iFlask: Isolate flask security system from dangerous execution environment by using ARM trustzone

Diming Zhang, Shaodi You

PII: S0167-739X(17)32223-9
DOI: https://doi.org/10.1016/j.future.2018.05.064
Reference: FUTURE 4239

To appear in: *Future Generation Computer Systems*

Received date : 29 September 2017
Revised date : 21 April 2018
Accepted date : 27 May 2018

Please cite this article as: D. Zhang, S. You, iFlask: Isolate flask security system from dangerous execution environment by using ARM trustzone, *Future Generation Computer Systems* (2018), https://doi.org/10.1016/j.future.2018.05.064

iFlask: Isolate Flask Security System From Dangerous Execution Environment by Using ARM TrustZone

Diming Zhang[a,b,*], Shaodi You[c,d]

[a]210023, Nanjing University, Nanjing Jiangsu, China
[b]212003, Jiangsu University of Science and Technology, Zhenjiang Jiangsu, China
[c]2601 Data61-CSIRO, Australia
[d]2601 Australian National University, Australia

Abstract

Security is essential in mobile computing. And, therefore, various access control modules have been introduced. However, the complicated mobile runtime environment may directly impact on the integrity of these security modules, or even compels them to make wrong access control decisions. Therefore, for a trusted Flask based security system, it needs to be isolated from the dangerous mobile execution environment at runtime. In this paper, we propose an isolated Flask security architecture called iFlask to solve this problem for the Flask-based mandatory access control (MAC) system. iFlask puts its security server subsystem into the enclave provided by the ARM TrustZone so as to avert the negative impacts of the malicious environment. In the meanwhile, iFlask's object manager subsystems which run in the mobile system kernel use a built-in supplicant proxy to effectively lookup policy decisions made by the back-end security server residing in the enclave, and to enforce these rules on the system with trustworthy behaviors. Moreover, to protect iFlask's components which are not protected by the enclave, we not only provide an exception trap mechanism that enables TrustZone to enlarge its protection scope to protect selected memory regions from the malicious system, but also establish a secure com-

---

[*] Corresponding author

Email addresses: `diming.zhang@gmail.com` (Diming Zhang), `shaodi.you@anu.edu.au` (Shaodi You)

munication channel to the the enclave as well. The prototype is implemented on SELinux, which is the widely used Flask-based MAC system, and the base of SEAndroid. The experimental results show that SELinux receives reliable protection, because it resists all known vulnerabilities (*e.g.*, CVE-2015-1815) and remains unaffected by the attacks in the test set.The propose architecture have very slight impact on the performance, it shows a performance degradation ranges between 0.53% to 6.49% compared to the naked system.

Keywords: iFlask, TrustZone, isolation
2010 MSC: 00-01, 99-00

1. INTRODUCTION

With the rapid growth in the mobile computing, mobile security has has become increasingly severe. Because mobile devices are designed as open and programmable and they can provide a large number of information services, including web browsing, instant messaging, financial transaction and so on. Therefore, access control has been employed by the mobile system as safeguard [1, 2, 3, 4]. However, the threats which can compromise the mobile system would breach these security services, because they run on the same privilege level and memory address space.

Isolation is a feasible option for elimination of this ripple effect. Currently, either separating the security module as a distinct system process with a smaller attack surface (*i.e.*, small Trusted Computing Base) [1] or using hypervisor-assisted isolation mechanism [5] is used to solve this problem. However, the former cannot provide enough protection if the kernel is compromised, while the latter may cause significant performance overhead and additional security problems in practice.

To solve these problems, we redesign the Flask security architecture based on ARM TrustZone technology [6], called iFlask . TrustZone-based devices can create an enclave, which is also named ”secure world”, to isolate the sensitive data from the normal world rich execution environment (REE). iFlask puts the security server into the secure world trusted execution environment (TEE), which can prevent adversaries from changing the policy decision behaviors. In iFlask , much of the communication between the object managers and the migrated security server is based on the cross-world calling from a built-in supplicant proxy, which runs in the normal world as a front-end part of iFlask. The responsibility of the *proxy* is to receive requests from the object managers, then forward them to the secure world and send back the results. Lowest level of cross-world calling is built on ARM Secure Monitor Call (SMC) instruction. Whenever an SMC instruction invokes, it will cause a switching to the monitor mode of the secure world, which can verify whether the call is safe or not. Thus, iFlask provides the required capabilities to do effective monitoring and protection for the policy decision process of its security server.

