ICE: A Passive, High-Speed, State-Continuity Scheme

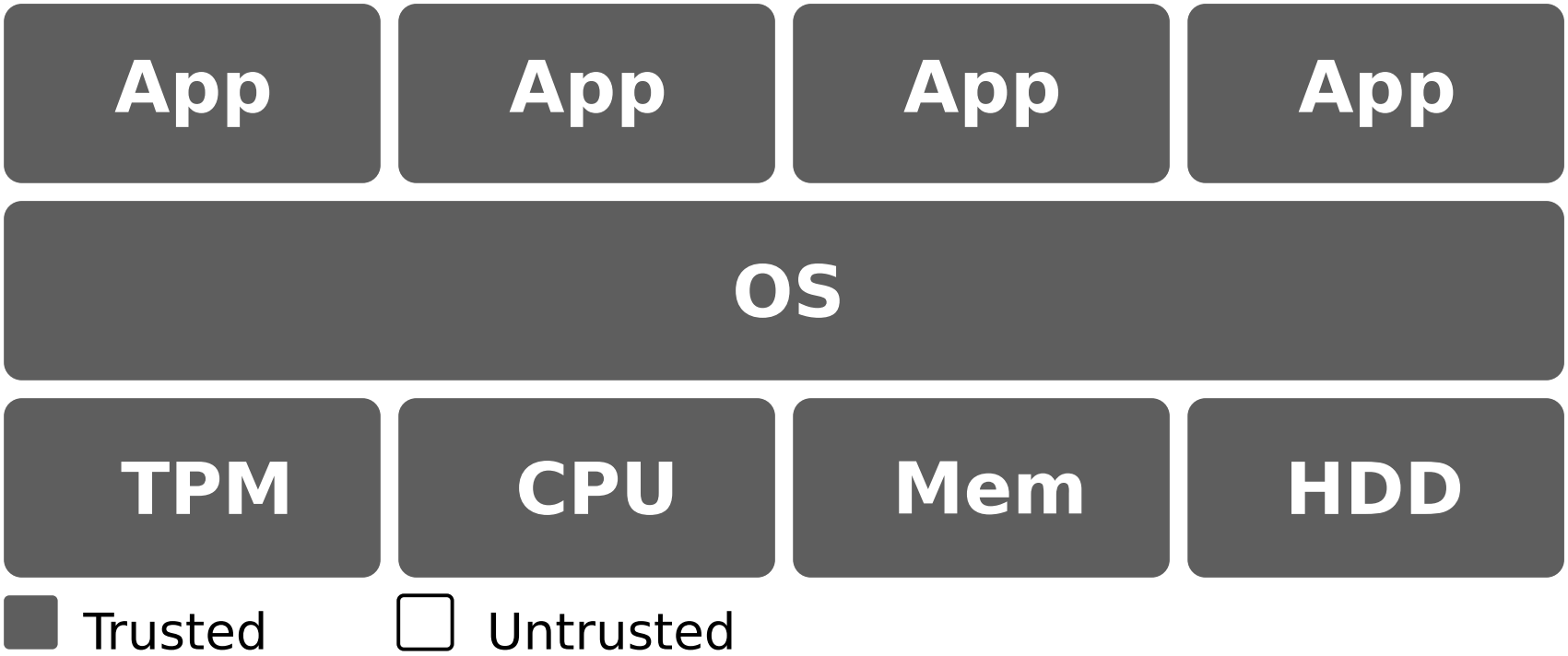
Raoul Strackx

KU Leuven

December 10, 2014

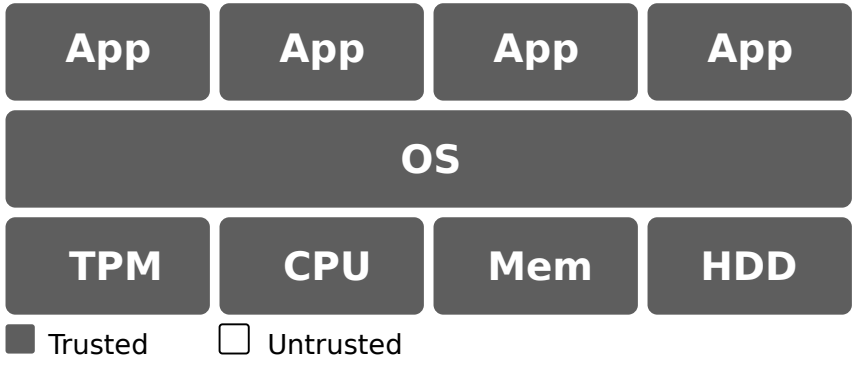
Outline

What we have. . .



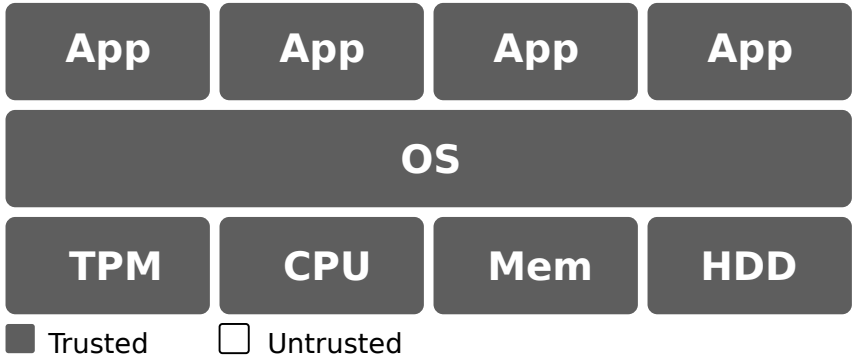
What we have. . .

- Windows 7: 40 million LoC
- Linux kernel: 12 million LoC
- Firefox: 6 million LoC



What we have. . .

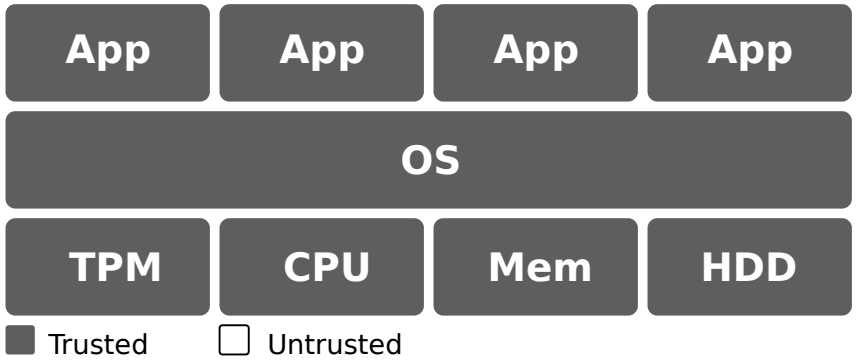
- More Secure Libraries
- Harden Legacy Software Automatically
- Memory-Safe Languages
- Software Verification
- OS Protection Features
- Hardware Security Modules



What we have. . .

