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

Min Hong Yun and Lin Zhong
Rice University
{mhyun, lzhong}@rice.edu

***Abstract*—Many mobile and embedded apps possess sensitive data, or secrets. Trusting the operating system (OS), they often keep their secrets in the memory. Recent incidents have shown that the memory is not necessarily secure because the OS can be compromised due to inevitable vulnerabilities resulting from its sheer size and complexity. Existing solutions protect sensitive data against an untrusted OS by running app logic in the Secure world, a Trusted Execution Environment (TEE) supported by the ARM TrustZone technology. Because app logic increases the attack surface of their TEE, these solutions do not work for third-party apps.**

This work aims to support third-party apps without growing the attack surface, significant development effort, or performance overhead. Our solution, called Ginseng, protects sensitive data by allocating them to *registers* at compile time and encrypting them at runtime before they enter the memory, due to function calls, exceptions or lack of physical registers. Ginseng does not run any app logic in the TEE and only requires minor markups to support existing apps. We report a prototype implementation based on LLVM, ARM Trusted Firmware (ATF), and the HiKey board. We evaluate it with both microbenchmarks and real-world secret-holding apps.

Our evaluation shows Ginseng efficiently protects sensitive data with low engineering effort. For example, a Ginseng-enabled web server, Nginx, protects the TLS master key with no measurable overhead. We find Ginseng's overhead is proportional to how often sensitive data in registers have to be encrypted and decrypted, i.e., spilling and restoring sensitive data on a function call or under high register pressure. As a result, Ginseng is most suited to protecting small sensitive data, like a password or social security number.

I. Introduction

Many mobile and IoT apps nowadays contain sensitive data, or secrets, such as passwords, learned models, and health information. Such secrets are often protected by encryption in the storage. However, to use a secret, an app must decrypt it and usually store it as cleartext in memory. In doing so, the app assumes that the operating system (OS) is trustworthy. OSes are complex software and have a large attack surface. Even techniques such as secure boot still leave the OS vulnerable to attacks after the boot, e.g., [11], [63]. Increasingly abundant evidence [10], [37], [42], [67], [74] suggests that prudent apps should not trust the OS with their secrets.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23327
www.ndss-symposium.org

There has been a growing interest in protecting app secrets against an untrusted OS, as summarized in Table I. Many reported solutions do not work for mobile and embedded systems that are based on the ARM architecture: they require either Intel SGX extension, e.g. Haven [8], SCONE [3] and Ryoan [32], or hypercall from userspace that is not available in ARM, e.g., Flicker [45], TrustVisor [44], and InkTag [31]. Others leverage ARM's TrustZone technology, which provides a hardware-supported trust execution environment (TEE) called the *Secure world*. Most of them run an entire or sensitive part of an app in the Secure world, e.g., [29], [40], [43], [60]. By doing so, they proportionally expand the attack surface of the TEE and as a result, do not support third-party apps. CaSE [76], a rare exception, only requires generic logic in the Secure world. Unfortunately CaSE requires a cache lockdown feature that is no longer available on the latest ARM architecture, i.e., AArch64. Moreover, CaSE keeps the entire app in the cache, which limits the app size to tens of KB; it also forbids concurrent OS activities, which incurs significant runtime overhead.

In this work, we present a new approach toward protecting app secrets against an untrusted OS, called Ginseng. Ginseng supports third-party apps without growing the attack surface, significant development effort, or performance overhead. It follows two principles to overcome the limitations of the prior work. First, app logic should not enter the TEE. Otherwise, the TEE's attack surface grows proportionally as the number of apps increases: app logic with a vulnerability opens the door to adversaries to compromise the TEE. Second, only sensitive data need to be protected. Protecting insensitive data incurs unnecessary overhead, which can be prohibitively high.

Following the two principles, Ginseng protects only *sensitive data*. The key idea is to keep them in *registers* only when they are *being used* and to save them in an encrypted memory region, called *secure stack*, when switching context. With the compiler's help, Ginseng infers which data are sensitive based on developer-provided hints and keeps them in registers. Whenever these registers have to be saved to the stack, e.g., due to function call or exception, Ginseng uses the secure stack to hide the sensitive data. We note some prior works also place secrets in registers in order to defend against cold-boot attacks [27], [28], [46], [47], [65]. These works, however, all trust the OS. Ginseng's runtime protection relies on a small, generic logic in the TEE, called GService. GService implements the behaviors of the secure stack, supports code integrity of functions processing sensitive data, and provides control-flow integrity (CFI) when sensitive data are in use. We enhance the security/safety of GService with a three-

pronged approach. First, we implement most of it in Rust, a safe language that guarantees software fault isolation by preventing unauthorized memory access and using a single ownership model [7]. Second, we minimize the unsafe part of GService to a small amount of assembly code, 190 lines in our implementation, which is amenable to formal verification by existing tools, e.g., Vale [9]. Finally, GService uses a statically-allocated memory region as its private heap, which further prevents it from affecting the existing TEE.

We report a Ginseng prototype using the HiKey board with ARM TrustZone. Combining Ginseng with known techniques that secure user input [41], [72], our prototype is the first to secure sensitive app data for its entire lifetime against an untrusted OS on ARM-based systems, without app logic in the TEE. The prototype has been made open-source [54].

Using both micro and macro benchmarks, we evaluate how much overhead Ginseng imposes and how app knowledge can help alleviate it. We build a two-factor authenticator processing a user's secret key to log in popular web sites such as Amazon and Facebook. We extend OpenSSL used by wpa_supplicant and the Nginx web server. Linux systems connecting to a Wi-Fi network use wpa_supplicant which saves Wi-Fi passwords in memory. Many IoT devices provide TLS-enabled web user interfaces using Nginx [35], [48], [71], which saves the TLS master key in memory. We also use Ginseng to protect a decision-tree-based classifier, which is popular on mobile and embedded apps and contains valuable intellectual property from the app vendor.

The evaluation shows Ginseng imposes a low overhead when computation with sensitive data does not contribute significantly to app execution time, which is usually true for I/O extensive apps. For example, Ginseng protects the TLS master key of Nginx with no measurable overhead for HTTPS transactions. Otherwise, Ginseng may impose a high overhead. For example, four functions processing the Wi-Fi password in wpa_supplicant dominate the app's execution time; this results in an overhead of 6 B cycles for protecting the password due to repeated function calls with a naïve implementation. Nevertheless, 75 % of this overhead can be eliminated by aggregating the functions and inlining callees, using app-specific knowledge, as we discuss in VI-C.

While we target Ginseng for ARM-based systems, Ginseng can be also realized in x86 systems as the latter also meet Ginseng's architectural requirements (see III-A1). Importantly, by keeping secrets in registers, Ginseng naturally protects them from cold-boot attacks, which extract and analyze system memory by exploiting the remanence effect of DRAM [30], [47]. Indeed, some existing defenses against cold-boot attacks also place secrets in registers, e.g, [27], [28], [46], [47], [65]. All of them trust the OS and even require OS collaboration at runtime. In contrast, Ginseng does not trust the OS and as a result, a significant part of Ginseng's innovation goes into preventing sensitive register content from entering memory without OS collaboration.

Although our prototype supports apps written C or C++, Ginseng can also protect sensitive data in managed code such as Dalvik bytecode, which constitutes the majority of Android apps. The challenge to applying Ginseng to managed code is that the managed runtime, not the compiler, determines the registers used for sensitive data. This challenge, however, can be sidestepped by using ahead-of-time (AOT) compilation and pre-compiling sensitive functions to binary on a developer's machine.

TABLE I. COMPARISON WITH RELATED WORK DISTRUSTING THE OS

Solutions	HW Req.	ARM Support	Attack Surface	Perf. Overhead
Ginseng	TrustZone or x86	✔	Fixed	Low
TLR [60] TrustShadow [29]	TrustZone		Prop.†	Mid-High
CaSE [76]		No AArch64	Fixed	High (no concurrency)
Overshadow [14] Flicker [45], InkTag [31] TrustVisor [44]	Hypercall from userspace	✘	*	*
Haven [8], SCONE [3] Eleos [50], Ryoan [32]	Intel SGX			

†"Prop." means that the attack surface grows proportionally with the number of apps, a result from having app logic in the TEE.

In summary, we make the following contributions:

- We report Ginseng, the first system design that protects *third-party* app secrets on ARM-based systems. Ginseng does not install any app logic in the TEE, requires only minor markups in app source code, and selectively protects only sensitive data for efficiency.
- We report a Ginseng prototype based on LLVM and an AArch64 board with ARM TrustZone, protecting sensitive data for its entire lifetime against the OS, starting from user input.
- We report the evaluation of Ginseng's performance and engineering overhead using microbenchmarks and four real-world apps with various secrets. It shows that Ginseng imposes overhead only when sensitive data is being processed and as a result, an app using sensitive data only sparingly, like Nginx, suffers no measurable overhead.

II. THREAT MODEL

We seek to protect the confidentiality and integrity of an application's sensitive data against an adversary who can access privileged software, i.e., OS. We do so without any app logic in the TEE. Below, we elaborate our assumptions and rationales.

Trusted Computing Base (TCB): Like many prior works [4], [6], [25], our TCB includes the hardware, bootloader for a higher privilege mode than the OS, and the software running in the higher privilege mode. At the hardware level, we trust the processor along with its security extensions, e.g., ARM TrustZone. At the software level, we trust the bootloader for the higher privilege mode and software in the mode. We do not trust the bootloader for the OS, the OS, and any software running in the OS. Following the TCB definition of Lampson et al [38], we do not consider apps relying on the TCB as part of the TCB because their misbehavior does not affect the TCB.

