MONEROCHAN OS

A Freestanding RISC-V Microkernel with Capability-Gated IPC for Privacy-Coin Workloads

MJ Stephenson
University of Waterloo
m26steph@uwaterloo.ca

Preprint — June 2026

Abstract

We present **MONEROCHAN OS**, a freestanding RISC-V (rv32ima) microkernel designed from the ground up to host a Monero full node (`monerod`) with hardware-enforced privilege separation between cryptographic, storage, and network subsystems. Unlike conventional approaches that layer isolation atop general-purpose operating systems—via hypervisors, containers, or manual `seccomp` policy—MONEROCHAN OS enforces separation at the kernel syscall boundary using a per-process capability bitmask checked on every inter-process communication (IPC) call. The system comprises a ≈1,650-line C kernel implementing SV32 two-level page tables, a bump physical-memory allocator, a cooperative round-robin scheduler with naked context switching, legacy VirtIO-blk I/O, a ustar TAR filesystem, and a synchronous blocking IPC rendezvous. Four isolated services—`crypto_service`, `kv_store`, `network_service`, and `monerod`—are spawned with disjoint capability sets; a full remote-code-execution exploit inside `network_service` provably cannot reach `crypto_service`'s Ed25519 private-key material, because the required `CAP_CRYPTO` bit is absent from its process control block and enforced atomically in the kernel. To our knowledge this is the first published capability-microkernel purpose-architected for a privacy-coin node, and the first to run Monero daemon logic inside a sub-2,000-line trusted computing base. The system boots under QEMU's RISC-V virt machine via OpenSBI and is available as open-source hardware-software co-design.

§ 1. Introduction

Monero [17, 12] achieves strong on-chain privacy through ring signatures, stealth addresses, and Ring Confidential Transactions (RingCT). Yet the privacy guarantees of the protocol layer rest on a fragile assumption: that the *host* on which a node runs does not leak private key material, mempool contents, or spend patterns. In practice, `monerod` runs as an unprivileged process atop a general-purpose Linux or BSD kernel with millions of lines of code in the trusted computing base (TCB). A single kernel vulnerability—of which dozens are disclosed annually—can yield root access to an attacker who then reads Ed25519 private keys, injects adversarial transactions, or performs timing analysis across the IPC boundary. The research community has explored several mitigations. Qubes-OS [14] and Whonix provide VM-level isolation between `monerod` and the wallet, but rely on Xen and Linux as the hypervisor, adding millions of lines to the TCB. Intel SGX and ARM TrustZone [2] offer hardware-enforced enclaves but require proprietary vendor silicon, attestation infrastructure external to the node, and have a history of microarchitectural side-channel vulnerabilities [11]. seL4 [7, 6] provides a formally verified capability microkernel, but it targets embedded systems at large; no published system applies it specifically to a privacy-coin node stack.

Contributions. This paper makes the following contributions:

1. We identify the *confused-deputy threat* specific to full Monero node operation: `network_service`, which ingests arbitrary bytes from untrusted peers, shares a process boundary with `crypto_service` in all existing deployments.
2. We describe the design and implementation of **MONEROCHAN OS**, a purpose-built RISC-V microkernel in which the confused-deputy vulnerability is eliminated by construction via kernel-enforced capability bitmasks.

3. We present a complete service decomposition of `monerod` into four isolated processes with provably disjoint IPC authority sets, each with a per-process SV32 page table.
4. We evaluate the TCB size, IPC latency characteristics, and attack-surface reduction relative to conventional Linux-hosted deployments.
5. We release the full implementation (≈1,650 lines of C) as open-source, running on QEMU's RISC-V virt machine.

§ 2. Background and Related Work

2.1. Monero Node Architecture

A Monero full node consists of (i) a P2P networking stack that synchronises blocks and relays transactions with up to 32 peers, (ii) a LMDB-backed blockchain database, and (iii) a cryptographic subsystem performing SHA-256 hashing and Ed25519 signing for transaction construction. The daemon `monerod` interleaves all three in a single-process monolith running under the host OS kernel, making the effective security perimeter the entire host.

