Secure Execution of Mutually Mistrusting Software

Dongli Zhang
Stony Brook University
dozhang@cs.stonybrook.edu

Abstract

Commodity operating systems, e.g. Linux and Android, running on PC or smartphone, are ubiquitous in home, commercial, government, and military settings. The booming popularity of PC and smartphone makes the commodity operating system an attractive target for attacks. These systems are tasked with a variety of applications, e.g. from secure software provided by trusted enterprises to regular applications including games and web browsers downloaded from untrusted third-party website.

Since PC and smartphone are used both for working and entertainment, both trusted and untrusted applications are installed on the same commodity operating system. The complex interface between malicious applications and the operating system kernel makes the latter one vulnerable to malware. The compromised untrusted operating system is able to break both privacy and integrity of secure applications. The user mode secure application is not tamper-resistant and immune to the privileged malicious operating system kernel.

Various methods have been proposed to execute mutually mistrusting software on commodity operating systems. In this talk, we divide the state of the art research papers into three classes. First, we discuss how to protect the secure application from the untrusted operating system. Second, we discuss the isolation of untrusted application from the benign operating system. Third, we discuss how to remove the trust relationship between application and operating system, that is, neither application nor operating system trust each other. We finally propose a framework for the secure execution of sensitive code on ARM architecture with TrustZone technology.

Research Proficiency Exam Report, August, 2014, Stony Brook, NY, USA.

1. Introduction

Nowadays, PC and smartphone running the commodity operating systems, which are ubiquitous in home, commercial, government and military settings, become significantly indispensable in our life. According to a report from Gartner [1], the worldwide PC shipments totaled 82.6 million units just in the fourth quarter of 2013. The most popular operating systems on PC are Windows, Linux and Mac OS. Various applications are installed and executed on the commodity operating system every day. Besides PC, recent years have also experienced explosive growth of smartphone sales. Inevitably, the rise in the popularity of smartphones also makes them an attractive target for attacks. According to the report from Canalys [2], the year of 2011 marks as the first time in history that smartphones have outsold PCs. Their booming popularity can be partially attributed to their improved functionality and convenience for end users.

PCs are used for a variety of daily works such as checking emails, video conference and data processing. Smartphones are also no longer basic devices for making phone calls and receiving text messages, but powerful platforms, with comparable computing and communication capabilities to commodity PCs, for video conference, gaming, and even online shopping. Generally, different applications have disparate level of security requirement. For instance, on a smartphone running Android, the online banking application has a higher level of security requirement than the Angry Birds.

The execution of mutually mistrusting software brings security trouble to the commodity operating systems. Suppose the PC is running a Linux operating system. The user might use the PC to view a lot of PDF files every day. The PDF file injected with malware is able to infect the operating system because of the complex interface between the application and the operating system. Latter, the user may login into the online bank account to check the deposit. As the underpinning privileged operating system kernel, including keyboard driver, has already been compromised, the browser's privacy, i.e., the user's credentials, will be stolen by the attacker.

Another example is on a smartphone running an Android operating system. Users are suggested to download Android applications from official website, such as Google Online Store. However, many users download repacked ap-

plications, e.g., games, from the untrusted third-party website. For instance, without protection, the malicious repacked game is able to elevate the privilege, compromise the Android runtime and even the Linux kernel. The compromised Android operating system is able to infect the privacy and integrity of other applications, e.g., the user's identification of Facebook.

In this paper, we discuss the solutions for the problems introduced by the execution of mutually mistrusting software. We formulate the problems into three sub-problems. The first problem is to protect the secure application or sensitive PAL (Pieces of Application Logic) from the untrusted operating system. Commodity operating systems entrusted with securing sensitive data are remarkably large and complex, and consequently, frequently prone to compromise. The privacy and integrity of the application are expected to be protected even in the event of a total OS compromise. Solutions to this problem have a variety of applications in real life from protecting the privacy and integrity of the certificate generation process on a CA server to isolating the sensitive list of Transaction Authentication Numbers (TAN) from the untrusted Android operating system on a smartphone.

The second problem is to isolate the untrusted application or an untrusted piece of code from the operating system. Modern commodity operating systems are underpinning many applications from different sources. The complex interface between the application and the operating system opens a large attack surface for the misbehaving application to compromise the rest of the operating system. The operating system can be compromised by a piece of native code downloaded and executed by the web browser with mechanisms such as ActiveX and NPAPI, or an Android application infected by DKFBootKit [13].