We do not trust the OS, but we assume the OS, e.g., Linux, is integral at booting time, thanks to the chain of trust, or secure

boot [1], [2], which verifies the integrity of the OS during boot. Nevertheless, secure boot only verifies that the kernel image is not modified; it does not fix vulnerabilities [11], [63]. Moreover, our threat model allows kernel modules to be loaded after booting, which are not subject to boot-time verification.

Threat Model: After boot, an attacker can compromise the OS and gain access to sensitive data of an unsuspicious app. The attacker can completely control the kernel and install any software. Then, the attacker can access memory content of an app by mapping the latter's physical page frames to the kernel's address space; or the attacker can simply turn off the MMU and access the memory content with physical addresses. Importantly, we assume a user or attacker can load a loadable kernel module (LKM). We do not address attacks on OS availability, side-channel attacks, and sophisticated physical attacks and consider them as out of scope.

No App Logic in TEE Utilizing the TEE as an app's execution environment helps the app conceal its memory content from the untrusted OS. However, doing so increases the TEE's attack surface and may result in information leakage of the entire trusted environment as demonstrated in Secure world attacks [36], [55], [63]. These attacks exploit the OS vulnerabilities to use privileged instructions; then, they compromise an app with vulnerabilities in the TEE and then the secure OS. That is, an app with vulnerabilities in the TEE becomes the gate for the attackers to enter the trusted environment. To prevent these types of attacks, both trusted apps and the secure OS must have no vulnerability, which is unrealistic as shown by continuous CVEs since 2013, e.g., [16]–[23].

To reduce app logic in TEE, many prior research efforts partition an app into secure and non-secure parts and only push the secure parts into the TEE [40], [60], [77]. This approach is hardly applicable to third-party apps because it still increases the TEE's attack surface and there can be many third-party apps. Thus, we follow the *principle of the least privilege* [58]. App secrets must be protected by the *mechanisms* that the TEE provides, not by running the apps in the TEE with excessive privilege. Ginseng is an example of such mechanisms.

III. Ginseng Design

Ginseng's goal is to protect the confidentiality and integrity of sensitive app data in an untrusted OS. Its key idea is for an app to retain sensitive data only in registers. When the data have to be saved to the stack, e.g., context switching, Ginseng saves them in an encrypted memory region, called *secure stack*. The hashes of the encrypted data in the secure stack are kept in the TEE so that the OS shall not break confidentiality and integrity of the data. In doing so, Ginseng limits sensitive data to local variables or function parameters and employs both compile-time and runtime mechanisms to keep them from leaving registers unencrypted. Figure 1 illustrates Ginseng's architecture.

This section provides an overview of Ginseng's design and elaborates its programming model and compile-time support for static protection. We will elaborate the runtime protection in IV.

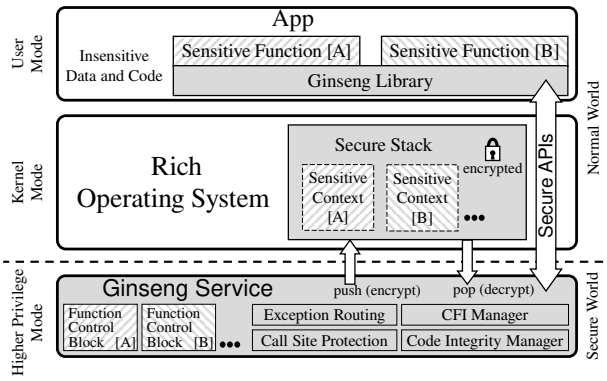

Figure 1. Ginseng Overview: a function with developer-marked sensitive variables directly communicates with GService through secure APIs. The service running with a higher privilege mode than the OS protects code integrity of the function, and confidentiality and integrity of sensitive data. Ginseng keeps sensitive variables in registers and uses the secure stack to protect them when switching context with the help from the Ginseng compiler. In ARM processors, the higher privilege mode corresponds to the Secure world supported by the TrustZone technology. Both the user and kernel modes are in the Normal world. The gray boxes are our contributions.

A. Design Overview

The basic unit of protection in Ginseng is a function. Ginseng provides a keyword `sensitive` for a developer to declare a local variable, parameter or return value of a function as sensitive, as shown in Figure 2. Once declared, we say that this variable and the corresponding function are *sensitive*. To ensure the confidentiality and integrity of sensitive data, Ginseng must ensure the code integrity of the sensitive functions via a collaboration of static and runtime protections. In addition to the code integrity, the collaboration also provides control-flow integrity (CFI) when sensitive data are in registers.

Ginseng implements static protection in its compiler. First, the register allocator only uses registers for the designated sensitive variables, never spilling them into the memory. This, however, is not enough because registers, as the function's execution context, can be saved to the memory, i.e., stack, when execution context changes, due to either function calls or context switching. Because it is the responsibility of a compiler to generate context-saving code for function calls, the Ginseng compiler saves and restores the registers with sensitive data to and from the secure stack.

Ginseng supports runtime protection with a small, passive piece of software with a higher privilege, i.e., running in the Secure world in ARM. The software, called GService, serves requests from a sensitive function. When registers with sensitive data must be saved/restored due to execution context change, GService encrypts/decrypts them before saving into/restoring from the memory, providing the abstraction of a *secure stack*. For the code integrity of a sensitive function, GService prevents the OS from mapping the function's code pages to the kernel address space. The service also ensures CFI by considering a function pointer as a sensitive variable.

Ginseng provides a small set of APIs for an app and GService to communicate with each other bypassing the untrusted OS.

1) Architectural requirements: Ginseng has three architectural requirements:

- a higher privilege mode than that of the untrusted OS to run GService;
- a direct call from an app to the higher privilege mode to bypass the OS;
- a way to trap writes to virtual memory control registers into the higher privilege mode.

We note that both x86 and ARM architectures meet the requirements. The hypervisor mode of Intel and AMD processors is suitable to run GService. A user process can use the hypervisor call instruction to directly communicate with GService in the hypervisor mode. It is also possible to trap modifications on virtual memory control registers using the virtual machine control fields in VMCS (or VMCB in AMD).

For clarity, we will use ARM 64-bit (AArch64) terminology to expose the design of Ginseng. We will use *Normal world* to refer to the rich OS and software running on top of it. When necessary, we will subdivide it into EL0 (user mode), EL1 (kernel mode), and EL2 (hypervisor mode). We will use *Secure world* to refer to the higher privilege mode, EL3. We will use AArch64 ISA and follow its calling convention. We will use *exception* to refer to asynchronous, i.e., interrupts, and synchronous exceptions caused by instructions, e.g., a permission fault by storing a register to read-only memory.

B. Programming model

Ginseng's unit of protection is a function. It protects local variables, parameters, and the return value of a function marked by developers as sensitive. We next elaborate this design choice and explain how it affects the programming model.

To use Ginseng, a programmer will mark a variable of any type as `sensitive` as shown in Figure 2. As protection comes with overhead, the smaller and the fewer `sensitive` variables, the more efficiently the program will run. Therefore, the incentive is clear and strong for the programmer to only mark the absolutely necessary variables. When compiling the program, the programmer indicates how many registers can be used for `sensitive` variables and the compiler will complain when the marked variables could not fit, which will be elaborated further in III-C.

The sensitivity of a variable is contagious both within and between functions. Ginseng's compiler performs static taint analysis to identify all variables that may carry the sensitivity. Because static taint analysis does not have the semantic information of all sensitivity propagations, it does so conservatively. For example, the return value of a local function taking a sensitive variable as a parameter will be considered sensitive. This may lead to an excessive number of sensitive variables, hurting performance. Therefore, we allow the app developer, who has semantic knowledge about the local function, to stop the sensitivity propagation by marking a variable as `insensitive`.

We limit sensitive data to local variables because protecting global variables is expensive and often unnecessary. To protect a global variable, the data page containing the variable needs to be encrypted and decrypted when the CPU enters and exits the kernel mode (EL1). For example, trusted software such as the secure monitor or trusted hypervisor [31] has to intercept all exceptions so that it can encrypt the data page before entering the kernel mode. When returning to the app, the trusted software has to decrypt the page, too. Because the kernel running on other cores can access the decrypted data page, only a single core could be online. Furthermore, TLB flushes and cache pollution further degrade the performance [50], [66]. On the other hand, using global variables for sensitive data is not absolutely necessary because a program can load sensitive data from their source on demand. For example, a program can load a secret key from the Secure world or a file system only when the key is needed. Therefore, Ginseng protects local variables only.

```
1  void hmac_sha1(sensitive long key_top,
2                 sensitive long key_bottom,
3                 const uint8_t *data,
4                 uint8_t *result) {
5    sensitive long tmp_key_top, tmp_key_bottom;
6    /* all other variables are insensitive */
7
8    /* HMAC_SHA1 implementation */
9  }
10
11 int genCode(sensitive long key_top,
12              sensitive long key_bottom) {
13   /* all other variables are insensitive */
14
15   /* use HAMC_SHA1 to compute 20-byte hash */
16   hmac_sha1(key_top, key_bottom, // sensitive data
17             challenge,           // current time/30sec
18             resultFull);         // (out) full hash
19
20   /* truncate 20-byte hash to 4-byte */
21   result = truncate(resultFull);
22
23   printf("OTP:_%06d\n", result);
24   return result;
25 }
26
27 void run() {
28   sensitive long key_top, key_bottom;
29
30   /* read a secret key from GService or a user */
31   s_read(TKN_KEY1_TOP, TKN_KEY1_BOTTOM, key_top);
32   s_read(TKN_KEY2_TOP, TKN_KEY2_BOTTOM, key_bottom);
33
34   genCode(key_top, key_bottom);
35 }
```

Figure 2. Simplified Two-factor Authenticator: a developer uses a keyword `sensitive` to mark sensitive variables and parameters.