- More Secure Libraries
- Harden Legacy Software Automatically
- Memory-Safe Languages
- Software Verification
- OS Protection Features
- Hardware Security Modules

. . . But software security is still a mess



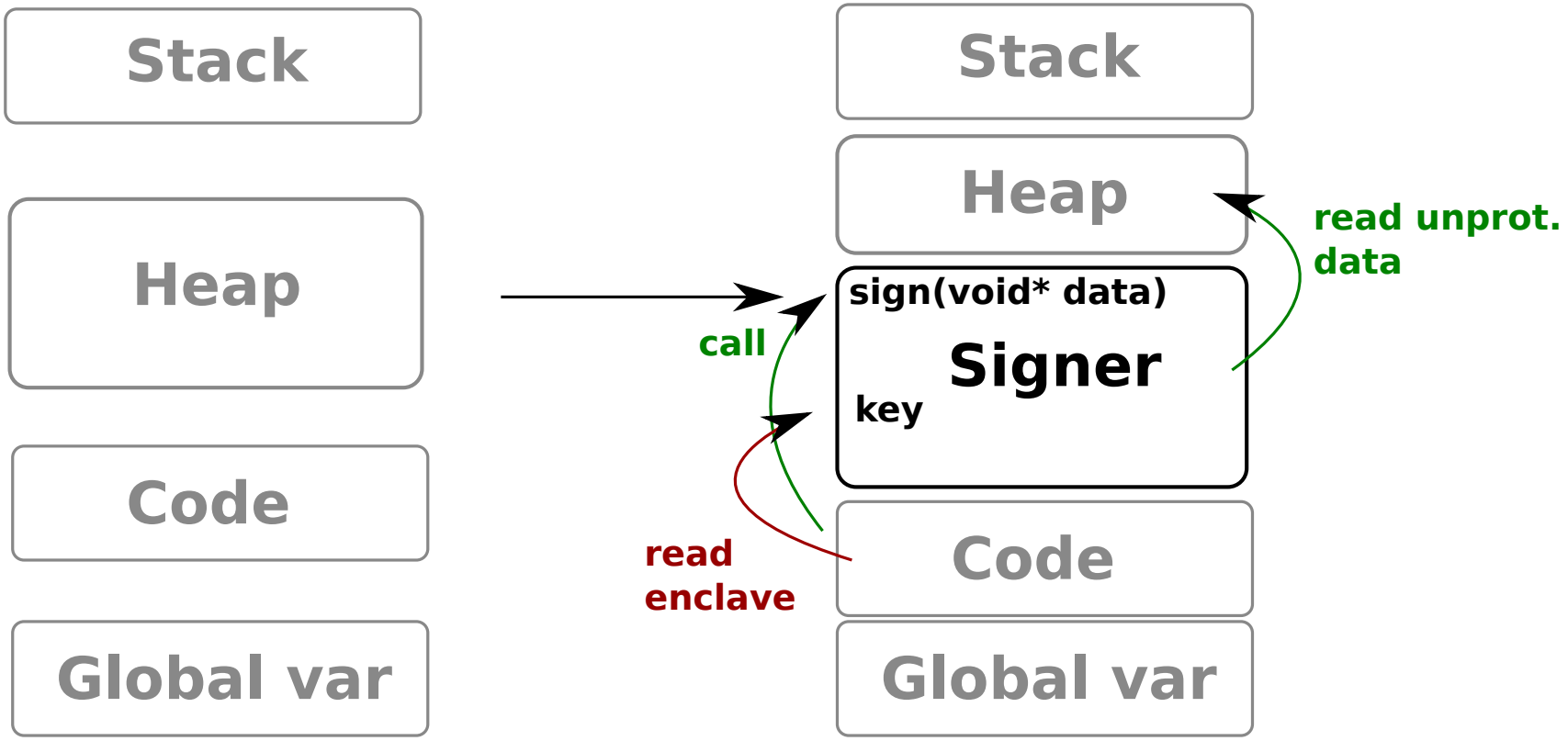
PMAs: A possible way out

Main idea:

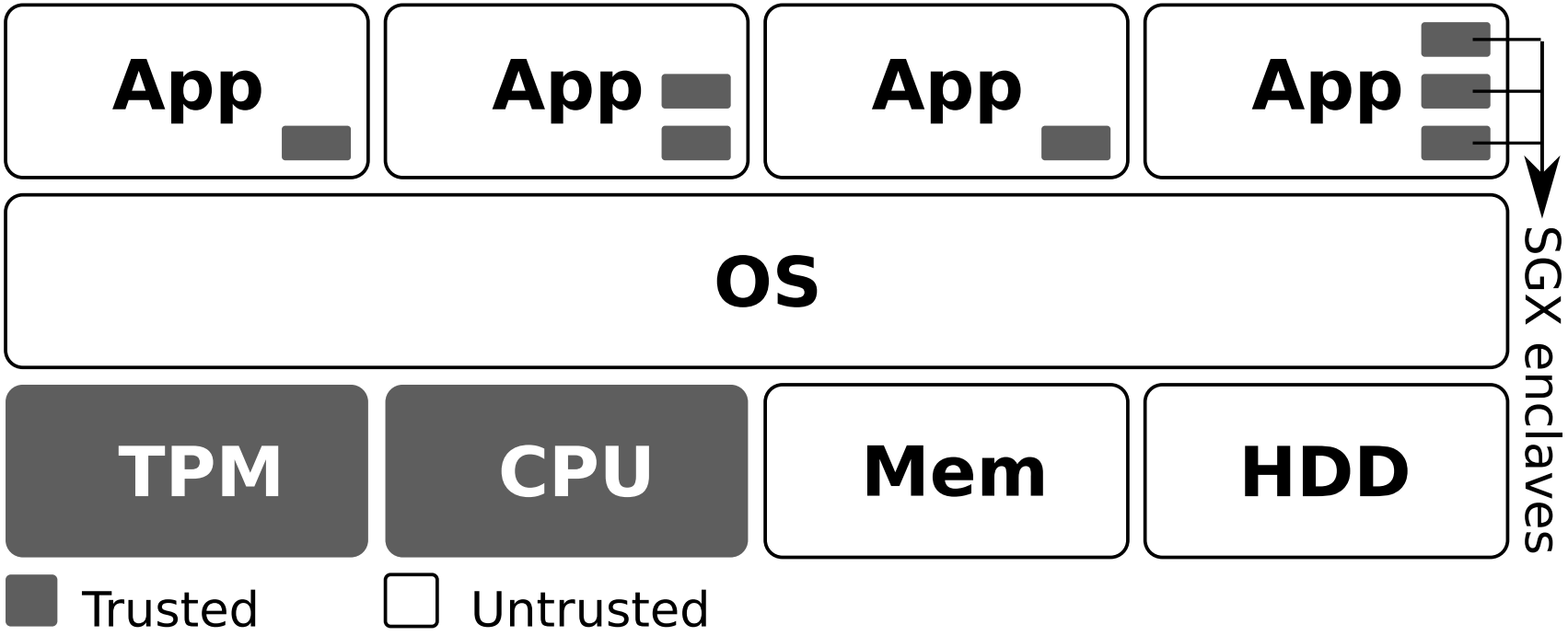
- Carve out sensitive parts of an application
- Apply hardware-based isolation to these *modules*
- Modules have complete control over their own security

This idea is known as a **Protected-Module Architecture** and will soon be available on commodity Intel processors (Intel SGX)