The third problem is how to establish the two-way protection, that is, to protect the application from the operating system while to protect the operating system from the application. We rethink the security model and argue that a two-way sandbox is desired. We discuss MiniBox [27], the first paper whose objective is to remove the trust between applications and the operating system on the Platform as a Service (PaaS) cloud computing.

In this paper, we discuss the evolution of prior works. We organize previous works based on the problems classification and essential design considerations when building solutions. The evolution graph of prior works on the three problems is in Figure 1. In section 2, we discuss the protection of secure application from the untrusted operating system. Section 3 is about the isolation of untrusted application from benign operating system. In section 4, we discuss MiniBox [27], the most up-to-date research paper on removing trust between application and the underpinning operating system.

2. Secure App on Untrusted OS

2.1 OS can be untrusted

Although commodity operating systems are developed by extremely professional software developers, the security provided by commodity operating systems is often inadequate. Trusted OS components include not only the kernel but also device drivers and system services that run with root privilege (e.g., daemons that run as root in Linux). Once such privileged code is compromised, an attacker gains complete access to sensitive data on a system. The privileged operating system kernel can read/write any code/data region of any user mode process. Both privacy and integrity are imperiled by the hostile operating system.

Besides directly manipulating a secure application's state, the hostile operating system kernel can also compromise the user mode application by Iago attack [17]. On commodity operating systems, the application and kernel are conceptually peers and the system call API defines an RPC interface between them. A carefully chosen sequence of integer return values to Linux system calls can lead a supposedly protected process astray. The following sample code demonstrates one Iago attack. It maps a 1024 byte region of memory via the mmap2 system call and then reads up to 1024 bytes into it from a file descriptor using the read system call. Since the kernel is responsible for memory management, instead of the address of the newly allocated memory region, it returns an address on the stack. The stack will be overwritten with up to 1024 bytes of the kernel's choice with the read system call. Therefore, a saved return address on the stack may be overwritten and the program can be coerced into executing a return-oriented program.

```
p = mmap(NULL,1024,prot,flags,-1,0);
read(fd,p,1024);
```

To protect the secure application from the malicious OS, a mechanism should be used to isolate the secure application from the OS. In the meantime, the isolated application should also be able to use the OS services. Ideally, the Iago attack is prevented. In the following sections, we will discuss different mechanisms on secure application protection. We divide the prior works into four classes as in Table 1.

2.2 Trusted Hardware based

Trusted Hardware allows the execution of pieces of application logic (PAL) in an isolated environment. Flicker [32] leverages the hardware support for Trusted Platform Module (TPM) [3] and late launch recently introduced from AMD's Secure Virtual Machine (SVM) technology. SVM chips are designed to allow the late launch of the software (e.g. Virtual Machine Monitor or Security Kernel) at an arbitrary time with the SKINIT instruction in CPU protection ring 0. As part of the SKINIT instruction, the processor first causes the TPM to reset the values of PCRs 17-23 to zero, and then transmits the contents of the PAL to the TPM so that it can

