhat dynamic)
- s/{e}PMP -> Register interface, limited number of rules
- MMU/MPT -> In-memory data structures

Filtering incoming memory accesses

Based on process id:

- IOMMU

Based on supervisor domain id:

- IOMPT Checker

Based on source id / RISC-V World ID (WiP):

- IOPMP/WorldGuard Checker

Other security controls/checks

Control Flow Integrity

- Shadow stacks
- Landing pads with label

Memory tagging (WiP)

Hardware Fault Isolation (HFI) (WiP)

About trust...

In the x86 era:
SoC provider -> Intel/AMD
In the arm64 era:
SoC provider -> Cloud provider (e.g. Graviton, Cobalt 100, Axion)
In the RISC-V era:
SoC provider -> everyone, maybe the attacker too !

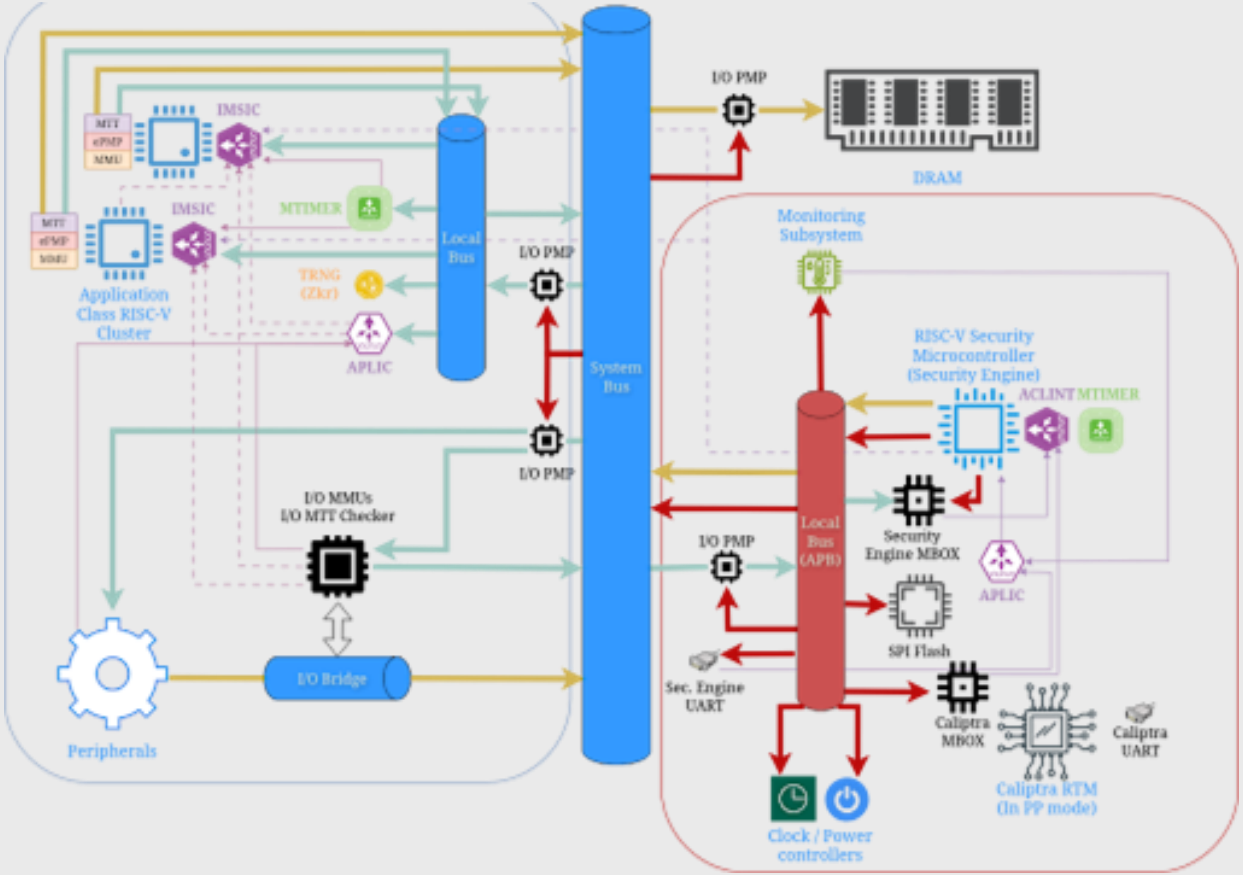
When everything was controlled by Intel/AMD, RoT / attestation was provided by them (e.g. fTPM, SGX etc), secure boot through UEFI (MIcrosoft holds the keys, the rest go through shim) etc. If we don't trust the cloud/service provider, we rely on Intel/AMD/Microsoft.