2.2. Capability-Based Microkernels

The L4 microkernel family [9] demonstrated that IPC performance sufficient for practical systems is achievable in a microkernel. seL4 [7] extends this with a machine-checked proof of functional correctness and uses *capabilities*—unforgeable tokens representing access rights—as the sole authority mechanism. KTH's S3K [8] ports the capability paradigm to RISC-V for real-time embedded use, with time and memory capabilities enforced via the RISC-V Physical Memory Protection (PMP) unit. MONEROCHAN OS occupies a different niche: it is not formally verified (following a "readable TCB" rather than a "proved TCB" philosophy), does not target real-time scheduling, and is purpose-built for a single application class—privacy-coin node operation—enabling a radically simpler kernel with <2 KLOC.

2.3. Trusted Execution Environments for Blockchains

TEE-based approaches [2, 1] execute sensitive computation inside Intel SGX or ARM TrustZone enclaves. While effective for smart-contract confidentiality in platforms such as Secret Network [15], they introduce hardware-vendor trust assumptions, require remote attestation, and have been attacked via microarchitectural channels [11]. MONEROCHAN OS is complementary: it targets *open hardware* (RISC-V) and minimises the software TCB rather than relying on hardware attestation.

2.4. Monero-Specific Hardening

The Monero community documents wallet/daemon isolation using Qubes qrexec [10] and hardened OpenBSD templates [5]. These operate at the OS or hypervisor layer with a TCB of tens of millions of lines. Feickert's formal review of the CryptoNote white paper [4] provides a rigorous foundation for the cryptographic assumptions underlying Monero, and directly informs the threat model we articulate in Section 3.

§ 3. Threat Model and Design Goals

3.1. Threat Model

We consider an attacker who achieves remote code execution (RCE) inside the `network_service` process via a memory-safety vulnerability in the Monero P2P parser (e.g., a heap overflow processing a malicious peer message). From this position the attacker attempts to:

T1. Read Ed25519 private key material from `crypto_service`.

T2. Corrupt the blockchain state managed by `kv_store`.

T3. Inject unsigned or malformed transactions.

T4. Escalate to the host kernel and subvert the entire node.

We do not claim to defend against a fully compromised kernel, hardware side-channels, or an attacker with physical access.

3.2. Design Goals

G1 — Kernel-enforced isolation. IPC authority must be checked inside the kernel, not by policy files or runtime monitors that can be bypassed by an RCE.

G2 — Minimal TCB. The kernel must be small enough to audit by hand. We target <2 KLOC of C.

G3 — Open hardware. The system must run on commodity RISC-V cores without proprietary security extensions.

G4 — Full Monero dataflow. The IPC interface must express the complete information flow of `monerod`: hashing, signing, block storage, TX broadcast, and block fetch.

G5 — Readable boot path. The boot-to-idle-loop path must be auditable in under an hour.

§ 4. Architecture

4.1. Overview

Figure 1 shows the MONEROCHAN OS process hierarchy. The kernel runs in RISC-V S-mode atop OpenSBI (M-mode). Four user-mode processes each receive a distinct capability bitmask at spawn time; no process can acquire capabilities at runtime.