Example: Figure 2, a snippet of a two-factor authenticator, reflects Ginseng's design decision. First, only local variables with sensitive data are protected. In `genCode()` (line 11-25), for example, the current time divided by 30 sec, `challenge`, and onetime password, `result` and `resultFull`, are not protected. The function declares only the two parameters as sensitive. We intentionally regard the onetime password as insensitive because this is refreshed at every 30 sec and not as important as the 80-bit secret key. Second, not all functions are sensitive. The bodies of `truncate()` and `printf()` called in line 21 and 23 are not protected because they do not contain a sensitive variable or parameter.

C. Static Protection

Ginseng extends the compiler to provide static protection of sensitive variables. The extension includes two parts. First, the compiler identifies all sensitive variables and keeps them

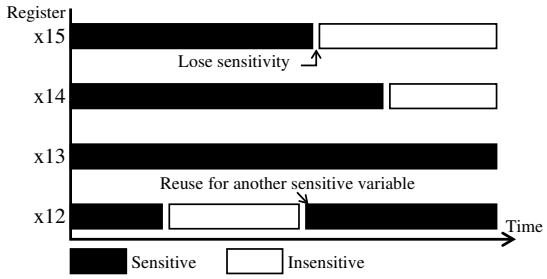

Figure 3. Dynamic Register Sensitivity Example: the sensitivity of a register follows a variable's liveness. The dynamic sensitivity enables the compiler to allocate a register to multiple sensitive or insensitive variables. GService does not know the true sensitivities of registers at runtime; hence, the service saves all registers that are potentially sensitive on an exception.

only in registers. Once a register holds the value of a sensitive variable, it becomes `sensitive`. Second, on a function call, the compiler emits instructions sending a request to GService which in turn encrypts sensitive data and saves them in the secure stack. Likewise, after the function call, the compiler emits similar instructions for restoring sensitive data from the secure stack to sensitive registers.

1) Allocating registers for sensitive variables: The register allocator of a compiler decides where a variable's value is stored: registers, stack or heap; Ginseng's compiler must keep sensitive variables in a set of predefined registers. With further help from the runtime protection (IV), Ginseng keeps sensitive variables away from memory that the untrusted OS can access.

The optimization goal of Ginseng's register allocation is to use as few sensitive registers as possible. This is because sensitive registers require special care by both the compiler and runtime so that they do not enter the memory and as a result, incur performance overhead.

To achieve this goal, Ginseng's compiler employs two complementary ideas. First, the sensitivity of a register should be dynamic, depending on whether it holds a *live* sensitive variable. At the variable's last use, the register loses its sensitivity and becomes free at which point the compiler can reuse it. Figure 3 shows an example. Register `x15` is allocated to a sensitive variable and becomes sensitive at the beginning of a function; as it loses sensitivity in the middle, it is allocated to an insensitive variable and becomes insensitive. The dynamic sensitivity allows the compiler to assign a register to multiple (sensitive) variables when their lifetimes do not overlap, e.g., `x12` in the figure.

The second idea is to allocate sensitive registers before others, a two-phase method. The compiler prioritizes sensitive registers to exclude them from the spillable register list and avoid unnecessary spills. If it allocates insensitive ones first, the compiler may not find a free register for a sensitive variable and need to introduce extra spills. In the first pass, the allocator allocates registers for sensitive variables and builds a sensitivity table that records the sensitivity ranges of registers. The table tells which registers are sensitive at a certain instruction. The second pass uses this information not to allocate them for insensitive variables. The call site protection (III-C2) also uses this information to determine which registers should be saved to the secure stack at a call

```
1  // x15 and x14 have sensitive data
2
3  // 1st argument (id_top)
4  mov     x0, #0x4e26
5  movk    x0, #0x577c, lsl #16
6  movk    x0, #0x2f99, lsl #32
7  movk    x0, #0x41e4, lsl #48
8
9  // 2nd argument (id_btm)
10 mov     x1, #0x7cc0
11 movk    x1, #0xcf91, lsl #16
12 movk    x1, #0x2362, lsl #32
13 movk    x1, #0x81e2, lsl #48
14
15 // 3rd argument (reg_vec):
16 //    sensitive data hide/move encoding
17 mov     x2, #0x2021000000000000
18
19 // 4th argument (dyn_dst):
20 mov     x3, xzr
21 bl      401ee4 <s_writeMoveV>
22
23 // prepare the last two arguments of hmac_sha1
24 // s_writeMoveV prepared the first two (x0 and x1).
25 sub     x2, x29, #24
26 add     x3, sp, #44
27 bl      400f20 <hmac_sha1>
```

Figure 4. Disassembled Call Site Protection Example: to protect sensitive data in registers `x15` and `x14`, the Ginseng compiler assigns a call site identifier, encodes sensitive data hide/move decision, and sends a request to GService through `s_writeMoveV()` before calling `hmac_sha1()`.

site. In the second pass, the allocator allocates registers for insensitive variables by referencing the table and excluding sensitive registers.

2) Protecting sensitive registers at a call site: In a function call, the caller and callee collaboratively save the caller's context in the stack following the calling convention. It is a compiler's responsibility to insert the code that saves/restores the context. When a function call is made within a sensitive function, Ginseng's compiler must save/restore sensitive registers only using the secure stack. That is, it emits instructions sending a request to GService through the secure APIs to be described in IV-A. Protecting sensitive registers at a call site is analogous to protecting them on an exception (IV-C1). Both have to save and restore the content of sensitive registers only using the secure stack. The difference is that the timing of an exception is unknown at compile time; thus, GService must intercept exceptions at runtime to protect sensitive registers.

The request sent to GService at a call site contains the call site identifier and information about sensitive registers and sensitive parameters of the callee. The identifier helps GService match the data saved in the secure stack with the corresponding call site. It is uniquely assigned to each call site by the compiler, à la AppInsight [53]. The compiler refers to the sensitivity table (III-C1) and inserts code to save all sensitive registers to the secure stack. It also decides which sensitive registers should be moved to registers for parameter passing according to the calling convention, i.e., `x0-x7` in AArch64.

One may wonder if a compromised Normal world entity could exploit the call site identifier to retrieve and modify sensitive data from the secure stack. GService addresses this by ensuring the code integrity of sensitive functions and that the request comes from a sensitive function, by checking the return address of the request saved in the exception link register (or the stack in x86). We will elaborate this in IV-C2.

TABLE II. GINSENG SECURE API

API	Description

# exposed to the Ginseng compiler 	
`s_writeMoveV (id_top, id_btm, reg_vec, dyn_dst)`
`s_readV (id_top, id_btm, reg_vec)`
	hides and restores sensitive data at a call site. The compiler assigns a unique identifier to each call site and encodes the hide/move information in the third parameter. `dyn_dst` is used for CFI and indicates whether the next branch instruction is for a function pointer.

`s_entry (id_top, id_btm)` `s_exit (id_top, id_btm)`	`s_entry()` checks code integrity. `s_exit()` sanitizes sensitive data from registers. The compiler assigns a unique identifier to each sensitive function.

# exposed to developers 	
`s_read (id_top, id_btm, reg_idx)`
`s_write (id_top, id_btm, reg_idx)`
	read and write a datum from and to GService. A 128-bit unique identifier (`id_top` and `id_btm`) specifies the datum to be read or written. The third parameter `reg_idx` decided by the Ginseng compiler specifies the source or target register.

Example: To call `printf()` with no sensitive parameter (line 23 of Figure 2), the compiler decides to hide all sensitive registers, `x15` and `x14` holding `key_top` and `key_bottom`. To call `hmac_sha1()` with two sensitive parameters (line 16), the compiler decides to move `x15` and `x14` to `x0` and `x1`, respectively. Once the compiler makes the register hide/move decision, it encodes the decision into the request to be sent to GService. Figure 4 is an example where the Ginseng compiler inserts a call to GService (`s_writeMoveV()`) and decides its arguments before calling `hmac_sha1()`. In line 4-13, the compiler inserts `mov[k]` instructions for the call site identifier. In line 17, the encoded information, `0x2021_0000_0000_0000`, tells GService to move `x15` to `x0` and `x14` to `x1`.

IV. RUNTIME PROTECTION

The static protection by the Ginseng compiler is necessary but not sufficient to protect sensitive variables in the Normal world. A compromised OS may modify a sensitive function to dump sensitive registers to memory, e.g., via code injection attacks. It can also access the stack when sensitive registers are saved to the stack as part of the execution context upon an exception such as a page fault or interrupt. When a sensitive function passes sensitive data to a callee, the OS may redirect the branching to a compromised function by compromising the control flow.

We now describe Ginseng's runtime protection against such accesses. The runtime protection heavily relies on GService, a passive, app-independent piece of software in the Secure world. GService ensures the *code integrity*, *data confidentiality* and *control-flow integrity (CFI)*. It does so only for sensitive functions to minimize overhead. It also modifies the kernel at three points, when booting, when modifying the kernel page table, and when handling an exception. Since we do not trust the OS, the kernel may overwrite the modifications. However, when any of these modifications is disabled, the kernel will infinitely trigger data aborts trying to modify read-only memory, thus ensuring sensitive data are always safe.

A. Ginseng internals

GService does not have any app logic. Instead, it tracks the execution of each sensitive function and provides a secure stack abstraction for each so that content of sensitive registers is encrypted before entering the memory. It exports a set of APIs for the Ginseng compiler (and developer) to insert into the app to track the execution of sensitive functions and to use their secure stacks properly.

Function Control Block (fCB): GService maintains a per-function data structure, called fCB, in the Secure world, to trace a sensitive function's execution. When accessing the secure stack, the service uses the trace to check a sensitive function's integrity and what registers can be sensitive on an exception (IV-C). GService allocates an fCB instance from a slab allocator when a sensitive function is first executed (IV-B). It adopts the least recently used (LRU) replacement policy when the allocator runs low on memory. The instance contains the information on physical memory address, code measurement, and a list of sensitive registers for the function. To trace execution of a sensitive function, the compiler-inserted code invokes GService at every entry point to a sensitive function, i.e., beginning of the function and returning from a callee.

