Proposal: CocoonFs format for SVSM persistence

Nicolai Stange `nstange@suse.de`

SUSE LLC

July 16, 2025

CocoonFs - Format

- Special purpose FS format suitable for TEE settings.
- Spec: `https://nicst.de/cocoonfs-format.html`
- All entities encrypted in CBC mode. Fresh IV for each entity encryption op. Unused storage randomized.
- Authenticated by means of a Merkle tree $\Rightarrow$ a single digest captures the state of the whole FS.
- Algorithm agility: any symcipher + hash(es) from the TCG algorithm registry may be used.
- Features a journal for robustness against service interruptions.
- File “names” are simply flat 32 bit inode numbers, organized in a B+-tree.
- (Support for online fs image growth would require a small format addition).

CocoonFs - Format

All entities encrypted in CBC mode. Fresh IV for each encryption op.

- $\Rightarrow$ No seeks, partial file reads or writes.
- TCG TPM reference implementation (`svsm/libtcgtpm/deps/tpm-20-ref`):
  - `TPMCmd/Platform/src/NVMem.c`
  - State size is a compiletime constant: `NV_MEMORY_SIZE`, 16 kB
  - State always read + written as a whole.
  - `seek(..., 0, SEEK_SET)` + `seek(..., 0, SEEK_END)`

CocoonFs - Implementation

- `https://github.com/coconut-svsm/cocoon-tpm/tree/main/storage`, `https://crates.io/crates/cocoon-tpm-storage` (for docs)
- SVSM runtime environment friendly:
  - No panic on allocation failures, etc.
  - Uses the `cocoon-tpm-crypto` crate throughout → bindings to RustCrypto + BoringSSL, with OpenSSL to come.
  - Needs only `SpinLock` + `RWLock`.
  - Generic over locking types + the block storage interface.
- Transaction based, i.e. everything is all-or-nothing.
- API is defined in terms of Rust's `async` `Future` framework. (With no anonymous `Future` types being created via `async fn`.)

CocoonFs - Implementation – Quick overview on Rust `Futures`

- Generic, execution environment agnostic framework for defining asynchronous APIs.
- `https://doc.rust-lang.org/std/future/trait.Future.html`
- Each operation/request is represented by some `impl Future` object. Progress is driven by

$$\texttt{Future::poll(self, ...)} \rightarrow \begin{cases} \texttt{Pending} \\ \texttt{Ready(result)} \end{cases}$$

- `Futures` may nest: outer `poll()` polls the inner `poll()`. The outermost `Future` is called a "task".
- `Future` objects represent the current execution state. May be stored on the heap (and passed around etc.).
- The actual execution environment provides
  - a "task executor" definition polling the top-level `Future` however it seems fit,
  - and IO primitive "leaf `Futures`" at the other end that play well with the executor.

  Anything inbetween can be made completely generic / agnostic of the execution environment.

CocoonFs - Implementation

The actual execution environment provides

- *a "task executor" definition polling the top-level* `Future` *however it seems fit,*
- *and IO primitive "leaf* `Futures`*" at the other end that play well with the executor.*

Anything inbetween is completely generic / agnostic of the execution environment.

- Start simple: busy-poll tasks ("FS ops") to completion, à la `https://crates.io/crates/pollster`.
- May evolve to anything more complex, e.g. interrupt driven, if needed: redefine the SVSM's task executor + the associated block device IO primitives.