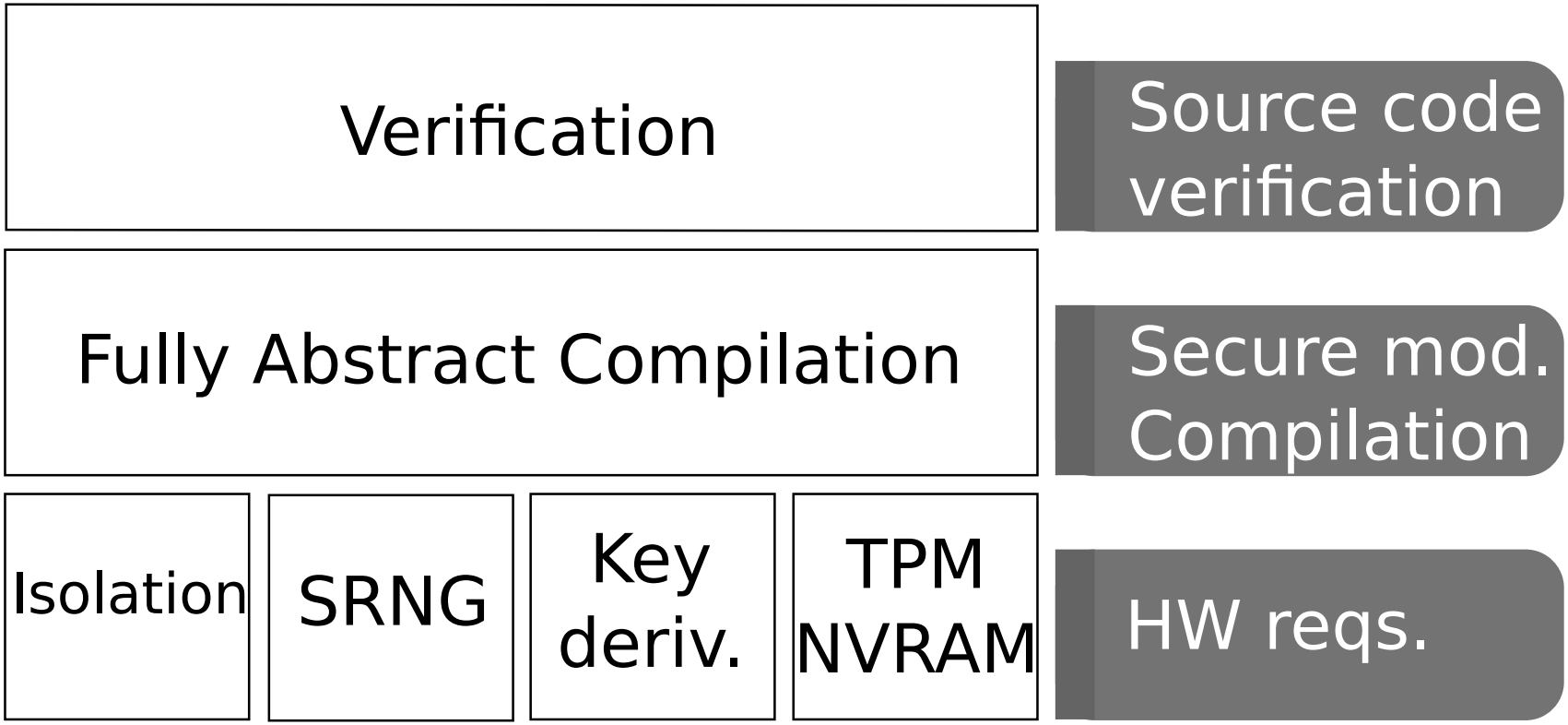
An example:


Stack
Heap
Code
Global var
call
Stack
Heap
sign(void* data)
Signer
key
read unprot.
data
read
enclave
Code
Global var

The result:


App
App
App
App
OS
SGX enclaves
TPM
CPU
Mem
HDD
Trusted
Untrusted

What we have. . .


Verification
Source code verification
Fully Abstract Compilation
Secure mod. Compilation
Isolation
SRNG
Key deriv.
TPM NVRAM
HW reqs.

What we can guarantee:

- SW properties of modules cannot be broken by SW attacks (Fully-abstract compilation[1] & verification[2])
- Resistance against powerful HW attacks

[1] Patrignani et al. "Secure compilation to Protected Module Architectures". *Accepted for publication in Transactions on Programming Languages and Systems (TOPLAS)*

[2] Agten, Jacobs, and Piessens. "Sound Modular Verification of C Code Executing in an Unverified Context". 2015. *Accepted for publication in Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*

What we can guarantee:

- SW properties of modules cannot be broken by SW attacks (Fully-abstract compilation[1] & verification[2])
- Resistance against powerful HW attacks

While the system executes continuously

. . . but in real-life systems crash, reboot and power goes down

[1] Patrignani et al. "Secure compilation to Protected Module Architectures". *Accepted for publication in Transactions on Programming Languages and Systems (TOPLAS)*

[2] Agten, Jacobs, and Piessens. "Sound Modular Verification of C Code Executing in an Unverified Context". 2015. *Accepted for publication in Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*

Outline

Providing State Continuity

Password-checking module

```
static int tries_left = 3;
static string password = "mypassword";

boolean check_password( string input ) {
   if ( tries_left == 0 )
      return false;

   if ( password == input ) {
      tries_left = 3;

      return true;
   } else {
      --tries_left;

      return false;
   }
}
```

Providing State Continuity

Password-checking module

```
static int tries_left = 3;
static string password = "mypassword";

boolean check_password( string input ) {
   if ( tries_left == 0 )
      return false;

   if ( password == input ) {
      tries_left = 3;
      store_state() //store state encrypted and signed
      return true;
   } else {
      --tries_left;
      store_state() //store state encrypted and signed
      return false;
   }
}
```

Providing State Continuity

Consider two possible attacks:

Attack 1: Providing stale state

- Store the initial state
- Guess the password 3 times
- Crash the system
- Provide the initial state stored on disk as being fresh

Password-checking module

```
static int tries_left = 3;
static string password = "mypassword";

boolean check_password( string input ) {
   if ( tries_left == 0 )
      return false;

   if ( password == input ) {
      tries_left = 3;
      store_state()
      return true;
   } else {
      --tries_left;
      store_state()
      return false;
   }
}
```

Providing State Continuity

Attack 2: Timing attack

- Guess the password
- Crash the system before storing the state if:

$$time_{correct} < time_{wrong}$$

- Restore the last state

Password-checking module

```
static int tries_left = 3;
static string password = "mypassword";

boolean check_password( string input ) {
   if ( tries_left == 0 )
      return false;

   if ( password == input ) {
      tries_left = 3;
      store_state()
      return true;
   } else {
      --tries_left;
      store_state()
      return false;
   }
}
```

Requirements

We need to add support for *state continuity*

Three key properties:

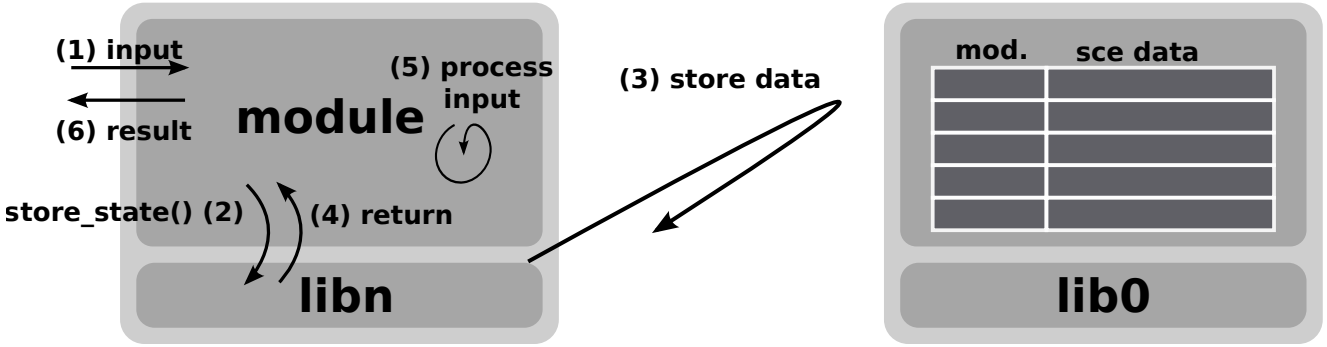
- **Safety**: prevent the use of stale data
- **Liveness**: a crash of the machine must not leave the machine unable to advance
- **Speed**: must be applicable to most applications