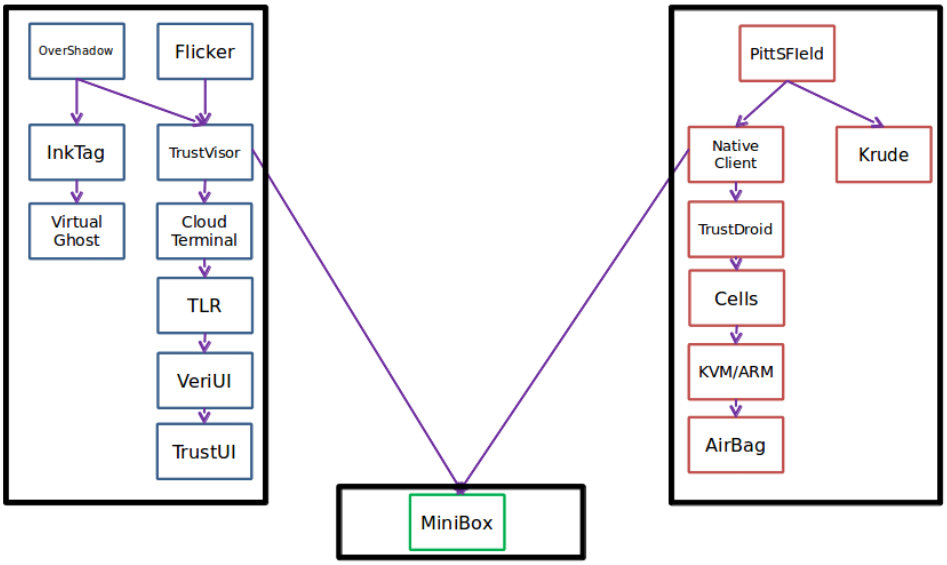

Figure 1. Evolution graph of prior works. The left branch includes prior works when application is trusted but OS is malicious. The right branch includes prior works when application is untrusted but OS is benign. MiniBox is the combination of two branches.

Solution Category	Research Papers
Trusted Hardware Based	Flicker [32], TrustVisor [31]
Hypervisor Based	Overshadow [18], InkTag [23], TrustVisor [31], Cloud Terminal [29]
Instrumentation Based	Virtual Ghost [19]
TrustZone Based	TLR [34], VeriUI [28], TrustUI [26]

Table 1. Solution categorization on the protection of secure application (PAL) from the untrusted OS.

be measured and extended into PCR 17. Software cannot reset PCR 17 without executing another SKINIT instruction. PALs can leverage TPM-based sealed storage to maintain state across Flicker sessions. Therefore, the sensitive task can be spit into multiple sessions.

However, Flicker's performance is not promising, especially for multi-session PAL. At the beginning of each session, the TPM should decrypt sealed data from persistent storage to recover the state of the previous session of this PAL. At the end of this session, the TPM seals the data again and stores the data on persistent storage. The frequent encryption/decryption undermines the performance of Flicker. TrustVisor [31] improves the performance of Flicker by simulating the TPM-based cryptography operations on a micro-TPM. The micro-TPM is simulated by CPU chip. TrustVisor, which is a tiny hypervisor, is booted with the late launch. It is responsible for the registration, invoke and unregistration of all PALs. TrustVisor isolates PAL from the untrusted operating system with the nested page table (NPT). Hardware TPM attests the integrity of the tiny hypervisor and the integrity of PAL is attested by the software-emulated micro-TPM. TrustVisor is the combination of both trusted hardware and hypervisor based solutions.

2.3 Hypervisor based

As we mentioned in last section, hypervisor can isolate the secure application from the untrusted operating system. Overshadow [18] utilizes a binary translation based hypervisor with a mechanism called cloaking to prevent the guest operating system from reading or tampering application code, data and registers. Cloaking is the mechanism to present an application context with a cleartexted view of its pages, and the OS context with an encrypted view. At any point in time, the page is mapped into only one shadow page table - either a protected application shadow used by cloaked user-space processes, or the system shadow used for all other accesses. Overshadow introduces a shim into the address space of the cloaked application, which cooperates with the VMM to mediate all interactions with the OS. The shim uses an explicit hypercall interface for interacting with the VMM. System call transitions between guest-user mode and guest-kernel mode are always trapped by a Binary Translation based VMM.

However, Overshadow has focused on simply isolating trusted code and data from the OS, with minimal support for securely using OS features, that is, it is not able to prevent Iago attack. It also doesn't support flexible access control

and crash consistency. InkTag proposes paraverification, a technique that simplifies the hypervisor by forcing the untrusted OS to participate in its own verification. InkTag requires the untrusted OS to provide information and resources to both the hypervisor and application that allow them to efficiently verify the operating system's actions. Verifying that the OS provides system services correctly allows InkTag to avoid having to reason about the OS's implementation of these services. Trusted application code executes in a high assurance process, or HAP, which is isolated from the OS. Nearly all application-level changes are contained in a small, 2000-line library (libinktag) the use of which is largely encapsulated in the standard C library.

Cloud Terminal [29] protects the secure application by running the software on the remote server, instead of locally. In Cloud Terminal, the only software running on the client, which the user interacts with, is a lightweight secure thin terminal whose primary functionality is to render the bitmap sent by the remote server. Most application logic is in a remote cloud rendering engine on the remote server. On the client side, the secure thin terminal is isolated and protected by the hypervisor. The tiny hypervisor helps supply a secure display and input path to remote software. The secure thin terminal has a very small TCB (23 KLOC) and no dependence on the untrusted OS. Therefore it can be easily checked and remotely attested to.

