ACE: Atomic Cryptography Extension for RISC-V

Roberto Avanzi [1], Ruud Derwig [2], Luis Fiolhais [3], Richard Newell [4], Barry Spinney [3], and Tolga Yalcin [1]

[1] Qualcomm, [2] Synopsys, [3] Independent Researcher, [4] Microchip

Abstract

We propose the Atomic Cryptographic Extension (ACE), an ISA extension which enables secure cryptographic implementations. ACE separates key configuration for use by software from key usage, allowing separated environments to perform these functions. For instance, setting the key can be delegated to a secure TEE applet. ACE also performs cryptographic operations in an atomic fashion (whence the name), in contrast to current round-based AES extensions, that by their nature cannot conceal the key. Keys are stored in Context Holding Registers *(CHR), architectural registers containing keys and metadata. However, in contrast to other architectural registers, CHRs cannot be read directly, and their contents can only be exported in encrypted, authenticated formats for reimport, enabling secure key usage across context switches and process migrations. ACE is work in progress of the High Assurance Cryptography (HAC) TG of RISC-V International.*

Introduction

Secure implementation of strong encryption is essential to many modern use cases. For instance, for content protection schemes it is desirable that even compromised processes cannot expose master keys, but, at worst, only individual content items. Secure cryptography is essential to identity management, communications, storage encryption, and emerging applications like ML model deployment to end-user devices.

Delegating cryptographic operations to Trusted Execution Environments (TEEs) can provide strong security for cryptographic operations. However, context switching, shared memory setup, and common execution restrictions in TEEs like disabled speculation or the use of non-cacheable memory introduce significant performance overheads. Even with hardware accelerators, TEE-based cryptography remains inefficient.

Modern CPUs commonly include cryptographic extensions, but they operate in a round-based fashion and cannot conceal keys. Thus, they fail to provide adequate security guarantees for sensitive applications. The missing elements are: (i) secure, local and *architected* key storage and (ii) state machines to safely implement cryptographic operations and protocols.

Architecture

Our approach allows software to perform cryptographic operations directly using protected keys that cannot be extracted or exposed. The *Atomic Cryptographic Extension (ACE)* implements it by providing the following functionality:

1. **Context Holding Registers (CHRs)** — Architectural registers that securely store cryptographic keys. Like integer and vector registers, CHRs are defined per-hart. Once configured with a key value, CHRs cannot be read to reveal that value directly. Beyond storing keys and their associated metadata, CHRs can maintain an internal state required by modes of operation or hashing.

2. **New CPU instructions:**

   (a) **Configuring a CHR with a key and associated metadata — or erasing and invalidating it.** The metadata cannot be set independently from the key, it binds the latter to a specific algorithm, and may set usage policies: For instance, a key may be used for decryption only, to prevent certain types of key misuse attacks.

   (b) **Atomic cryptographic operations using CHR-stored keys.** Some implementations may choose to provide only the AES and Galois Multiplication, while other may support more algorithms. The list of algorithm may also include threshold implementations hardened against side-channel attacks.

   (c) **Secure export/import of CHR contents as encrypted and authenticated *Sealed Cryptographic Contexts* (SCCs).** This is necessary for system software stacks to enable secure context switching. It also enables secure key request and delivery: For instance, a requesting process may ask a secure Applet to configure a CHR and export it to a SCC. Upon return, the requesting process be able to import the SCC into a

CHR any time it needs it, without having to switch contexts each time.
The key used to wrap and unwrap a SCC is called the *Context Transport Key* (CTK).

(d) **State management messages**, to let the algorithm associated with a CHR advance through the stages of authenticated encryption (AEAD) or hash function processing.