When we switch to a more open ecosystem, establishing trust in the cloud is much harder, this is still an open issue and we don't have a good answer...

About trust...

A part of the solution is to have a reference RoT platform...

Putting it all together



Steps towards confidential computing...

Phase 1: Software-based approach, no extra hardware

MCU comes out of reset, and its BootROM

- Fetches its firmware image from flash/network
- Unpacks and verifies it using a hardcoded built-in public key hash
  - Why hash: Fixed size regardless of key type
  - Public key is attached to the firmware image, image also acts as a certificate
- We'll use ED25519 for signing/measuring as Open DICE suggests in the software implementation.
- Following the Open DICE profile
  - From the Unique Device Secret (UDS) and the input values mentioned in the profile, it will generate two Compound Device Identifiers (CDIs), one for attestation (CDI0_att) and one for key sealing (CDI0_seal).
  - Using the UDS it'll generate a DeviceID key pair (DID_priv/pub) and will generate a CDI certificate CDI0_cert following the CBOR format (to avoid asn.1/x509). It will also generate a self-signed certificate DID_cert that includes DID_pub and is signed with DID_priv (since this will be fixed we can just pre-calculate it and append it to the BootROM image), that would act as our ECA certificate.
  - For now UDS and all inputs except the MCU Firmware's hash (that acts as a TCB Component Identifier - TCI) will be hardcoded in BootROM.
- Using PMP (CVA6 has support) we make sure memory is properly protected when doing signature verifications etc, and after we are done with everything, we clean up memory and block access to BootROM's code using the Lock bit, we'll only leave CDI0_att/seal/cert, and DID_cert behind on a predefined memory region.
- Jumps to the firmware.

Phase 1: Software-based approach, no extra hardware

- MCU firmware
  - Since our current MCU firmware is Linux, we can use the available Open DICE driver to expose CDI0_att/seal/cert and DID_cert to userspace (/dev/opendice-0). Firmware will then grab DID_cert, verify its signature, and then verify CDI0_cert's signature with DID_pub.
  - It then fetches any other firmware images from flash/network, verifies their signatures (same image format as BootROM, we'll mostly reuse the same code), unpacks them and brings them out of reset etc.
  - For the boot image of the application-core cluster it generates CDI1_att/seal/cert using CDI0_att, places them to a memory region (together with DID_cert and CDI0_cert) and adds that region to the provided device tree so that the Linux driver for Open Dice on the application cores uses it.
  - Since we'll use a combined image that includes both OpenSBI and the kernel that would work since CDI1 will describe the OS environment too.
- We are following a layered DICE architecture, without the presence of a Dice Protection Environment (DPE).
- At this point we'll have an environment compatible with the Open DICE profile, and we can demonstrate using e.g. CDI1_seal from Linux applications for key sealing. This approach is also used by recent Android phones and is probably the only currently implemented use case for DICE on Linux.