2.4 Instrumentation based

Virtual Ghost protects application from a compromised or even hostile OS. It leverages compiler instrumentation (with LLVM) to create ghost memory that the operating system cannot read or write. Virtual Ghost is based on the Secure Virtual Architecture (SVA) [20]. In SVA, all kernel and module code must first go through LLVM intermediate representation form (bitcode). The SVA VM translates code from virtual instruction set to the native instruction set of the hardware. SVA adds a set of instructions to LLVM called SVA-OS; these instructions replace the hardware-specific operations used by an OS to communicate with hardware and to do low-level state manipulation. During the translation from virtual instruction to native instruction, load/store operations are instrumented so that access to secure memory pages can be prevented from OS without unmapping or encrypting secure pages. Virtual Ghost also enforces Control Flow Integrity (CFI) [14] on kernel code in order to ensure that the compiler instrumentation of kernel code is not bypassed.

2.5 TrustZone based

Considering the limited computing resources on smartphones, hypervisor and instrumentation based solutions are not applicable to the smartphone. TrustZone [4] is utilized on smartphone to protect the secure application. TrustZone is a hardware security technology incorporated into recent ARM processors. With TrustZone, the processor can execute instructions in one of two possible security modes, referred to as the normal world, where untrusted code executes, and the secure world, where secure services run as in Figure 2. These processor modes have independent memory address spaces and different privileges. While code running in the normal world cannot access the secure world address space, code running in the secure world can access the normal world address space in certain conditions. Besides memory, peripherals and interrupt are also world-sensitive. World switch is done via a special instruction called the Secure Monitor Call (smc).

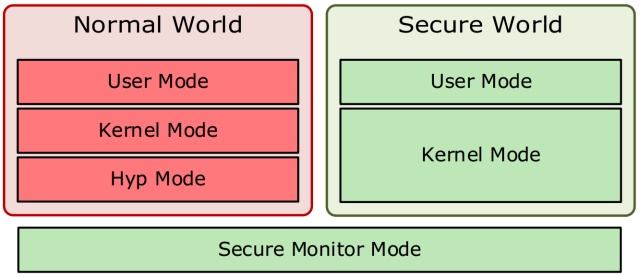

Figure 2. Split CPU Mode with TrustZone Support

Trusted Language Runtime (TLR) [34] is a system that protects the confidentiality and integrity of .Net mobile application from OS security breaches by separating and isolating the application's security-sensitive logic from the rest of the application. TLR and security-sensitive code are in the secure world of TrustZone. TLR is a small runtime engine that is capable of interpreting .Net managed code inside a trusted secure environment. It is carefully crafted by borrowing parts of the runtime engine design from the .NET Micro Framework (NETMF) so that the TCB is significantly smaller than a full-blown .NET framework and a full-featured OS. Security-sensitive code and data are in a Trustbox which is an isolation runtime environment that protects the integrity and confidentiality of code and data. The Trustlet specifies the secure data and an interface that defines what data can cross the boundary between the Trustbox and the untrusted world. With TLR, the developer should manually split the application into sensitive and nonsensitive part. A secure application can package the code handling sensitive data into TrustLet and run it in the TrustBox in the Secure World.

However, TLR does not support direct I/O within the Secure World. VeriUI [28] is able to securely handle user inputs (i.e., passwords) and communication with remote servers. Smartphone applications often augment their functionality by accessing user data mentioned by services such as Twitter and Facebook. VeriUI is proposed to prevent phishing attacks by untrusted OS through a secure and isolated environment for password input and transmission. An app can invoke a web browser running in the secure environment of TrustZone to retrieve an OAuth token after the user successfully authenticates. Even the malicious OS cannot have access to the password data. The secure kernel running in

secure environment can use its protected resources (i.e., a vendor-installed public-key pair) to generate a signed attestation that includes a hash of the Secure World's system software as well as information about the user's login request.

As VeriUI runs a Linux in TrustZone secure world to provide the attested login for users, it has a very large TCB. TrustUI [26] takes a step further by excluding drivers for user-interacting devices like touch screen from its trusted computing base. TrustUI is a new trusted path design for mobile devices that enables secure interaction between end users and services. It is based on ARM's TrustZone technology and requires no trust of the commodity software stack. TrustUI adopts a mechanism that logically splits a device driver into two parts: a backend running in the normal world and a frontend running in the secure world. The backend part is the unmodified driver and its corresponding wrapper in the normal world, while the frontend part works on top of it and provides safe access to device for secure pages. The two parts communicate through corresponding proxy modules running in both worlds which exchange data through shared memory.