Outline

Side note:

- First store input, then process
- Modules are deterministic

→ We only need to consider state-continuous storage

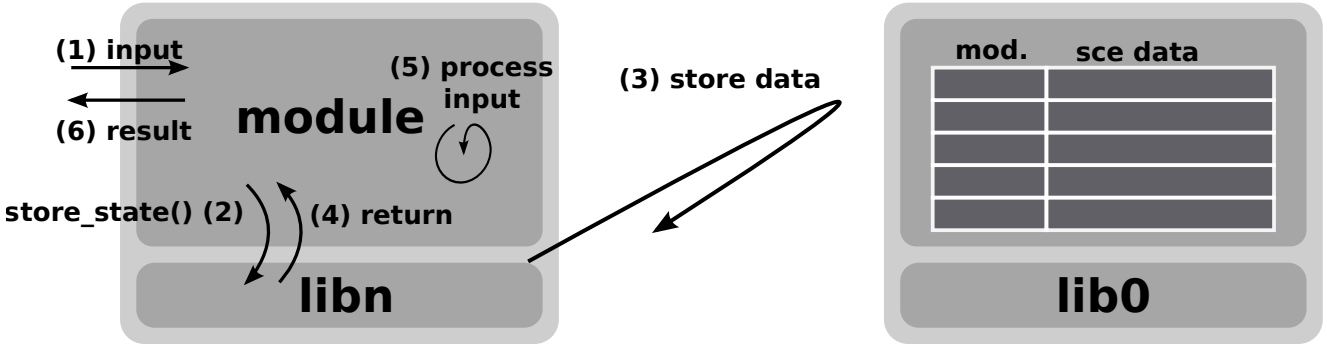


Side note:

- First store input, then process
- Modules are deterministic

→ We only need to consider state-continuous storage

- (Only provide state-continuous execution to a single module)



Building blocks: Guards

Definition:

$$\text{guard}_i(n) = (\underbrace{\text{Hash}^i(n)}_{\text{guard value}}, \underbrace{i}_{\text{guard index}})$$

Incrementing a guard:

$$\begin{aligned}\text{guard}_0(n) &= (n, 0)\\ \text{guard}_i(n) &= (\text{Hash}^i(n), i)\\ \text{guard}_{i+1}(n) &= (\text{Hash}^{i+1}(n), i+1)\end{aligned}$$

Comparison of guards:

$$(n,i) \leq (m,j) \Leftrightarrow \begin{cases} n = m & \text{if } i = j \\ (\text{Hash}(n), i+1) \leq (m,j) & \text{if } i < j \end{cases}$$

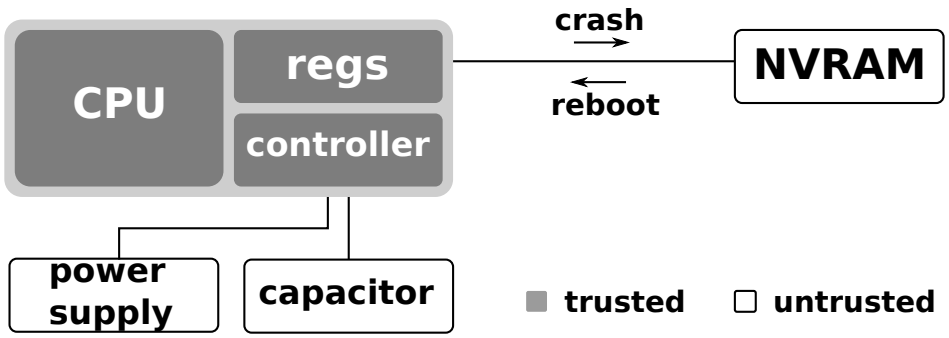
Building Blocks: Cubes

State stored on disk = Cubes:

Building blocks: Guarded memory

"Guarded Memory"

- Non-volatile memory
- Contents remains secret *only* while power is on
- Power-off will make the contents world-readable
- Stores freshness information



Main Idea

- Enclaves keep their state between invokations
- Store a guard that acts as a “pointer” to the fresh cube in guarded memory
- This pointer *only* leaks on crash/reboot/loss of power
- As no previous guard leaked, knowledge of this pointer proves freshness of the cube

Init Step

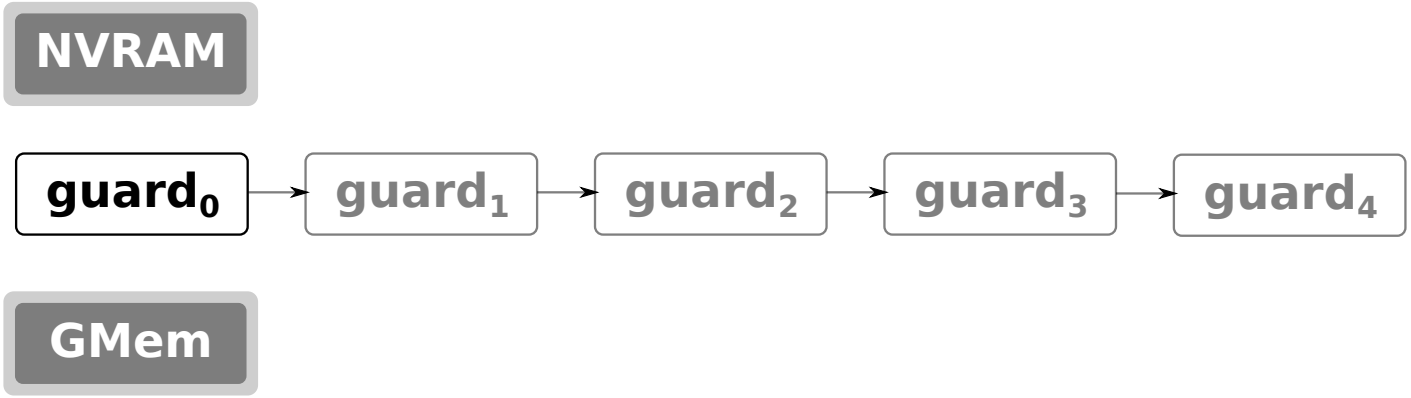
NVRAM

GMem

initial_state(State s0)

```
gmem.enable_protection();
guard = gen_guard();
key = gen_key();
hdd.write( new cube( s0, guard, key ) );
gmem.write( guard );
nvram.nvwrite( key, guard );
```

Init Step



initial_state(State s0)

```
gmem.enable_protection();
guard = gen_guard();
key = gen_key();
hdd.write( new cube( s0, guard, key ) );
gmem.write( guard );
nvram.nvwrite( key, guard );
```

Init Step

initial_state(State s0)

```
gmem.enable_protection();
guard = gen_guard();
key = gen_key();
hdd.write( new cube( s0, guard, key ) );
gmem.write( guard );
nvram.nvwrite( key, guard );
```

Init Step

initial_state(State s0)

