Notary: A Device for Secure Transaction Approval

Anish Athalye Adam Belay Frans Kaashoek Robert Morris Nickolai Zeldovich

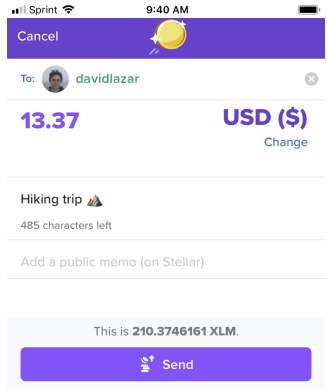
MIT CSAIL

How to securely approve transactions?

- Users perform sensitive transactional operations
  - Bank transfers
  - Cryptocurrency transactions
  - Deleting backups
  - Modifying DNS records

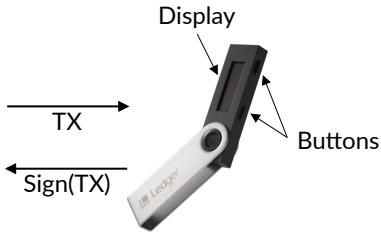
Common solution: smartphone apps

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



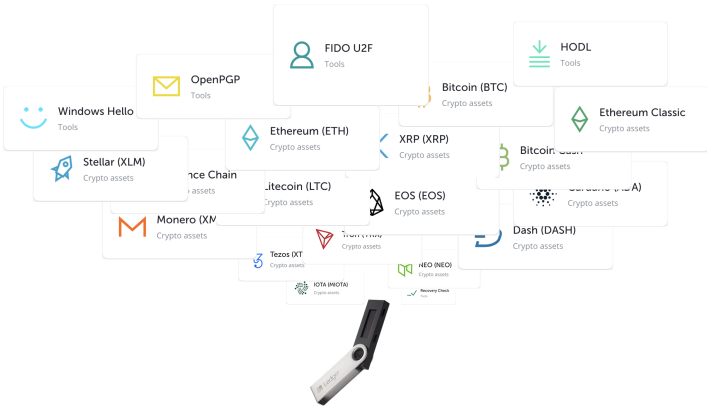
Approval **agent** on smartphone

Hardware wallets for transaction approval



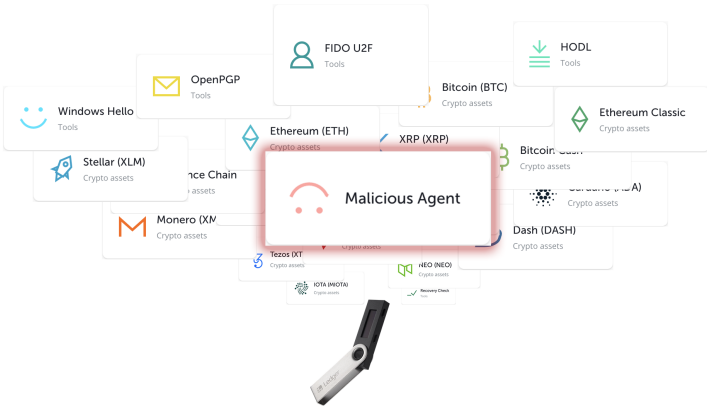
Ledger wallet

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)

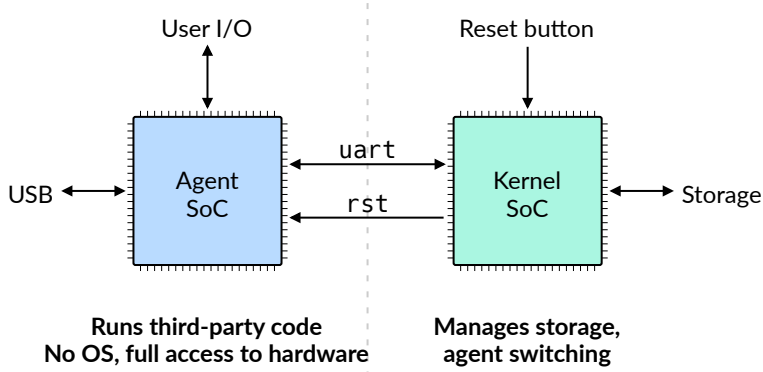
Contribution: Notary

- Agent separation architecture
  - **Reset-based switching**
  - **Verified deterministic start**
- Physical hardware wallet prototype