Secure Stack: GService provides a secure stack abstraction for a sensitive function to save sensitive registers upon context switch such as a function call and exception. The secure stack is actually in the Normal world memory but its content is encrypted by GService. GService exposes two APIs for the compiler to encrypt/push and pop/decrypt data to and from the secure stack, respectively.

Secure APIs: The two APIs mentioned above to operate on the secure stack must bypass the OS. In the ARM architecture, Ginseng utilizes *security violation* to bypass the OS. A security violation occurs when a Normal world program accesses the Secure world memory. The processor can handle the violation in three different ways: ignoring the violation, raising an external abort (EA) in the Normal world, or raising an EA in the Secure world (EL3). We configure the system to raise an EA in the Secure world (EL3) by setting the external abort bit of the secure configuration register (`SCR_EL3.EA`). Thus, whenever an app attempts to access the Secure world memory, GService in the Secure world directly catches it and banishes the OS (in the Normal world) from the communication. GService allocates a unique Secure world memory address for each secure API.

Table II summarizes the three pairs of secure APIs supported by Ginseng. They are implemented as a library that will be linked with the app at compile time. The first pair are used by the compiler for accessing the secure stack. The second pair are used by the compiler to check the code integrity of a sensitive function at its entry point and to sanitize the function's sensitive data from registers at its exit point, respectively. The last pair are used by app developers to read and write sensitive data from and to the Secure world. We note that the compiler cannot use call-by-value scheme and pass the address of a sensitive variable because the variable is essentially a register without an address. Instead, the Ginseng compiler finds out the index of a sensitive register from the sensitivity table (III-C1) and passes the index. For example,

when the compiler works on `run()` in Figure 2, it knows that `x15` and `x14` are used for `key_top` and `key_bottom`. Thus, it passes `15` and `14` as the third argument of `s_read()` in line 31-32.

B. Code integrity of a sensitive function

Without code integrity, the OS can modify a sensitive function to access its sensitive data. Ginseng preserves code integrity by hiding the code pages of sensitive functions from the kernel with two complementary techniques.

First, we deprive the kernel of its capability of modifying its page table. At boot time, we make the kernel page table read-only. We modify the kernel so that when it needs to modify its page table, the kernel sends a request to GService via a higher privilege call, secure monitor call in ARM or hypervisor call in x86. For example, we modify `set_pte()` to send a request for setting the attributes of a page table entry. When GService receives the request, it changes the page table only when the modification would not result in mapping the sensitive function's code pages. This technique, assuming secure boot [1], [2], has been used in prior work [4], [6], [25]. It treats the kernel and app page tables in the same way so that it can monitor all physical-to-virtual mappings and avoid double mapping [4] to a sensitive function. We note that secure boot does not reduce the attack surface of the OS; nor does it deal with kernel modules loaded after boot, which is allowed by Ginseng's threat model.

We also forbid the kernel from overwriting its page table base register so that it cannot swap the table with a compromised one that the kernel can modify. To do so, we trap writes to the register to GService. On the ARM architecture, we first trap a write to the hypervisor mode (EL2) and then immediately forward the control to GService in EL3. In a way, EL2 functions as a relay to EL3, even when there is no hypervisor. Our benchmark shows the overhead for this trap is not only small (1.8 K cycles on ARM), but also occurs only once per booting.

Ginseng's approach to prevent the page table swap is different from prior work because of its uniquely powerful threat model, with the least restriction on attackers to the best of our knowledge. Prior work [4], [6], [25] statically removes the instruction modifying the kernel page table base register from the kernel image; and disables support for loadable kernel modules or does runtime verification on loadable kernel modules. However, with Ginseng's threat model, these are not enough. In our threat model, a user-installed root-privileged app can load a kernel module with instructions modifying the base register for kernel page table. Thus, Ginseng traps the instructions at runtime, instead of removing them statically.

Second, at a sensitive function's entry point, GService hides sensitive function code pages from the kernel and checks code integrity, invoked by `s_entry()` inserted by the Ginseng compiler. GService first walks the kernel page table to ensure no mapping to the function's code pages. During the page table walk, other page table modification requests from the kernel are delayed to avoid TOCTOU attacks. Then, GService hashes the function code and compares it with one that the compiler supplies and signs. Only after both checks pass, GService allocates and initializes an fCB instance for the function. We note that the page table walk, code hashing, and fCB initialization only happen when a sensitive function is invoked for the first time. Therefore, the overhead occurs only once for each function.

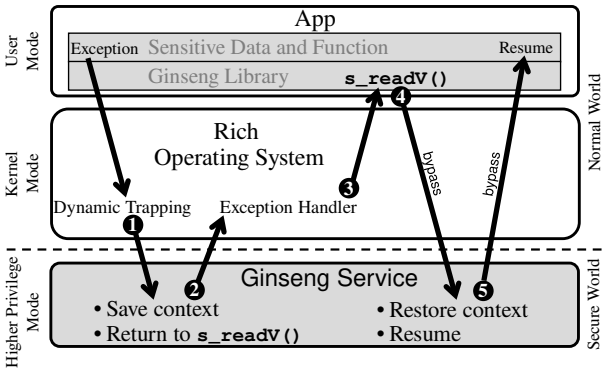

Figure 5. Exception Routing: GService intercepts all exceptions when sensitive data in registers. The service uses the secure stack to hide potentially sensitive registers. After the OS handles an exception, the service restores the data and resumes the function.

C. Data confidentiality

Statically allocating sensitive data to registers is not enough for data confidentiality. In the event of an exception or a function call, sensitive registers must be saved to the memory as a part of execution context. Ginseng's secure stack provides storage for sensitive registers, encrypted in the Normal world memory.

1) Exceptions within a sensitive function: When a CPU core is executing a sensitive function, an exception may transit the core into the kernel mode and save sensitive registers to the memory. GService intercepts all exceptions using *dynamic trapping* as shown in ❶ in Figure 5. Before handing exception handling to the OS, GService must save the sensitive registers to the secure stack (❷). Once the exception is handled by the OS, it must restore these registers and resume the interrupted function (❸-❺). To achieve the above, GService must do three things.

First, when an exception occurs, GService must intervene before the OS handles it (❶). Ginseng achieves this with a technique called *dynamic trapping*. In Ginseng, sensitive data can enter registers only via two secure APIs: `s_readV()` or `s_read()`. When either one of them is invoked, GService inserts the higher privilege call to the beginning of all exception vectors so an exception will immediately invoke GService. Dynamic trapping is a runtime modification on the kernel code in the memory but requires a source code change to reserve room for the higher privilege call. We insert the `NOP` instruction at the beginning of the exception vector source code in order to reserve the room. When the compiler or a developer uses the secure API to load sensitive data, GService replaces it with the Secure Monitor Call instruction, i.e., `SMC`. The replacement is possible because all instructions have the same length, 32 bits, in AArch64. In the x86 architecture, due to the variable length of instructions, one can reserve the room for the hypervisor call by inserting a three-byte `NOP` instruction [34], `NOP DWORD ptr [RAX]`, into the vectors. GService avoids unnecessary trapping by removing the replacement when sensitive data

```
1  // x15 and x13 have sensitive data
2  // x14 is a function pointer
3
4  /* omitted: preparing the first two arguments */
5
6  // 3rd argument (reg_vec):
7  //     sensitive data hide/move encoding
8  mov      x2, #0x210000000000
9  movk     x2, #0x2030, lsl #48
10
11 // 4th argument (dyn_dst):
12 orr      x3, xzr, #0x1
13 bl       401ee4 <s_writeMoveV>
14
15 // branching to a function pointer
16 blr      x14
```

Figure 6. Disassembled CFI Example: the compiler makes the callee's address in x14 sensitive and sets `x3` to indicate that the next branch instruction for a function pointer. GService checks the callee's integrity when servicing `s_writeMoveV()`.

leave registers, i.e., when `s_exit()` or `s_readWriteV()` is invoked.

Second, GService must save sensitive registers to the secure stack and then return to the OS' exception handler (❷). However, at runtime, GService only knows which registers are *potentially* sensitive based on the function's fCB. This is because the service does not have the sensitivity table (III-C1) used at compile time. Thus, the service saves all of potentially sensitive registers in the secure stack.

Finally, once the OS serves the exception, the control must be handed back to the app after restoring the sensitive data from the secure stack (❸-❺). For this, we change the return address twice to redirect the control flow. At ❷, GService saves the return address and replaces it with `s_readV()`'s address. This makes the kernel return to `s_readV()`(❸), not to the instruction when the exception occurs. When GService is invoked through `s_readV()`, the service restores the saved return address to return to the resume point of the sensitive function (❺).

2) Call site protection: Protecting sensitive data at a call site is similar to using the secure stack on an exception. In the call site protection, the Ginseng compiler does the most of the job (III-C2) because the call site is known at compile time. GService's goal for the call site protection is to restore sensitive data to the right call site, as instances of the same function call can be recursive or concurrent.

The compiler-assigned call site identifier helps GService recognize which sensitive data belong to which call site, but it is insufficient because a call site called recursively or concurrently will have the same identifier. If GService with no app logic cannot distinguish those calls with the same identifier, it may restore wrong sensitive data after a call site. To support recursive calls with thread-safety, the service employs two techniques. It first identifies a call site by its identifier and stack pointer, which differs per thread. Second, GService uses a LIFO-style storage to support recursive calls. As a result, it restores the most recently saved sensitive registers first when recursive calls return with the same identifier.