3. Untrusted App on Benign OS

In this section, we discuss the protection of the operating system from an untrusted application or a piece of untrusted code. In this paper, we call both untrusted application and untrusted code as untrusted module. The untrusted module can be pieces of native code downloaded by a web browser, an application uploaded and executed on the PaaS server, or an Android application downloaded from an untrusted third-party website. Although the isolation of untrusted module can prevent it from infecting the operating system, it is far from enough. There are other challenges. First, the isolated code module wants to interact with the operating system services via system calls. Second, the isolation (sandbox) should not impact the performance of program execution. Third, a low implementation overhead is expected, that is, the modification to compiler, linker, application source code and operating system kernel source code should be minimized. Last, since the smartphone has limited resources, the isolation should be lightweight.

In this paper, we categorize the prior works according to the granularity of isolation as in Table 2. The granularity of isolation varies, including intra-process, inter-process, inter-namespace and inter-VM.

3.1 Inter-VM based

The naive approach is to isolate each untrusted module into its corresponding VM. There are a variety of virtual machine monitors, including Xen [5], KVM [6], Qemu [7], and VMWare [8]. Recently, the hardware virtualization extension has been added into the ARM and the ARM based KVM [21] is integrated into the Linux kernel since Linux 3.9. Since this approach is clear and self-explained, we will not discuss it in detail in this paper.

3.2 Intra-Process based

Intra-Process protection is to isolate the untrusted module from other memory regions in the same address space. SFI [35] is proposed to sandbox the untrusted module by rewriting the untrusted code at the instruction level, that is, to instrument store/load and control flow instructions. However, it only works for RISC architectures. PittSFIeld [30] presents sandboxing technique that can be applied to CISC architecture e.g. IA-32, and whose application can be checked at load-time to minimize the TCB. Unlike RISC architectures, whose instructions have the same length, the x86 has variable-length instructions that might start at any byte. To avoid this problem, PittSFIeld divides memory into segments whose size and location is 16-byte aligned. New instructions are instrumented before store/load and control flow instructions to check that the sandboxed module cannot read/write data outside sandbox and transfer to illegal control flow target outside sandbox.

A weakness of PittSFIeld is it cannot effectively mediate the access from untrusted module to operating system services. Besides isolating the untrusted module, Native Client [37] also allows the module to interact with services, such as file I/O and local database access, by the combination of intra-process and inter-process approaches. An Intra-Process based sandbox is used to isolate the untrusted module from the runtime service, which resides in the same address space as the sandboxed untrusted module. Runtime service mediates the communication between the untrusted module and other processes including web browser and other services.

3.3 Inter-Process based

Krude et al. [25] propose an inter-process based approach to sandbox the untrusted module. It is especially designed for PaaS architectures, where code execution needs to be isolated to protect tenants from unauthorized access to their data by other tenants and to protect the host system from any type of intrusion by other tenants. The untrusted module is uploaded to the PaaS server and it is isolated in a new process. Krude et al. use the process barrier and the seccomp filter mechanism to restrict access to memory and to the system call interface. Almost all system calls are blocked for the isolated process. Besides memory allocation and deallocation, the isolated process can communicate with OS by sending to request to a supervisor process via pipe, which is the IPC mechanism on Linux. The supervisor process will process the request and send the response back to the isolated process also via pipe.

3.4 Inter-Namespace based

The Inter-Namespace based approach is primary proposed for smartphone running Android. Nowadays, smartphones are ubiquitous. Many people use the smartphone both for

Solution Category	Research Papers
Inter-VM Based	KVM/ARM [21]
Intra-Process Based	SFI [35], PittSFIeld [30], Native Client [37]
Inter-Process Based	Native Client [37], Krude et al. [25]
Inter-Namespace Based	TrustDroid [16], Cells [15], AirBag [36]

Table 2. Solution categorization on the protection of OS from the untrusted application.

working and personal needs. However, the personal applications downloaded from the untrusted website can compromise the application issued by the trusted enterprise. Therefore, many users carry multiple phones to accommodate work, personal and geographic mobility need. Cells [15] proposes a smartphone virtualization solution so that multiple virtual smartphones can run simultaneously on the same physical smartphone in an isolated, secure manner.