In the same time, the integrity of iFlask front-end components hosted by the normal world is also under protection. iFlask builds an exception trap mechanism to prevent unauthorized modification to critical CPU registers, such as *SCTLR, TTBRs, and TTBCR.* Meanwhile, by instrumenting the mobile system kernel, page tables cannot be directly modified by the kernel. Any modification is hooked by the SMC instruction and sent to the secure world to verify the legitimacy. Furthermore, iFlask prevents the physical memory double mapping to avoid bypassing the memory protection mechanism and object reuse. Thus, the security of the front-end components in the normal world are guaranteed, even though they are on the same privilege level and memory address space as the mobile system. Additionally, the communication channel to the secure world is secured by cryptography technology as well.

It is also notable that the performance overhead may particularly be of great concern in the mobile devices that are mostly restricted by severe resource constraints. Although the TrustZone hardware-assisted isolation does not cause significant system-wide performance degradation compared to other software-based solutions, the processor still needs to perform context switches to the secure world before accessing the isolated resources, so that frequently access to the secure world will inevitably impact on the overall performance. Hence, two measures are taken by iFlask for performance optimization. First, in the normal world, iFlask uses an access vector cache (AVC) to minimize the performance impact caused by the TrustZone-based isolation mechanism. The AVC allows the *proxy* to cache access decisions made by the back-end isolated MAC service in the secure world in order to minimize the performance overhead, because the MAC supplicant client does not usually need to perform additional lookups outside of that cache. And Second, iFlask makes secure world computation SoC-bound, indicating that the data, code, heap and stack have to be stored only in the cache. We also make use of the hardware-assisted cache locking function to pin down portions of the cache, without significantly impacting on the overall performance.

In summary, our contributions in this paper are:

- We design a feasible iFlask security architecture by using ARM TrustZone technology. Compared to Flask, iFlask shows superior security.
- Performance is also under consideration in iFlask, different optimization methods are adopted respectively in the normal world and the secure world.
- We implement the prototype based on SELinux, and test it thoroughly to verify the effectiveness of iFlask.

The remainder of this paper is organized as follows. Section 2 introduces the background knowledge. Section 3 presents iFlask design in detail. Section 4 describes our implementation based on SELinux. Section 5 discusses our experimental evaluation. Section 6 summarizes related work. Section 7 concludes this paper.

2. BACKGROUND

2.1. Flask Security Architecture

Flask [7] is a security architecture that restricts the level of control that users or subjects have over the objects that they create. Unlike in a discretionary access control (DAC), where users have full access control over their own files, directories, etc., Flask adds additional security contexts to all file system objects. Users and processes must have the appropriate access right to these categories before they can interact with these objects. However, adversaries can provoke unusual modifications to the Flask's behavior by tampering its execution state variables, such as some critical CPU registers and data. Moreover, due to the current monolithic kernel design, the Flask architecture is hard to guarantee its security thoroughly. The malicious code, such as kernel-level rootkits, can embed itself into the compromised kernel and stealthily inflicts damages with full, and unrestricted control to the Flask resources, like policies. Therefore, we propose iFlask to solve this problem in this paper.

2.2. ARM TrustZone Security Extension

ARM TrustZone is a security extension which enables the ARM devices to operate in both the normal and secure world in a time-sliced fashion. It introduces the notion of privilege separation to build an enclave. Diverse extensions are integrated across the system to isolate two worlds so as to ensure the confidentiality and integrity of the secure world. For example, the normal and secure worlds have their own CPU modes, ARM TrustZone uses an NS-bit in the secure configuration register (SCR) to control and indicate whether a CPU is executing in the normal world or the secure world. This bit is also used by TrustZone components to manage access to resources (*i.e.*, memory and peripherals) out of the CPUs. TrustZone memory adapter (TZMA) and TrustZone Address Space Controller (TZASC) partition the memory address corresponding to DRAM and SRAM into several memory regions, each of which is marked as either non-secure or secure. Moreover, the secure world has a higher privilege than the normal world so that it can freely access the resources in the normal world such as CPU registers, memory, and peripherals, but not vice versa. In addition, the monitor mode is the highest privileged mode, it is added alongside the existing privileged modes to coordinate and arbitrate between normal world and secure world.