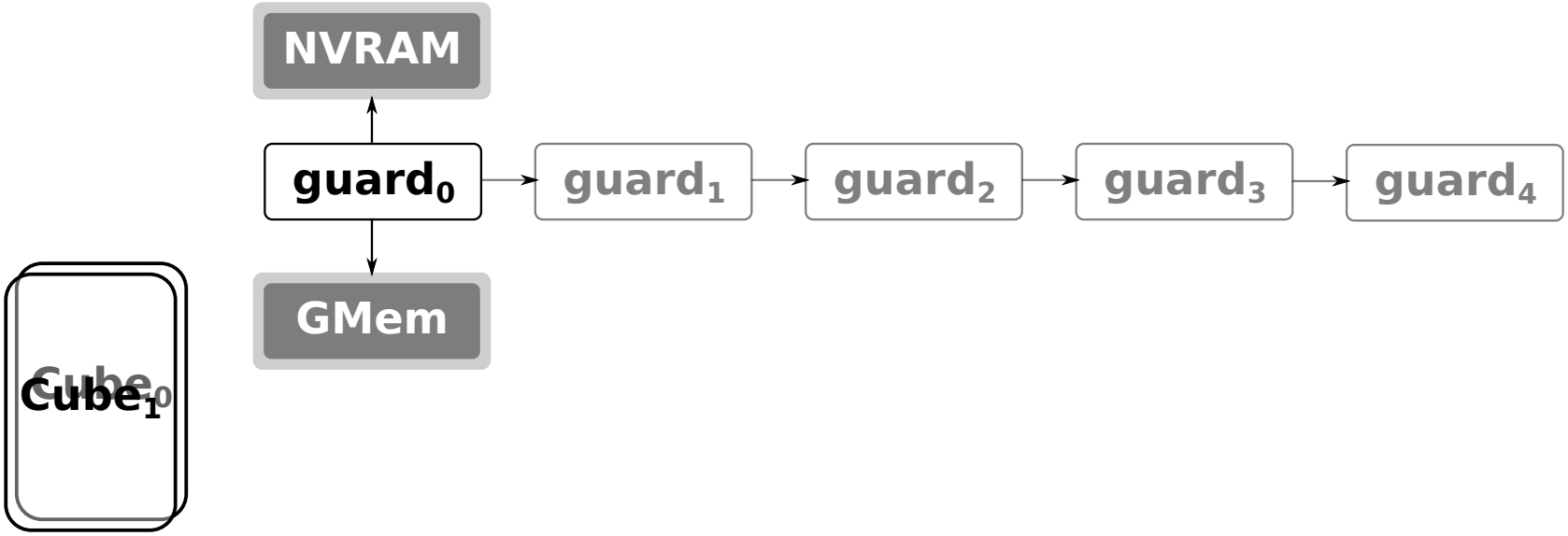
```
gmem.enable_protection();
guard = gen_guard();
key = gen_key();
hdd.write( new cube( s0, guard, key ) );
gmem.write( guard );
nvram.nvwrite( key, guard );
```

Init Step

initial_state(State s0)

```
gmem.enable_protection();
guard = gen_guard();
key = gen_key();
hdd.write( new cube( s0, guard, key ) );
gmem.write( guard );
nvram.nvwrite( key, guard );
```

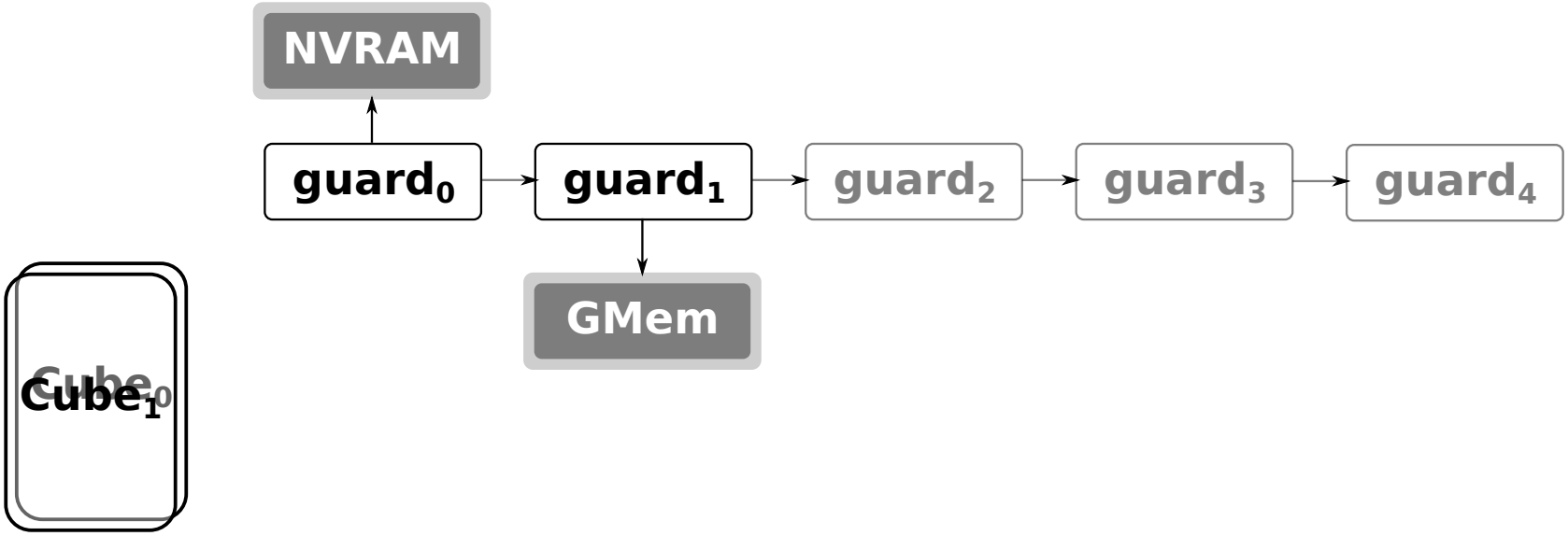
Next Step



store_state(State s1)

```
++guard;
hdd.write( new cube( s1, guard, key ) );
gmem.write( guard );
```

