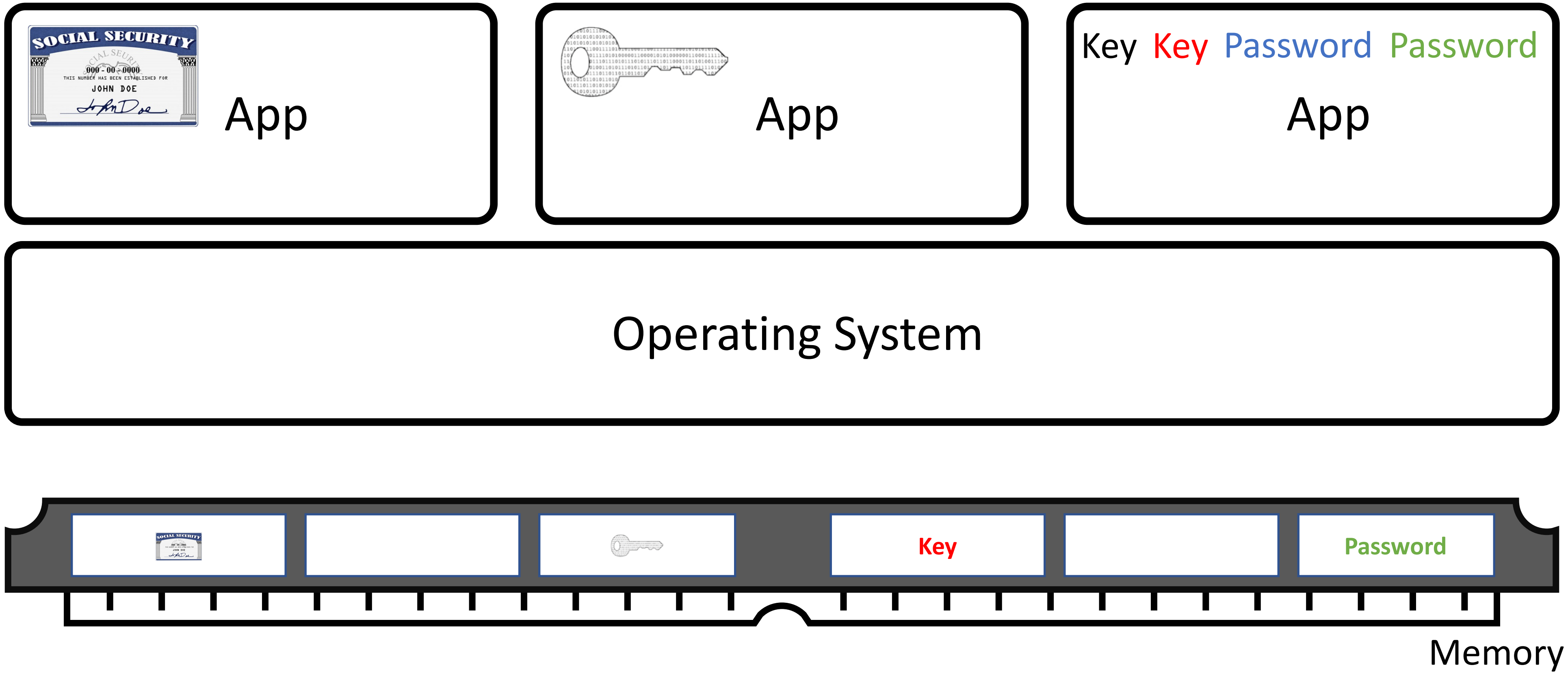
Ginseng: Keeping Secrets in Registers When You Distrust the Operating System

