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

Operating System

Memory

Key

Password

App

Key

Password

App

Key

Password

App
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

- **Exception Level 0**: Trusted App
- **Level 1**: Secure OS
- **Level 3**: Monitor
- **Hardware**: HW_0, HW_1, HW_2, HW_3
  - Accessible to SW/NW

- **Secure World**
- **Normal World**
  - App
  - Rich OS
ARM TrustZone

Secure World
- Trusted App
- Secure OS
- Monitor

Normal World
- App
- Rich OS

Hardware
- HW_0
- HW_1
- HW_2
- HW_3

Accessible to SW
Accessible to SW/NW
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]
All proportionally increase the secure world
All proportionally increase the secure world
All proportionally increase the secure world

Secure World

Fingerprint app

DRM APP

Fingerprint app

DRM APP

Password app

Trusted Sensors

Touchscreen Driver

Video Driver

Unmodified app support

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

Kernel
Core 0

hmac()
Core 1

Password
Core 2

App

Function call, context switch, exception

Memory
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
   • 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
   • 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

Confidentiality

Code Integrity

CFI
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
  • Use as less registers as possible for sensitive variables
  • Protect registers with sensitive data at a call site

Register allocator
Secure stack
Where to save sensitive data?

```c
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 / 30 sec
    resultFull); // (out) full hash
```

Problem: Sensitive registers must not be saved to stack.
Sensitive registers → secure stack

```c
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 / 30 sec
    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

```c
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 / 30 sec
    resultFull); // (out) full hash
```

printf() implementation

/* No sensitive data in regs or memory*/

Sensitive data -> secure stack

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 $\rightarrow$ GService intercepts exceptions

No sensitive data $\rightarrow$ No intercept
Dynamic Exception Trapping

/* 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]

...
Dynamic Exception Trapping

**CPU Core**

**VBAR: Vector Base Address Register**

**Exception Vectors**

```Assembly
/*
* 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**

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

![Graph showing latency (\(10^3\) cycle) vs. number of sensitive registers.](image)

- **Exception**: saving + restoring
- **Call site**: saving + restoring
- **Call site**: saving
## End-to-end Overhead

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Authenticator</th>
<th>wpa_supplicant</th>
<th>Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>37 K</td>
<td>219 M</td>
<td>1.7 M</td>
</tr>
<tr>
<td>Code Integrity</td>
<td>45,356 K</td>
<td>45 M</td>
<td>23 M</td>
</tr>
<tr>
<td>Callsite</td>
<td>680 K (17 times)</td>
<td>6,429 M (131,078)</td>
<td>1,640 M (40,988 times)</td>
</tr>
<tr>
<td>Exception</td>
<td>9 K (0.13 times)</td>
<td>6 M (99.40)</td>
<td>6 M (78.52 times)</td>
</tr>
<tr>
<td>GService overhead</td>
<td>851 K</td>
<td>661 M</td>
<td>411 M</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>46,933 K</td>
<td>7,361 M</td>
<td>2,299 M</td>
</tr>
</tbody>
</table>

Nginx: **no meaningful overhead**
## Development Effort

<table>
<thead>
<tr>
<th>In SLoC</th>
<th>Authenticator</th>
<th>wpa_supplicant</th>
<th>Classifier</th>
<th>Nginx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>250</td>
<td>400 + 513 K</td>
<td>5 K</td>
<td>145 + 513 K</td>
</tr>
<tr>
<td>Modified (added)</td>
<td>10</td>
<td>25 + 90&lt;sup&gt;Ⅰ&lt;/sup&gt;</td>
<td>6</td>
<td>0 + 200&lt;sup&gt;Ⅰ&lt;/sup&gt;</td>
</tr>
<tr>
<td>Time</td>
<td>0</td>
<td>1 d</td>
<td>3 h</td>
<td>1 d</td>
</tr>
</tbody>
</table>

<sup>Ⅰ</sup>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

```c
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);
}
```

```c
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 */
}
```

```c
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 / 30 sec
              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

```c
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);
}
```

```c
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 */
}
```

```c
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 / 30 sec
              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

```c
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);
}
```

```c
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 */
}
```

```c
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 / 30 sec
              resultFull);         // (out) full hash

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

    printf("OTP: %06d\n", result);
    return result;
}
```
An exception is handled by the kernel after sensitive registers are saved to secure stack.
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.
GService directly returns to the function
GService directly returns to the function

- Ginseng
- Library
- Normal World
- App
- Sensitive Function
- Ginseng Library
- Insensitive data / code
- Rich Operating System
- Secure World
- Secure Stack

- Secure Stack
- Ginseng Service
- Restore sensitive data
- Directly return to the sensitive function

Core
hmac() regs
Password

Protected by

Exception Vector
smc
/* exception * handle */
Microbenchmark: Overhead for code integrity

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

Onetime overhead per function

Re-enter a sensitive function:
3 K cycles