Threat model

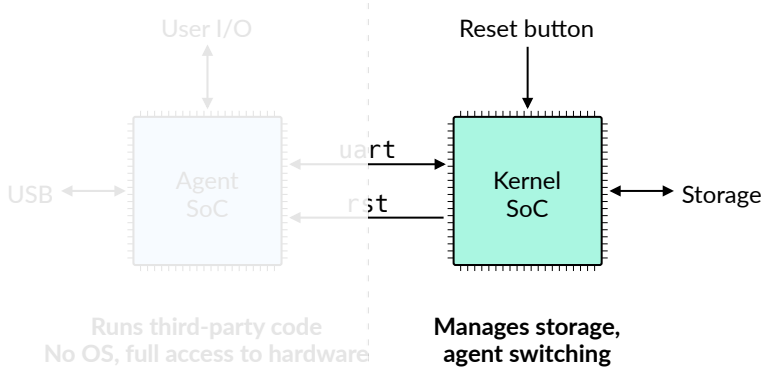
- Some agents are malicious
- Physical attacks out of scope
  - Could be addressed by tamper-proof hardware

Separation architecture provides isolation



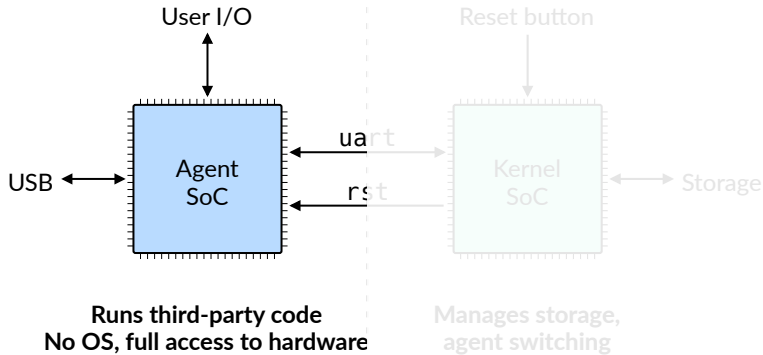
Notary separation architecture

Separation architecture provides isolation



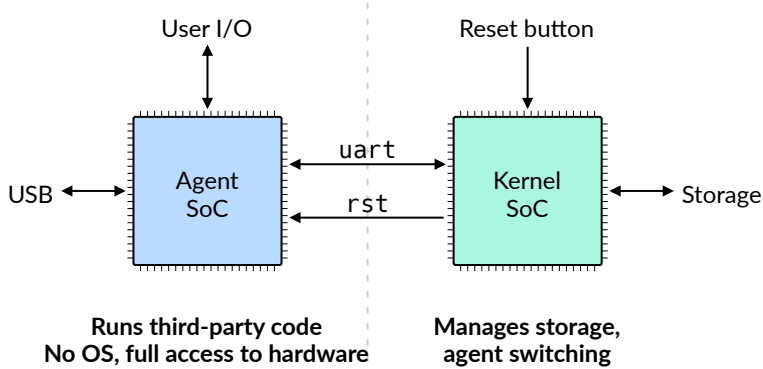
Kernel SoC

Separation architecture provides isolation



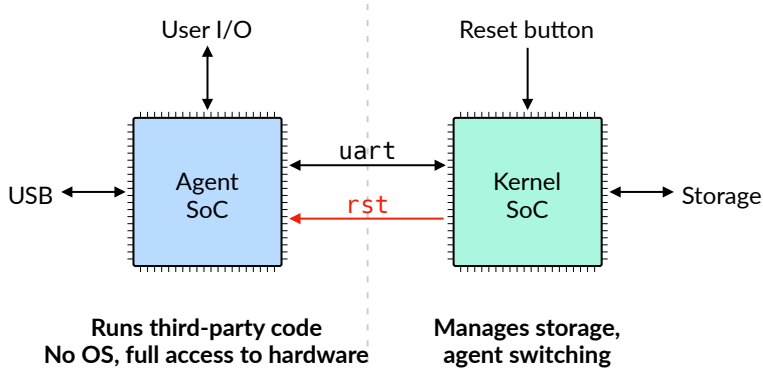
Agent SoC

Separation architecture provides isolation



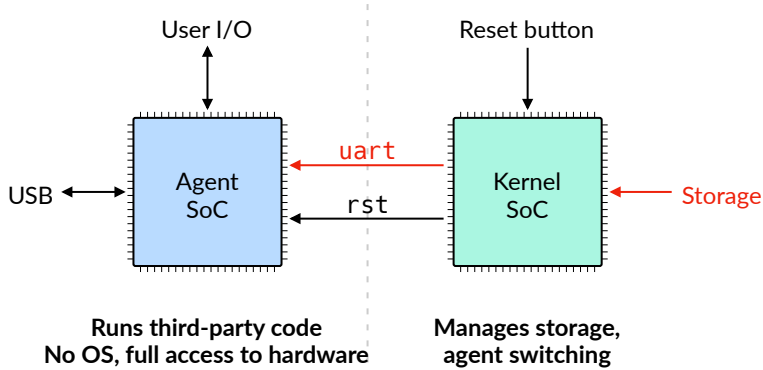
Connected only by UART (and reset wire)

Separation architecture provides isolation



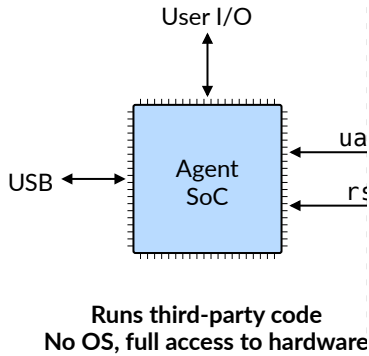
Kernel resets Agent SoC

