ACE: Atomic Cryptographic Extensions

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

Motivation

Selected Use Cases and Application Domains

- Content Protection, Digital Identity, Storage Encryption
  - Allow to efficiently encrypt/decrypt without exposing keys.
  - Separate the keys per domain.
- Cloud Computing
  - Allow migration of VMs together with their keys, protecting the latter.

Issues

- Need to manage keys
- Lack of trust
- Need efficient cryptography
- Physical Side Channel Attacks
- μarchitectural SCA

Inadequate Current Solutions

- Key provisioning mechanisms have different interfaces and security properties ⇒ added complexity for SW.
- Cryptographic accelerators slow to set up, shared with leakage risks, lack transparent algorithmic agility, and...
- — are system-specific ⇒ SW porting needs extra effort.
- Current crypto ISAs expose keys, lack algorithmic agility and SCA protection ⇒ security risks; and
- — do not have access to system keys ⇒ limited.

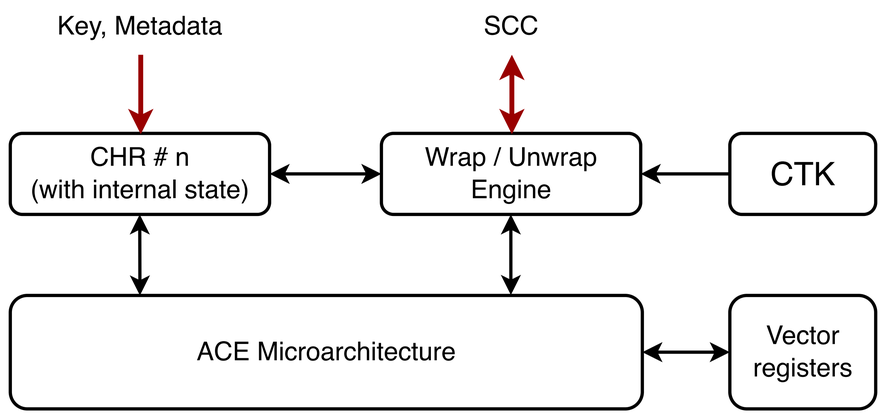
Solution

Architecture

- **Context Holding Registers (CHRs)** store keys and metadata that only ACE can access. # registers ∈ [8..32],
- **Context Transport Key (CTK)**: Special register only accessed by Machine Mode as WO. Used to Wrap/Unwrap CHR states (encrypted and authenticated) into:
- **Sealed Cryptographic Context (SCC)**: Data chunk containing a CHR's state. Can only be unwrapped with same CTK used to wrap.
- *Key and metadata can be written in cleartext to a CHR, but not extracted!*
- Data read/written from vector regs, but SCCs only from/to memory.
- Operations are atomic, invoked by `ace.execute`.
- Modes of operation can be supported: `ace.message` to switch stages.
- *Lifecycle CTKs* can be derived from the "master" CTK and *lifecycle strings* to bind SSCs to specific lifecycles (see next column).

Instructions

- `ace.set`
- `ace.invalidate`
- `ace.import`
- `ace.export`
- `ace.execute`
- `ace.size`
- `ace.available`
- `ace.message`



Architecture (segue)

ACE maintains a list of *lifecycle strings* which are: permanent/per device/per power cycle, etc.

The *index* of the lifecycle string is put in CHR/SSC metadata. The string is combined with the CTK to use a *derived CTK* which is used in place of the original CTK for import/export.

If a lifecycle string is modified, all lifecycle CTKs derived from it change — and the old values lost. SSCs wrapped with the old Lifecycle CTKs are thus invalid.

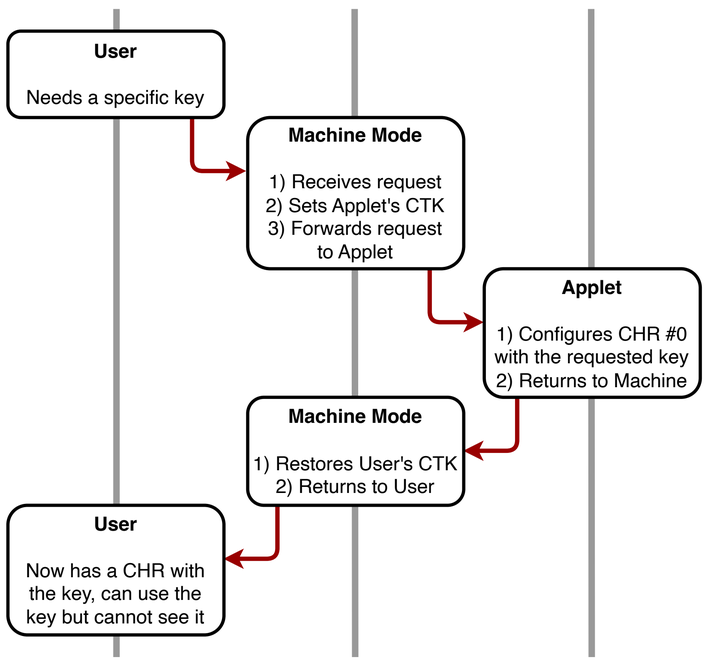
Microarchitecture

- ACE can provide many algorithms, which must be all discoverable by `ace.available`. Algorithms not initially supported can be added later and trapped to Machine Mode.
- Side channel protected versions of these algorithms can be provided and exposed.
- ACE can provide access to some system specific keys through CHRs. The table of such keys is microarchitecture specific.
- Foreign key-providing HW blocks can be supported, just add a HW wrapper to create SCCs.

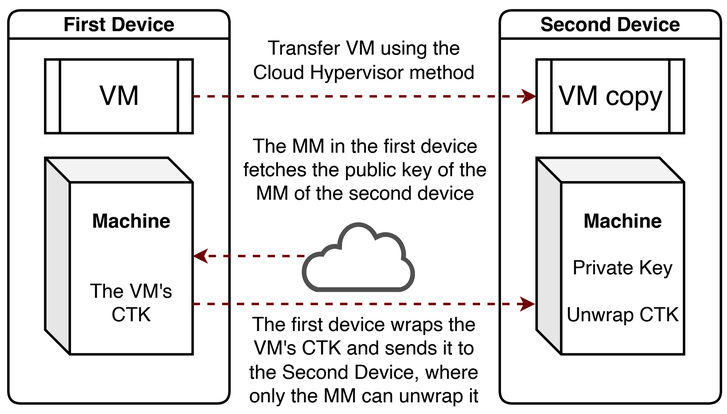
Usage

Example Software Flow

Getting a Key from a Key-Provisioning Applet



Migration



The MM can configure per-VM or per-World CTKs to provide key isolation.
It also support migration of VMs together with the SSCs in its memory: the MM of the source device will wrap the VM's CTK using the public key of the target device's MM, and the latter will be able to configure it.

Conclusion

ACE is an ACE!

The ACE instruction set extension and architecture is a solid proposal to support a variety of high assurance cryptographic operations in a secure way — while providing a streamlined and uniform interface to SW.