CAFE: A Virtualization-Based Approach to Protecting Sensitive Cloud Application Logic Confidentiality

Sungjin Park, Chung Hwan Kim, Junghwan Rhee, *Member, IEEE*, Jong-Jin Won, Taisook Han, *Member, IEEE*, and Dongyan Xu

Abstract—Cloud application marketplaces of modern cloud infrastructures offer a new software deployment model, integrated with the cloud environment in its configuration and policies. However, similar to traditional software distribution which has been suffering from software piracy and reverse engineering, cloud marketplaces face the same challenges that can deter the success of the evolving ecosystem of cloud software. We present a novel system named CAFE for cloud infrastructures where sensitive software logic can be executed with high secrecy protected from any piracy or reverse engineering attempts in a virtual machine even when its operating system kernel is compromised. The key mechanism is the end-to-end framework for the execution of applications, which consists of the secure encryption and distribution of confidential application binary files, and the runtime techniques to load, decrypt, and protect the program logic by isolating them from tenant virtual machines based on hypervisor-level techniques. We evaluate applications in several software categories which are commonly offered in cloud marketplaces showing that strong confidential execution can be provided with only marginal changes (around 100-220 lines of code) and minimal performance overhead. The results demonstrate the effectiveness and practicality of CAFE in cloud marketplaces.

Index Terms—Cloud computing marketplace, secure execution environment, code confidentiality protection

1 Introduction

Cloud computing infrastructures are becoming increasingly popular and mature. Gartner estimated the size of public cloud service market to grow to $131 billion by 2017 from $111 billion in 2012 [42]. As the technologies for cloud infrastructures have become mature, there is an increasing demand for software services especially in Infrastructure-as-a-Service (IaaS) clouds, where computers (physical or virtual) are provided to tenants with full flexibility. It is, however, difficult for cloud providers to fulfill all of the diverse needs of software that are continuously increasing.

Consequently, major cloud computing services (such as Amazon Web Services (AWS), IBM Cloud, and Microsoft Azure) operate marketplaces where application developers can upload and retail software, and cloud users can purchase the software that they need.

- *S. Park and J.-J. Won are with the System Research Division, Attached Institute of ETRI, Daejeon 34129, Korea. E-mail: {taiji, wonjj}@nsr.re.kr.*
- *C.H. Kim is with the Computer Security Department, NEC Labs America, Princeton, NJ 08540. E-mail: chungkim@nec-labs.com.*
- *J. Rhee is with the Autonomic Management NEC Labs America, Princeton, NJ 08540. E-mail: rhee@nec-labs.com.*
- *T. Han is with the School of Computing, Korea Advanced Institute of Science and Technology, Daejeon 34141, Korea. E-mail: han@cs.kaist.ac.kr.*
- *D. Xu is with the Department of Computer Sciences, Purdue University, West Lafayette, IN 47907. E-mail: dxu@cs.purdue.edu.*

Manuscript received 5 Apr. 2016; revised 9 Mar. 2018; accepted 16 Mar. 2018. Date of publication 22 Mar. 2018; date of current version 9 July 2020. (Corresponding author: Sungjin Park.)
Digital Object Identifier no. 10.1109/TDSC.2018.2817545

The ecosystem of cloud marketplaces and services in general involves three parties: cloud users, application developers, and cloud providers. Cloud users seek and purchase the cloud applications suitable to their needs in terms of functionality, price, the easiness of management, etc. Compared to traditional software that requires installation and management specific to each user (e.g., desktop applications), cloud applications are optimized to run on a cloud platform utilizing various services delivered from the cloud provider.

Fig. 1 illustrates how AWS Marketplace works as an example of a cloud marketplace. AWS application developers submit their packages to the marketplace after placing program binaries and dependent components in a disk image. Cloud users search for the software that meets their needs in the marketplace and purchase them. When the cloud users create a virtual machine (VM), they are prompted with a list of the disk images that include the purchased applications. The selected disk image is then written to the virtual disk of the VM, so the application can be used by the cloud users. The applications can be easily deployed using VM disk images (a.k.a., VM images) without tedious installation procedures that traditional applications require.

While this new form of distribution simplifies the deployment of software, one of the key problems in software distribution still remains in cloud marketplaces: *the deployed software faces the risk of piracy and reverse engineering*, similar to what conventionally distributed software is facing. Because cloud users typically own an entire VM with all privileged permissions given, technically they have no

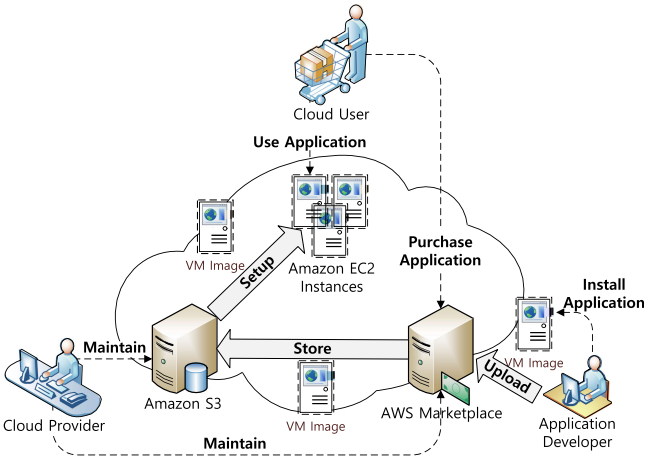

Fig. 1. An example of cloud application marketplace model: Amazon Web Services (AWS).

restriction on the inspection and replication of the applications installed in the VM.