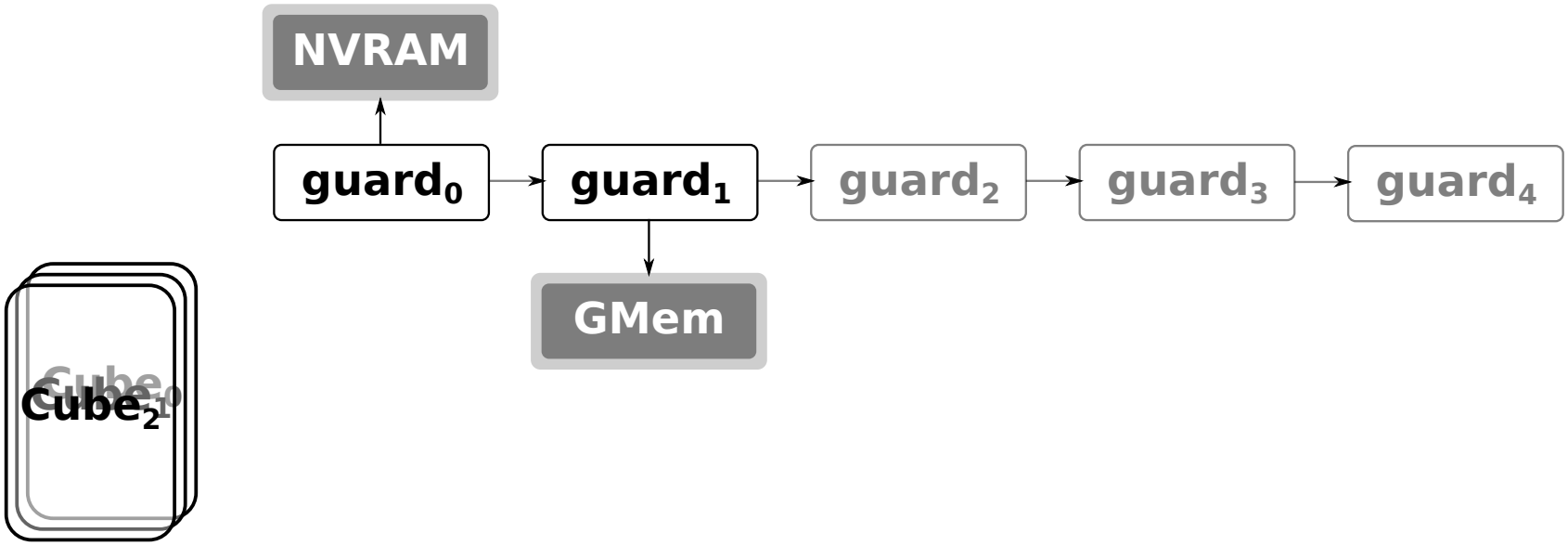
Next Step



store_state(State s1)

```
++guard;
hdd.write( new cube( s1, guard, key ) );
gmem.write( guard );
```

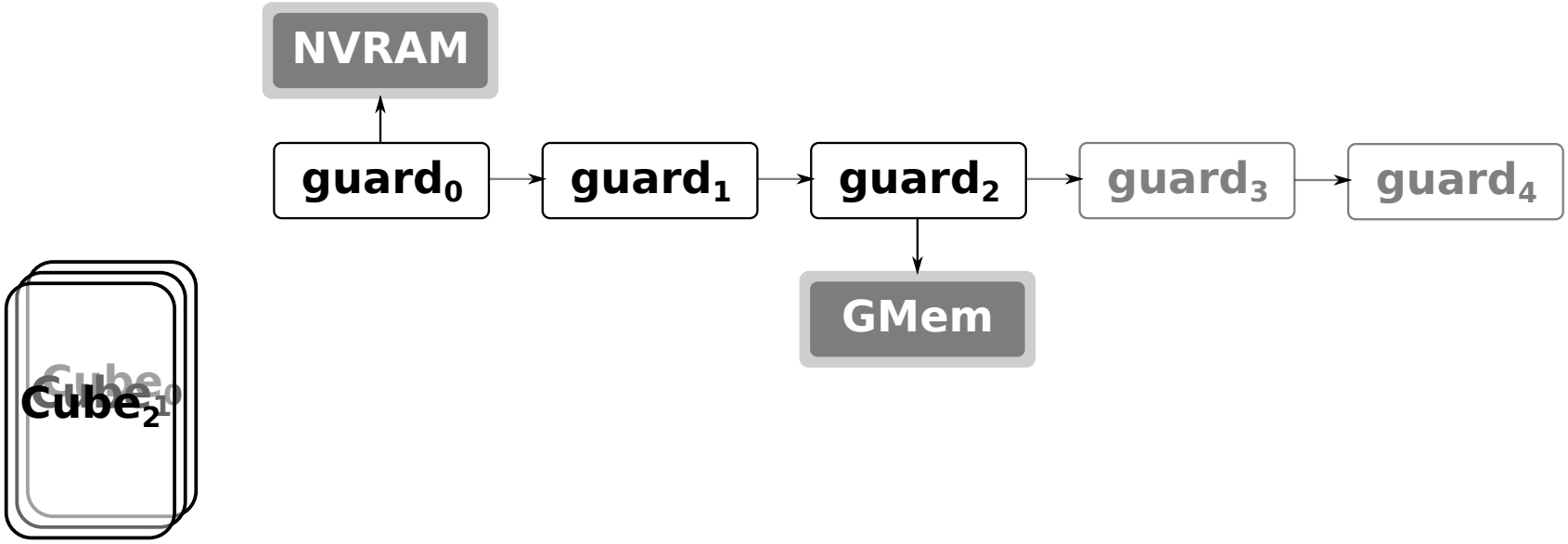
Next Step



store_state(State s2)

```
++guard;
hdd.write( new cube( s2, guard, key ) );
gmem.write( guard );
```

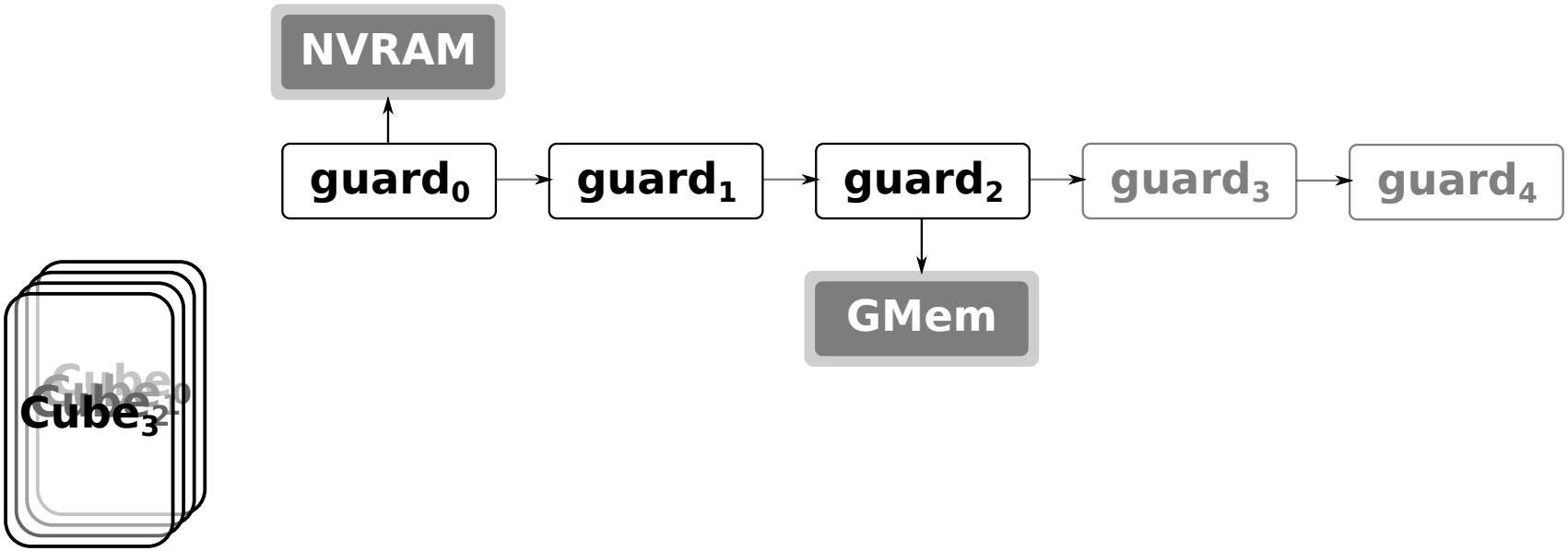
Next Step



store_state(State s2)

```
++guard;
hdd.write( new cube( s2, guard, key ) );
gmem.write( guard );
```