Phase 1.5: Adding secure storage and UEFI in the mix

- Instead of hardcoding things in BootROM we'll use **OTP storage**. On our way to use Caliptra we'll need to introduce Caliptra's Fuse Controller (coming from OpenTitan) anyway, so it's a good opportunity to go step-by-step and introduce the controller both in QEMU and the SoC design before going all-in.
    - We'll use OTP storage for the UDS, the fw vendor's public key hash, to track lifecycle (both Caliptra and Open DICE mention 8 stages -one byte in OTP-), platform provisioning data (e.g. MAC addresses etc), and maybe for other things as well.
    - The controller is included in Caliptra's emulator (written in rust), and the rtl code is also open source. The only difficult part is the fuse "macro" that's used for the actual fuses in ASIC (I have no idea how that works on the FPGA). In the actual hw this obviously can't be off-chip.
- This will allow us to work on provisioning processes that we can later re-use when we switch to Caliptra since Caliptra uses ~3kbits for its fuse map, we can go for an 8kbit fuse macro and use the remaining space.
- We'll also add 256KB of NV storage (e.g. an SPI flash), that we'll encrypt using AES GCM in the MCU firmware, with a key derived from CDI0_seal.
- This will allow us to provide secure storage to the platform, that we can at this point use to implement UEFI's secure boot variables. To do that we'll need to expose that storage via an SBI extension on top of RISC-V RPMI, and also propose that extension to become part of the SBI/RPMI spec. It's a great contribution opportunity, especially if we also contribute patches for the reference UEFI implementation (EDK2) and OpenSBI too.
    - Note that the RISC-V RPMI also has a hw component, already available on Linux, basically a mailbox.
- With the UEFI approach, since the UEFI firmware won't follow Open DICE, in the measurements / CDIs we'll get through /dev/opendice-0 the kernel won't be included, so it breaks attestation but we can still demonstrate key sealing.

Phase 2: Switching to Caliptra

- Caliptra provides a **Dice Protection Environment (DPE)** so we won't be using the layered DICE architecture anymore, and won't be handling CDIs and keys directly. Instead we'll use handles / contexts and the DPE API to request attestation/sealing keys, and certificates. The DPE will keep CDIs hidden, as well as the UDS etc.
- This also fixes an issue with CoVE that requires platform token to be signed using UDS, since we can ask Caliptra to sign it for us (the MCU Firmware, after we do the measurements etc) with it, without exposing it.
- The switch will reduce the code size and complexity of both the BootROM and the MCU firmware since all crypto operations (except for LUKS) will happen in Caliptra. Since we'll go for Caliptra 2.0, it'll also save us from reading the images and pushing them in chunks to the mailbox, since Caliptra 2.0 also supports DMA via AXI (and mailbox is there mostly for commands).
- We'll already have the OTP/Fuses hw parts ready from phase 1.5 so that Caliptra can use it as required, but we'll need to also add the Caliptra's mailbox / PQC KV SRAM, an APB bus to talk to it and other required blocks.
- We won't be using Open Dice anymore (since we won't be exposing CDIs), so we won't be able to demonstrate much on the Linux side, since there are currently no users for DPE, most tools/services, and the kernel itself currently use TPM 2.0. However we should expose Caliptra's DPE to the platform so that it can be used later on. So we can introduce a new RISC-V RPMI extension to talk to the DPE over SBI, and implement the proxy between that and Caliptra's mailbox interface in the MCU firmware. Note that there is also a standard interface from TCG to talk to a DPE in a similar way to a TPM, but that requires extra hardware. Fortunately DPE is a simple API compared to TPM 2.0, so the proxy wouldn't be that much to implement (and we'd need to implement it for the CRB approach - the TCG standard DPE interface- too anyway).
- With that interface in place we can then implement the EFI_CC_MEASUREMENT_PROTOCOL in the UEFI and have measured boot the UEFI way, that we can demonstrate. This currently assumes a TPM but we can showcase how to use a DPE instead by matching PCRs to DPE handles.
- Another addition would be to introduce a DPE driver for Linux and expose a /dev/dpe0, that can be used as a TPM alternative. If we have the time and resources we can even use that to e.g. demonstrate how it can be used by systemd-cryptenroll and LUKS2 to provide encrypted storage. This way we'll introduce the first DPE users on Linux, that would be a significant contribution.
- We'll use ECDSA with the standard 256 NIST curve, and maybe in the future go for PQC (just to say we are "quantum resistant").