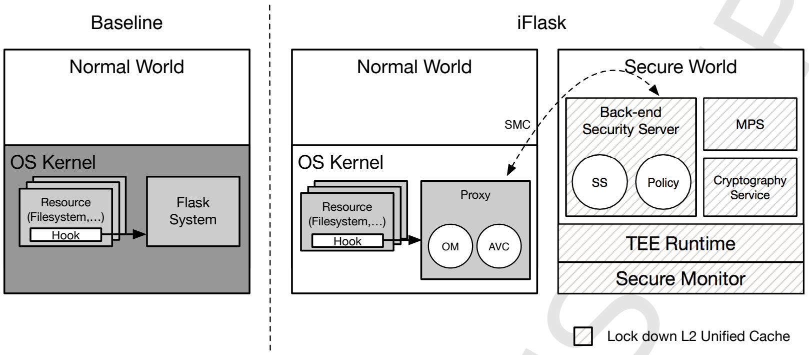

Figure 1: iFlask is a TrustZone based cross-world architecture. Its security server is hosted by the secure world, while its object managers are running in the normal world. The interaction between two worlds is based on SMC instruction. Memory protection service (MPS) and cryptography service are designed to protect iFlask's components outside the secure world. Additionally, the AVC module and the locking down L2 cache lines method are used to improve the system performance.

3. DESIGN

In this section, we describe the design of iFlask. First, we introduce the architecture overview of iFlask. Second, we present memory protection mechanism provided by iFlask. Third, we describe the secure communication mechanism used by iFlask. These three parts make up iFlask so as to build a comprehensive protection framework.

3.1. iFlask Overview

Figure 1 provides an overview of iFlask architecture. Compared to the *Baseline*, the architecture components of iFlask are across worlds: a back-end security server hosted in the secure world and the front-end object managers as well as the *proxy* serving in the normal world.

As we mentioned in Section 2.1, the core component of the Flask is the security server, which provides policy decisions based on security policies. Therefore, iFlask isolates the security server from the normal world in order to protect it from potential dangerous from the complicated execution environment. In the simplest implementation, the back-end security server need to provide security policy decisions, to maintain the security policy logic and policy-independent data (*i.e.*, security context and security identifier map), and to manage the AVC of the *proxy*. The back-end security server also provides functionality for loading and changing policies, moreover, its computation is SoC-bounded. This can prove advantageous because the security server can improve its response time by using cached results. Meanwhile, the security policy is stored in the secure world for security as well. All the MAC system components hosted by the back-end MAC service are isolated from the normal world so that adversaries are unable to directly launch attacks against them.

The *proxy* is responsible to route access control requests to the back-end security server, to cache policy decisions and to invoke object managers to enforce security policy decisions in the normal world. The object managers are used to enforce security policy decisions. The *proxy* provides three primary elements for object managers. First, the architecture provides interfaces for retrieving policy decisions from the security server. Second, the architecture provides object managers abilities to register and to receive notifications of changes to the security policy. Third, it provides an AVC module that allows object managers to access the policy decisions cached in the AVC to reduced the performance overhead. Since the *proxy* is not built in the secure world, iFlask provides memory protection based on exception trap mechanism for the components running in the normal world to enhance the rich execution environment security. Additionally, the communication channel to the secure world is also secured. Section 3.2 and Section 3.3 describe the design of these protection mechanisms in detail.

3.2. Memory Protection Outside the Secure World

The mobile system virtual address (VA) is divided into 3 distinct regions: user memory, the kernel code, and the kernel *const_map_mem*. The *const_map_mem* describes the physical address (PA) which is constantly mapped to the kernel. The kernel allocates memory from the *const_map_mem* region when it needs to create new objects like page tables. The user space memory regions are

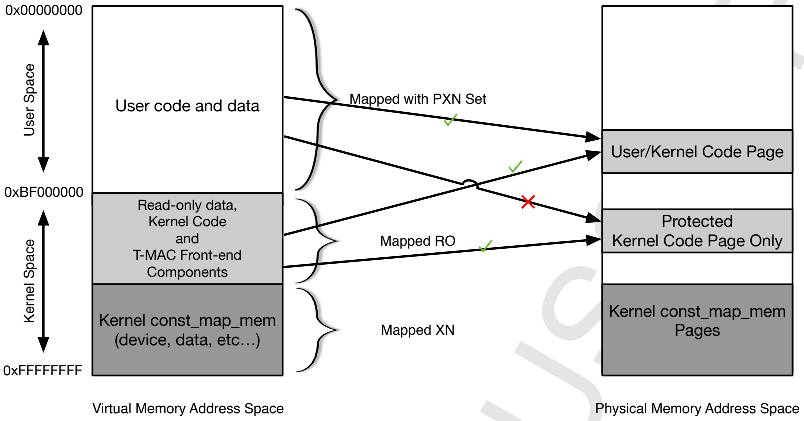
0x00000000
User Space
0xBF000000
Kernel Space
0xFFFFFFFF
User code and data
Read-only data, Kernel Code and T-MAC Front-end Components
Kernel const_map_mem (device, data, etc…)
Mapped with PXN Set
Mapped RO
Mapped XN
User/Kernel Code Page
Protected Kernel Code Page Only
Kernel const_map_mem Pages
Virtual Memory Address Space
Physical Memory Address Space