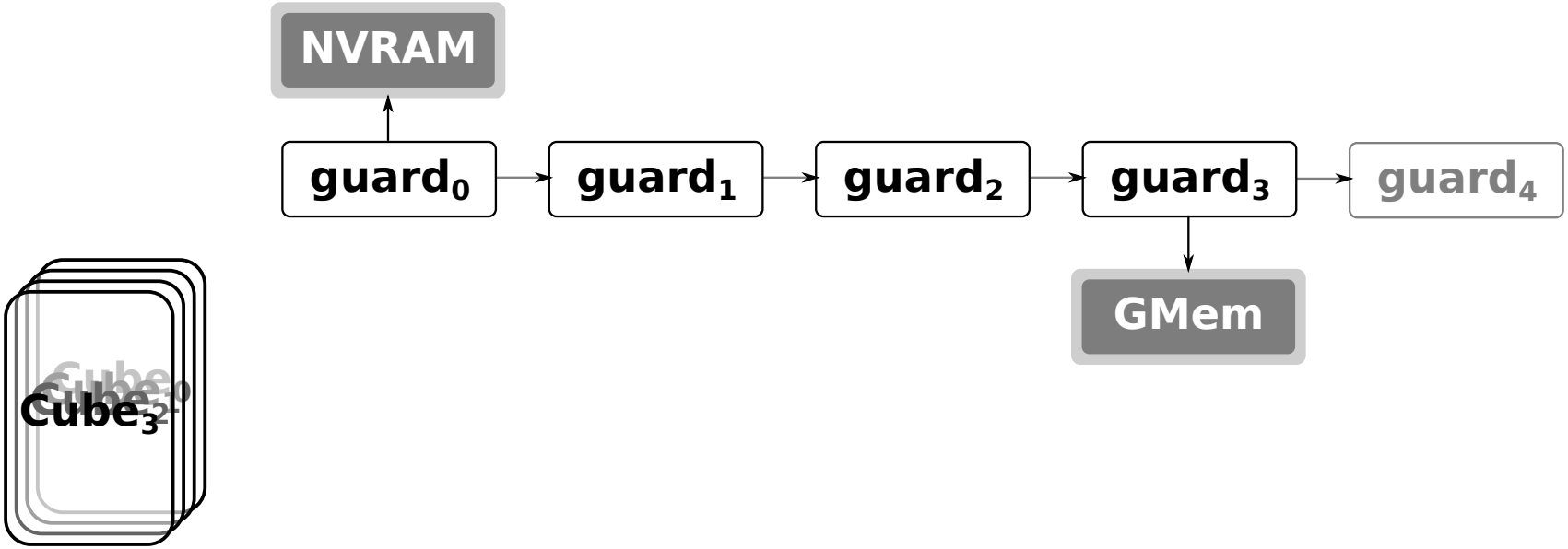
Next Step



store_state(State s3)

```
++guard;
hdd.write( new cube( s3, guard, key ) );
gmem.write( guard );
```

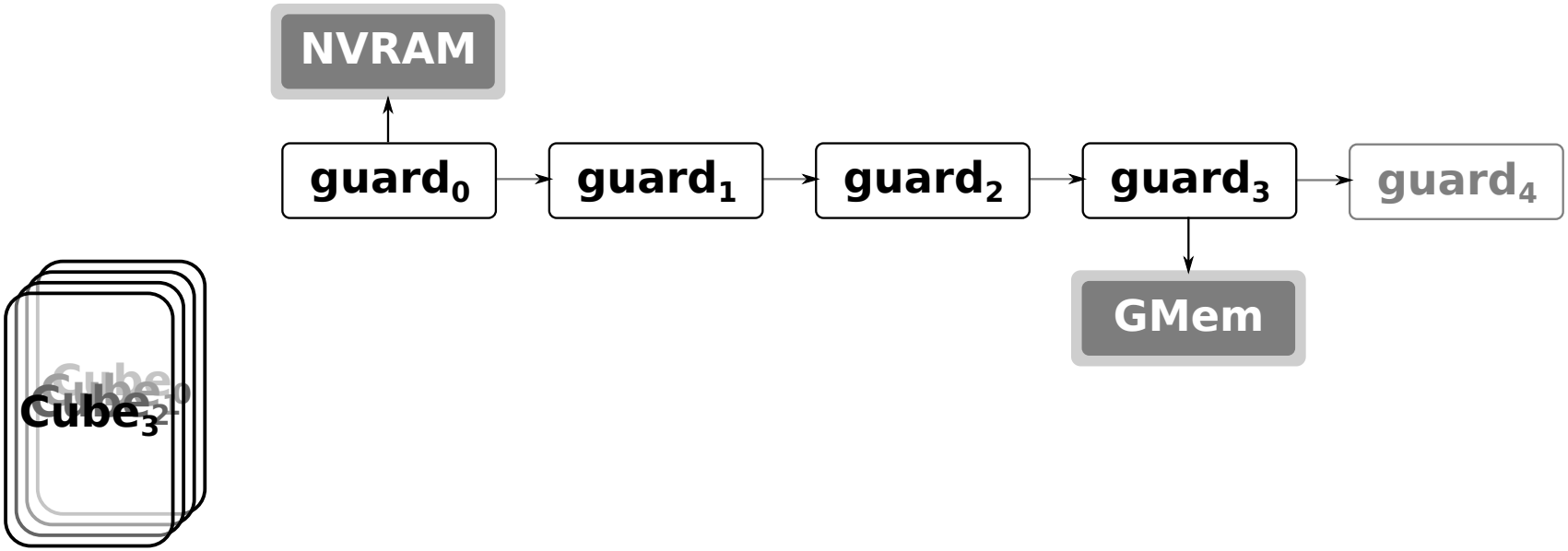
Next Step



store_state(State s3)

```
++guard;
hdd.write( new cube( s3, guard, key ) );
gmem.write( guard );
```

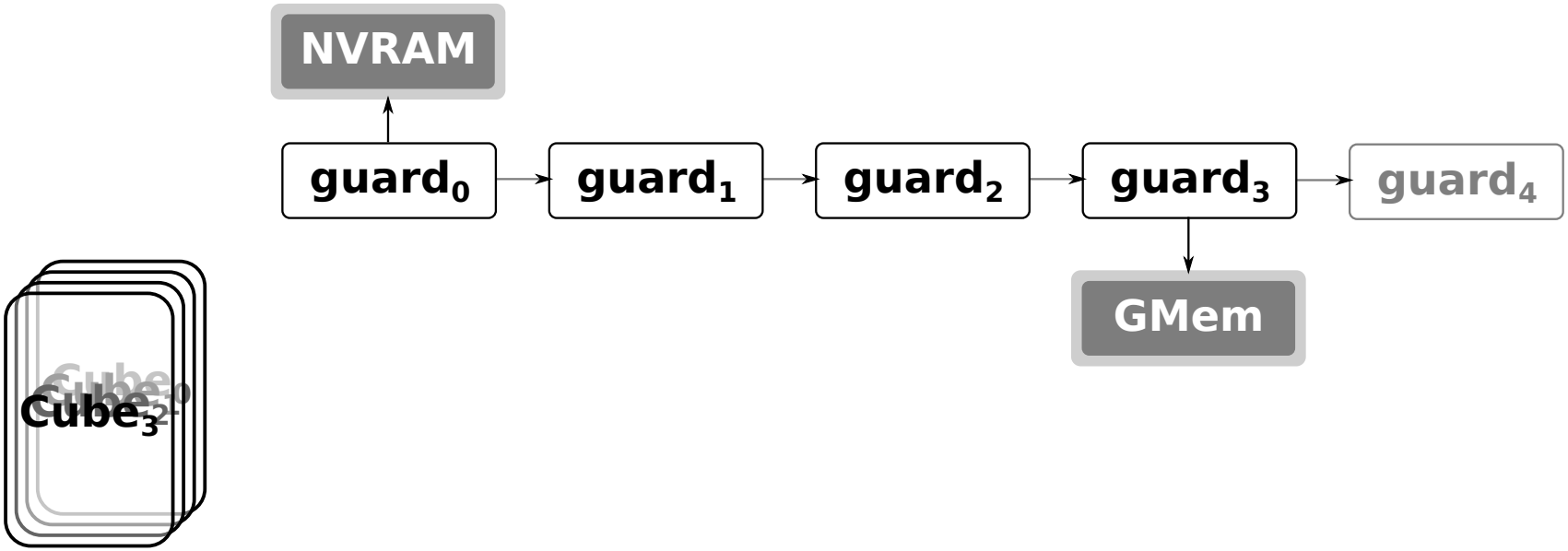
Crash!