Min Hong Yun and Lin Zhong

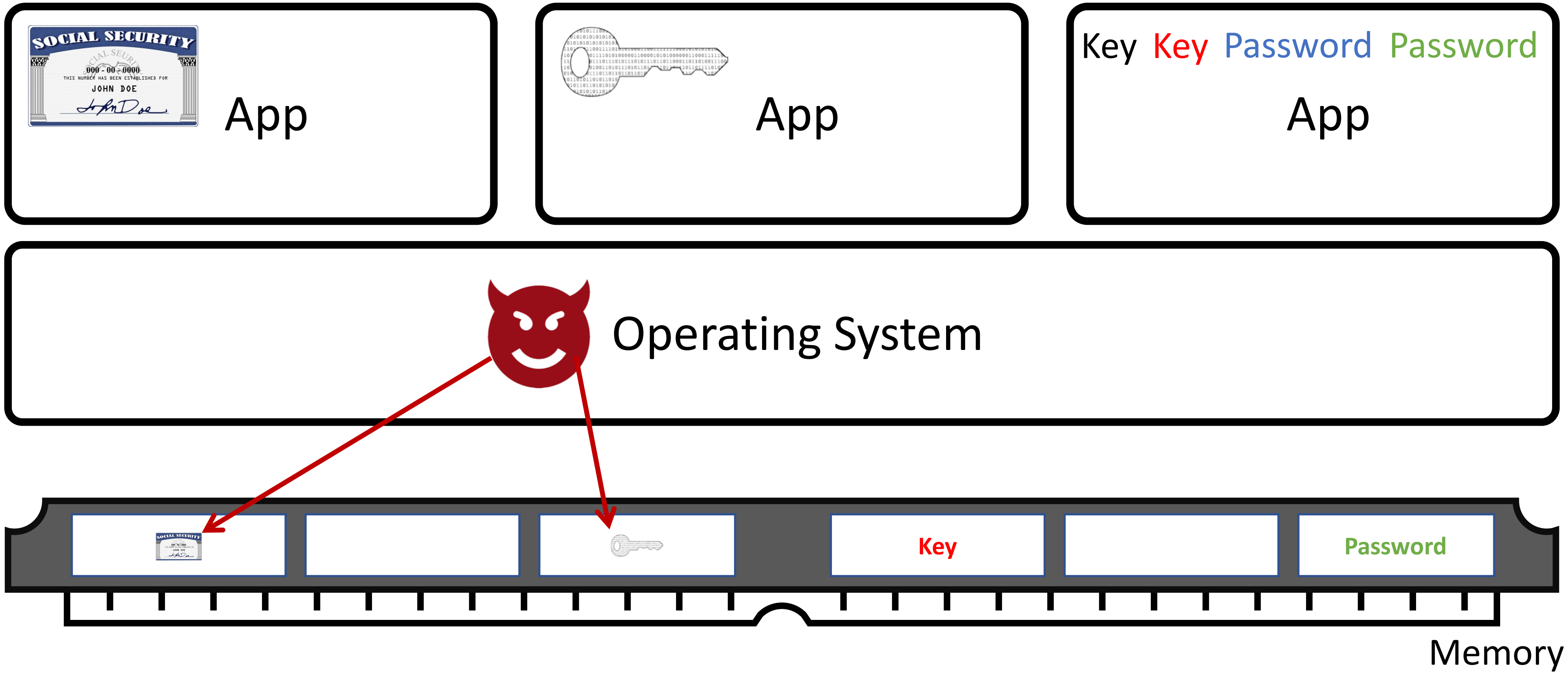
Rice University

Feb 25, 2019

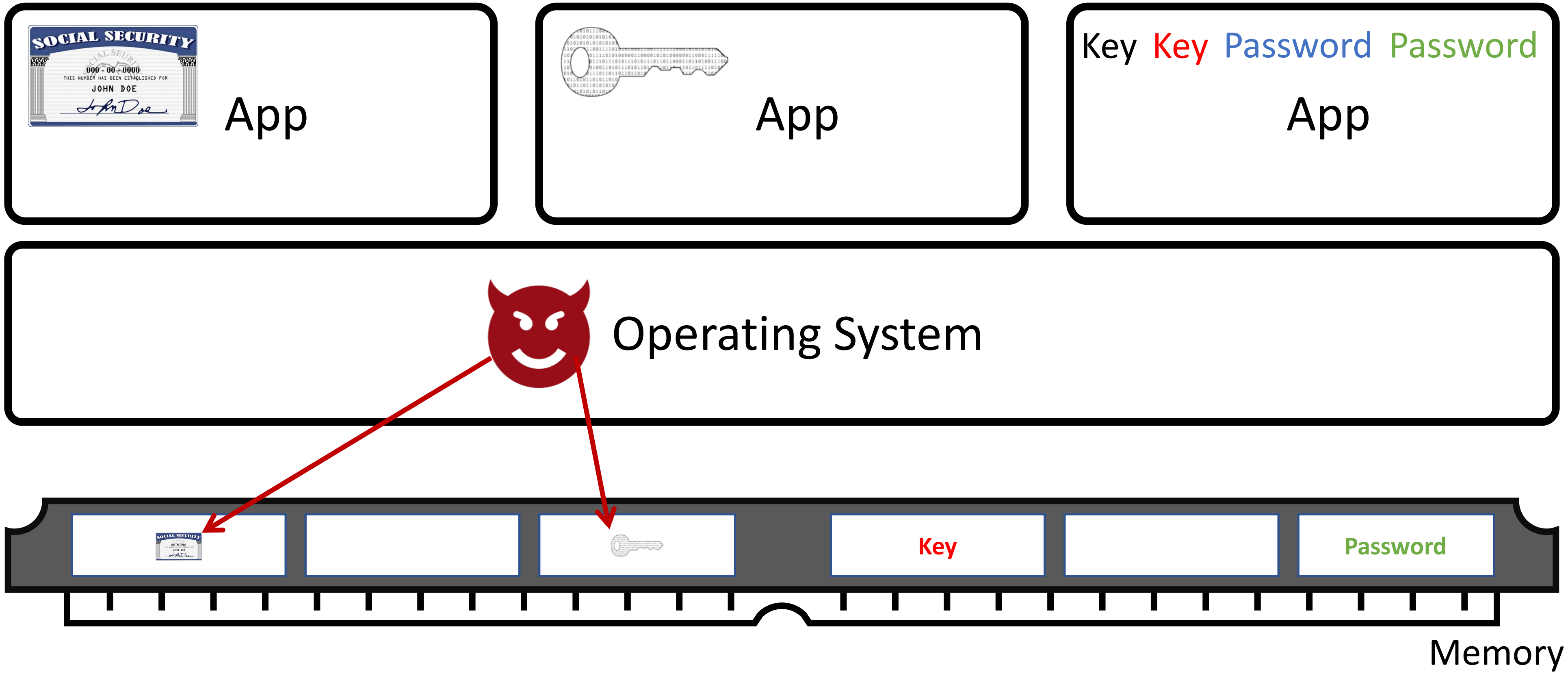
Sensitive data in memory



When the OS is compromised



Goal: Protect sensitive data against a compromised OS



Threat Model

- Trusted
  - Hardware
  - ARM TrustZone
  - A chain of trust

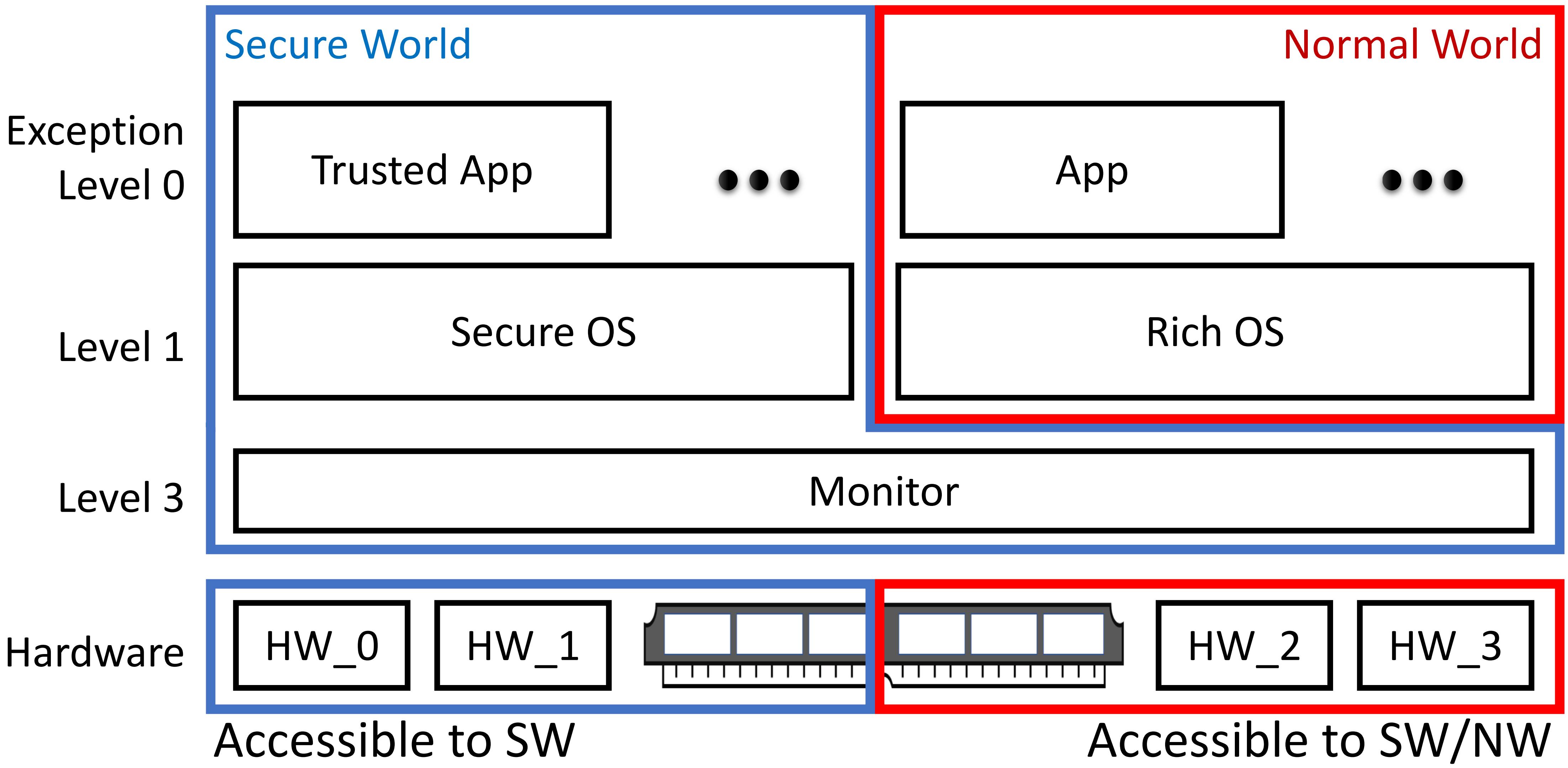
Threat Model

- Trusted
  - Hardware
  - ARM TrustZone
  - A chain of trust

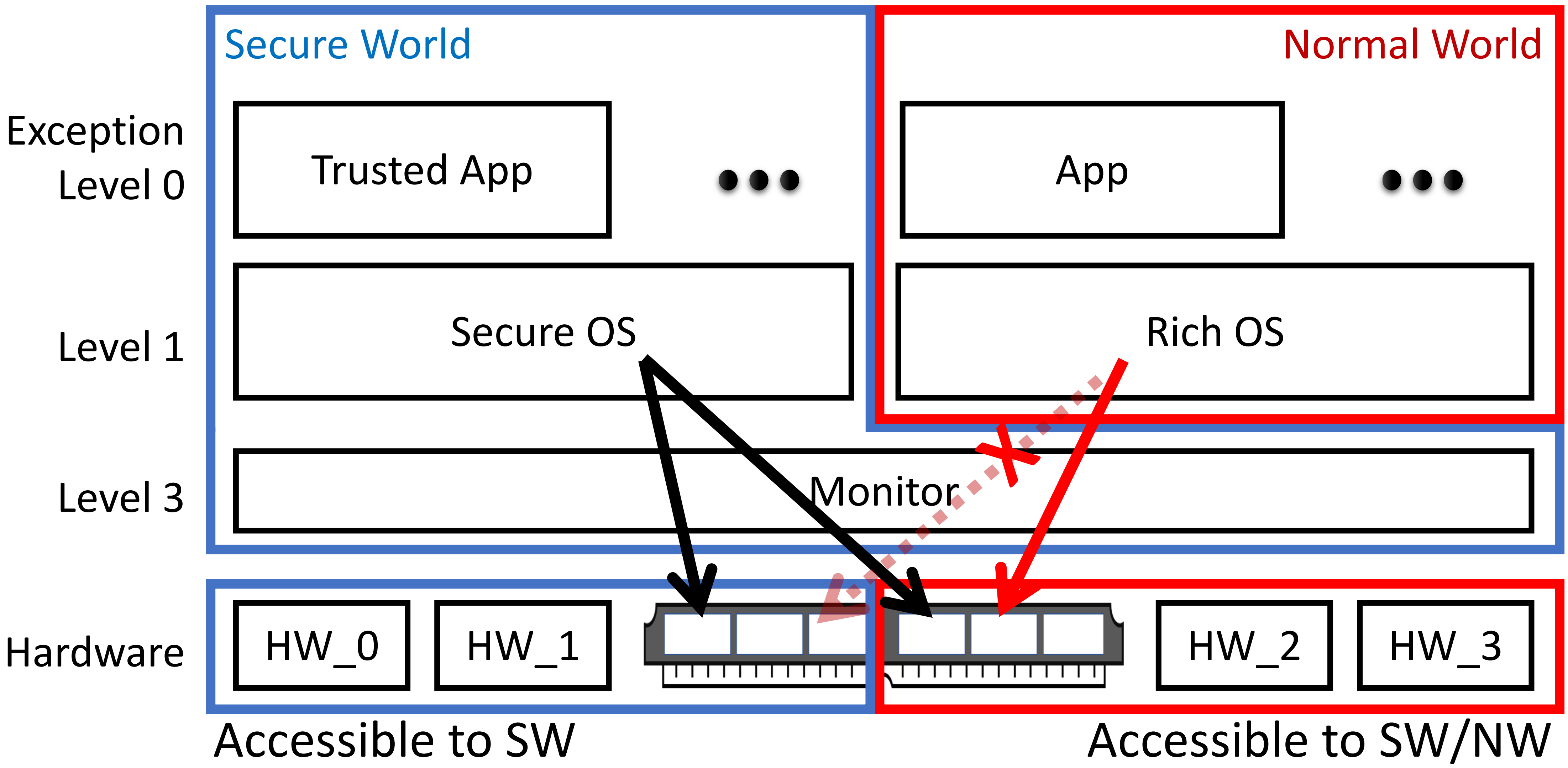
- Untrusted

Everything else
i.e., apps, system software, and OS

ARM TrustZone

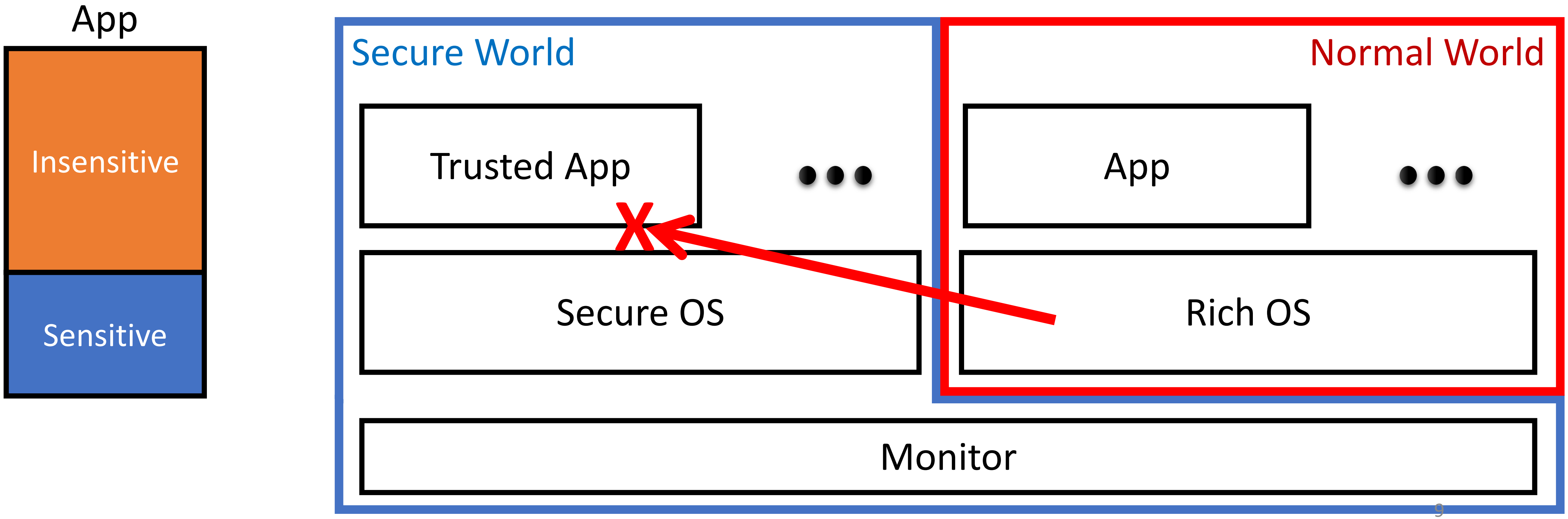


ARM TrustZone



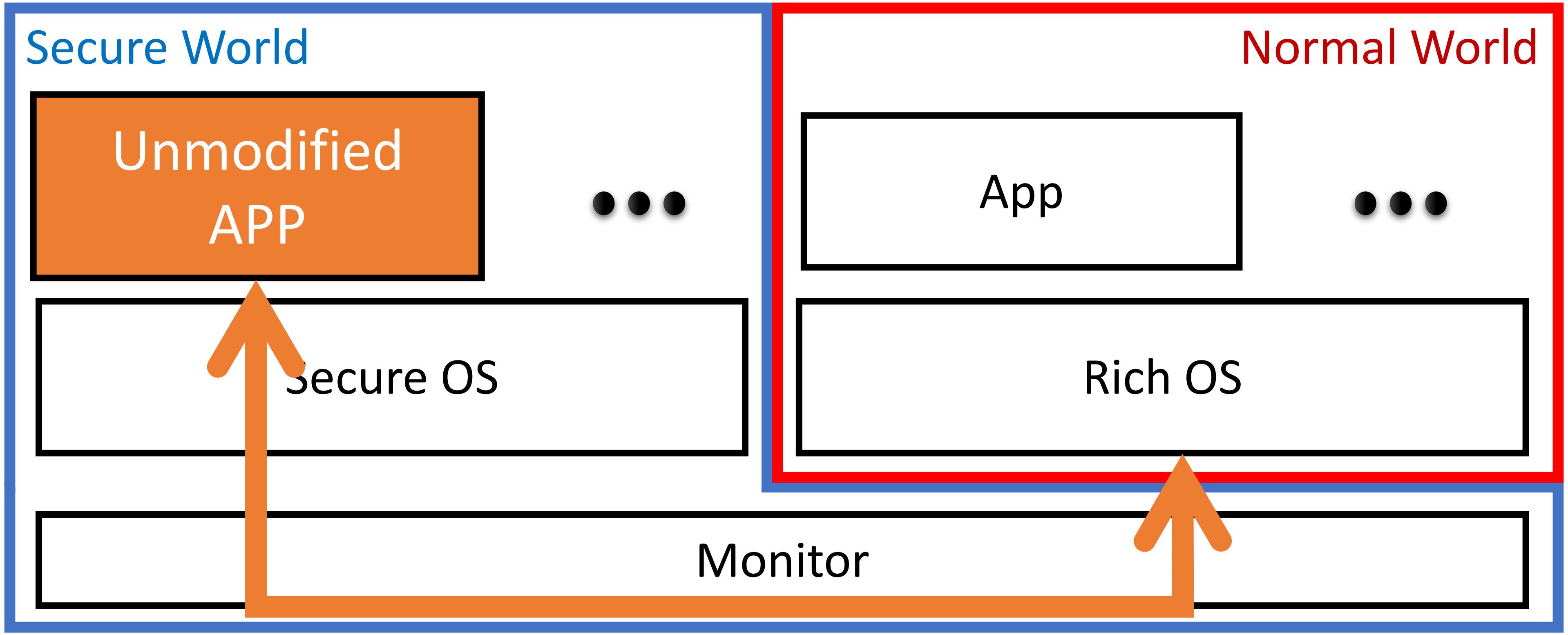
State of the Art #1: Divide an app into sensitive and insensitive parts