Separation architecture provides isolation



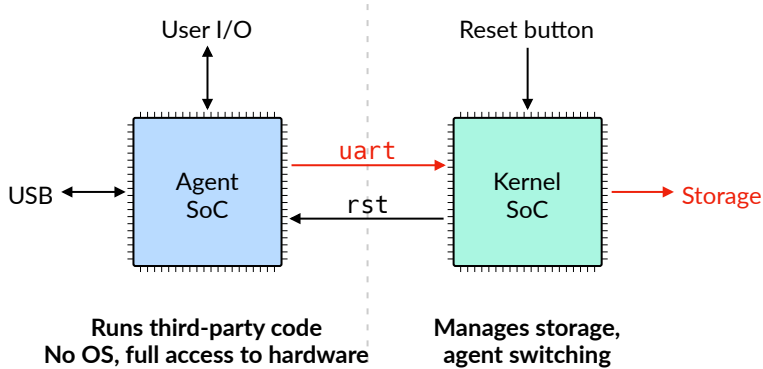
`launch()`: load agent code + data

Separation architecture provides isolation



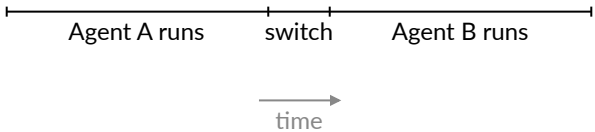
Agent runs on Agent SoC, independently of Kernel SoC

Separation architecture provides isolation

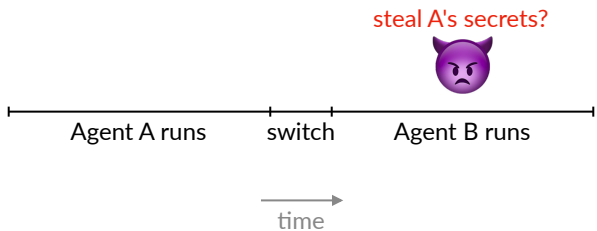


`exit(state)`: save state and terminate

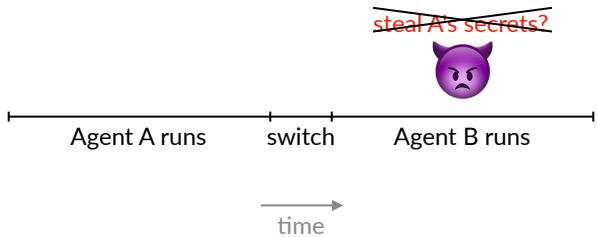
Desired property: noninterference



Desired property: noninterference



Desired property: noninterference



Deterministic start ensures noninterference

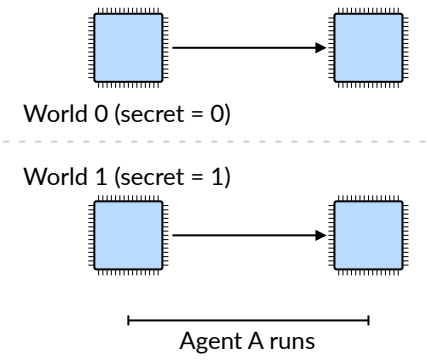
- Run before starting any agent
- Clears state in SoC (puts chip in deterministic state)