D. Control-flow integrity (CFI)

Ginseng ensures CFI only when sensitive data are in registers, i.e., when calling a function with sensitive parameters, returning a sensitive value, and restoring sensitive data from the secure stack.

1) Function pointer and return: When a sensitive function invokes another function through a function pointer and passes sensitive data through parameters, the sensitive data in registers for parameter passing are at risk. For example, in line 34 of Figure 2, calling `genCode()` as a function pointer. Because the address of a function is dynamically determined at runtime, an attacker may tamper with the function pointer to access sensitive data in registers. Similarly, when a function returns a sensitive datum, an attacker may try to compromise the return address. Direct function calls (not through function pointers) are included in the code pages, e.g., line 27 in Figure 4; thus, the code integrity of a sensitive function (IV-B) provides CFI for them.

To provide the CFI with a function pointer, Ginseng treats the pointer as sensitive and checks the integrity of the pointed function. Within a sensitive function, the compiler promotes a function pointer with sensitive parameters to a sensitive variable. Once the callee's address is loaded from memory to a sensitive register, the address is no longer accessible to the OS because it is a value in a sensitive register. When the caller sends a request to GService to push sensitive registers to the secure stack before a call site, the service checks the callee's integrity along with the call site protection (IV-C2). The service moves sensitive data to registers for parameter passing, only when the callee is sensitive and integral.

When a function returns a sensitive value to the caller, `s_exit()` inserted by the compiler at the exit point (IV-A) sends a request to GService, which in turn checks the returning-to function's integrity.

Example: Figure 6 is disassembled code when compiling Figure 2 after changing `genCode` of line 34 to a function pointer. The hide/move encoding in `x2` tells that register `x14` is a function pointer. The forth argument of `s_writeMoveV()` is `1` and indicates that the callee is pointed by a function pointer. The service searches the very next branch instruction, `blr x14` in line 16, and checks whether the callee is a sensitive function and integral at runtime.

2) Restoring sensitive data from the secure stack: GService restores sensitive data from the secure stack after exception handling and a call site, which can lead to leakage if the service returns to a compromised function, not to an integral sensitive function.

Since a compiler-assigned call site identifier is coded in a function body, an attacker may read an identifier from a binary file and try to illegally get sensitive data with the retrieved identifier. On an exception, however, GService always returns to the saved resumption address of a sensitive function (❺ in Figure 5); thus, an attack has no chance to break the CFI. Thus, GService provides the CFI after a call site, by checking whether the return address belongs to a sensitive function.

To provide CFI for a call site and restore sensitive data only for the right call site, the service must identify who requests the data to restore. GService finds the information on the requester from two addresses in the exception link register and the link register. The exception link register tells who sends the request to the service, which must be `s_readV()`. The link

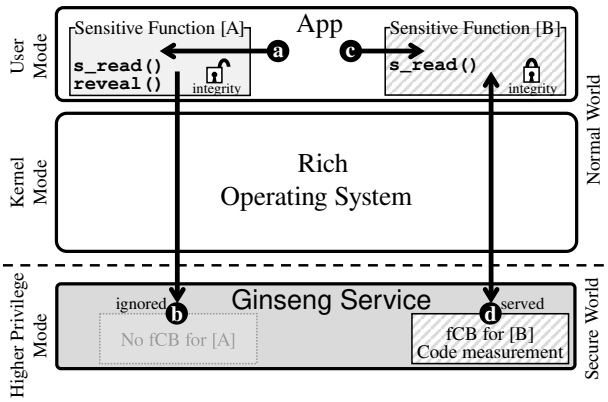

Figure 7. Protection Against Compromised Insensitive Functions: the service ignores the `s_read()` request from an unchecked sensitive function (❷) without an fCB instance. One from a checked sensitive function (❹) is served, but Ginseng keeps the sensitive data from leaving registers and sanitizes the data from registers at the exit point.

register tells who calls `s_readV()` to send the request. Thus, only when the former address belongs to `s_readV()` and the latter address belongs to a sensitive function with an fCB instance, the service decrypts sensitive data from the secure stack to sensitive registers. When checking the latter address, the service translates the address to a physical address and then compares it with the start address and size in an fCB instance (IV-B).

E. Attack Surface Control & Analysis

We discussed how Ginseng protects sensitive data against an attacker fabricating a call site identifier and exploiting control flow, in IV-D. In this section, we further discuss how Ginseng isolates GService from the existing software in the Secure world, why the dynamic trapping is integral, why (compromised) insensitive functions do not risk sensitive data, and the limitation of Ginseng.

1) Make GService bullet-proof: Ginseng requires GService to run in the Secure world, which is the TEE. An attacker may try to exploit the service's vulnerabilities to compromise the TEE as demonstrated by previous attacks [36], [55], [63]. We take three measures to reduce the attack surface from GService. First, we implement GService with a safe language, specifically Rust, which is type and memory safe [7]. Second, GService uses a statically allocated, private heap. These two measures confine GService's memory accessibility and enforce software fault isolation [70]. Finally, we minimize the use of unsafe code, i.e., assembly code in GService for accessing system registers. Given its small size, 190 SLOC in our implementation, its correctness can be easily verified using tools like Vale [9].

2) Attack Surface Analysis: Ginseng effectively prevents the OS from compromising dynamic trapping by its design. Because the higher privilege mode and code integrity forbid the OS from influencing ❷, ❹, and ❺ in Figure 5, the OS may try to compromise ❶ or ❸, instead. GService ensures the higher privilege call before loading sensitive data (IV-A); thus, whenever sensitive data are in registers, exception trapping (❶) is always enabled. Ginseng does not assure ❸; however, it does not break data confidentiality nor integrity because GService will not restore sensitive data unless requested via `s_readV()`. Moreover, once the service loads sensitive data for an exception, it always bypasses the kernel and returns to the resume point.

An attacker can completely rewrite an insensitive function. While this does not risk sensitive data, thanks to the code integrity of sensitive functions, the attacker can jump into the middle of a sensitive function as illustrated in Figure 7 by compromising an insensitive function. At ⓐ, a compromised control flow jumps into a sensitive function that has not been executed and calls `s_read()` to loads sensitive data from the Secure world. In this case, GService has not checked the code integrity nor created an fCB instance for the function; it simply ignores (ⓑ) any requests from the function. If the sensitive function's code integrity is already checked, at ⓒ, the compromised control flow can legitimately read sensitive data using `s_read()` (ⓓ) in the sensitive function. However, since the sensitive function's code integrity is ensured (IV-B), the read sensitive data will reside in registers and remain inaccessible to the attacker.

What Ginseng cannot prevent is replay attacks with `s_write()`. These attacks may break the integrity of sensitive data but cannot break confidentiality. A hypothetical example is that a function declares a nonce or salt as sensitive data and updates it using the secure API. Because the jump ⓒ in Figure 7 can legitimately read and write sensitive data, this attack may overwrite the value. Thus, if a nonce or salt is overwritten, a transaction will be ignored and the hashed result will differ. However, the overwriting does not affect the confidentiality.

V. Implementation

This section reports our ARM-based implementation of Ginseng. The EL3 of TrustZone provides a higher privilege than that of the OS. The direct communication between an app and GService leverages the security violation (IV-A), which is triggered by the app accessing Secure world memory and captured by GService, therefore bypassing the OS. Finally, by using EL2 as a trap relay point (IV-B), the service can trap writings to the virtual memory control registers.

We modify the Linux kernel v4.9 to make the kernel page table read-only and to relay page table modifications to GService, in a way similar to TZ-RKP [4] and SPROBES [25]. In a complete system, GService should verify that the modifications do not map a sensitive function's code pages to the kernel address space, also similar to TZ-RKP. Because our current prototype does not implement this verification, the overhead we report later does not include that of the verification. The verification overhead, however, would be negligible for our benchmarks: it is small [6] and happens rarely. The verification is necessary only when there is an update to the kernel page table such as system booting and process creation.

A. Static Protection

For the static protection, we prototype Ginseng compiler based on LLVM v6.0. The compiler allocates up to seven registers `x15-x9` for sensitive data. It can be easily extended to support floating point values and more registers for sensitive data.

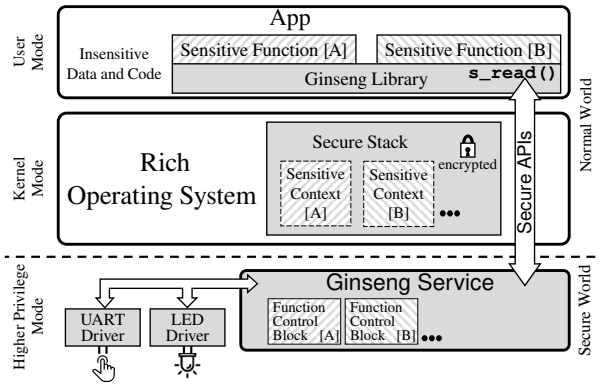

Figure 8. Secure Input-to-App Data Path: when an app requests sensitive user input, GService reads data from the dedicated input device (UART) and delivers them to sensitive registers specified in `s_read()` bypassing the OS. The service uses the dedicated LED as a secure input indicator by turning it on only when the secure input device is being used.

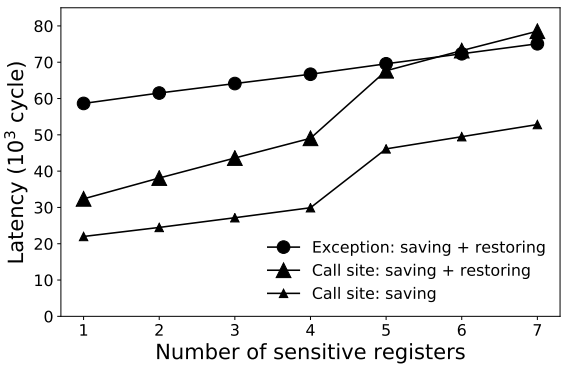

Figure 9. Overhead for Accessing the Secure Stack: the main source of the overhead for the secure stack at a call site is to encrypt and decrypt sensitive data and allocate a storage for them. On an exception, the four-time world switching, encryption, and decryption imposes the additional overheads. The standard deviations for a call site and exception are 0.66 K cycles and 1.03 K cycles at maximum, respectively.