Figure 2: Normal World Virtual Memory Layout and Mapping Principle.

mapped as Privileged eXecute Never (PXN). The kernel code memory regions are mapped as read-only. The kernel *const_map_mem* region is configured as non-executable privileged memory. Meanwhile, the memory protection service (MPS) records the state of every page of the physical memory. Whenever a new VA to PA mapping is built, the MPS checks the new access permission given to the physical page against the control data to verify that there is no violation to the memory protection strategy. The physical memory state is saved in an array named the *phy_map*, each entry of this array corresponds to a 4KB physical page. The *phy_map* determines physical pages of whom need to be protected. The *phy_map* marks physical pages which are used by kernel code or page tables as protected. Any request to build a writable mapping to the protected physical pages will be rejected by MPS. Likewise, mapping a physical page which is already mapped writable to user space will be rejected by the same logic.

Furthermore, we deprive the kernel from its own capability to modify page tables. We replace the operation writing to critical CPU registers, such as *SCTLR, TTBRs, and TTBCR* with the SMC instruction. Then, modifications to protected page tables are obliged to request appropriate operations from the secure world. This is achieved by modifying the access permission in the page table entry so that the memory hosting the modified page tables become read-only region. Such a process begins when the mobile system performs initialization and is mandatory during each modify operation to the protected page table. If a page table is read-only to the kernel, usually, write to a read-only page is ended in a descriptor indicating a lack of permissions. When a page fault occurs, the page fault handler in the kernel will then generally pass a segmentation fault to the offending process, indicating that the update was invalid. The code of the page fault handler is part of the kernel. Its job is to analyze the cause of the fault and to do something about it. Therefore, we can replace the conventional page fault handler by SMC instruction so that it is able to make specific page faults trap into the secure world. The key point here is that TZASC and other bus peripherals can grant access for the secure world to read/write normal world memory. In addition, important to note that intercepting page table modifications does not need the kernel in a safe state even though the trap relies on the kernel to send the request for page table modification from the secure world. Since the read-only page tables are non-writable to the kernel, it is not possible for a compromised kernel to skip this mechanism without TrustZone knowledge.

According to the logic described above, iFlask front-end components such as object managers and the *proxy* can be protected by the MPS by modifying their *const_map_mem* mapping to be read-only.

3.3. Security Cross-World Communication

The Weakness of Communication Channel for TrustZone: When the normal world requests resources from the secure world, a communication channel is required for messaging between the two worlds. The channel simply use a block of world-shared memory area that is not secure as it is used by both normal and secure world.

iFlask provides a cryptographic service built in the secure world to protect the shared data. The key pair are generate in the secure world. The public key is sent to the *proxy* to encrypt and sign the shared data, and the private key is only saved in the secure world for decryption and verification. Therefore, adversaries cannot steal the private key, and the data in the shared memory can be sent to the secure world safely. Additionally, the remote attestation of the *proxy* is checked by the secure world, whenever it builds the session to the back-end security server. As long as the verification is invalid, the secure world will reject the further requests from it and reboot the device. Hence, we can guarantee that adversaries cannot breach the secure world by exploiting the insecure communication way.

3.4. Secure World Cache-Assisted Acceleration

Cache is one of the primary units to improve system performance in modern processor architecture. iFlask uses cache-assisted acceleration mechanism to reduce the response latency of the secure world application. Due to the tiny volume of the trusted execution environment in the secure world, it can be loaded into L2 unified cache entirely in order to improve execution efficiency. However, different from the physical memory resource which is divided into two distinct regions for two worlds respectively, the whole L2 unified cache is shared. It means that the data belongs to the normal world can replace the the cached data which belongs to the secure world.

In order to keep the cached secure world data will not be replaced by the normal world data, we should lock down L2 cache lines. Locking down cache for special usage will inevitably have an impact on the system performance. Hence, it is important to control the usage of locked cache lines. In iFlask, secure monitor has the highest priority to be locked into the cache, because it is responsible to handle the SMC exception and switch the context which is the first thing have to be done before entering the secure world. The other security services such as security server, MPS, and cryptography service are the second highest priority. The priority of security policy set is the lowest, but we can lock down most common policies. To the parts which are not locked in the L2 unified cache will be store in the SRAM, which is faster than the physical memory.