AdAttester [MobiSys '15], Liu et al. [MobiSys '12], TLR [ASPLOS '14], and so on



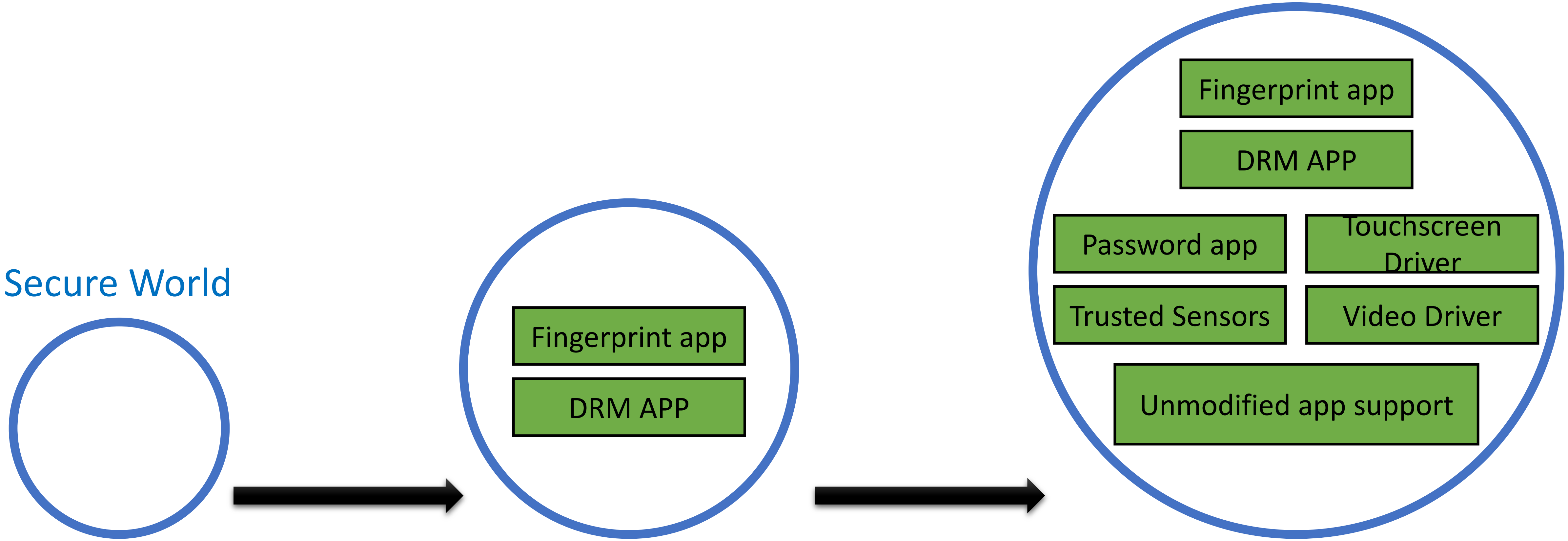
State of the Art #2:
Run an unmodified app in the secure world

TrustShadow [MobiSys`17]

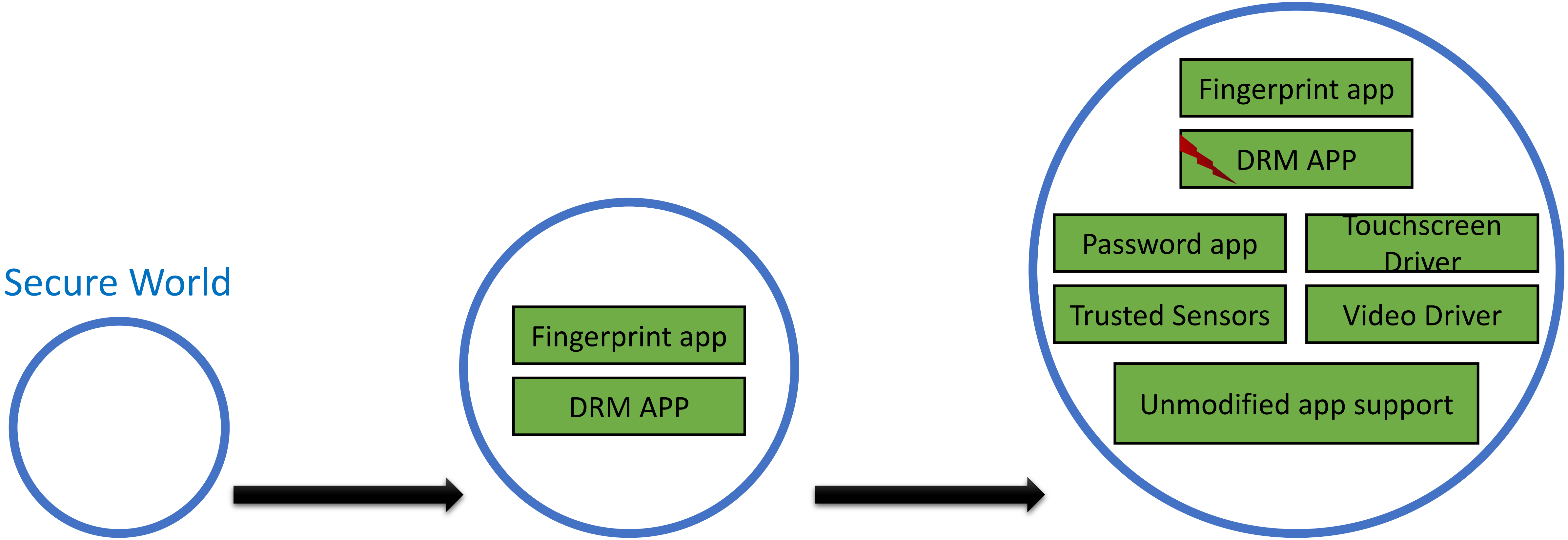


System calls, exceptions, library calls, and etc.

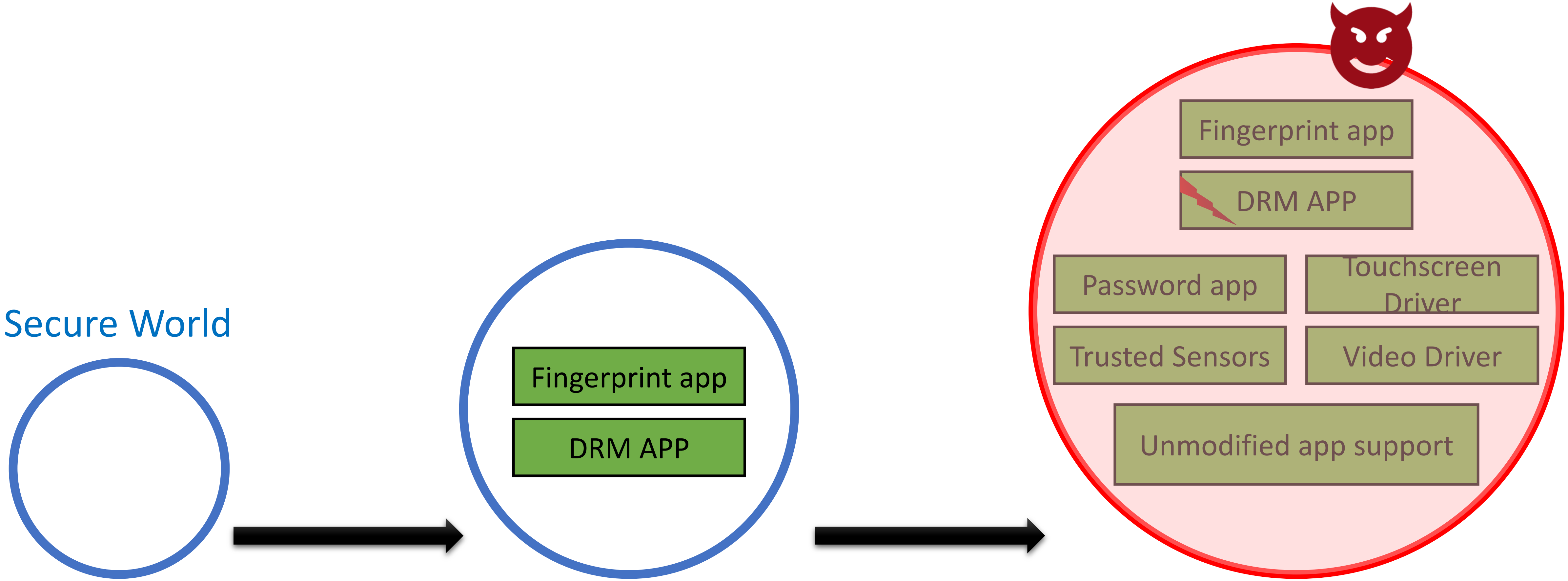
All proportionally increase the secure world



All proportionally increase the secure world



All proportionally increase the secure world



QSEE Integer Overflow [BlackHat14]
TEE API bug [BlackHat15]
TRUSTNONE [TR15]

Two Principles

1. No app logic in the Secure world
    - We should not include third-party apps in the secure world
    - It leads to vulnerabilities
      e.g., CVE-2015-6639, CVE-2015-8999, CVE-2015-9007, CVE-2016-1919,
      CVE-2016-1920, CVE-2016-2431, CVE-2016-3996, CVE-2016-5349, and so on

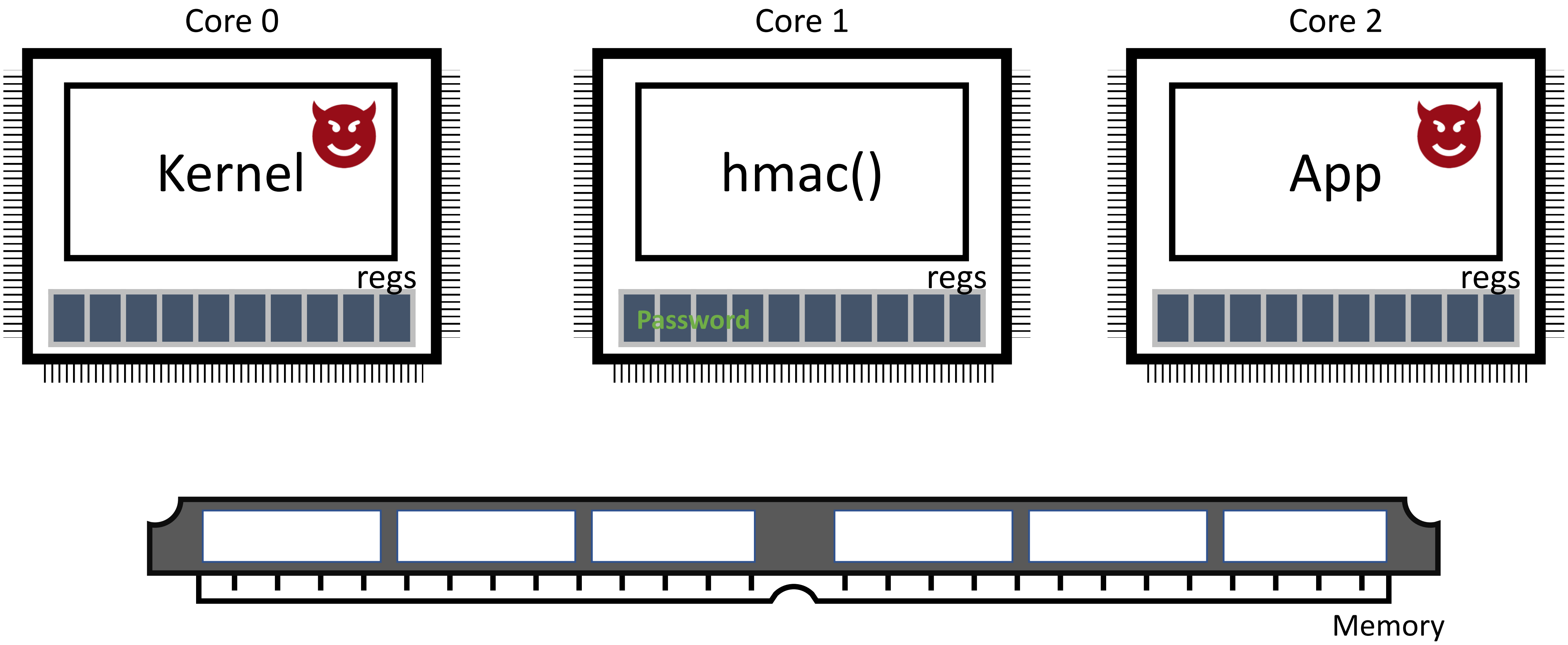
2. Protect only sensitive data
    - Protecting insensitive data only increases overhead
    - Not all data are important. E.g, time vs. password