Deterministic start ensures noninterference

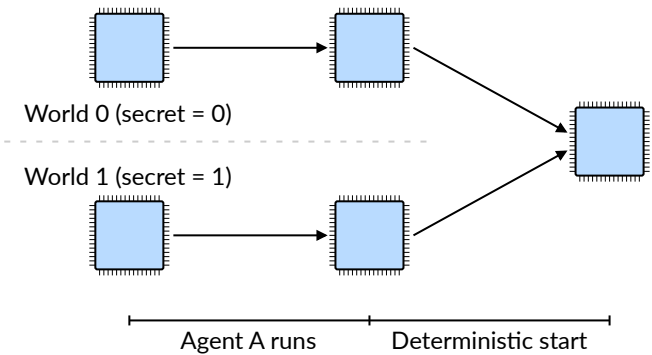
World 0 (secret = 0)

World 1 (secret = 1)

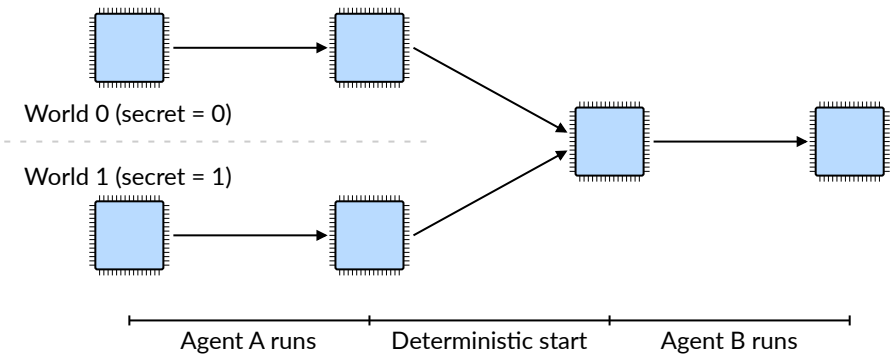
Deterministic start ensures noninterference



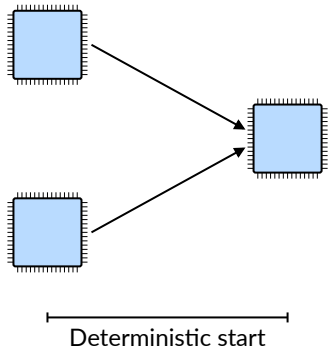
Deterministic start ensures noninterference



Deterministic start ensures noninterference



Deterministic start ensures noninterference



Challenge: completeness

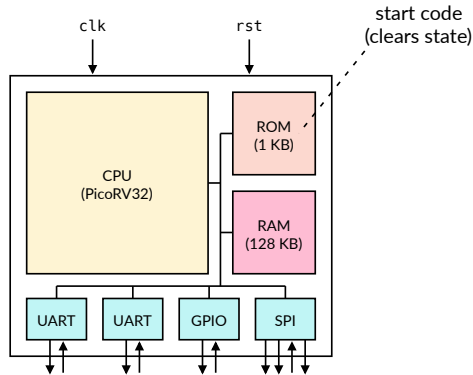
- Lots of state
  - Registers
  - Microarchitectural state: CPU caches, ...
  - RAM
  - SoC peripherals: UART, SPI, ...
- Must work for all states