4. IMPLEMENTATION

4.1. SELinux and Kernel Instrumentation

SELinux uses the Linux security Module (LSM) in the Linux kernel to achieve mandatory access control in the REE. The main components of SELinux are the SELinuxFS, object managers, access vector cache, security server, and security policy. We move security server, security policy, and configuration files to the secure world, and keep the SELinuxFS, object managers, and the access vector cache staying in the normal world. We build a TEE to host the security server and the security policy in order to make them work properly. The object managers and the *proxy* are merged into the AVC module. The SELinuxFS is an extension module of the filesystem. Both the AVC and the SELinuxFS are stored in the kernel code memory region, hence, we mark their memory regions as protected so as to secure their integrity even though the REE is compromised.

The Inter-Process Communication (IPC) is the original way of communication between the SELinux components. By employing the iFlask architecture, the IPC is replaced by the Remote Procedure Call (RPC) which is responsible to cross-world communication. The *proxy* can invoke the back-end service. In our prototype, the RPC is based on GlobalPlatform TEE Client API in the REE and GlobalPlatform TEE Internal API in the TEE, which are the industry standard.

We directly modify the source code of the kernel to place hooks upon modifying page tables and upon writing to critical CPU registers. The hooks execute an SMC instruction to switch to the secure world. We use a command ID which is placed in a general purpose register upon the SMC call to differentiate the SMC instructions called by kernel hooks from those requesting the back-end security server. Whenever the execution switches to the secure world, the security monitor checks that register value to determine the call type. In addition to kernel instrumentation, we use a binary analysis tool to ensure that all critical CPU registers writes are replaced by hooks, which is a basic requirement for memory protection as discussed in Section 3.2. Likewise, we can insert an SMC

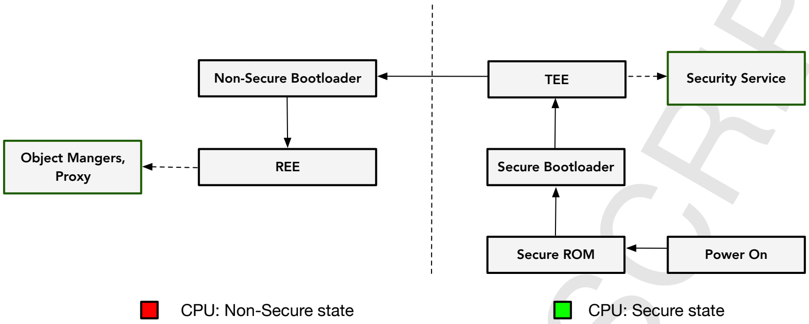

Figure 3: The secure boot sequence of the prototype.

instruction in place of the page table exception handler to implement the hooks so that page fault can execute an SMC instruction to switch the secure world.

4.2. Secure Booting

Before iFlask working properly, two booting phases are required on the device. The first phase stage is the secure world initialization. iFlask components in the secure world are loaded into the secure world as long as the TEE boots up and remains in the secure world memory until the system powers off or restarts. Figure 3 shows the secure boot sequence. Firstly, after power on, the hardware platform loads the secure bootloader from the secure ROM to the SDRM. Then, the secure bootloader initializes the Secure Monitor layer and secure world runtime layer in an orderly manner, and loads images of trusted service from the secure non-volatile storage into the memory of the secure world. Finally, the secure bootloader switches the primary CPU state from the secure to the non-secure so as to finish the first phase. The second phase stage is the normal world initialization. After the primary processor switching from the secure state to the non-secure state, it launches the non-secure bootloader to initialize the REE in the normal world. The last step of secure boot is initializing the iFlask components in the normal world. The most important work of the normal world part initialization is to interact with the back-end services, which is already running in the secure world, for object labeling. Then the booting finished. Notice that the system will panic if any step fails. The booting in

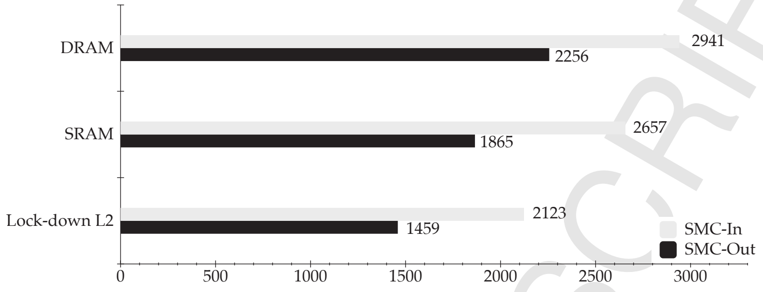

Figure 4: Different World switching overheads caused by three distince memory place.

our implementation is trusted, it prevents malicious firmware from running on the memory by authenticating all boot firmware images, as well as the normal world bootloader.