In **Ginseng**,

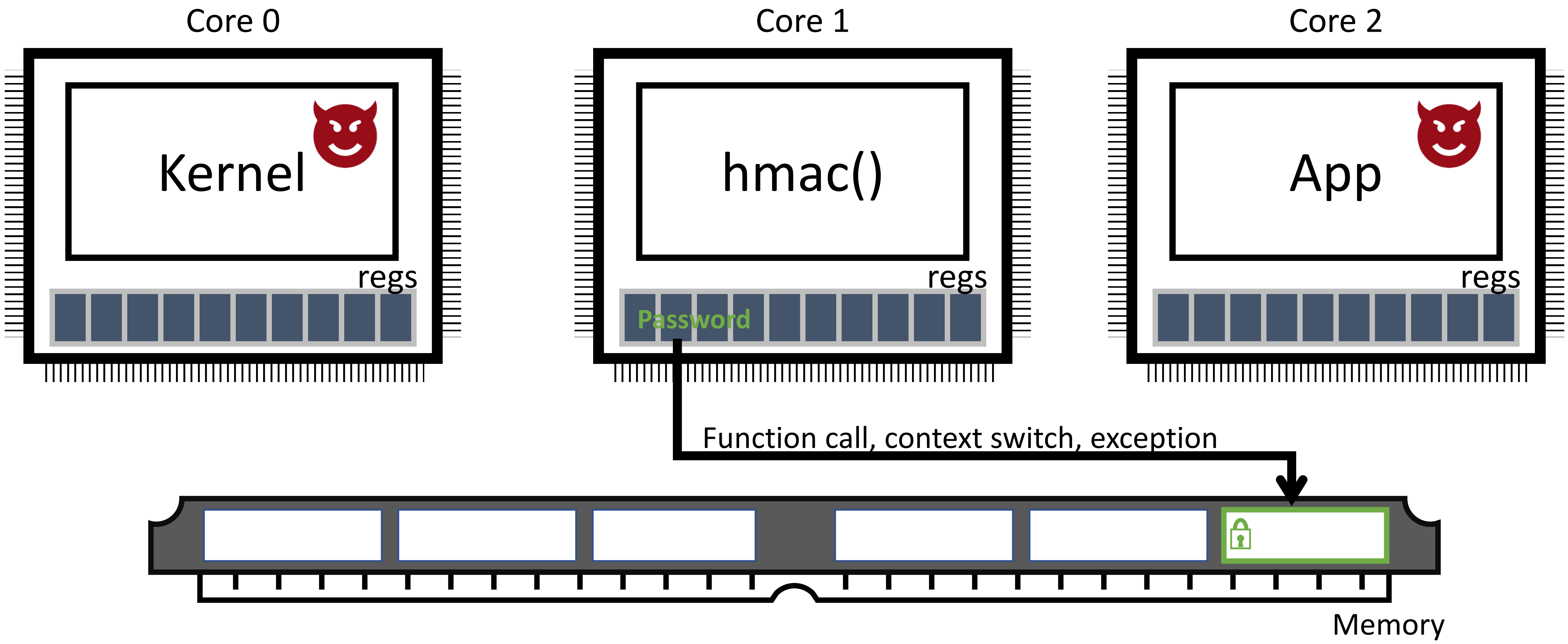
- Secure world : a trusted computing base for the normal world
- Normal world : the execution environment for apps

➡ Protect secrets of third-party apps in the Normal world

Idea: Keep sensitive data in registers only when being used



Idea: Encrypt sensitive data to memory when not being used



Challenges

1. Data must be saved in memory, or stack
    - on a subroutine call,
    - on an exception, e.g., page fault and interrupt

2. A function with sensitive data can be compromised
    - E.g., code injection by the kernel

3. A function with sensitive data can jump to a compromised function

Challenges

1. Data must be saved in memory, or stack → Confidentiality
   - on a subroutine call,
   - on an exception, e.g., page fault and interrupt

2. A function with sensitive data can be compromised → Code Integrity
   - E.g., code injection by the kernel

3. A function with sensitive data can jump to a compromised function → CFI

Challenges

1. Data must be saved in memory, or stack → Confidentiality
   - on a subroutine call,
   - on an exception, e.g., page fault and interrupt

today

2. A function with sensitive data can be compromised → Code Integrity
   - E.g., code injection by the kernel

3. A function with sensitive data can jump to a compromised function → CFI

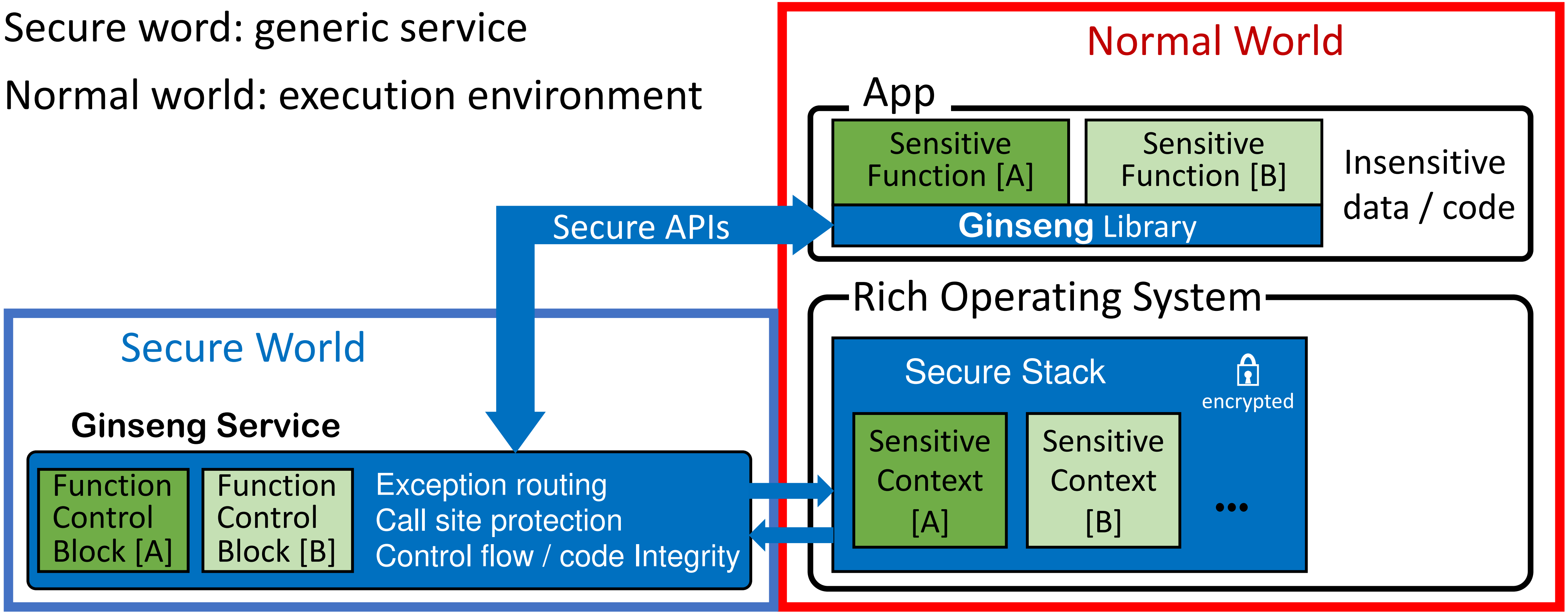
paper

Design

- Programing Model ➡ Developer-marked sensitive variables
- Static Protection ➡ Sensitive variables are always in registers
- Dynamic Protection ➡ Runtime support for functions with sensitive data
- Secure Stack ➡ Encrypted memory in the Normal world

Software Architecture

- Secure word: generic service
- Normal world: execution environment



Static Protection:
The compiler keeps sensitive data in registers

- Goals:
  - Allocate registers to sensitive variables and **never spill** them
  - Use as **less registers** as possible for sensitive variables
  - Protect registers with sensitive data at a **call site**

Static Protection:
The compiler keeps sensitive data in registers

- Goals:
  - Allocate registers to sensitive variables and **never spill** them ← Register allocator
  - Use as **less registers** as possible for sensitive variables
  - Protect registers with sensitive data at a **call site** ← Secure stack

Where to save sensitive data?

x14 x15

```
sensitive long key_top, key_bottom;
// all other variables are insensitive

/* computing with key_top and key_bottom */


  printf("Generating code...\n");

  /* use HAMC_SHA1 to compute 20-byte hash */
  hmac_sha1(key_top, key_bottom, // sensitive data
            challenge,           // current time / 30sec
            resultFull);         // (out) full hash
```

Problem Sensitive registers must not be saved to stack

Sensitive registers ⟶ *secure stack*

```
                      x14        x15
sensitive long key_top, key_bottom;
// all other variables are insensitive

/* computing with key_top and key_bottom */


  printf("Generating code...\n");

  /* use HAMC_SHA1 to compute 20-byte hash */
  hmac_sha1(key_top, key_bottom, // sensitive data
            challenge,           // current time / 30sec
            resultFull);         // (out) full hash
```

Callsite Protection req.

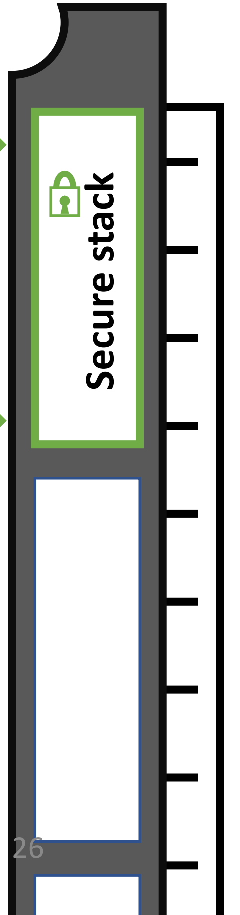
Call site ID

Hide x14 and x15

Callsite Protection req.

Call site ID

Restore x14 and x15



Sensitive data are protected at a call site

```
                  x14         x15
sensitive long key_top, key_bottom;
// all other variables are insensitive

/* computing with key_top and key_bottom */


  printf("Generating code...\n");

  /* use HAMC_SHA1 to compute 20-byte hash */
  hmac_sha1(key_top, key_bottom, // sensitive data
            challenge,           // current time / 30sec
            resultFull);         // (out) full hash
```

Sensitive data -> secure stack

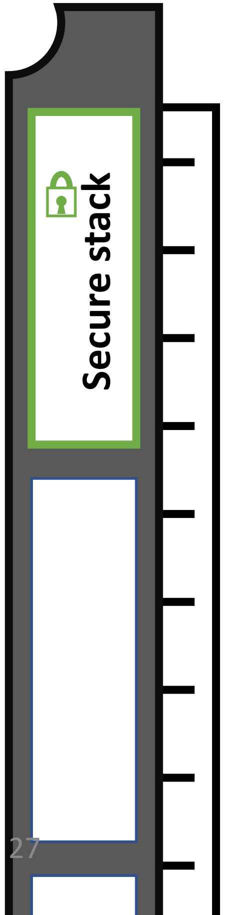
↓

`printf()` implementation

```
/* No sensitive data
 in regs or memory*/
```

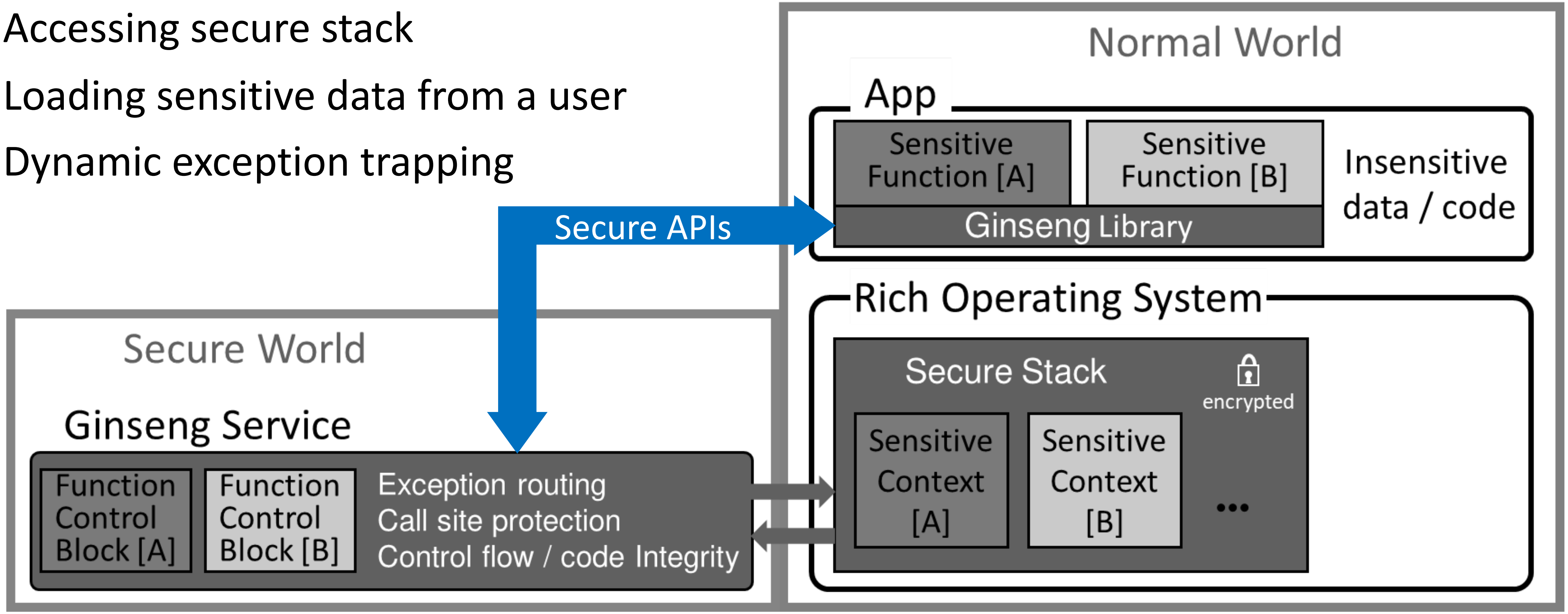
↓

Sensitive data <- secure stack



Secure API bypasses the OS

- Accessing secure stack
- Loading sensitive data from a user
- Dynamic exception trapping