In addition to the program logic of individual software, each cloud provider may have its own value-added services, such as improved security or performance optimization. If a competing cloud provider acquires the binary code of such services, they should be able to understand their logic leveraging reverse engineering techniques and can thus create similar services. Furthermore, cloud users may be able to copy and use the binary code in another cloud infrastructure or in his own machine without major effort.

Several previous work attempt to solve this problem via *code obfuscation*, *memory protection*, *secure RPC*, and *DRM*, but it has not been fully solved.

Binary code obfuscation makes reverse engineering more difficult by transforming binary code while its semantics are preserved [21], [30], [32]. However, it only impedes reverse engineering and cannot prevent the theft of software logic in the long term. Once the binary code is in the hands of an attacker, many approaches can be applied to understand or to reuse the logic [53].

Existing approaches [20], [33], [34] leveraging virtualization-based memory protection can provide a certain level of confidentiality to software. They are, however, designed to protect *partial* code confidentiality or do not address necessary issues in a practical cloud marketplace setting where code confidentiality must be fully protected throughout the entire life span of software after its submission.

Secure remote procedure call (RPC) [3] can be used to protect the confidentiality of the service logic executed in a remote server. A client invokes the execution of a server procedure by sending the input parameters and it receives the output data from the server through a typically encrypted session channel. Yet, the security of the mechanism relies on middleware, operating system (OS) that may be compromised and its performance often suffers from the fact that input/output data must be transmitted at every request. Furthermore, another concern is the possibility of vulnerability or software bugs due to large trusted computing base (TCB).

Many software vendors use digital right management (DRM) [12] tools to make illegal usage difficult. However, such attempts appear to be unsuccessful as in reality the pirated software circumvents the copyright protection [48]. In DRM, typically a cryptographic key stored in memory is used to decrypt the encrypted binary code. Thus, if the attacker figures out the location of the key in memory, it is possible to extract the plain binary code. Moreover, even if the decryption of the binary file fails, the decrypted binary code must be present in memory while the code is being executed due to the stored-program computer architecture. If the attacker has the privilege to access the program's memory, he can acquire the program image using a memory dump.

As a new alternative, we propose a system named CAFE[1] to address these challenges. CAFE provides a cloud application execution environment with code confidentiality so that it can protect sensitive cloud application logic from any piracy or reverse engineering attempts performed by cloud users, even in the case that the guest OS of the cloud user VM is compromised.

The key idea of CAFE is *the end-to-end execution environment of cloud applications that protects the confidential logic in the applications in their entire lifetime*: license verification, binary code transmission, loading and execution.

CAFE works in the following way. First, developers create software or port existing software in two groups of program binary files: a group that can be open to cloud users and the other that contains *confidential logic* that needs to be protected. We name the binary files of the former *public binaries* and the latter *secret binaries*. The public binaries are submitted in the form of a VM image that also contains other files related to the application (e.g., configuration files) and the VM environment but does not contain the secret binaries. A cloud user may be able to copy the files or extract the in-memory images of the public binaries as in the existing cloud application distribution.

In contrast, the secret binaries are submitted in the form of separate files and are automatically managed by the cloud providers with protection. When the application is run in the user VM, the secret binaries are fetched on demand at runtime for execution via a secure deployment protocol by the hypervisor. The hypervisor securely loads the secret binaries through a cryptographically protected channel after the authentication of the VM. The end-to-end framework ensures that the sensitive logic is completely isolated from cloud users at all times. Throughout the whole lifetime of the VM, the binary and runtime states of the sensitive logic stay confidential and are strictly protected from the entire guest OS in the VM by the hypervisor.

Contributions. The contributions of this paper are as follows.

- *A secure license verification and transmission layer for confidential deployment of secret binaries:* To support the confidential execution of cloud application logic in real-world marketplaces, a secure authentication and transmission layer is necessary for the management of secret binaries that are requested by a large number of user VMs in the cloud infrastructure for confidential execution. We propose a new secure layer for application license verification and secret binary transmission to enable the authentication and distribution of security sensitive program logic in a confidential way—which is required for a practical marketplace environment.