5. EVALUATION

In this section, we present the evaluation on our prototype. First, we use a micro-benchmark to measure the execution time required for a full context switching to and from the secure world. Second, we use a set of benchmarking tools to evaluate the overhead caused by employing iFlask. Third, we use some real world exploits to test the effectiveness of our design, and describe the security analysis about iFlask from multiple vectors. Performance evaluation was performed on the Hikey board, which is built around the HiSilicon Kirin 620 SOC whose microarchitecture is the Cortex-A53 with a 64-bit ARMv8-A instruction set.

5.1. Overhead of World Switching

Our first experiment is a micro-benchmark to measure the execution time required for a full context switching to and from the secure world. To enable a more accurate analysis, we use cycle counters and ARM cycle count register (CCNT) to ensure consistency across multiple CPUs in the analysis of micro-benchmark cases. Instruction barriers were utilized before and after tak-

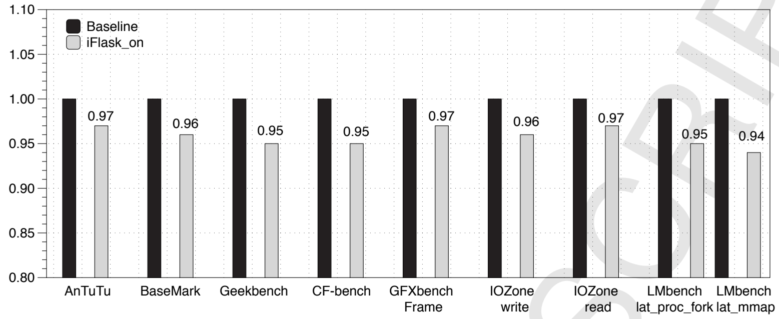

Figure 5: Mainstream benchmarking results.

ing timestamps to avoid out-of-order execution or pipelining from skewing our measurements.

The *SMC-In* micro-benchmark is to measure costs of the switch from the normal world to the secure world by directly issuing the SMC instruction. The *SMC-Out* micro-benchmark is for the opposite direction. The experimental results show that the cost of *SMC-In* was 2123 cycles and the cost of *SMC-Out* was 1459 cycles, indicating that the number of cycles of a full round trip of switch was around 3500 cycles. Exactly what needs to be saved and restored for each switch depends on the hardware design and the software mechanism used for inter-world communications.

Additionally, we measure the switching overhead without cache-assisted optimization. We put the secure monitor in three another two memory places in our platform, including DRAM and SRAM. As shown in Figure 4, the overhead caused by the secure monitor can be reduced by placing its code and data in fast memory close to the processor.

5.2. Performance Impact on Mainstream Benchmarking

Our second measurement is to use popular benchmarks to evaluate the performance overhead of TEE-perf implementation. We measured the performance with widely used mobile benchmark tools: AnTuTu, BaseMark and Geekbench. In the meanwhile, we also utilized other synthetic workload benchmark tools: CF-bench, IOZone, GFXBench and LMbench.

Table 1: APP Loading Time (in seconds)

Application	Baseline		iFlask		Extra Cost	
	1st	2nd	1st	2nd	1st	2nd
Firefox	1.58	1.26	1.77	1.38	0.19	0.12
Calculator	1.01	0.72	1.23	0.86	0.22	0.14
Calendar	0.98	0.64	1.09	0.75	0.11	0.11
Disk Utility	1.13	0.93	1.34	1.08	0.21	0.15
Search	1.03	0.47	1.24	0.58	0.21	0.11
Contacts	1.04	0.48	1.21	0.51	0.17	0.03
Gallery	1.12	0.72	1.36	0.83	0.24	0.11
Chess	1.11	0.88	1.29	1.01	0.18	0.13
Advanced Setting	0.99	0.81	1.21	0.89	0.22	0.08

To investigate the performance impact on the system, we tested the two cases: *Baseline* and *iFlask_on*. Note that the performance of *Baseline* represents the performance of applications when SELinux is running in the REE, while the performance of *iFlask_on* represents the performance of applications when iFlask is employed by SELinux.

Figure 5 provides the experimental results. In the figure, 1 represents the performance of the *Baseline*, and higher values indicate lower latency or higher throughput. When the iFlask is employed, overall performance is degraded slightly. As shown in the experimental results, iFlask shows a low overhead that ranges between 0.53% to 6.49%. The results are expected because these benchmarks involves a comprehensive evaluation of the overall system performance, which includes CPU, memory, and I/O. iFlask adds overhead to a small portion of these operations.

5.3. Latency of Application Loading Time

User application loading time is an important aspect of the performance of mobile devices since it impacts user experience. Therefore, the second measurement is to measure the impact of loading Linux user space applications.