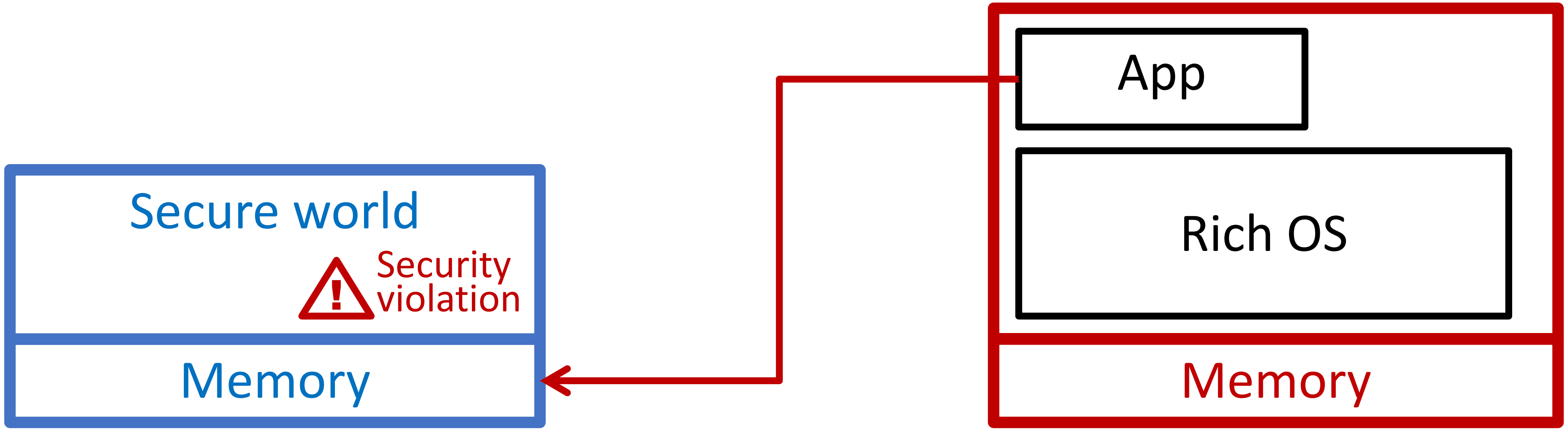
Secure Monitor Call is not enough.

Secure Monitor Call (SMC) instruction

- invoked the Secure world
- only available to the kernel

Problem	We must not send cleartext data to secure stack via the untrusted OS

Idea for Secure API:
Trigger a security violation from the Normal world



The rich OS is unaware of the communication

Static Protection is not enough

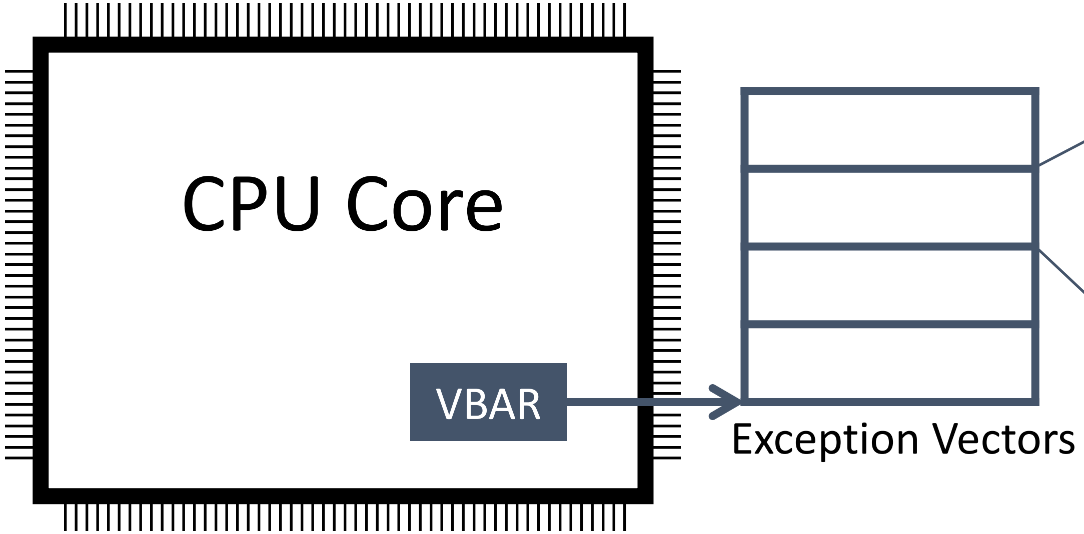
Challenge #1: Data must be saved in memory, or stack

- ~~on a subroutine call,~~
- on an exception, e.g., page fault and interrupt

Dynamic Exception Trapping

Any sensitive data	→	**GService** intercepts exceptions
No sensitive data	→	No intercept

Dynamic Exception Trapping



VBAR: Vector Base Address Register

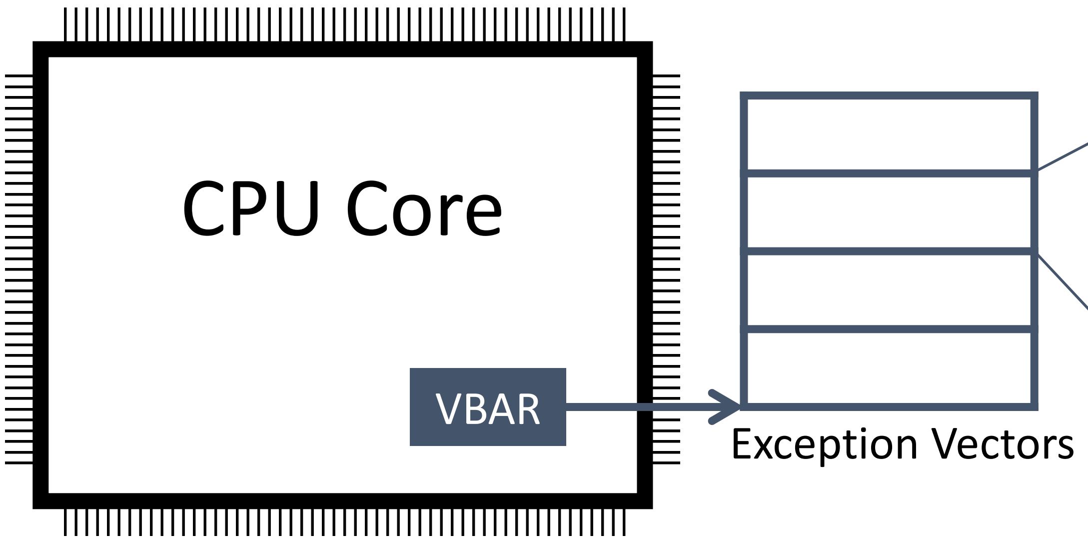
entry.S

```
/*
 * Common to all vectors
 * Save registers to the stack
 */
sub       sp, sp, #S_FRAME_SIZE
stp       x0, x1, [sp, #16 * 0]
stp       x2, x3, [sp, #16 * 1]
stp       x4, x5, [sp, #16 * 2]
stp       x6, x7, [sp, #16 * 3]
          ...


/* Exception
   Service
   Routine */
```

Preprocessed and simplified

Dynamic Exception Trapping



VBAR: *Vector Base Address Register*

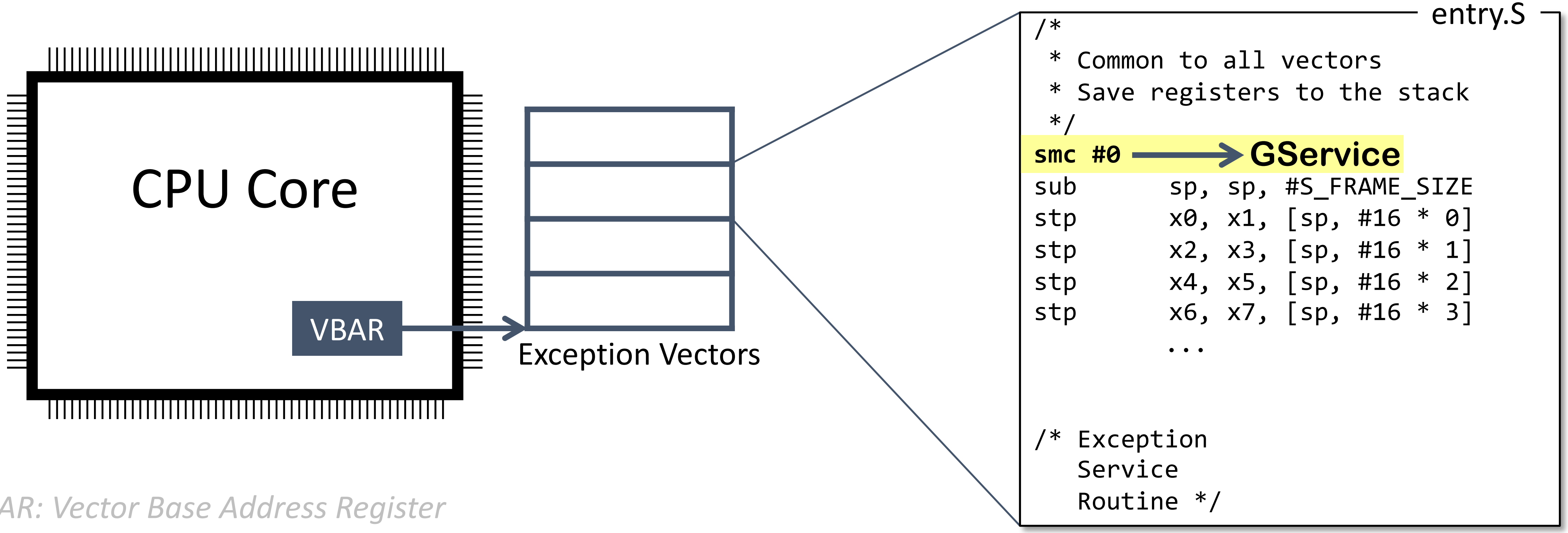
entry.S

```
/*
 * Common to all vectors
 * Save registers to the stack
 */
nop         // place holder
sub         sp, sp, #S_FRAME_SIZE
stp         x0, x1, [sp, #16 * 0]
stp         x2, x3, [sp, #16 * 1]
stp         x4, x5, [sp, #16 * 2]
stp         x6, x7, [sp, #16 * 3]
            ...


/* Exception
   Service
   Routine */
```

Preprocessed and simplified

Before loading any sensitive data, **GService** inserts **smc**



VBAR: Vector Base Address Register

Preprocessed and simplified

The OS handles an exception, but **GService** encrypts data.