B. Runtime Protection

GService: We prototype GService running on ARM Trusted Firmware (ATF) v1.4. As designed, we implement most of it in Rust. Instead of linking the Rust standard library, we use `linked_list_allocator` [49] to implement its heap with statically allocated memory. There are only 190 lines of assembly code for accessing system registers such as the secure configuration register. We import SHA1 and AES assembly implementations from the Google BoringSSL project and use them as a cryptographic library. It helps reduce overhead by using ARM's cryptographic instructions with 128-bit vector registers.

The Ginseng library implements the secure APIs that GService exposes (Table II). Each API is implemented with four lines of assembly code; it accesses a dedicated secure address to trigger a security violation. The address range is `0xF400_0000-0xF400_5000` in the HiKey board. The library, statically linked to the app, increases the binary size by 2.7 KB.

C. Secure User Input

To demonstrate Ginseng's practicability, we implement a secure input-to-app data path as illustrated in Figure 8, similar to TrustUI [41] and TruZ-Droid [72]. This results in a first-of-its-kind system that protects sensitive data for its entire lifetime on ARM-based systems without app logic in the TEE. We map the addresses of UART2 and LED2 to the Secure world's address space in `hikey_init_mmu_el3()` of ATF and use the devices as secure I/O. We also implement their device drivers in the Secure world. If GService cannot find a datum matching with the identifier of `s_read()`, the service turns on LED2, receives input from UART2 and delivers it to the sensitive register specified in `s_read()`. The drivers are also implemented in Rust with inline assembly of 161 SLOC.

VI. Evaluation

We evaluate Ginseng to answer the following questions.

- How much overhead does Ginseng impose to protect sensitive data? How much does each design component contribute?
- What is end-to-end performance overhead in practical applications? How much does each design component contribute to the overhead?
- How hard is it to apply Ginseng?

A. Microbenchmark

We report overheads due to the Ginseng interposition in a sensitive function. In each benchmark, we conduct 1,000 iterations and measure CPU cycles using ARM's performance monitoring unit (PMU). We make PMU count cycles at all exception levels in both Normal and Secure worlds through the PMU filter register (`PMCCFILTR_EL0`). We reset and read PMU from the user space to measure the overheads from an application's perspective, and theses operations take less than 10 cycles.

Overhead at entry and exit points: We first report the overhead at a sensitive function's entry point. As mentioned in IV-B, Ginseng compiler inserts a call to `s_entry()` at a sensitive function's entry point, and GService walks the kernel page table, hashes the function code, and allocates an fCB instance. The service does this only when a sensitive function is first executed; afterward, it references the fCB instance. We evaluate the overhead with varying the function size from 64 to 4096 bytes The overhead at an initial visit to a function is 11.81 M cycles. 96 % (11.33 M cycles) of the overhead is due to the kernel page table walk and 0.12 % (13.92 K cycles) is for hashing function code. The remaining cycles are for instantiating an fCB instance. When the same function is re-executed, the overhead becomes 2.95 K cycles, which is for searching the fCB instance and checking whether the program counter of the user space in `ELR_EL3` is within the function's range. At an exit point of a sensitive function, the compiler inserts a call to `s_exit()`. The overhead at an exit point is 3.10 K cycles as the work is simple such as sanitizing sensitive registers and deactivating the exception redirection.

Overhead at a call site: Before calling a function within a sensitive function, GService invoked by `s_writeMoveV()` encrypts and saves sensitive registers to the secure stack, and moves sensitive data to registers for parameter passing if used

as parameters. After the call, GService invoked by `s_readV()` decrypts and restores the registers. We plot the overhead to protect sensitive data on a call site in Figure 9. The overhead to save sensitive data starts from 22 K cycles and increases along with the number of sensitive registers. The figures shows a small jump from four to five sensitive registers. It is because the vector implementation used for the storage increases its capacity from four to eight when saving the fifth elements. Saving sensitive registers takes longer than restoration because of the secure stack frame allocation. The overhead for restoring registers starts from 10 K cycles and increases linearly along the number of registers to be restored.

Overhead on an exception: Exception redirection to protect sensitive data imposes an overhead of 58-74 K cycles depending on the number of sensitive registers, as shown in Figure 9. Unlike the overhead at a call site, the overhead does not jump when saving five registers because GService preallocates the secure stack frame at a function's entry point not on an exception. For this benchmark, we use a synchronous exception using the `SVC` instruction in our benchmark program. Because it is impossible to time an interrupt in the user space, we instead trigger a synchronous exception and measure the overhead to handle the exception. We invoke the `getppid()` system call using `SVC` as LMBench does for the null system call benchmark and exclude the cycles for the OS to serve the system call.

Overhead on `s_read()` and `s_write()`: A developer uses the two APIs to read and write sensitive data from and to GService. Both APIs impose the least overhead of 2.8 K cycles in the microbenchmark as their roles are simple. In both APIs, GService first reads two addresses in the exception link register and link register and identifies the requester like in IV-D2. If an fCB with the latter address is found, the service reads or writes a sensitive datum and returns to the application.

B. Applications

We apply Ginseng to four practical apps to quantify the engineering effort and measure end-to-end overhead. In their original forms, these apps save their secrets in the memory as cleartext and as a result, are vulnerable to an untrusted OS. Using Ginseng, we revise them so that the secrets are stored in their secure stacks and never enter the memory in the Normal world as plaintext.

1) Two-factor Authenticator: The two-factor authenticator, based on RFC 6238 [33], enables a service provider to test a user's credentials with both a regular password and a time-based onetime password (OTP) that changes every 30 seconds. To use the two-factor authentication, the service provider and the user share a secret key and generate the OTP by performing HMAC-SHA1 on the current time and the key. At the user end, the authenticator in a mobile device can encrypt the key in a file system. However, to generate an OTP, the authenticator must decrypt and store the key in the memory, making it vulnerable to a compromised OS.

Development effort: We implement the authenticator with 100 lines of C code and 150 lines of assembly implementation of SHA1 imported from the Google BoringSSL project. Since we implemented the authenticator without Ginseng ourselves, applying Ginseng to it took us a trivial amount of time. The authenticator processes a key and reversible intermediate keys in four sensitive functions, so we added the `sensitive` keyword at ten places of these functions. Ginseng protects the key read from GService. We have tested its correctness by logging into popular web sites, Facebook, Amazon, and Twitter, and it can be used for workstation login through PAM [59].

TABLE III. OVERHEAD BREAKDOWN (CYCLE)
Kernel page table walk and call site protection are dominant overhead factors

		Authenticator	wpa_supplicant		Classifier
Baseline		37 K	219 M		1.7 M
Overhead	Kernel page table walk	45,356 K	45 M	23 M	11.3 M
	Call site protection	680 K (17 times)	6,429 M (131,078 times)	1,640 M 40,988 times)	4.4 M (137 times)
	Exception redirection	9 K (0.13 times)	6 M (99.40 times)	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 (naïve)	2,299 M (optimized)	19.6 M

Performance: As shown in Table III, Ginseng imposes an overhead of 46 896 K cycles to the authenticator. The kernel page table walk constitutes 97 % of the overhead. It is because GService walks the kernel page table four times for four sensitive functions. The call site protection causes 1.5 % of the overhead due to 17 times of function calls within the sensitive functions. Since its binary size is 10 KB and sensitive data resides in registers only for 831 us, we observe no page fault and only two interrupts during the 15 iterations. The exception redirection causes the overhead of 9 K cycles on average. Although the overhead due to the kernel page table walk is seemingly high, it is less than 50 ms and, more importantly, onetime overhead when the sensitive functions are first called. Afterwards, when the authenticator generates a new OTP, the overhead becomes 1540 K cycles, less than 2 ms.

2) wpa_supplicant: wpa_supplicant is open-source software that is used by numerous Linux systems, including Ubuntu and Android, to connect to a wireless network using the Wi-Fi Protected Access (WPA) protocol. Vanilla wpa_supplicant reads a cleartext password from a configuration file or a network manager [26] and saves it in the memory. It derives the key for encryption/decryption from the password. Because the password is stored as cleartext, it is vulnerable to an untrusted OS.

Development effort: The Ginseng-enabled wpa_supplicant saves the password in GService; the configuration file only contains the UUID to retrieve it. We modify 25 SLOC out of 400 K SLOC in wpa_supplicant so that it reads UUID for a password from the configuration file. We also modify 90 SLOC out of 513 K SLOC in OpenSSL so that OpenSSL protects the password with Ginseng. We import BoringSSL's assembly implementation of SHA1 like in two-factor authenticator. It took one day for the first author, who was unfamiliar with wpa_supplicant and OpenSSL, to apply Ginseng.

Performance: We measure CPU cycles and time span from its start to when the WPA association is completed. We discuss the performance in a naïve implementation in this section and defer discussion on an optimized implementation to VI-C.

In the naïve implementation, the call site protection causes 90 % of the overhead. This is because wpa_supplicant calls a sensitive function that repeatedly calls `libc` and other sensitive functions. This leads to 131 K function calls and constitutes six billion cycles for the call site protection. The kernel page table walk introduces the same amount of overhead as that of the two-factor authenticator because both have four sensitive functions. Its binary size and running time, 4.2 MB and 20 sec respectively, lead to more exception redirections. During the 20 sec execution, we observe two types of exceptions, instruction and data aborts (a.k.a. page fault) and per-core timer interrupts. This exception redirection constitutes 0.1 % of the overhead. GService contributes 9 % of the overhead due to the repeated entries to and exits from sensitive functions, and internal storage (de)allocation thereof.