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

```
always @(posedge clk) begin
        if (!resetn || trap) begin
                if (!resetn)
                        mem_state <= 0;
                if (!resetn || mem_ready)
                        mem_valid <= 0;
                mem_la_secondword <= 0;
                prefetched_high_word <= 0;
        end else begin
                if (mem_la_read || mem_la_write) begin
                        mem_addr <= mem_la_addr;
                        mem_wstrb <= mem_la_wstrb & {4{mem_la_write}};
                end
                if (mem_la_write) begin
                        mem_wdata <= mem_la_wdata;
                end
                case (mem_state)
                        0: begin
                                if (mem_do_prefetch || mem_do_rinst || mem_do_rdata) begin
                                        mem_valid <= !mem_la_use_prefetched_high_word;
                                        mem_instr <= mem_do_prefetch || mem_do_rinst;
                                        mem_wstrb <= 0;
                                        mem_state <= 1;
                                end
                                if (mem_do_wdata) begin
                                        mem_valid <= 1;
                                        mem_instr <= 0;
                                        mem_state <= 2;
                                end
                        end
                        1: begin
                                assert(mem_wstrb == 0);
                                assert(mem_do_prefetch || mem_do_rinst || mem_do_rdata);
                                assert(mem_valid == !mem_la_use_prefetched_high_word);
                                assert(mem_instr == (mem_do_prefetch || mem_do_rinst));
                                if (mem_xfer) begin
                                        if (COMPRESSED_ISA && mem_la_read) begin
                                                mem_valid <= 1;
                                                mem_la_secondword <= 1;
                                                if (!mem_la_use_prefetched_high_word)
                                                        mem_16bit_buffer <= mem_rdata[31:16];
                                        end else begin
                                                mem_valid <= 0;
                                                mem_la_secondword <= 0;
                                                if (COMPRESSED_ISA && !mem_do_rdata) begin
                                                        if (~mem_rdata[1:0] || mem_la_secondword) begin
                                                                mem_16bit_buffer <= mem_rdata[31:16];
                                                                prefetched_high_word <= 1;
                                                        end else begin
                                                                prefetched_high_word <= 0;
                                                        end
                                                end
```

$\Longrightarrow$ SMT-compatible format
(for symbolic circuit simulation)

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

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

```
nop
nop
nop
```

Verifying deterministic start for Notary's SoC

```
nop
nop
nop
```

error, state not cleared:
`soc.cpu.cpuregs[1]`

Verifying deterministic start for Notary's SoC

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

Verifying deterministic start for Notary's SoC

```
  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

```
  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
```

error, state not cleared:
`soc.uart.cr0`

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
/* clear uart control register */
  la t0, _uart0
  sw zero, 0(t0)
```

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
/* clear uart control register */
  la t0, _uart0
  sw zero, 0(t0)
```

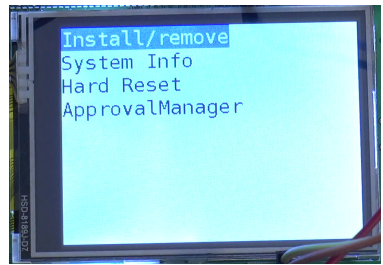
deterministic start verified!

$n = 180342$ cycles, < 10 ms

(mostly spent clearing RAM)

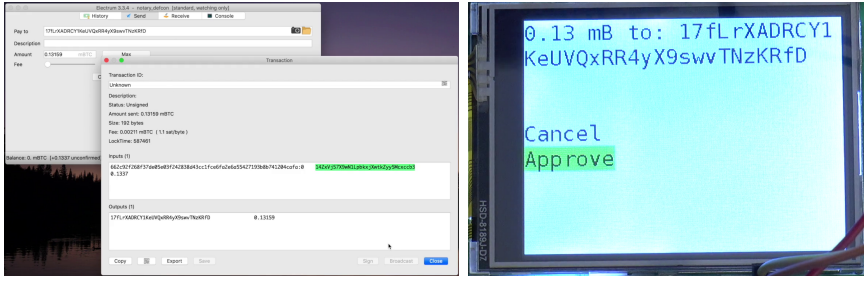
Notary hardware and system software

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



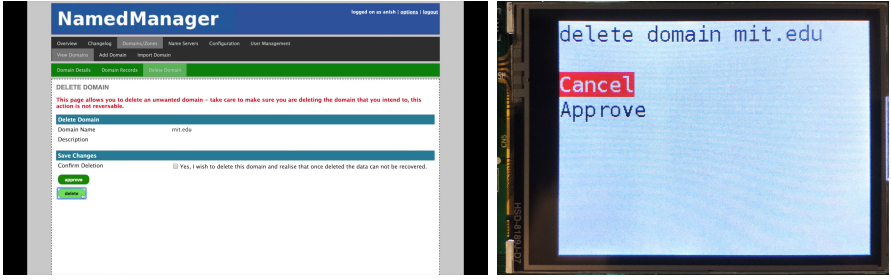
Notary prototype

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