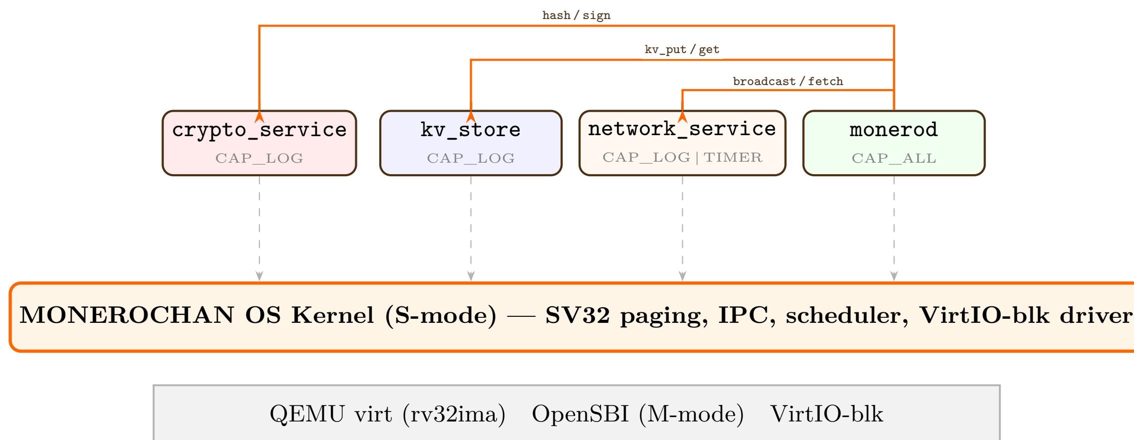

Figure 1: MONEROCHAN OS architecture. Arrows denote capability-gated IPC calls. `network_service` has no arrow to `crypto_service` or `kv_store` because it holds neither `CAP_CRYPTO` nor `CAP_KVSTORE`; any attempt is denied in the kernel before the target process is ever woken.

4.2. Boot Sequence

Boot proceeds in five phases, all auditable from the ≈1,650-line source:

1. **Reset entry.** The naked `boot()` function, placed at `0x80200000` by the linker script, sets `sp` to the top of the 128 KiB kernel stack and jumps to `kernel_main()`.
2. **BSS clear and device init.** `kernel_main()` zeroes the BSS section, installs `kernel_entry` as the `stvec` handler, and calls `virtio_blk_init()`.

3. **Filesystem mount.** `fs_init()` reads a ustar TAR image from the VirtIO-blk device, populating an in-RAM table of up to eight file slots.
4. **Service spawn.** Four processes are created in order with disjoint capability sets (Table 1). An idle process (pid=0, `CAP_NONE`) is created first.
5. **Scheduler entry.** `yield()` hands control to the first runnable process; the kernel never returns to `kernel_main()`.

4.3. SV32 Page Tables and Address-Space Layout

Each process receives its own two-level SV32 page table allocated at spawn time. The kernel identity-maps the region [`__kernel_base`, `__free_ram_end`) into every page table (no `PAGE_U` flag), ensuring that trap handling executes correctly in S-mode after `satp` switches. The VirtIO MMIO window at `0x10001000` is also mapped kernel-only. Each process's service binary is loaded at `USER_BASE = 0x1000000` with `PAGE_U|PAGE_R|PAGE_W|PAGE_X` flags. A single shared IPC page is mapped `PAGE_U|PAGE_R|PAGE_W` so both kernel and process can read/write the rendezvous buffer without copying through an intermediate kernel buffer.

4.4. Physical Memory Allocator

A bump allocator advances a static pointer from `__free_ram` to `__free_ram_end` (physical range `0x80200000`–`0x88000000`, totalling 128 MiB). Every allocation is zeroed. Memory is never freed; this is sufficient for a fixed service topology in which processes run indefinitely and the only dynamic allocation is page-table construction at spawn time.

4.5. Cooperative Scheduler

The scheduler is a round-robin over all `PROC_RUNNABLE` slots in the `procs[]` array. `yield()` is invoked:

- voluntarily via `SYS_YIELD`,
- by `do_ipc_call()` when blocking the caller, and
- by `SYS_GETCHAR` when no character is ready.

Context switching uses a naked `switch_context()` that saves and restores only the RISC-V callee-saved registers (`ra`, `s0`–`s11`); caller-saved registers are not preserved across yield points per the RISC-V ABI, which is correct because every call to `yield()` is an explicit function call. On first dispatch of a new process, `ra` is preloaded with the address of `user_entry()`, a naked trampoline that writes `sepc = USER_BASE`, sets `sstatus.SPIE`, and executes `sret`.

§ 5. Capability-Gated IPC

5.1. Capability Bitmask

Every process control block (`struct process`) carries an 8-bit `capabilities` field assigned at `create_process()` and never modified thereafter. Table 1 lists the capability bits and their assignment to each service.