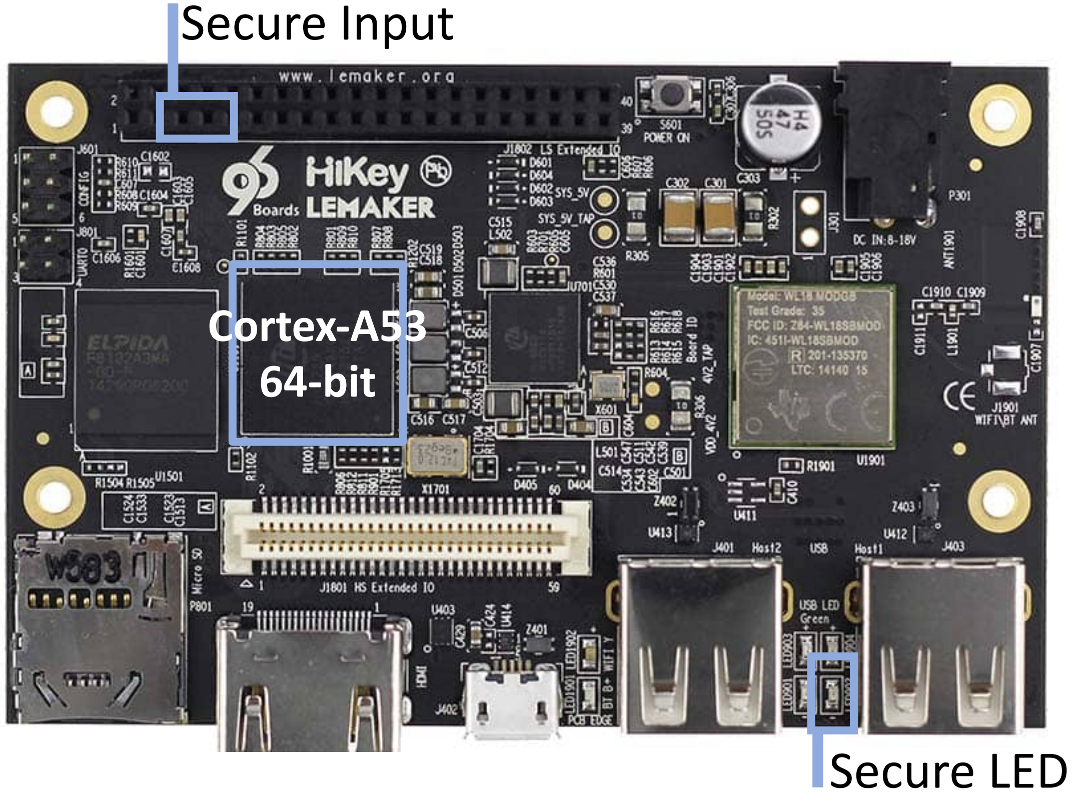
Design Summary

• Programing Model	➡	the *sensitive* keyword
• Static Protection	➡	**Ginseng** compiler
• Dynamic Protection	➡	Code and control flow Integrity
• Secure Stack	➡	Encrypted Normal world memory

Implementation

- LLVM v6.0
- **Ginseng Service** in Rust
- Linux v4.9

- Benchmark
  - Two-factor authenticator
  - wpa_supplicant
  - Learned classifier (C4.5)
  - Nginx



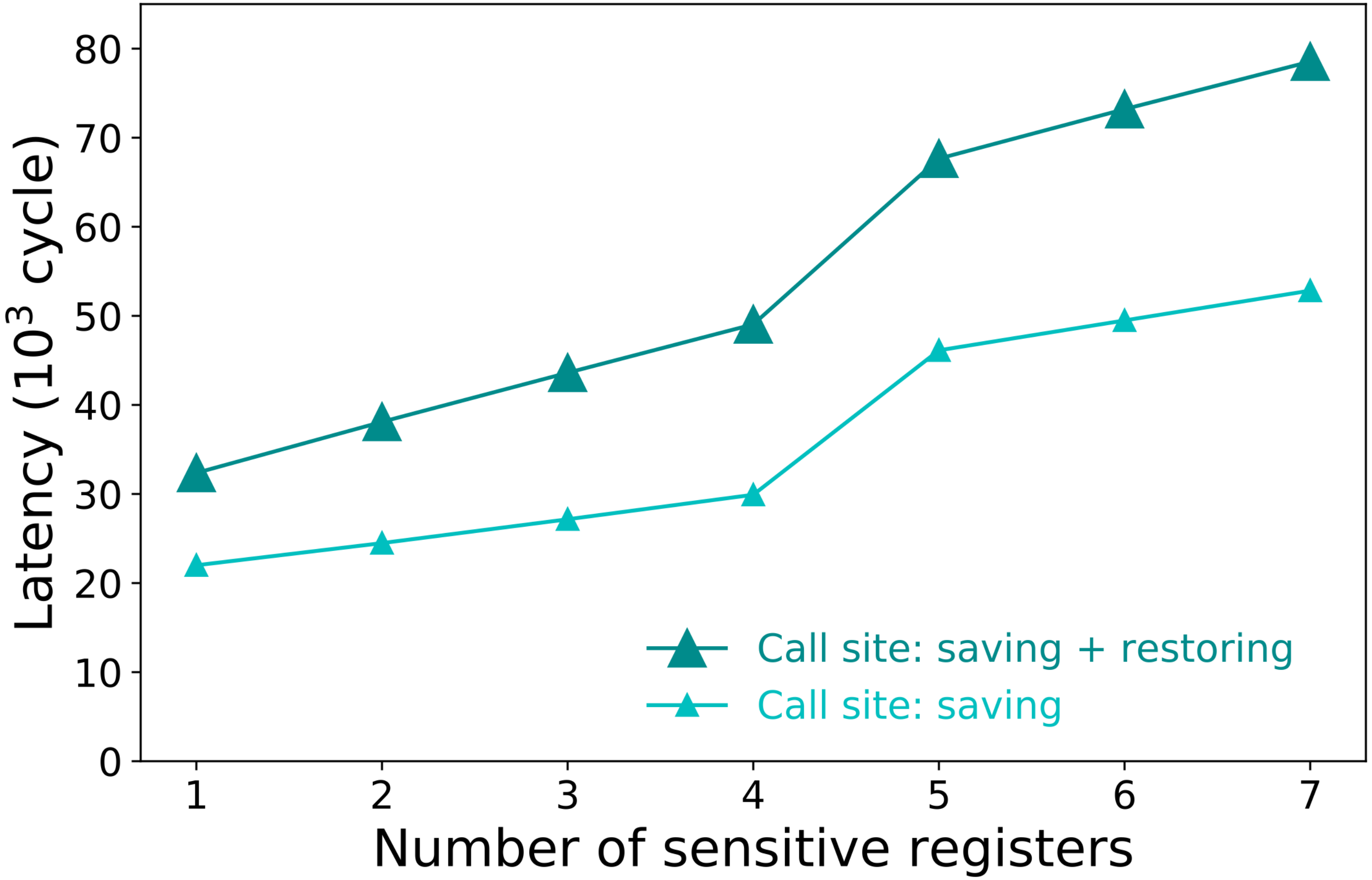
Evaluation

Q1. Microbenchmark for the protections

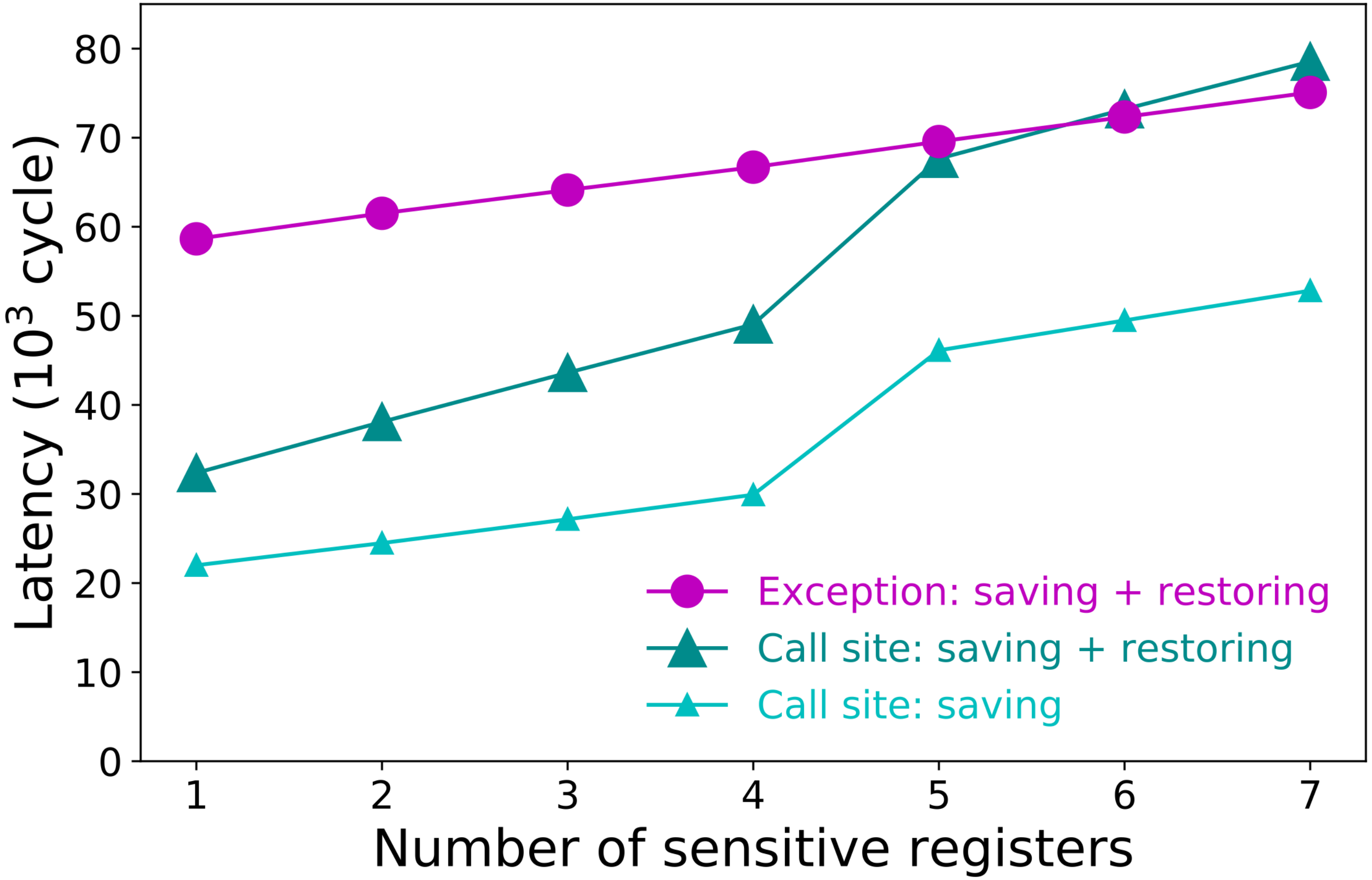
Q2. End-to-end overhead in real applications

Q3. Difficulty of applying **Ginseng**

Microbenchmark: Overhead for accessing secure stack



Microbenchmark: Overhead for accessing secure stack



End-to-end Overhead

(cycle)		Authenticator	wpa_supplicant		Classifier
Baseline		37 K	219 M		1.7 M
Overhead	Code Integrity	45,356 K	45 M	23 M	11.3 M
	Callsite	680 K (17 times)	6,429 M (131,078	1,640 M 40,988 times)	4.4 M (137 times)
	Exception	9 K (0.13 times)	6 M (99.40	6 M 78.52 times)	0.4 M (5.4 times)
	GService overhead	851 K	661 M	411 M	1.7 M
Total		46,933 K	7,361 M	2,299 M	19.6 M
			naïve	*optimized*	

Nginx: no meaningful overhead

Development Effort

In SLoC	**Authenticator**	**wpa_supplicant**	**Classifier**	**Nginx**
Baseline	250	400 + 513 K	5 K	145 + 513 K
Modified (added)	10	25 + 90†	6	0 + 200†
Time	0	1 d	3 h	1 d