Phase 2.5: Adding TPM 2.0 in the mix

- Since there are no DPE users on Linux but there are multiple TPM 2.0 users (e.g. IMA, SSH, dm-crypt/LUKS2 etc), we can expose a TPM 2.0 interface to support those users out of the box, until DPE becomes more popular.
- Implementing the TPM 2.0 API is a serious effort, even with the reference implementation in place. Even creating a proxy for it via SBI/RPMI could be a lot of work (but we can still explore that option I guess). Instead we can go for a simple quick-n-dirty workaround:
  - Create a TCG Command Response Buffer interface which is the expected way for platform users to talk to the TPM (this requires extra hardware, but it should be simple to implement), and Linux already has a driver for it.
  - Create a simple Linux driver on MCU firmware to proxy commands sent via the CRB to a socket.
  - Run the TPM 2.0 emulator (part of the reference implementation) on MCU, listening to that socket, using the LUKS-encrypted storage from phase 1.5 as a storage backend, transparent to the emulator.
  - Effectively we'll be implementing an fTPM.
- This can also be used by UEFI's EFI_CC_MEASUREMENT_PROTOCOL and EFI_TCG2_PROTOCOL.

Phase 3: RISC-V CoVE

- Hard truth: CoVE is still WiP
  - It's not ratified yet, it's still in draft state. At this point it's basically a moving target.
  - There is practically no sw support for it, other than an HS-mode TSM implementation called Salus, and some old off-tree kernel patches to use that from KVM. There is no M-mode TSM implementation available and even the SBI interface is not implemented in OpenSBI.
  - The spec is incomplete, it says it uses the layered DICE architecture but e.g. requires the platform token to be signed by UDS which in our case (where the RTM contains both Caliptra and the MCU) it can only work with a DPE.
  - The SBI extension is overengineered.
  - The current spec / ABI also doesn't support sealing, only attestation. Sealing is scheduled for the v2 ABI.
  - CoVE-IO is also in early development and there are not that many devices that support SPDM to play with anyway.
  - On the other hand that gives us many opportunities to contribute both to the spec and the sw ecosystem.

Phase 3: RISC-V CoVE

- Hard truth 2: CoVE requires significant effort also on the hardware side
  - Harts need to implement the Smmpt extension
  - We'll also need the I/OMMU to know about supervisor domain ids, and also have an I/OMPT checker on the side.
  - We'll need AIA support too, to be able to route interrupts to guests via IMSIC, and the relevant ISA extensions (e.g. indirect CSRs etc) also implemented.
  - On the other hand we'll need AIA and the I/OMMU anyway to support VMs properly.
  - Also even if we don't go for the CoVE API/ABI we'll still need Smmpt and I/OMPT checker to provide confidential VMs.
  - Smmpt is also not ratified yet but it's getting close.

Phase 3: RISC-V CoVE

- The plan:
  - Keep an eye for CoVE and participate in the development process, providing confidential VMs functionality in a clean way is a requirement for modern server / cloud environments. Let's work within the RISC-V technical committee to at least get Smmpt ratified so that we can add it to our platform and be future-proof.
  - Keep an up to date virtual prototype (a QEMU setup) to help in the development process on the software side, and exploration in general.
  - When the extension gets mature, we'll see the sw ecosystem evolve, Linux, QEMU and other projects require extensions to be ratified before they go upstream, also contributing to upstream is part of the ratification process.
  - Familiarize ourselves with the existing sw ecosystem, and also pKVM and projects from confidential computing consortium.
  - Treat this as a research topic.

Questions ?

Thank you !

https://higher-project.eu