Unlike the virtualization techniques mentioned in section 3.1, Cells leverages a lightweight OS-level virtualization by the utilization of namespace. Linux namespace is being used by OpenVZ [9] and LXC [10]. A set of processes can be grouped into the same namespace. Each Linux namespace has PID namespace isolation, network namespace isolation, UTS namespace isolation, mount namespace isolation and IPC namespace isolation.

Cells observes that smartphones display only a single application at a time, and introduces a usage model which has one foreground Virtual Phone (VP) that is displayed and multiple background VPs that are not displayed at any given time. The foreground VP is always given direct access to hardware devices while the background VPs are given shared access to hardware devices when the foreground VP does not require exclusive access. Cells provides novel kernel-level and user-level device namespace mechanisms to efficiently multiplex hardware devices across multiple VPs. Therefore, untrusted application inside personal VP (namespace) will not be able to compromise the trusted application inside enterprise VP.

While Cells aims to embrace the emerging Bring-Your-Own-Device (BYOD) paradigm, each VP is treated equally and the isolation is achieved at the coarse-grained VP boundary. Unlike Cells, AirBag [36]'s objective is to boost the smartphone's defense capability against the malware infection by isolating the untrusted application in the AirBag environment. By dynamically creating an isolated runtime environment with its own dedicated namespace and virtualized system resources, AirBag is able to protect the OS from the malicious untrusted applications, e.g., an Android game repacked with the malware. AirBag creates and decouples an Application Isolation Runtime (AIR) from the native Android runtime, which contains Java & Native Libraries, Application framework (e.g., SurfaceFlinger service) and Dalvik virtual machine. AIR does not need to be trusted as it might be potentially compromised by untrusted application. AirBag multiplexes hardware resources between the AIR and native runtime by either creating a second resource (e.g., memory buffer) or creating a proxy between runtimes and hardware to mediate access from different runtimes.

4. Two-Way Protection

We discussed the protection of secure application from untrusted OS (e.g., TrustVisor [31]) in section 2 and the protection of benign OS from untrusted application (e.g., Native Client [37]) in section 3. In this section, we discuss the removing of trust between the application and operating system. Within my knowledge, currently, MiniBox [27] is the first and the only attempt toward a practical two-way sandbox for x86 native applications by combining TrustVisor and Native Client.

Platform-as-a-Service (PaaS) is one of the most widely commercialized forms of cloud computing. According to Google, in 2012, there were 1M active applications running on Google App Engine [11], where untrusted applications are sent by customers. Therefore, it is critical to protect the cloud platform from the untrusted applications. Besides cloud provider such as Google, security on PaaS is also a concern for cloud customers. People should rethink the security model of PaaS cloud computing because a two-way sandbox is desired.

Although it seems promising to combine a one-way sandbox (e.g., TrustVisor) and a two-way memory isolation mechanism (e.g., Native Client) to establish two-way protection, there are many challenges. First, a deliberate system design is required. Second, the interface between software modules for OS protection and the application should be minimized and secure. Finally, the design of TrustVisor doesn't support Iago attack prevention. The final system design should be able to protect applications against Iago attack.

MiniBox [27] combines the one-way sandbox for x86 native code and hypervisor-based two-way memory isolation. As in figure 3, the sandbox is split into service runtime modules and OS protection modules. The service runtime is included in the isolated memory space with the application together to support application execution. The original disassembly based sandbox is not required anymore because the hypervisor not only isolates the application from OS, but also isolates the OS from the application. To prevent Iago attacks, the system calls are divided into sensitive calls and non-sensitive calls. All sensitive calls are handled directly by a LibOS [33], which the application trusts, residing at