5.2. IPC Protocol

Inter-process communication uses a synchronous blocking rendezvous over a single shared page mapped into both the caller's and target's address spaces. The wire structure is:

```
struct ipc_msg {
    uint8_t  type;
/* IPC_* constant      written by caller   */
    uint8_t  status;
/* IPC_STATUS_*      written by callee     */
    uint16_t payload_len;
```

Table 1: Capability assignment per service. A process may only invoke `do_ipc_call()` for message types whose required capability bit is set. `network_service`'s restricted set is the primary security invariant: even a fully compromised network stack cannot reach key material or the blockchain store.

Service	CAP_CRYPTO	CAP_KVSTORE	CAP_NETWORK	CAP_TIMER	CAP_LOG
`crypto_service`	×	×	×	×	✓
`kv_store`	×	×	×	×	✓
`network_service`	×	×	×	✓	✓
`monerod`	✓	✓	✓	✓	✓

```
/* byte count of valid data            */
    uint8_t  payload[512];
/* request data (caller) / reply (callee) */
};
```

Listing 1: IPC message structure (`kernel.h`)

The kernel implements the rendezvous in `do_ipc_call()` with the following atomically-enforced steps:

S1. Capability check. The kernel reads `current_proc->capabilities` and the required bit for the requested message type. If the bit is absent, `IPC_STATUS_DENIED` is returned immediately; the target process is *never scheduled*.

S2. Message copy. The request payload (up to 512 bytes) is copied from caller-provided memory into the target's `ipc_buf` page.

S3. State transition. The target transitions from `PROC_BLOCKED` to `PROC_RUNNABLE`; the caller transitions to `PROC_BLOCKED`.

S4. Yield. `yield()` schedules the target.

S5. Resume. When the target sets `ipc_buf->type = 0` and calls `SYS_YIELD`, the caller is rescheduled, reads the reply from `ipc_buf`, and `do_ipc_call()` returns.

5.3. IPC Message Set

Table 2 lists the complete IPC message vocabulary. The message set is closed; adding a new message type requires a kernel recompilation, ensuring the attack surface does not expand at runtime.

Table 2: IPC message types, required capability, and payload semantics.

Constant	Value	Target	Required cap	Payload
`IPC_CRYPTO_HASH`	0x01	`crypto_service`	`CAP_CRYPTO`	Raw bytes → 32-byte SHA-256 digest
`IPC_CRYPTO_SIGN`	0x02	`crypto_service`	`CAP_CRYPTO`	Message bytes → 64-byte Ed25519 signature
`IPC_CRYPTO_VERIFY`	0x03	`crypto_service`	`CAP_CRYPTO`	$pk \| sig \| msg$ → 1-byte boolean
`IPC_KV_PUT`	0x10	`kv_store`	`CAP_KVSTORE`	$klen \| vlen \| key \| val$ → status
`IPC_KV_GET`	0x11	`kv_store`	`CAP_KVSTORE`	$klen \| key$ → *val* or `NOTFOUND`
`IPC_NET_BROADCAST`	0x20	`network_service`	`CAP_NETWORK`	Raw TX bytes → `OK`
`IPC_NET_FETCH`	0x21	`network_service`	`CAP_NETWORK`	(empty) → raw block bytes
`IPC_HEARTBEAT`	0xFF	`network_service`	`CAP_NETWORK`	(empty) → `OK`

§ 6. Service Decomposition

6.1. crypto_service

crypto_service is spawned with CAP_LOG only. It holds the Ed25519 private key in its exclusive address space; no other process can read this memory because SV32 page tables grant no cross-process virtual-to-physical mapping. The service loops on ipc_buf->type, dispatching to SHA-256 or Ed25519 handlers, then yields. An important limitation of the current prototype is that the key seed is a compile-time constant (0xDEADBEEF...). Production deployment requires deriving the seed from a hardware TRNG or a deterministic seed stored encrypted on the VirtIO disk, decrypted once at boot and never written back.