3) Learned Classifier (Decision Tree): Decision trees are light-weight classifiers that are widely used in practice [52]. A node in a tree selects an attribute of data based on the criteria and branches down to a leaf node for classification. The tree is an important intellectual property by the software vendor since it often requires valuable training dataset and efforts [69]. The software vendor often includes the tree in the software package distributed to users, e.g., installed in a smartphone. While the tree can be encrypted in storage, it must be decrypted to classify data. Once in memory, its confidentiality or integrity can be compromised by the OS [51]. As the criteria in tree nodes, i.e., attributes to be chosen at a node, determine a decision tree, Ginseng protects criteria in each node of a decision tree.

Development effort: We adopt the C4.5 decision tree implementation used by recent works on model security [12], [24] and use Ginseng to protect the information on what attribute to be chosen at a decision node. We only need to add six lines of code into the 5 K SLOC for declaring a single sensitive variable and reading the attribute information to the variable.

Performance: We train the tree with the voting dataset supplied with the implementation. As shown in Table III, 63 % of the overhead is due to the kernel page table walk and 25 % is for the call site protection. During the 15 iterations, we observe 81 exceptions which constitutes 2 % of the overhead. The implementation has a single sensitive function which is recursively called with no other function calls; thus, the 137 call site protections are only for recursive calls. The total overhead due to Ginseng is 18 M cycles; however, 63 % of the overhead is a one-time overhead due to the kernel page table walk, and the reoccurring overhead of 7 M cycles is less than 10 ms and barely perceptible when used interactively [61].

4) Nginx web server: Many IoT devices such as IP camera and wireless router [35], [48], [71] embed a web server, Nginx. For secure communications, the web server uses the Transport Layer Security (TLS) protocol, which derives session keys from a master key and uses them to encrypt and decrypt communications. Although OpenSSL, a TLS library used by Nginx, sanitizes session keys when a session ends, it saves the the master key in the memory for five minutes for session resumption [39], which is vulnerable to a compromised OS.

Development effort: In our modification, we protect the master key and derive the session keys without storing the master key in the memory. We modify 200 SLOC in OpenSSL so that it reads the master key from GService. We again import BoringSSL's assembly implementation of SHA512, which is used for SHA384 and has 980 SLOC. The modification is necessary largely due to limitations in our compiler prototype, which does not yet support an array with sensitive data. As a result, we have to manually modify OpenSSL to break down a 48-byte master key in an array into multiple variables. A full-fledged compiler supporting large or complex data types will eliminate this manual effort; and we believe extending the compiler for this only requires engineering effort. It took two days for the first author to apply Ginseng despite the complexity of OpenSSL.

Performance: We perform Apache benchmark (`ab`) 10 000 times on transferring 1-1024 KB. We do not observe any meaningful difference. For example, when we send 1 KB data over HTTPS, the vanilla web server processes 31.18 transaction per sec and the Ginseng-enabled server processes 31.35 transaction per second. This is because network and file system activities readily mask Ginseng's overhead. We note related efforts aiming at x86-based systems such as Fides [68] and TrustVisor [44] incur measurable overhead, up to 16 % [68] for the Apache web server.

C. Optimizing Ginseng's overhead

Following our experience of applying Ginseng to the above applications and quantifying its overhead, we now discuss ways that Ginseng's overhead can be reduced. Ginseng's recurring overhead mainly comes from the use of secure stacks. Two factors determine the cost of secure stack uses: the number of function calls within a sensitive function, the number and size of sensitive variables within a sensitive function.

The number of function calls inside a sensitive function determines the overhead from call site protection. For example, in wpa_supplicant (VI-B2), the overhead with the naïve implementation increases the execution time by 4 sec. The third row of Table III shows that 90 % of the overhead comes from multiple call sites repeatedly invoked in loops. To reduce the overhead due to the 131 K call site protections, we reduce the number of call sites within sensitive functions by inlining the small callees, e.g., `memset()` and `memcpy()`. This reduces the overhead by 75 %, from 6.4 B cycles to 1.6 B cycles as shown in the table.

When sensitive variables are many or large (V-A), the compiler may have to spill some of them to the secure stack, incurring an overhead similar to that of call site protection. Therefore, developers should leverage their knowledge about their apps to minimize the number and size of sensitive variables (III-B). For example, in wpa_supplicant, the Wi-Fi password cannot be reversely engineered from a derived key of 32 bytes. Therefore, there is no need to protect the derived key.

Ginseng incurs a onetime overhead when walking the kernel page table to protect the code integrity of a sensitive function. This overhead is determined by the number of sensitive functions (IV-B). By aggregating multiple sensitive functions into one, one can reduce this onetime overhead. For example, we aggregate the four sensitive functions of wpa_supplicant into two to reduce the overhead from the

kernel page table walk by $50\,\%$, as shown in the second row of Table III. However, such aggregation creates a sensitive function with more sensitive registers, which increases register pressure and may need higher recurring overhead. Therefore, developers must balance between onetime and recurring overheads using their knowledge of the app and the number of sensitive registers allowed by the compiler. For example, when aggregating sensitive functions of wpa_supplicant, we are wary that the compiler allows up to seven sensitive registers (V-A). Thus, we aggregate sensitive functions only when the aggregation does not lead to more than seven sensitive registers in the aggregated function.

VII. Related Work

Ginseng protects sensitive data against the untrusted OS, with a fine granularity to reduce the overhead. Ginseng also protects the data against cold-boot attacks without trusting the OS. In this section, we discuss how previous works protect sensitive data against the untrusted OS and cold-boot attacks.

A. Protecting secrets against untrusted OS

1) App Logic in Isolated Execution Environment: Previous works provide a special, isolated execution environment based on a higher privilege mode, e.g., hypervisor mode, or hardware security extension, i.e., Intel SGX. The environment is inaccessible to the rest of the system and provides protections on code and data in it [8]. Thus, previous works run applications with sensitive data or their sensitive parts in this isolated environment. Ginseng does not provide such an execution environment; instead, it protects only sensitive data by keeping them in registers. By not protecting insensitive data, Ginseng reduces the overhead of protection. Using registers, Ginseng also protects sensitive data against Iago attacks [13], which compromise an application through manipulated system call return values.

One can protect an unmodified application by simply running it inside an isolated environment. However, blindly protecting the entire application imposes high overhead, which motivates us to focus on only sensitive data. Overshadow [14] uses a hypervisor to present different memory views to the OS and a protected application, a technique called memory cloaking. CloudVisor [73] extends this idea to virtual machines; trusted software in the hypervisor mode interposes interactions between guest VMs and the existing VMM that is deprivileged from the hypervisor mode to the kernel mode. These hypervisor-based solutions are susceptible to the Iago attack and incur prohibitively high overhead due to excessive intervention for memory cloaking. SICE [5] constructs an isolated execution environment using the x86 system management mode (SMM) and a RAM resizing feature available only in AMD processors. Moreover, it must suspend the OS when the isolated application is running, incurring additional overhead.

Recent hardware extensions allow an isolated execution environment to be constructed with lower overhead. For example, Intel SGX supports such isolated environments as *enclaves*. Haven [8] runs a protected application in a dedicated enclave together with a trusted library OS. Panoply [64] reduces the TCB size by removing the OS from the enclave; instead, it provides a thin container in the enclave through which an application in the enclave can access the OS outside of the enclave. Ryoan [32] extends the isolated execution environment supported by Intel SGX to a distributed system. However, the use of SGX enclave still incurs significant overhead, up to $34\times$ [50], because enclave exits are expensive. SCONE [3] and Eleos [50] reduce this overhead. Unlike Haven, SCONE excludes the library OS from the enclave and issues asynchronous system calls through shared memory to avoid enclave exits. Eleos goes even further by handling page faults within an enclave through ActivePointers [62] and avoids enclave exits due to page faults.

All these works have a common problem, which motivates Ginseng: while protecting sensitive data, they also protect insensitive data and as a result, incur disproportionally high overhead. For example, despite clever optimization, Eleos slows down memcached $3.2\times$, and SCONE degrades Apache throughput by $20\,\%$. None of these works defend the secrets against cold-boot attacks as Ginseng does.

One can reduce the overhead of protection by only running the sensitive part of an application in the isolated execution environment. Flicker [45], TrustVisor [44], and InkTag [31] encapsulate sensitive functions and their data and run the sensitive code in an isolated execution environment provided by a hypervisor. The hypervisor's roles is similar to that of Overshadow and CloudVisor; however, this approach is only available in x86 systems because the ARM architecture disallows an application to directly use the hypervisor call as is required by previous works. Importantly, for SGX-based isolated execution environments, this approach can also incur high overhead because communication between app partitions is expensive due to enclave exits.

2) App Logic in Trusted Execution Environment: Related to Intel SGX, ARM TrustZone technology also provides a hardware-isolated environment; this environment is called the trusted execution environment (TEE) because the rest of the system is not only inaccessible to the environment but trusts it. TrustShadow [29] runs an entire app in TrustZone's TEE, substantially increasing the latter's attack surface. Liu et al. [43], AdAttester [40], TrustTokenF [77], and TLR [60] partition apps into sensitive and insensitive parts and only run the sensitive parts in the TEE, which still increases the latter's attack surface. Ginseng does not require any app logic in the TEE and provides a finer protection granularity, sensitive data, not all data in a sensitive function.

