Notary: A Device for Secure Transaction Approval

Anish Athalye    Adam Belay    Frans Kaashoek    Robert Morris    Nickolai Zeldovich

MIT CSAIL
How to securely approve transactions?

• Users perform sensitive transactional operations
  • Bank transfers
  • Cryptocurrency transactions
  • Deleting backups
  • Modifying DNS records
Common solution: smartphone apps

- Suffers from isolation bugs (e.g. jailbreaks)

Approval agent on smartphone
Hardware wallets for transaction approval

![Diagram of a Ledger wallet](image)

- Display
- Buttons

TX

Sign(TX)

Ledger wallet
Challenge: wallets need to isolate agents

Ledger app store: 50+ third-party agents
Challenge: wallets need to isolate agents

Ledger app store: 50+ third-party agents
Problems with existing hardware wallets

• OS bugs
  • Over 10 found in Ledger and Trezor wallets

• Potential hardware bugs
  • Shared hardware state could leak secrets (e.g. Spectre)
Contribution: Notary

• Agent separation architecture
  • Reset-based switching
  • Verified deterministic start

• Physical hardware wallet prototype
Threat model

• Some agents are malicious

• Physical attacks out of scope
  • Could be addressed by tamper-proof hardware
Separation architecture provides isolation

Notary separation architecture

- Runs third-party code
- No OS, full access to hardware

- Manages storage, agent switching

Agent SoC (Blue)
- User I/O
- USB
- uart
- rst
- Notary

Kernel SoC (Green)
- Reset button
- Storage

Diagram: Agent SoC connected to User I/O and USB. Kernel SoC connected to Reset button and Storage. UART and RST signals between the components.
Separation architecture provides isolation

Agent
SoC
Kernel
SoC
Runs third-party code
No OS, full access to hardware
Manages storage,
agent switching
User I/O
Reset button
Storage
uart
rst
USB

Kernel SoC
Separation architecture provides isolation

Agent SoC

Runs third-party code
No OS, full access to hardware

User I/O

Reset button

USB

uart

rst

Storage

Manages storage, agent switching

Kernel SoC

Agent SoC
Separation architecture provides isolation

- **Agent SoC**: Runs third-party code. 
  - No OS, full access to hardware.

- **Kernel SoC**: Manages storage, agent switching.
  - Connected only by UART (and reset wire).

- **User I/O**
- **Reset button**
- **USB**
- **Storage**
- **uart**
- **rst**
Separation architecture provides isolation

Kernel resets Agent SoC

Runs third-party code
No OS, full access to hardware

Manages storage, agent switching

User I/O
Reset button

USB
uart
rst
Storage
Separation architecture provides isolation

- Agent
- SoC
- Kernel
- SoC

- Runs third-party code
- No OS, full access to hardware
- Manages storage, agent switching
- User I/O
- Reset button
- USB
- UART
- RST
- Storage

launch(): load agent code + data
Separation architecture provides isolation

Agent runs on Agent SoC, independently of Kernel SoC
Separation architecture provides isolation

- User I/O
- Reset button
- USB
- Agent SoC
  - Runs third-party code
  - No OS, full access to hardware

- Kernel SoC
  - Manages storage, agent switching

exit(state): save state and terminate
Desired property: noninterference

Agent A runs  switch  Agent B runs

time
Desired property: noninterference

steal A's secrets?

Agent A runs

switch

Agent B runs

time
Desired property: noninterference

Agent A runs  switch  Agent B runs

steal A's secrets?

Time
Deterministic start ensures noninterference

- Run before starting any agent
- Clears state in SoC (puts chip in deterministic state)
Deterministic start ensures noninterference

World 0 (secret = 0)

World 1 (secret = 1)
Deterministic start ensures noninterference

World 0 (secret = 0)

World 1 (secret = 1)

Agent A runs
Deterministic start ensures noninterference
Deterministic start ensures noninterference

World 0 (secret = 0)

World 1 (secret = 1)

Agent A runs
Deterministic start
Agent B runs
Deterministic start ensures noninterference
Challenge: completeness

- Lots of state
  - Registers
  - Microarchitectural state: CPU caches, ...
  - RAM
  - SoC peripherals: UART, SPI, ...