recovery_step(void)

```
...
bool is_fresh( Cube *cube, Guard fresh ) {
  return ( check_mac( cube, tpm.keys ) &&
    fresh.value == cube->guard.value &&
    tpm.guard <= cube->guard );
}
```

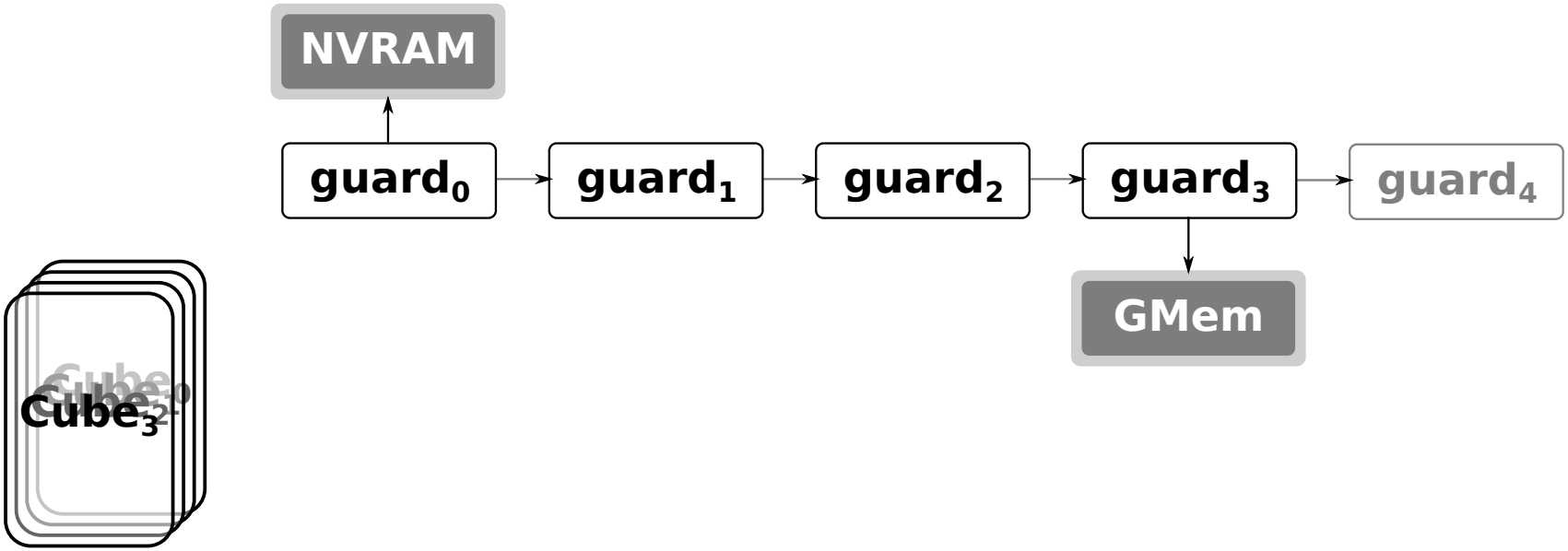
Crash!



recovery_step(void)

```
...
bool is_fresh( Cube *cube, Guard fresh ) {
  return ( check_mac( cube, tpm.keys ) &&
    fresh.value == cube->guard.value &&
    tpm.guard <= cube->guard );
}
```

Crash!



recovery_step(void)

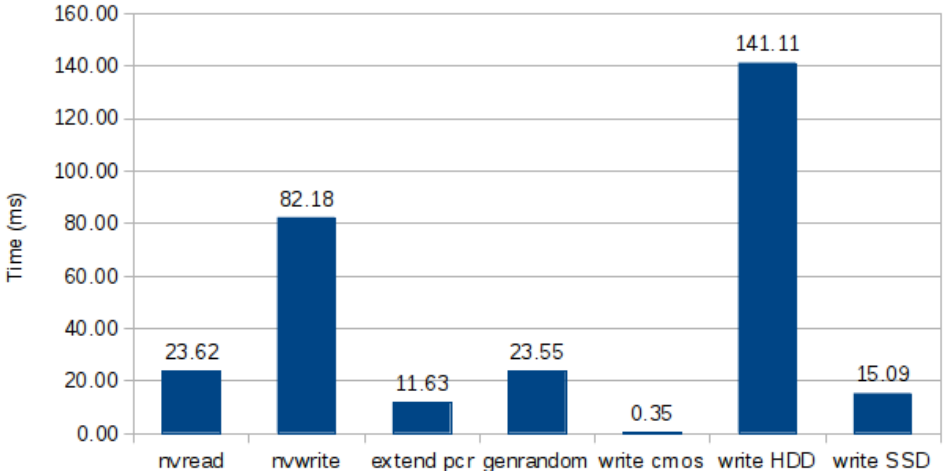
```
...
bool is_fresh( Cube *cube, Guard fresh ) {
  return ( check_mac( cube, tpm.keys ) &&
    fresh.value == cube->guard.value &&
    tpm.guard <= cube->guard );
}
```

Outline

Evaluation: Security

- System can do down at *any* point in time
- Formal + machine checked proof
- (More details: see paper/talk to me offline)

Evaluation: Performance



	Password	
SSD (in ms)	`-lice0`	`-licen`
computation	0.06	0.13
writing guard	0.33	0.33
writing cubes	14.61	17.42
total	15.00	17.87

HDD (in ms)	`-lice0`	`-licen`
computation	0.06	0.12
writing guard	0.35	0.35
writing cubes	112.80	183.23
total	113.21	183.71

With the right hardware support:

587 times faster than TPM NVRAM accesses!

Why is this important:

- It's applicable in practice *without* reducing SGX's attack model
- Speed of TPM NVRAM is *not* an issue
- Opens up the way to rely on even less HW requirements
- Enables a large range of applications[3]

[3] Strackx and Lambrigts. "Idea: State-Continuous Transfer of State in Protected-Module Architectures". 2015. *Accepted in Engineering Secure Software and Systems (ESSoS'15)*

Outline

Conclusion

Provide state-continuous execution that is

- A practical security primitive for PMAs
- Proven secure
- Fast
- Very limited HW requirements

Questions?

raoul.strackx@cs.kuleuven.be

https://distrinet.cs.kuleuven.be/software/pcbac