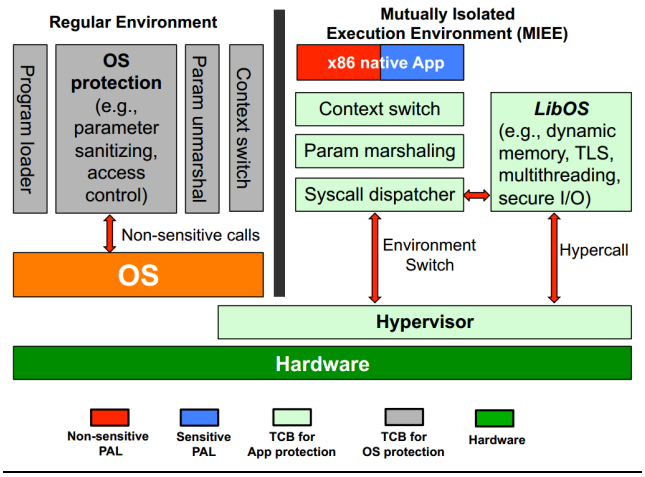

Figure 3. MiniBox Architecture

the Mutually Isolated Execution Environment (MIEE). Non-sensitive calls will be forwarded to the Regular Environment (RE). The OS in RE handles the non-sensitive calls mediated by the OS protection module in RE.

5. Discussion

5.1 More Dimensions

In this paper, we discuss prior works along just one dimension, the trust between the application and OS. In this dimension, we divide prior works into ”App does not trust OS”, ”OS does not trust App” and ”Mutually Mistrusting”. There are also other dimensions to discuss the prior works. For instance, some works [18, 19, 23, 25, 27, 29–32, 35, 37] are proposed for PC and some works [15, 16, 26, 28, 34, 36] are proposed for smartphone. While some works are proposed for a local PC or smartphone, some works are proposed for cloud environment, such as PaaS [25, 27]. Flicker and AirBag only work for single application, others such as TrustVisor, Cells and MiniBox can support many applications from different users simultaneously. Some works are designed to support the protection and isolation of just pieces of application logic (PAL), while Overshadow and AirBag’s protection mechanisms are in the granularity of the whole application. Overshadow does not require the modification to the source code of the application while InkTag requires the user to change the way of programming.

5.2 Limitations

Although many prior works have already solved the problems with a variety of mechanisms, there still exist some limitations. While MiniBox [27] is the first known attempt to remove the trust between the application and OS on PaaS, it supports only a single guest OS at this time. Besides, There is no two-way protection on Android. Android makes the problem more complex. Unlike Linux, where one task is implemented as a single application, the task on Android is usually accomplished by a set of applications together. The compromised Android runtime is able to infect the application in a way that is similar to Iago attack. It is expected that the Android application cannot be compromised even its underpinning runtime is malicious. We hope to investigate how Linux kernel helps verify the behavior of a compromised Android runtime.

The hypervisor-based isolation solutions, including TrustVisor, InkTag and MiniBox, cause overhead in context switch. The VMFUNC instruction released on the latest Intel 4th Generation Processor enables the software in guest OS to switch the hardware Extended Page Table (EPT) without the VM exit. Since VM exit is one of primary reasons for performance overhead of VM, we hope the investigation on how to perform secure environment switch using the VMFUNC instruction will improve the performance of hypervisor based solution.

6. Future Work

Most prior works on PC are based on x86 architecture, e.g., Intel and AMD. x86-based servers are not energy efficient. To handle hundreds of millions of users and their associated transactions, companies such as Amazon, Facebook, and Google run immense data centers with until-recently unimaginable computation and storage capacities. As online services become pervasive, projections indicate that electricity consumed in global data centers worldwide in 2010 is more than 200B KWh, between 1.1% and 1.5% of worldwide electricity use [24]. Three years ago, Google announced that their facilities have a continuous electricity usage equivalent to powering 200,000 homes [22].Therefore, it is promising to replace x86 with ARM architecture, which is more energy efficient, to build the next generation of servers in the cloud. There are already many ARM development boards published, such as Raspberry Pi, BeagleBone and Cubitruck. Recently, AMD announces plans to sample 64-bit ARM Opteron processors [12].

In the future, we want to explore the security problems on ARM architecture. Although prior works such as TLR [34], VeriUI [28] and TrustUI [26] also use security features of ARM, they are designed especially for smartphones. We propose the first solution to execute the secure PAL for ARM architecture either on a single server or in the cloud environment. Virtualization can effectively achieve the mutually memory isolation between application and OS. However, currently the hardware virtualization extension is not supported by all ARM CPUs. For instance, the ARM Cortex-A8 Processor does not support hardware virtualization.