B. Protecting secrets against cold-boot attacks

When an attacker can physically access the hardware, the memory image can be extracted and analyzed by cold-boot attacks [30], [56], [57]. Countermeasures use alternative types of memory. Sentry [15] uses on-chip SRAM (iRAM) for inaccessibility and boot-time zero-filling by low-level device firmware. Cache is also used as an alternative to iRAM [28], [75], [76] or a combined way [15]. TRESOR [46], Amnesia [65], and ARMORED [27] use CPU registers. Sentry [15] and all the prior works using registers trust the OS. CaSE [76] does not trust the OS; it locks a sensitive application in the cache. Not only is the application limited to the size of the cache, the OS has to be stopped on all memory-coherent cores when the application runs. Ginseng distrusts the OS and is

deployable to all ARM systems that support ARM TrustZone and all x86 systems that support the hypervisor mode.

VIII. Concluding Remarks

In this work, we present Ginseng which protects sensitive data on the untrusted OS. We identify its architectural requirements and show that both ARM TrustZone and x86 hypervisor mode meet the requirements. To reduce unnecessary overheads, Ginseng keeps sensitive data in registers only when they are being processed. When the data have to be saved to the stack, Ginseng uses the secure stack which the OS cannot decrypt. We build Ginseng prototype based on ARM TrustZone and show Ginseng can be applied to practical applications with reasonable efforts and overhead. For example, the Nginx web server protects the TLS master key with modification to only $0.2\,\%$ of the source code.

Although we prototype Ginseng in an ARM-based device, we envision that adopting a different architecture could make the implementation simpler and lead to less overhead. With the hypervisor level of x86, for example, the secure APIs (IV-A) can be implemented with hypercalls, instead of security violation. Moreover, there is no need for forwarding the kernel page table modification trapped in the hypervisor to a higher level, i.e., the Secure world (IV-B). Finally, dynamic exception trapping (IV-C1) will not be necessary because the hypervisor can directly intercept exceptions.

Acknowledgements

This work was supported in part by NSF Awards #1611295, #1701374 and #1730574. Dr. Dan Wallach and Dr. Nathan Dautenhahn provided useful pointers to related work, especially possible attacks. Sicong Liu pointed us to the decision-tree classifier used in the evaluation. The authors are grateful to the anonymous reviewers and the paper shepherd Dr. Adam Bates for constructive reviews that made the paper better.

References

[1] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proc. IEEE Symp. Security and Privacy (SP)*, 1997.

[2] ARM, "ARM security technology: Building a secure system using TrustZone technology," http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[3] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell *et al.*, "SCONE: Secure Linux containers with Intel SGX," in *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2016.

[4] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[5] A. M. Azab, P. Ning, and X. Zhang, "SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2011.

[6] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning, "SKEE: A lightweight secure kernel-level execution environment for ARM," in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2016.

[7] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System programming in Rust: Beyond safety," in *Proc. Wrkshp. Hot Topics in Operating Systems (HotOS)*. ACM, 2017.

[8] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," in *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2014.

[9] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *Proc. USENIX Security Symp.*, 2017.

[10] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory," in *Proc. USENIX Security Symp.*, 2017.

[11] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on Android," in *Proc. ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, 2011.

[12] Y. Cao and J. Yang, "Towards making systems forget with machine unlearning," in *Proc. IEEE Symp. Security and Privacy (SP)*, 2015.

[13] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2013.

[14] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2008.

[15] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting data on smartphones and tablets from memory attacks," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2015.

[16] CVE-2015-6639, https://nvd.nist.gov/vuln/detail/CVE-2015-6639.

[17] CVE-2015-8999, https://nvd.nist.gov/vuln/detail/CVE-2015-8999.

[18] CVE-2015-9007, https://nvd.nist.gov/vuln/detail/CVE-2015-9007.

[19] CVE-2016-1919, https://nvd.nist.gov/vuln/detail/CVE-2016-1919.

[20] CVE-2016-1920, https://nvd.nist.gov/vuln/detail/CVE-2016-1920.

[21] CVE-2016-2431, https://nvd.nist.gov/vuln/detail/CVE-2016-2431.

[22] CVE-2016-3996, https://nvd.nist.gov/vuln/detail/CVE-2016-3996.

[23] CVE-2016-5349, https://nvd.nist.gov/vuln/detail/CVE-2016-5349.

[24] H. Gao, Y. Chen, K. Lee, D. Palsetia, and A. N. Choudhary, "Towards online spam filtering in social networks." in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2012.

[25] X. Ge, H. Vijayakumar, and T. Jaeger, "SPROBE: Enforcing kernel code integrity on the TrustZone architecture," in *Proc. IEEE Mobile Security Technologies (MoST)*, 2014.

[26] GNOME, "GNOME network manager," https://wiki.gnome.org/Projects/NetworkManager, 2017.

[27] J. Gotzfried and T. Muller, "ARMORED: CPU-bound encryption for Android-driven ARM devices," in *Proc. IEEE Int. Conf. on Availability, Reliability and Security (ARES)*, 2013.

[28] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with private keys without RAM." in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2014.

[29] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "TrustShadow: Secure execution of unmodified applications with ARM TrustZone," in *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2017.

[30] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," in *Proc. USENIX Security Symp.*, 2008.

[31] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "InkTag: Secure applications on an untrusted operating system," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2013.

[32] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data." in *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2016.

[33] IETF, "TOTP: Time-based one-time password algorithm," https://tools.ietf.org/html/rfc6238, 2011.

[34] Intel, *Intel 64 and IA-32 Architectures Software Developers Manual*, 2017, ch. 4.3.

[35] B. Japenga, *Security for Web-Enabled Devices*. Circuit Cellar, 2016.

[36] U. Kanonov and A. Wool, "Secure containers in Android: the Samsung KNOX case study," in *Proc. ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, 2016.

[37] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Security and Privacy (SP)*, 2019.

[38] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, 1992.

[39] J. Lee and D. S. Wallach, "Removing secrets from Android's TLS," in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2018.

[40] W. Li, H. Li, H. Chen, and Y. Xia, "AdAttester: Secure online mobile advertisement attestation using TrustZone," in *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2015.

[41] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proc. Asia-Pacific Workshop on Systems (APSys)*, 2014.

[42] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *Proc. USENIX Security Symp.*, 2018.

[43] H. Liu, S. Saroiu, A. Wolman, and H. Raj, "Software abstractions for trusted sensors," in *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2012.

[44] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proc. IEEE Symp. Security and Privacy (SP)*, 2010.

[45] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proc. The European Conf. Computer Systems (EuroSys)*, 2008.

[46] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *Proc. USENIX Security Symp.*, 2011.

[47] T. Müller and M. Spreitzenbarth, "FROST: Forensic recovery of scrambled telephones," in *Int. Conf. on Applied Cryptography and Network Security (ACNS)*, 2013.

[48] OpenWrt Project, https://openwrt.org/.

[49] P. Oppermann, "linked_list_allocator," https://crates.io/crates/linked_list_allocator, 2018.

[50] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS services for SGX enclaves." in *Proc. The European Conf. Computer Systems (EuroSys)*, 2017.

[51] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *Proc. IEEE Symp. Security and Privacy (SP)*, 2006.

[52] J. R. Quinlan, "Induction of decision trees," *Machine learning*, pp. 81–106, 1986.

[53] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "AppInsight: Mobile app performance monitoring in the wild," in *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2012.

[54] RECG download page, http://download.recg.org.

[55] D. Rosenberg, "QSEE TrustZone kernel integer overflow vulnerability," in *Black Hat Conference US*, 2014.

[56] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "GUITAR: Piecing together Android app GUIs from memory images," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[57] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. Richard III, "Screen after previous screens: Spatial-temporal recreation of Android app displays from memory images," in *Proc. USENIX Security Symp.*, 2016.

[58] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," in *Proc. of the IEEE*, 1975.

[59] V. Samar, "Unified login with pluggable authentication modules (PAM)," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 1996.

[60] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM TrustZone to build a trusted language runtime for mobile applications," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2014.

[61] S. C. Seow, *Designing and engineering time: The psychology of time perception in software*. Addison-Wesley Professional, 2008, ch. 3.

[62] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: a case for software address translation on GPUs," in *Proc. Int. Symp. Computer Architecture (ISCA)*, 2016.

[63] D. Shen, "Exploiting TrustZone on Android," in *Black Hat Conference US*, 2015.

[64] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux applications with SGX enclaves." in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2017.

[65] P. Simmons, "Security through Amnesia: a software-based solution to the cold boot attack on disk encryption," in *Proc. ACM Annual Computer Security Applications Conference (ACSAC)*, 2011.

[66] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2010.

[67] W. Song, H. Choi, J. Kim, E. Kim, Y. Kim, and J. Kim, "Pikit: A new kernel-independent processor-interconnect rootkit," in *Proc. USENIX Security Symp.*, 2016.

[68] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2012.

[69] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction APIs." in *Proc. USENIX Security Symp.*, 2016.

[70] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 1994.

[71] H. Xu, F. Xu, and B. Chen, "Internet protocol cameras with no password protection: An empirical investigation," in *Proc. Int. Conf. Passive and Active Measurement (PAM)*, 2018.

[72] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "TruZ-Droid: Integrating TrustZone with mobile operating system," in *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2018.

[73] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 2011.

[74] H. Zhang, D. She, and Z. Qian, "Android root and its providers: A double-edged sword," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[75] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "CacheKit: Evading memory introspection using cache incoherence," in *Proc. IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[76] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-assisted secure execution on ARM processors," in *Proc. IEEE Symp. Security and Privacy (SP)*, 2016.

[77] Y. Zhang, S. Zhao, Y. Qin, B. Yang, and D. Feng, "TrustTokenF: A generic security framework for mobile two-factor authentication using TrustZone," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, 2015.