To simulate the real feelings, the process of loading time measurement was done by using a Canon EOS 5D Mark III to record a video for displaying the open process, and the exact time spent in loading an app to display on the screen was extracted from the video by playing it back. Twice measurements were done by us for one application. The first measurement was for creating a fresh process, including making the policy decision, loading the binary code from the storage device to memory and displaying for the first time. The second measurement was made when the first measurement was finished and the target application was closed.

The results of the measurement are listed in Table 1, the extra cost indicates the overhead added by iFlask. It is observed that there exists an obvious difference between two measurements. The overhead in the first loading is higher than that of the second loading. For further investigation, we found that the high overhead in the first measurement is because there is no corresponding policy decisions cached in the AVC. While most of latency caused in the second measurement is due to kernel just executes the application process and displays it on the screen.

5.4. Security Analysis

iFlask passed through rigorous testing and evaluation that validate the effectiveness of its protection. We tested iFlask using the real world exploits, including CVE-2007-5495, CVE-2007-5496, CVE-2015-1815, and an attack called ”troubleshooter” published in Github. We also wrote our own attack code that writes to the physical memory using the */dev/mem* interface. Exploiting this vulnerability allows a user space process to trick the kernel into maliciously modifying its own memory hosting iFlask front-end components. We use these exploits to trick the kernel to write the protected memory region, page tables and parts of its data. All of these attacks failed because the protected memory region is mapped read-only. We also failed to modify the page tables to change the protected kernel memory’s read-only access permission.

iFlask is based on the ARM TrustZone technology, therefore all its security properties are contingent on the security of TrustZone. Recent reports [8, 9] showed that any user space application is able to execute shellcode in the secure world. These attacks require two basic conditions: (1) the normal world TrustZone driver accepts malformed *ioctl* command will allow installed application the execute arbitrary code in the kernel. (2) the secure world runtime layer has the mistake in input structure bound check may lead to an arbitrary code execution vulnerability in the secure world. In iFlask, the secure world runtime layer does not provide the public interface for the applications running in the normal world to load any executable code into the secure world. All applications running in the secure world are loaded by the secure bootloader and launched by the secure world runtime layer. There is no application dynamic loading feature in our TEE design. Therefore, we can guarantee that the basic conditions of attacks against TrustZone do not exist in our system.

Since the policy decisions are made in the secure world, the malicious code in the normal world has no privileges to access or modify any resources of the security server directly at runtime. The access policy set used in the security server are stored on the secure non-volatile storage. They are loaded into the secure world memory region by the secure bootloader when the system starts up. Therefore, the adversaries is unable to access the secure world part of iFlask from either the non-volatile storage or the secure memory except for the hardware attacks. In addition, the adversaries may try to tamper with the control flow of security server. But, due to the code of security server runs in the secure world, the attacks cannot modify its code. Moreover, since all the secure interrupts are triggered in the secure world, the adversaries cannot intercept the workflow of the secure world part through interrupts. Although a world-shared memory region that can be modified by the mobile system, iFlask provides the cryptography method to protect the security of communication channel between two worlds. Therefore, we can guarantee that the information to and from the secure world is safe.

6. RELATED WORK

Hypervisor-assisted isolation methods are widely researched and applied. They use virtualization technology to provide high privilege and isolation for protecting security services. Nevertheless, hypervisors have security challenges of their own. They are expected to do more tasks for system resource management and distribution. Their source codes are too big to ensure safety. Therefore, a number of vulnerabilities in hypervisors increase the risks of attacks on isolated security services. It is difficult to ensure the absence of exploitable vulnerabilities in hypervisors that could be utilized to disable security checks and access sensitive data.

In mobile computing, ARM is the most widely used instruction set architecture in terms of quantity produced, with over a hundred billion ARM processors produced as of 2017. Therefore, There are many researches [10, 11, 12, 13, 14, 15, 16, 17, 18] have managed to use TrustZone to protect their sensitive code and data of applications in an isolated execution environment against a potentially compromised mobile system. Most of them focus on building a thinner, more secure environment dedicated to process sensitive data, such as cryptography and authentication. In this paper, we provide an isolation mechanism to the MAC system by migrating it to the TrustZone secure world. In addition, we present techniques that focus on data integrity outside the secure world and on performance optimization to make iFlask a real world solution.

7. CONCLUSION

iFlask provides Flask system real-time protection based on ARM TrustZone technology. It migrates the security server to the secure world, and establish the memory protection, cryptography and digital signature to protect the rest part in the normal world. Compared to other feasible solutions, our design not only diminishes the performance overhead caused by isolation, but also reduces the Flask's attack surface so as to make most of specific attacks invalid. In other words, iFlask does not have to trade off isolation and effectiveness.