- Must work for all states
Simple approaches fail

- Reset pin
  - Clears minimal state necessary to restart

- Power cycling
  - State takes minutes to decay (cold boot attacks)
Notary’s approach: use software

• Reset returns control

• Software in boot ROM can clear internal state

• How to write this code?
  • Must clear every single bit of internal state
Gate-level description captures all internal state

RTL (e.g. Verilog): all digital state is explicit

⇒ SMT-compatible format
(for symbolic circuit simulation)
Verifying deterministic start for Notary’s SoC
Verifying deterministic start for Notary's SoC

/* no reset code */
Verifying deterministic start for Notary's SoC

/* no reset code */

error, state not cleared:
soc.cpu.latched_rd
Verifying deterministic start for Notary's SoC
Verifying deterministic start for Notary's SoC

error, state not cleared: soc.cpu.cpuregs[1]
Verifying deterministic start for Notary's SoC

```assembly
nop
nop
nop
/* clear registers */
li x1, 0 /* ... */
li x31, 0
```
Verifying deterministic start for Notary's SoC

```assembly
nop
nop
nop
/* clear registers */
li x1, 0 /* ... */
li x31, 0
```

error, state not cleared:
soc.cpu.mem_wdata
Verifying deterministic start for Notary's SoC

nop
nop
nop

/* clear registers */
li x1, 0 /* ... */
li x31, 0

/* clear buffer */
sw zero, 0(zero)
Verifying deterministic start for Notary's SoC

nop
nop
nop
/* clear registers */
li x1, 0 /* ... */
li x31, 0
/* clear buffer */
sw zero, 0(zero)

error, state not cleared:
soc.ram.data[0]
Verifying deterministic start for Notary's SoC

```
nop
nop
nop

/* clear registers */
  li x1, 0 /* ... */
  li x31, 0

/* clear buffer */
  sw zero, 0(zero)

/* clear ram */
  la t0, _sram_start
  la t1, _sram_end

loop:
  sw zero, 0(t0)
  addi t0, t0, 4
  bne t0, t1, loop
```
Verifying deterministic start for Notary’s SoC

```plaintext
n
n
/* clear registers */
li x1, 0 /* ... */
li x31, 0
/* clear buffer */
sw zero, 0(zero)
/* clear ram */
la t0, _sram_start
la t1, _sram_end
loop:
sw zero, 0(t0)
addi t0, t0, 4
bne t0, t1, loop
```

error, state not cleared: soc.uart.cr0
Verifying deterministic start for Notary's SoC

```assembly
nop
nop
nop
/* clear registers */
li x1, 0 /* ... */
li x31, 0
/* clear buffer */
sw zero, 0(zero)
/* clear ram */
la t0, _sram_start
la t1, _sram_end
loop:
    sw zero, 0(t0)
    addi t0, t0, 4
    bne t0, t1, loop
/* clear uart control register */
la t0, _uart0
sw zero, 0(t0)
```

Verifying deterministic start for Notary's SoC

```assembly
nop
nop
nop
/* clear registers */
li x1, 0 /* ... */
li x31, 0
/* clear buffer */
sw zero, 0(zero)
/* clear ram */
lal t0, _sram_start
la t1, _sram_end
loop:
sw zero, 0(t0)
addi t0, t0, 4
bne t0, t1, loop
/* clear uart control register */
la t0, _uart0
sw zero, 0(t0)
```

deterministic start verified!

\[ n = 180342 \text{ cycles, } < 10 \text{ ms} \]

(mostly spent clearing RAM)
Notary hardware and system software

- Additional hardware: $8 (extra chips)
- TCB: 4000 LOC (mostly drivers)
Notary agent: Bitcoin

Bitcoin app (left) and agent (right)
Notary agent: web-app approval

Web app (left) and agent (right)
Evaluation summary: Notary is practical

Notary’s design prevents bugs while preserving developer and user experience.

(see paper)
Related work

• Non-wallet security devices [iOS enclave, Yubikey]

• Verified kernels [SeL4, Hyperkernel, Nickel, CertiKOS]

• Verified hardware [Kami, Hyperflow]

(see paper)
Conclusion

• Notary separation architecture
  • Reset-based switching: clearing state between switching agents
  • Verified deterministic start: ensuring state clearing is correct

• Notary prototype
  • RISC-V-based prototype
  • 2 agents: Bitcoin, web-app approval

anish.io/notary