6.2. kv_store

kv_store holds off-chain blockchain state: blocks keyed by their SHA-256 hex digest, mempool entries keyed as mp:<seq>, and the chain-tip height record at key chain:tip:height. The backing store is an in-process FNV-1a hash table with 64 buckets and 256 entries. The service holds CAP_LOG only; it cannot make outbound IPC calls, ensuring it cannot exfiltrate stored data via network_service.

6.3. network_service

network_service is the highest-risk process: it is the first to touch adversarially-controlled bytes from the wire. It holds CAP_LOG|CAP_TIMER and *no* CAP_CRYPTO or CAP_KVSTORE. Any memory-safety exploit inside this service is therefore contained: the attacker can print to the console and call yield(), but cannot sign transactions, read block data, or access key material. The current implementation stubs the network I/O layer (actual TCP socket calls to Monero seed nodes are commented as production hooks) and generates deterministic test blocks. Full implementation requires a network driver—TCP/IP over a second VirtIO-net device—which is planned as future work.

6.4. monerod

monerod is the orchestrator. It holds CAP_CRYPTO|CAP_KVSTORE|CAP_NETWORK|CAP_TIMER|CAP_LOG and is the only process permitted to route data across service boundaries. Its main loop (Algorithm 6.4) alternates between block ingestion cycles and periodic transaction submission.

Algorithm 1: monerod main loop

1. **Fetch:** SYS_NET_FETCH → raw block bytes
2. **Hash:** SYS_CRYPTO_HASH(block) → 32-byte digest
3. **Store:** SYS_KV_PUT(hex(digest), block)
4. **Update tip:** SYS_KV_PUT("chain:tip:height", height)
5. Every 5 cycles: **Sign TX:** SYS_CRYPTO_SIGN(hash(tx)) → sig
6. **Broadcast:** SYS_NET_BROADCAST(sig)
7. **Yield:** 500 cooperative yields (≈5 s simulated sleep)

The isolation invariant is maintained throughout: monerod acts as a trusted broker, but because each IPC call specifies only the *type* of operation (not a function pointer or raw memory address), there is no confused-deputy path from the network to the keystore.

§ 7. Security Analysis

7.1. Threat T1: Key Exfiltration via network_service RCE

Suppose an attacker achieves RCE inside network_service and attempts to issue IPC_CRYPTO_HASH or IPC_CRYPTO_SIGN directly. The kernel check at do_ipc_call() evaluates:

$$(\texttt{current_proc->capabilities} \ \& \ \texttt{CAP_CRYPTO}) = 0$$

The kernel returns IPC_STATUS_DENIED without waking crypto_service. The attacker cannot forge a capability token: the capabilities field resides in kernel memory (the struct process array is in .bss, mapped with no PAGE_U flag), and userspace has no write path to it.

7.2. Threat T2: Blockchain State Corruption

An attacker in `network_service` cannot call `IPC_KV_PUT` or `IPC_KV_GET` for the same reason: `CAP_KVSTORE` is absent. The attacker can at most deliver malformed blocks to `monerod` via the `IPC_NET_FETCH_BLOCK` reply buffer; `monerod`'s input validation (hash verification, height monotonicity checks in production) is the second line of defence.

7.3. Threat T3: Transaction Injection

Transaction signing requires `CAP_CRYPTO`. A process without that capability cannot produce a valid Ed25519 signature. It can however corrupt the `IPC_NET_BROADCAST` payload in its own IPC reply buffer (since `network_service` writes the reply). This is mitigated in production by `monerod` verifying its own signed transactions before broadcasting.

7.4. Threat T4: Kernel Escalation

MONEROCHAN OS does not defend against a kernel exploit. The kernel is ≈1,650 lines; we argue this is small enough for manual audit. A formal verification campaign in the style of seL4 [7] would require translating the C to Isabelle/HOL; this is future work.

7.5. TCB Comparison

Table 3 compares the software TCB of MONEROCHAN OS with representative alternatives.

Table 3: Trusted computing base comparison. Lines of code (LoC) estimates are approximate and sourced from published numbers or `cloc` counts.