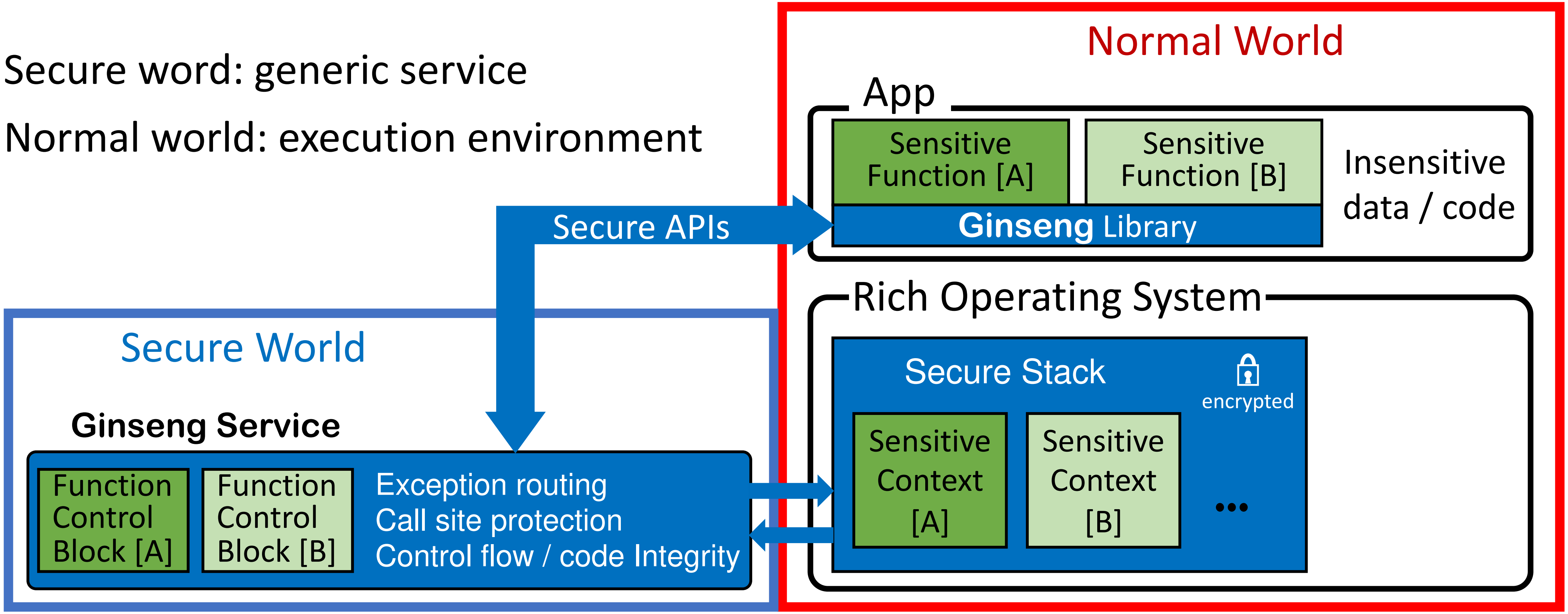
†OpenSSL

Mainly due to the prototype's limitation: supporting only primitive types

➔ can be reduced only by *engineering* effort

Ginseng protects sensitive data with no app logic in the Secure world

- Secure word: generic service
- Normal world: execution environment



backup

Programming Model: A developer marks a sensitive variables

- Not all data are sensitive
- Not all function are protected

```
void hmac_sha1(sensitive long key_top,
               sensitive long key_bottom,
               const uint8_t *data, uint8_t *result) {
  sensitive long tmp_key_top, tmp_key_bottom;
  /* all other variables are insensitive */

  /* HMAC_SHA1 implementation */
}

int genCode (sensitive long key_top,
             sensitive long key_bottom) {
  /* all other variables are insensitive */

  /* use HAMC_SHA1 to compute 20-byte hash */
  hmac_sha1(key_top, key_bottom, // sensitive data
            challenge,           // current time / 30sec
            resultFull);         // (out) full hash

  /* truncate 20-byte hash to 4-byte */
  result = truncate(resultFull);

  printf("OTP: %06d\n", result);
  return result;
}
```

```
void run () {
  sensitive long key_top, key_bottom;

  /* read a secret key from GService or a user */
  s_read(TKN_KEY1_TOP, TKN_KEY1_BOTTOM, key_top);
  s_read(TKN_KEY2_TOP, TKN_KEY2_BOTTOM, key_bottom);

  genCode(key_top, key_bottom);
}
```

A simplified two-factor authenticator

Programming Model: A developer marks a sensitive variables

- Not all data are sensitive
- Not all function are protected

```
void run () {
  sensitive long key_top, key_bottom;

  /* read a secret key from GService or a user */
  s_read(TKN_KEY1_TOP, TKN_KEY1_BOTTOM, key_top);
  s_read(TKN_KEY2_TOP, TKN_KEY2_BOTTOM, key_bottom);

  genCode(key_top, key_bottom);
}
```

A simplified two-factor authenticator

```
void hmac_sha1(sensitive long key_top,
               sensitive long key_bottom,
               const uint8_t *data, uint8_t *result) {
  sensitive long tmp_key_top, tmp_key_bottom;
  /* all other variables are insensitive */

  /* HMAC_SHA1 implementation */
}
```

```
int genCode (sensitive long key_top,
             sensitive long key_bottom) {
  /* all other variables are insensitive */

  /* use HAMC_SHA1 to compute 20-byte hash */
  hmac_sha1(key_top, key_bottom, // sensitive data
            challenge,           // current time / 30sec
            resultFull);         // (out) full hash

  /* truncate 20-byte hash to 4-byte */
  result = truncate(resultFull);

  printf("OTP: %06d\n", result);
  return result;
}
```

Programming Model: A developer marks a sensitive variables

- Not all data are sensitive
- Not all function are protected

```
void hmac_sha1(sensitive long key_top,
               sensitive long key_bottom,
               const uint8_t *data, uint8_t *result) {
  sensitive long tmp_key_top, tmp_key_bottom;
  /* all other variables are insensitive */

  /* HMAC_SHA1 implementation */
}
```

```
int genCode (sensitive long key_top,
             sensitive long key_bottom) {
  /* all other variables are insensitive */

  /* use HAMC_SHA1 to compute 20-byte hash */
  hmac_sha1(key_top, key_bottom, // sensitive data
            challenge,           // current time / 30sec
            resultFull);         // (out) full hash

  /* truncate 20-byte hash to 4-byte */
  result = truncate(resultFull);
    insensitive
  printf("OTP: %06d\n", result);
  return result;
}
```

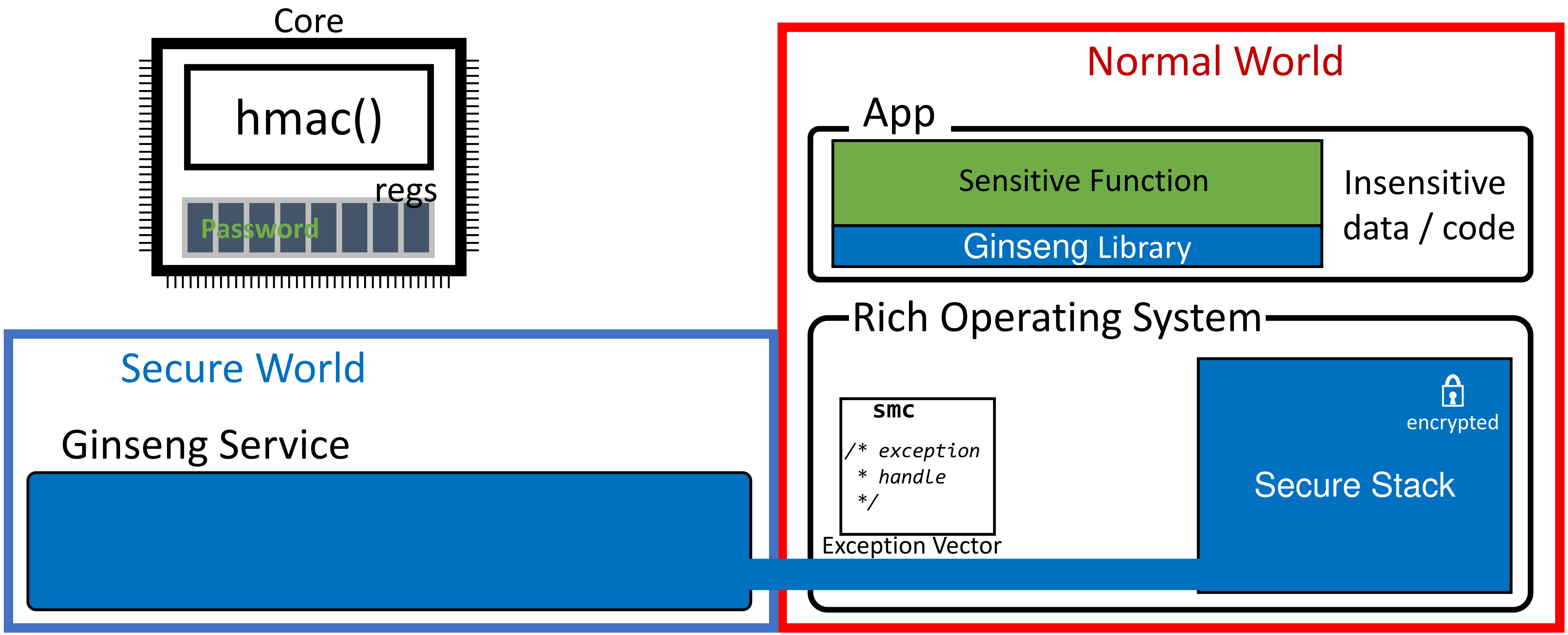
```
void run () {
  sensitive long key_top, key_bottom;

  /* read a secret key from GService or a user */
  s_read(TKN_KEY1_TOP, TKN_KEY1_BOTTOM, key_top);
  s_read(TKN_KEY2_TOP, TKN_KEY2_BOTTOM, key_bottom);

  genCode(key_top, key_bottom);
}
```

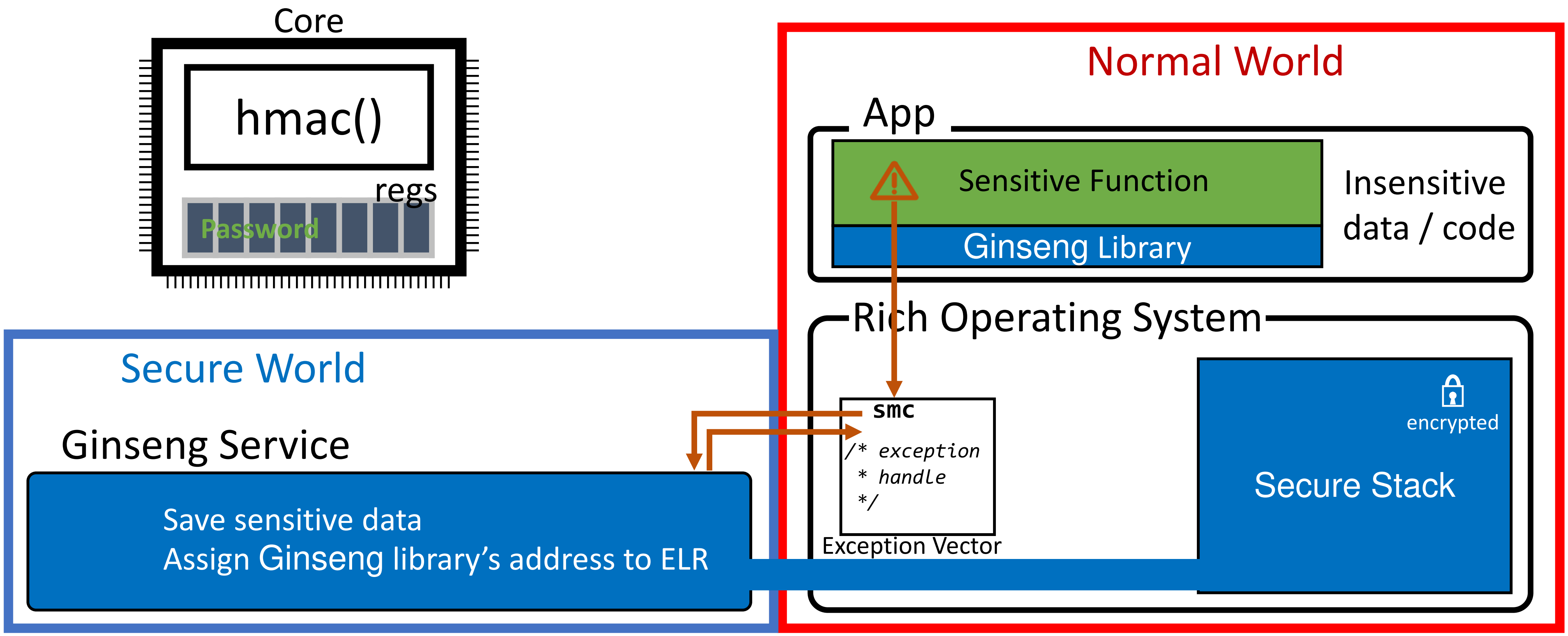
A simplified two-factor authenticator

An exception is handled by the kernel after sensitive registers are saved to secure stack