Our proposal, as in Figure 4, leverages the TrustZone, which is supported by most ARM CPUs, to isolate the secure PAL in the secure world. The regular OS is running in the normal world while the secure PAL is executed in the secure world. Unlike virtualization which can create more than one isolated environments, there is only one secure world with TrustZone. To prevent the secure PAL of one

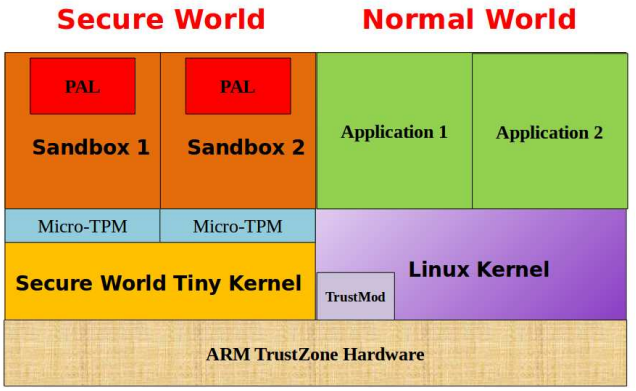

Figure 4. Secure execution of PAL on ARM Architecture

application from compromising the PAL of another, all PALs are sandboxed in the secure world. We will use TrustZone to emulate the secure boot, late launch and TPM operations. As ARM boards usually have limited resources, the secure world tiny kernel will not be loaded into memory unless the execution of PAL is registered and triggered. To prevent Iago attack, we divide the system calls into sensitive calls and non-sensitive calls. All sensitive calls, which can be used by malicious OS to mount the Iago attack, will be handled directly by the tiny kernel in secure world. Non-sensitive calls will be redirected to the untrusted OS in normal world. As the tiny kernel is only responsible for supporting the Micro-TPM, memory management and handling sensitive system calls, the TCB is small.

7. Conclusion

The protection of secure application from malicious OS and the sandbox of untrusted application from benign OS on the PC are two relatively mature research topics that have made substantial advances over the last decade. However, how to remove the trust between the application and OS on both PC and smartphone is still not fully explored. Many of prior works have limitations, e.g., the environment switch generates nontrivial performance overhead. In this paper, we discuss the evolution of prior works on the three problems. More future works are expected especially for smartphone and ARM-based server/cloud. In the future, the design of cloud computing framework should balance security, performance, cost and mobility.

References

[1] `http://www.gartner.com/newsroom/id/2647517`.

[2] `http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011`.

[3] `http://www.trustedcomputinggroup.org`.

[4] `http://www.arm.com/products/processors/technologies/trustzone/index.php`.

[5] `http://www.xenproject.org`.

[6] `http://www.linux-kvm.org`.

[7] `http://wiki.qemu.org`.

[8] `http://www.vmware.com`.

[9] `http://openvz.org`.

[10] `http://linuxcontainers.org`.

[11] `http://appengine.google.com`.

[12] AMD announces plans to sample 64-bit ARM Opteron. `http://goo.gl/au7BlB`.

[13] Security Alert: New Android Malware DKFBootKit Moves Towards The First Android BootKit. `http://www.csc.ncsu.edu/faculty/jiang/DKFBootKit`.

[14] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[15] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[16] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[17] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[18] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[19] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[20] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[21] Christoffer Dall and Jason Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[22] James Glanz. Google Details, and Defends, Its Use of Electricity. The New York Times, online at `http://www.nytimes.com/2011/09/09/technology/`

google-details-and-defends-its-use-of-electricity.html?_r=0.

[23] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[24] Jonathan G. Koomey. My new study of data center electricity use in 2010. www.koomey.com/post/8323374335, 2011.

[25] Johannes Krude and Ulrike Meyer. A versatile code execution isolation framework with security first. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop (CCSW)*.

[26] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Bingyu Zang, Cheng-Kang Chu, and Tieyan Li. Building trusted path on untrusted device drivers for mobile devices. In *4th Asia-Pacific Workshop on Systems (APSys)*, 2014.

[27] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC, 2014.

[28] Dongtao Liu and Landon P. Cox. Veriui: Attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2014.

[29] Lorenzo Martignoni, Pongsin Poosankam, Matei Zaharia, Jun Han, Stephen McCamant, Dawn Song, Vern Paxson, Adrian Perrig, Scott Shenker, and Ion Stoica. Cloud terminal: Secure access to sensitive applications from untrusted systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*.

[30] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium*, 2006.

[31] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*.

[32] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*.

[33] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[34] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[35] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[36] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. Airbag: Boosting smartphone resistance to malware infection. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[37] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*.