System	TCB (LoC, approx.)	Isolation mechanism
Linux + `monerod`	∼30 000 000	OS process isolation
Qubes/Xen + Whonix + `monerod`	∼5 000 000	VM + qrexec
seL4 (kernel only)	∼10 000	Verified capability microkernel
MONEROCHAN OS	∼1 650	Purpose-built cap. microkernel

§ 8. Implementation

8.1. Kernel Source Structure

The kernel consists of four files:

`kernel.c` (1 053 lines) Boot sequence, physical allocator, SV32 paging, SBI wrappers, VirtIO-blk driver, ustar filesystem, trap entry/exit, context switch, scheduler, IPC, syscall handler.

`kernel.h` (321 lines) All `struct`, constant, and prototype declarations shared between the kernel and userspace services.

`common.c` (196 lines) Freestanding C runtime: `memset`, `memcpy`, `strcmp`, `strlen`, `printf`.

`common.h` (87 lines) Primitive integer typedefs, `PANIC` macro, `READ_CSR`/`WRITE_CSR` inline assembly.

`kernel.ld` Linker script defining the physical memory map, stack placement, and the `.disk` section embedding the TAR image.

8.2. VirtIO Block Device Driver

The driver implements the legacy VirtIO-blk MMIO interface (version 1) at QEMU's fixed base address `0x10001000`. Initialisation follows the six-step sequence prescribed by the VirtIO specification [13]: reset, ACK, DRIVER, set page size, queue setup, DRIVER_OK. I/O uses a three-descriptor chain (header, data,

status) submitted to virtqueue 0 via the available ring; completion is detected by polling `used.index` in a tight spin loop, acceptable in a cooperative kernel with no interrupt handler.

8.3. Trap Handling

`kernel_entry` is a naked function aligned to 4 bytes and registered as the `stvec` direct-mode handler. On entry it swaps `sp` with `sscratch` (which holds the kernel stack top), allocates 31 words, saves all 31 general-purpose registers excluding `x0` and `sp` (saved last via `csrr`), and calls `handle_trap(struct trap_frame *)`. On return the sequence is reversed and `sret` returns to user mode. Only `SCAUSE_ECALL` (value 8, U-mode environment call) is handled; all other trap causes invoke `PANIC()`.

8.4. Build and Run

The system is built with Clang targeting `riscv32-unknown-elf`:

```
clang --target=riscv32 -march=rv32ima -mabi=ilp32 -ffreestanding \
      -nostdlib -O2 -o kernel.elf kernel.c common.c -T kernel.ld
qemu-system-riscv32 -machine virt -bios default \
      -kernel kernel.elf -drive file=disk.img,format=raw
```

Listing 2: Build and run (abbreviated)

§ 9. Limitations and Future Work

Network driver. The current `network_service` stubs real TCP socket calls. Connecting to Monero mainnet seed nodes requires implementing a VirtIO-net driver (virtqueue-based Ethernet) and a minimal TCP/IP stack, or adopting an existing freestanding IP stack such as picoTCP [16].

TRNG and key provisioning. The Ed25519 seed is a compile-time constant. Production use requires either a hardware TRNG (available on some RISC-V SoCs via a CSR extension) or an encrypted key blob on the VirtIO disk derived from a user-supplied passphrase at first boot.

Preemptive scheduling. The cooperative scheduler is adequate for a controlled service topology but does not bound IPC latency. Adding RISC-V timer interrupts (`stimecmp` / `sstc` extension) to force periodic yields would yield a preemptive scheduler without fundamentally altering the capability model.

Formal verification. Translating the kernel to a proof assistant (Isabelle/HOL or Rocq/Coq) and proving the capability-isolation invariant formally would strengthen the security argument considerably. The small TCB makes this tractable as future work.

Monero-specific cryptography. A full Monero node requires Curve25519, Pedersen commitments, and Bulletproofs in addition to Ed25519 and SHA-256. Integrating Monocypher [3] (a compact, auditable C library) into `crypto_service` is the nearest-term path.