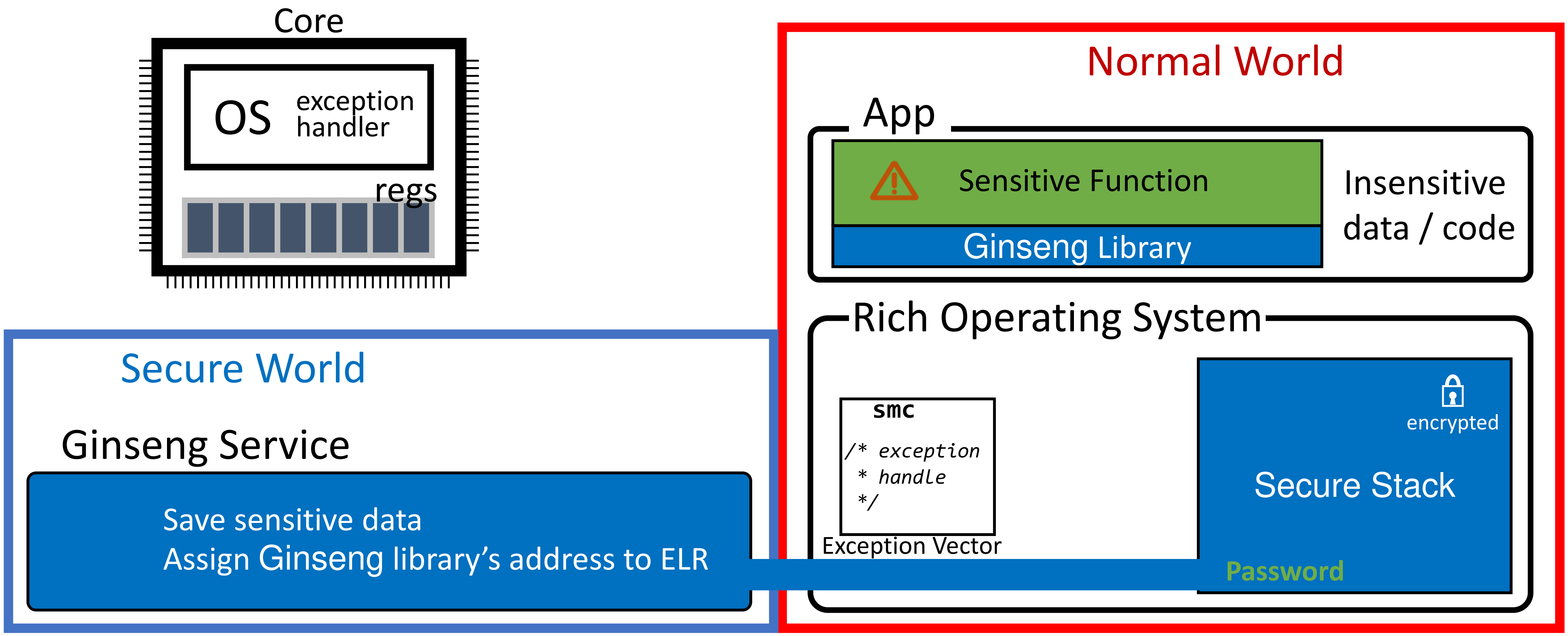
ELR: Exception Link Register

An exception is handled by the kernel after sensitive registers are saved to secure stack



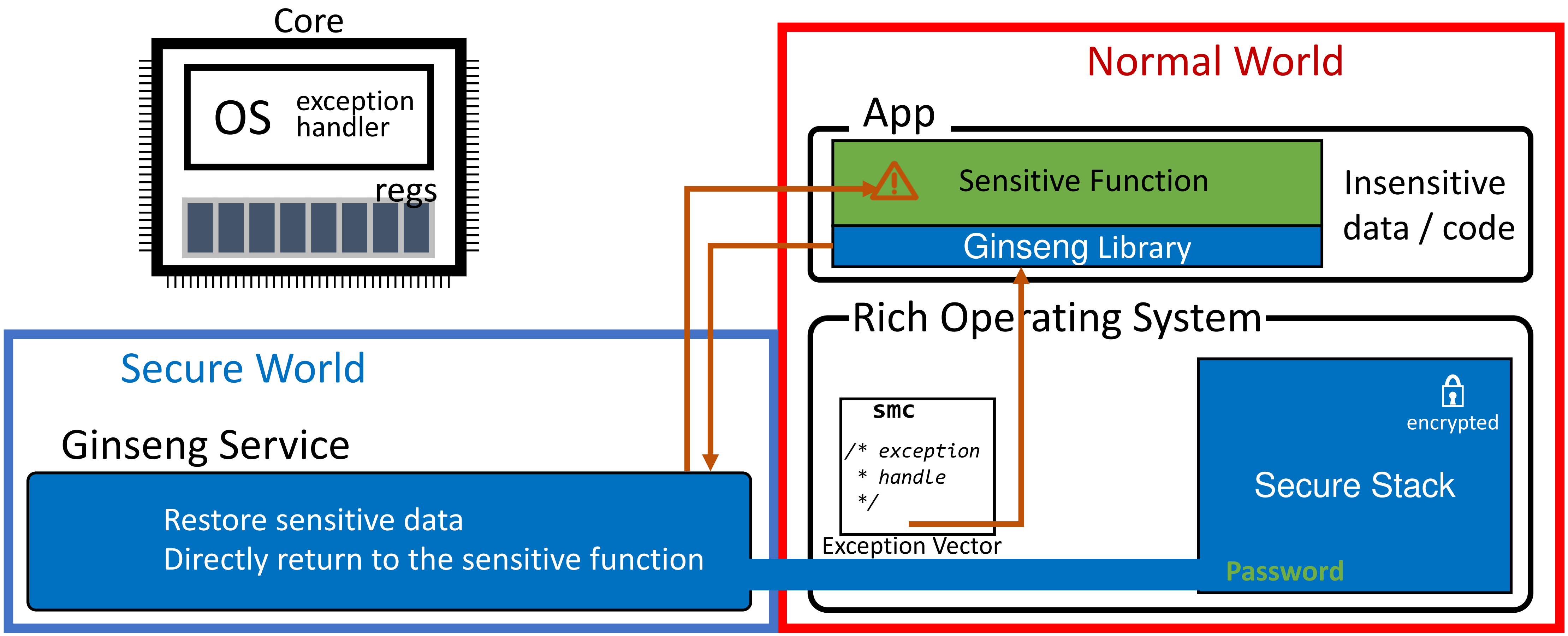
ELR: Exception Link Register

An exception is handled by the kernel after sensitive registers are saved to secure stack

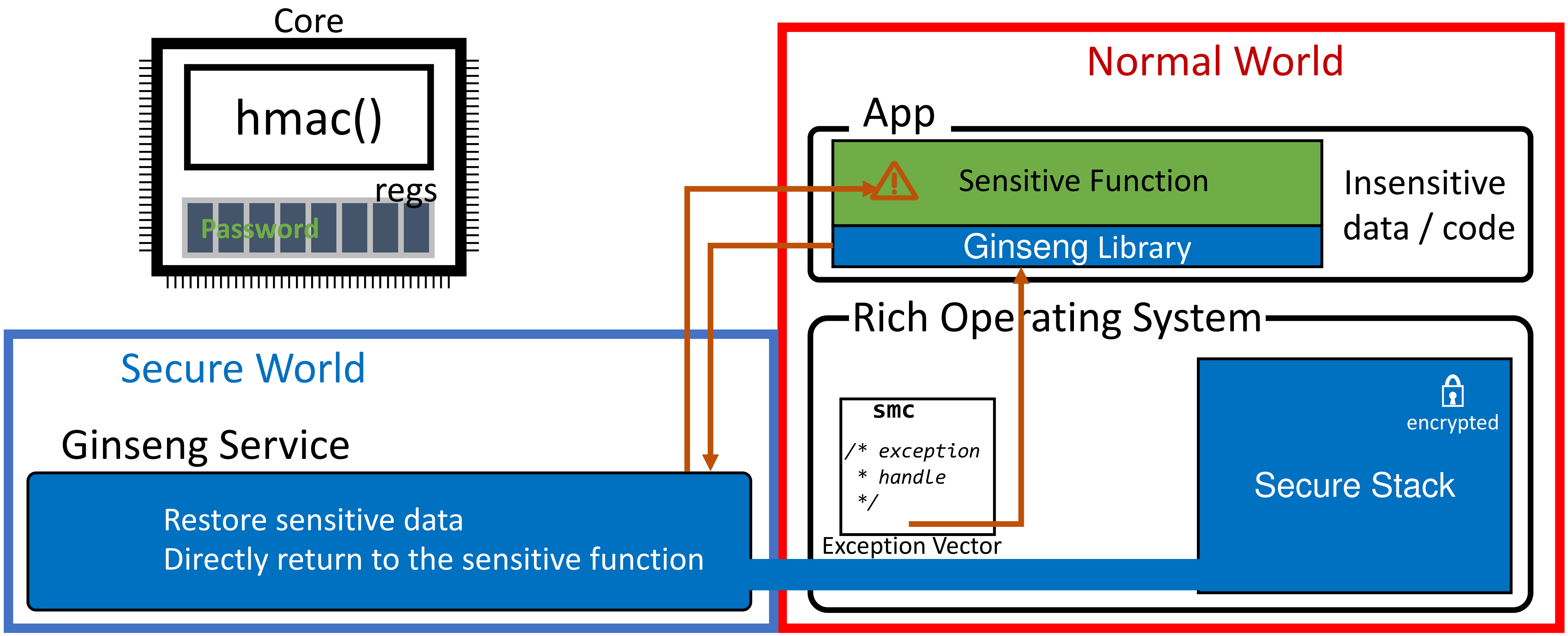


ELR: Exception Link Register

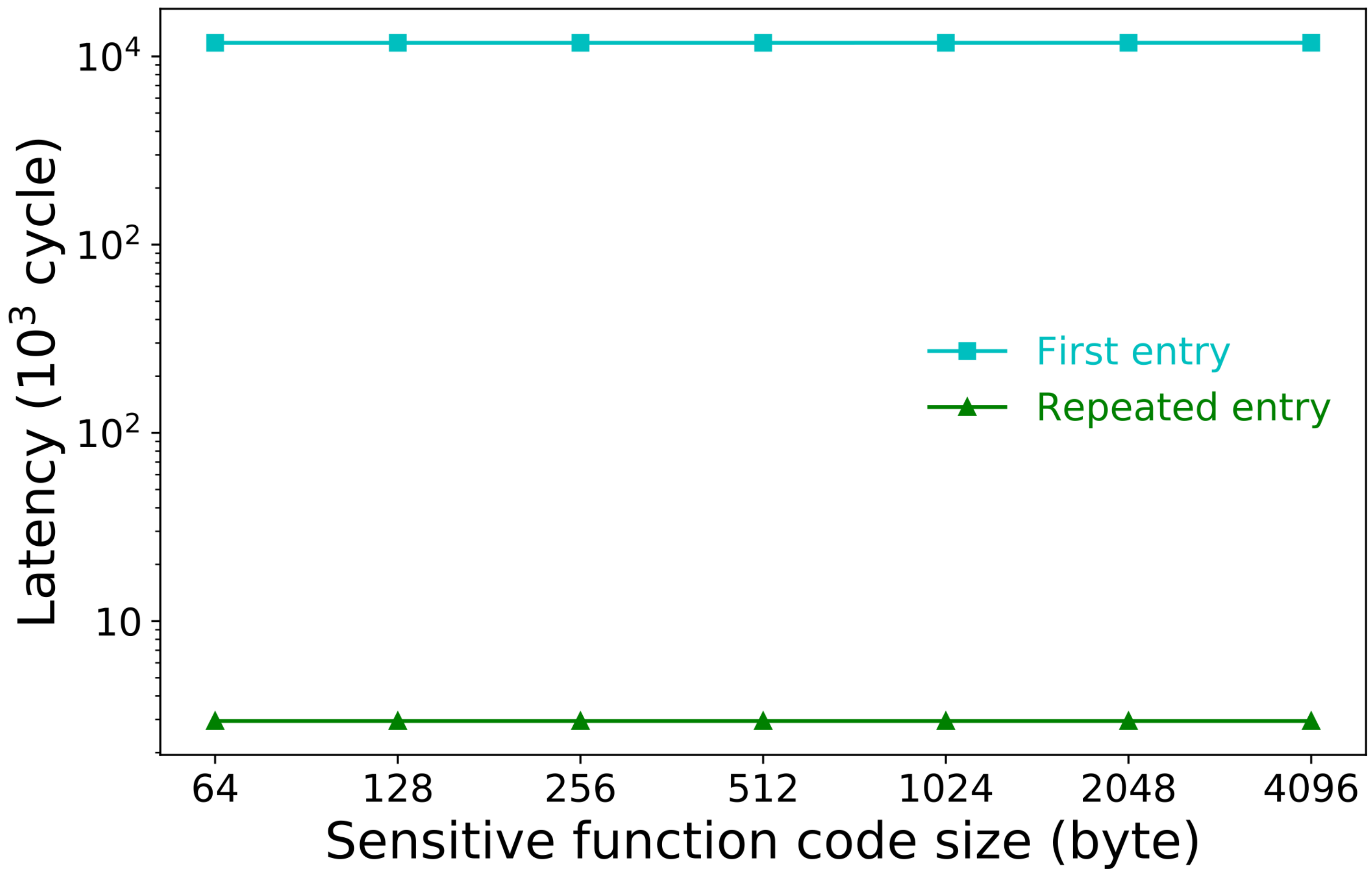
GService directly returns to the function



GService directly returns to the function



Microbenchmark: Overhead for code integrity



Kernels pagetable walk:
11 M cycles (≤ 10ms)

Onetime overhead per function

Re-enter a sensitive function:
3 K cycles