3. **The configuration instructions may configure CHRs to use Immutable System Keys (ISKs).** The ISKs are the keys generated during manufacturing, provisioning, or at boot. The table is collated by the system's RoT at boot and pushed to an internal RAM of the ACE implementation without direct software exposure. Only the RoT can write to this RAM.
4. **Support for Lifecycle Management and Migration.** ACE can bind a key to a specific lifecycle. Some keys should only be valid until the next shutdown or soft reset, whereas other keys are permanently available. If a given key may only be used on one device, it should become useless if the process/VM is migrated to a different device.

Lifecycles are enforced as part of the CHR metadata, and by the fact that they determine the keys used for SCC import and export, as we discuss next.

Context Transport Keys and Lifecycle Policies. A SCC is created by encrypting and authenticating a CHR using a 256-bit *Context Transport Key* (CTK) in the AES-GCM-SIV mode (RFC 8452). The format of the SIV differs from RFC 8452 only in that the 8 least significant bits of the 128-bit SIV are replaced by the *Lifecycle* field.

The default CTK is the *Root CTK*, and it can only be set by Machine Mode. Import and export operations use either this Root CTK or a derived *Lifecycle-Specific Context Transport Key* (LSCTK). The LSCTK is derived from the Root CTK and a Lifecycle Secret stored in the *Lifecycle Secrets Table* (LST), which is part of the ISK Table. Each entry in the LST corresponds to a value of the Lifecycle field in the SIV.

Example Lifecycles include: keys valid until next boot, keys bound to specific devices, bound to device models, or OEMs. Modifying an LST entry invalidates all keys wrapped with its previous value, as they can no longer be re-imported. This simple mechanism enables the Root of Trust to manage key lifecycles through the LST, which then apply to the SCCs.

Isolation and Migration of Process Keys are not architected by ACE, but all the underpinnings that are required to implement them are provided.

In place of configuring a single Root CTK for the entire system, Machine Mode can also program a different Root CTK for each World/Supervisor Domain (as per `smmtt`) to prevent the leakage of SCCs from one Domain to another. Supervisor Domains may ask MM to reconfigure the Root CTK before running a VM or Applet, to mutually isolate the latter.

If a specific Root CTK is associated with a VM and the VM must be migrated to a different device, the MM can use a Key Encapsulation Mechanism (KEM) to transfer this Root CTK to the destination device. The destination device's MM will then be able to set the same root CTK when running the migrated VM.

All the keys that are stored as SCCs in the memory of the VM (for instance, all the keys used by user processes are exported and reimported for them by the parent OS upon context switch) can then be used on the target machine if their lifecycle setting allows. For instance, if the CTK is the Root CTK, then any SCC will be reimported. However, if a key is device bound, the derived LSCTK will be different and import will fail. This enables use cases such as subscriptions being tied to a device, or services associated only to given product tiers.

Instructions. The instructions supported by ACE have self-explanatory names, such as `ace.set` (to set key and metadata), `ace.invalidate`, `ace.export`, `ace.import`, `ace.execute`, `ace.size` (to get the size of the SCC, which may vary depending on the algorithm), `ace.available` (to allow algorithm discoverability). A further instruction, `ace.message`, serves to change the state machine of a CHR in a controlled way, for instance to move from initialization, to message absorption and then to finalization in a hash function. A privilege level specific system register (`ACE_CHR_VALID`) keeps track of whether a given CHR is actually in use, in order to accelerate context switch operations. Therefore, software developers shall use `ace.invalidate` when a given key is no longer needed.

The Register File. As for the number of registers, encoding allows up to 32 CHRs, with number 31 tied to the Root CTK and only accessible by Machine Mode. It is not mandatory to architect 32 registers: We estimate that a typical system will need around 8 CHRs. Since CHRs are (relatively) seldom reconfigured and they should not be speculated upon for security reasons, they do not require (much) renaming. As a result, even if they contained, say, 1000 bits (to allow keys in multiple shares), their hardware cost would be minor in comparison to the integer, floating point and vector register files of modern microarchitectures.

Conclusions

ACE is a solid architecture to support a variety of high assurance cryptographic operations in a secure way with a streamlined and uniform interface.