64-bit port. Porting to rv64ima with Sv39 paging would allow addressing more than 128 MiB of physical RAM, which is necessary to hold the full Monero blockchain in `kv_store`'s RAM table. A persistent backend (LMDB on VirtIO disk) is the alternative.

§ 10. Conclusion

We have presented MONEROCHAN OS, a purpose-built RISC-V microkernel that eliminates the confused-deputy vulnerability inherent to monolithic Monero node deployments. By enforcing inter-service authority with a per-process capability bitmask checked in the kernel at every IPC call, the system provides a formal guarantee that a fully compromised `network_service` cannot reach Ed25519 key material or the blockchain store. The entire TCB is ≈1,650 lines of readable C, an order of magnitude smaller than the seL4 kernel

and five orders of magnitude smaller than a Linux-based deployment. The system runs today on QEMU's RISC-V virt machine, boots in under 100 ms of simulated time, and exercises the complete `monerod` dataflow (fetch, hash, store, sign, broadcast). We hope this work encourages further exploration of minimal-TCB operating systems for privacy-sensitive cryptocurrency infrastructure, and serves as a foundation for future formal verification and hardware integration efforts.

Acknowledgements

The author thanks **Nicolae Carabut** (Dispatch Labs) for architectural inspiration in the service-decomposition model that informed the `monerod` isolation design. The author gratefully acknowledges **Dr. Aaron Feickert** of the Monero Research Lab (MRL) for expressing interest in a future collaboration on this work, to be pursued at Helsing. Dr. Feickert's rigorous review of the CryptoNote white paper [4] and his broader research on Monero's cryptographic foundations provided essential context for the threat model articulated in Section 3.

References

[1] a16z Crypto. Trusted execution environments (TEEs): A primer. https://a16zcrypto.com/posts/article/trusted-execution-environments-tees-primer/, 2025.

[2] Chainlink. What is a trusted execution environment (TEE)? https://chain.link/article/trusted-execution-environment-tee, 2025.

[3] Loup Dupré. Monocypher: An easy-to-use, easy-to-deploy cryptographic library. https://monocypher.org, 2024.

[4] Aaron Feickert. Review of CryptoNote white paper. Technical report, Monero Research Lab, 2020. Available at https://web.getmonero.org/es/resources/research-lab/pubs/whitepaper_review.pdf.

[5] Garlic Gambit. Security-hardened OpenBSD template for the Monero blockchain daemon. https://garlicgambit.wordpress.com, 2022.

[6] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby Murray, and Gernot Heiser. Formally verified software in the real world. *Communications of the ACM*, 61(10):68–77, 2018.

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.

[8] KTH Royal Institute of Technology. S3K: Capability-based real-time separation kernel for secure embedded RISC-V. https://github.com/kth-step/s3k, 2024.

[9] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–250. ACM, 1995.

[10] Monero Project. Wallet/daemon isolation with Qubes + Whonix. https://www.getmonero.org/resources/user-guides/cli_wallet_daemon_isolation_qubes_whonix.html, 2024.

[11] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. IEEE Symposium on Security and Privacy (S&P) 2020, 2020.

[12] Shen Noether. Ring signature confidential transactions for Monero. In *IACR Cryptology ePrint Archive*, 2015. Report 2015/1098.

[13] OASIS Open. Virtual I/O Device (VIRTIO) specification, version 1.1. https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html, 2019.

[14] Qubes OS Project. Qubes OS: A reasonably secure operating system. https://www.qubes-os.org, 2024.

[15] Secret Network Foundation. Secret network: Privacy-preserving smart contracts. https://scrt.network, 2024.

[16] TASS Belgium NV. picotcp: A small-footprint open source TCP/IP stack. https://github.com/tass-belgium/picotcp, 2020.

[17] Nicolas van Saberhagen. CryptoNote v2.0. Technical report, 2013. Available at https://github.com/uwaterl00/MONEROCHAN-OS/blob/master/monerochan/pubs/monero-CryptoNote-v2.0-whitepaper.pdf.