References

[1] S. Bugiel, S. Heuser, A.-R. Sadeghi, Flexible and fine-grained mandatory access control on android for diverse security and privacy policies., in:

Usenix security, 2013, pp. 131–146.

[2] S. Smalley, R. Craig, Security enhanced (se) android: Bringing flexible mac to android., in: NDSS, Vol. 310, 2013, pp. 20–38.

[3] S. Bugiel, S. Heuser, A.-R. Sadeghi, Towards a framework for android security modules: Extending se android type enforcement to android middleware. 05.12. 2012//cased. nr, Tech. rep., TUD-CS-2012-0231.

[4] A.-R. Sadeghi, Mobile security and privacy: The quest for the mighty access control, in: Proceedings of the 18th ACM symposium on Access control models and technologies, ACM, 2013, pp. 1–2.

[5] S.-M. Lee, S.-B. Suh, B. Jeong, S. Mo, A multi-layer mandatory access control mechanism for mobile devices based on virtualization, in: Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE, IEEE, 2008, pp. 251–256.

[6] A. ARM, Security technology-building a secure system using trustzone technology, ARM Technical White Paper.

[7] S. Ray, S. Stephen, L. Peter, H. Mike, A. Dave, L. Jay, The flask security architecture: System support for diverse security policies (1999) 123–140.

[8] D. Rosenberg, Qsee trustzone kernel integer over flow vulnerability, in: Black Hat conference, 2014.

[9] D. Shen, Exploiting trustzone on android, Black Hat US.

[10] N. Santos, H. Raj, S. Saroiu, A. Wolman, Using arm trustzone to build a trusted language runtime for mobile applications, in: ACM SIGARCH Computer Architecture News, Vol. 42, ACM, 2014, pp. 67–80.

[11] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, W. Shen, Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world, in: Proceedings of the 2014 ACM SIGSAC

Conference on Computer and Communications Security, ACM, 2014, pp. 90–102.

[12] X. Ge, H. Vijayakumar, T. Jaeger, Sprobes: Enforcing kernel code integrity on the trustzone architecture, arXiv preprint arXiv:1410.7747.

[13] M. Pirker, D. Slamanig, A framework for privacy-preserving mobile payment on security enhanced arm trustzone platforms, in: Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on, IEEE, 2012, pp. 1155–1160.

[14] W. Li, H. Li, H. Chen, Y. Xia, Adattester: Secure online mobile advertisement attestation using trustzone, in: Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, ACM, 2015, pp. 75–88.

[15] B. Yang, K. Yang, Y. Qin, Z. Zhang, D. Feng, Daa-tz: an efficient daa scheme for mobile devices using arm trustzone, in: International Conference on Trust and Trustworthy Computing, Springer, 2015, pp. 209–227.

[16] H. Sun, K. Sun, Y. Wang, J. Jing, H. Wang, Trustice: Hardware-assisted isolated computing environments on mobile devices, in: Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on, IEEE, 2015, pp. 367–378.

[17] H. Sun, K. Sun, Y. Wang, J. Jing, Trustotp: Transforming smartphones into secure one-time password tokens, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 976–988.

[18] A.-A. Reineh, G. Petracca, J. Uusilehto, A. Martin, Enabling secure and usable mobile application: Revealing the nuts and bolts of software tpm in todays mobile devices, arXiv preprint arXiv:1606.02995.

Diming Zhang receives his B.E. degree in science from the Soochow University, Jiangsu, China, in 2009, and the Master degree in software engineering from the Southeast University, in 2011. In 2011, he joined College of Computer Engineering, Jiangsu University of Science and Technology, as a Lecturer. His current research interests are operation system, parallel computing and architecture.

Shaodi You received his Ph.D. and M.E. degrees from The University of Tokyo, Japan in 2015 and 2012 and his bachelor's degree from Tsinghua University, P. R. China in 2009. He is currently a research scientist at Data61-CSIRO (formerly known as NICTA), Australia. He also serves as adjunct lecturer at Australian National University, Australia. His research interests are physics based vision, deep learning and high-performance computing. He is best known for physics based vision in bad weather, especially in rainy scenes. He serves as reviewer for TPAMI, IJCV, TIP, CVPR, ICCV, SIGGRAPH and etc. He is currently the Chair of IEEE Computer Society, Australian Capital Territory Section, Australia.}

Highlights

- We design a feasible iFlask security architecture by using ARM TrustZone technology. Compared to Flask, iFlask shows superior security.
- Performance is also under consideration in iFlask, different optimization methods are adopted respectively in the normal world and the secure world.
- We implement the prototype based on SELinux, and test it thoroughly to verify the effectiveness of iFlask.