1. It stands for "Cloud Application Function Enclaving."

- *A guest-transparent, hypervisor-level runtime environment:* The confidential execution of sensitive application logic should not rely on the in-guest mechanisms for constructing a runtime environment, such as binary loading and relocation, because the guest OS cannot be trusted. We provide a novel loading and relocation mechanism based on hypervisor-level techniques for confidential execution.
- *Application and evaluation of CAFE in cloud marketplaces:* CAFE is applied to various cloud applications which are available or similar to the ones in marketplaces. The evaluation results show the effectiveness of CAFE in the provision of complete secrecy for sensitive application logic in cloud infrastructures.

This paper is structured as follows: Section 2 presents the definition of our problem. The design of CAFE is presented in Section 3. Section 4 describes the implementation details, and the evaluation of CAFE is presented in Section 5. Section 6 discusses possible attack scenarios. Related work is discussed in Section 7, and Section 8 concludes this paper.

2 Problem Definition

2.1 Goals

Our goal is to provide a secure execution environment that provides a confidentiality service to sensitive program logic in cloud applications, so the applications released to cloud marketplaces are protected from malicious cloud users who attempt to commit software thefts. More specifically, this goal can be presented as the following two objectives:

Secure Execution of Sensitive Application Logic Confidential to User VMs. In a typical cloud infrastructure, cloud users have the privilege to perform any operation in the user VM. For instance, a cloud user is free to access the binary image of the OS kernel that his VM boots up with. With this capability in mind, providing an end-to-end environment for confidential execution is challenging. Program execution includes multiple steps such as delivery of a binary, loading it into memory to construct a runtime environment, and scheduling it for execution. If a cloud user with administrative privilege obtains the program image, for example, from the memory space of the process or simply from binary files at any point of the aforementioned steps, it may lead to the compromise of confidential execution of the program.

Scalable and Practical Distribution of Secret Binaries for Cloud Marketplaces. Towards a practical cloud marketplace system, the distribution and deployment of application binaries are important. Today's cloud infrastructures manage about hundred thousands of VMs [39] running various kinds of cloud applications. In order to support such a diverse set of applications, it is essential to have a secure and scalable environment for application binary distribution. For the confidential execution of binary code, its content must be delivered and deployed with confidentiality. Thus, the framework for application binary distribution must ensure that the content of the binary remains confidential end-to-end from its submission to its execution.

2.2 Adversary Model

We present our adversary model based on a reasonable cloud environment in modern systems. The main goal of an adversary would be to obtain the content (in any form) of program binaries that are protected by our system. We note that the guest OSes running in cloud user VMs are *untrusted*, which means that an attacker can execute arbitrary executable code at any privilege. That is, an attacker can compromise all software including kernel, drivers, libraries, and applications in the user and kernel mode running in the user VM. More specifically, an attacker can attempt to obtain and reverse-engineer the binary codes containing sensitive application logic using the following methods.

Access to File System. As described in Section 1, applications are distributed to user VMs as a set of files written to the VM's disk image. The most obvious way to obtain the program is to access the files inside the file system. An attacker can acquire them using a file I/O tool or via file system APIs such as `fread`. Based on our adversary model, the attacker has privilege to mount and access any file system. Therefore, once the program is stored in a file system in a plain form, he should be able to obtain it.

Access to Runtime Process Memory. A more elevated attack to obtain the program binary is to capture the runtime states of the program. We consider that the attacker can access the memory of application processes. For instance, he can attach a debugging tool to a target process or inject malicious code into its memory. Using such methods, the attacker can obtain the binary code from the runtime memory of the target processes.

Access to Network. Another attack method to obtain the binary would be using the network layer. We assume that the attacker can eavesdrop, modify, inject, and block network traffic between cloud provider servers and cloud user VMs. For example, if the cloud provider transfers a binary file to the user VMs over the network in a plain form, then the attacker may be able to obtain the binary from network packets.

The aforementioned attack surfaces are based on interactions between a tenant and a VM, and possible attack scenarios that are related. We further discuss these attack scenarios in Section 6.

3 Design

3.1 System Overview

To support confidential execution of various kinds of applications in a practical cloud marketplace setting, the confidentiality of sensitive logic should be systematically maintained in the entire work-flow from the development of programs to the delivery to the cloud users and their execution. CAFE achieves the aforementioned goals using *an end-to-end framework for the confidential execution of cloud applications* by using hypervisor-level techniques. This is one of the key novelties of this paper compared to previous work which only focus on a local view of protection [20], [21], [30], [32], [33], [34]. Fig. 2 gives an overview of the CAFE architecture in the three stages of the distribution and the use of a cloud application.

Application Development and Submission. As mentioned in Section 1, cloud application developers build their program code into two separate groups to be supported by CAFE: the public binary and secret binary groups. In our model, application developers have the responsibility to determine which part of application logic needs confidentiality. The application is annotated to use the user level APIs that CAFE provides,

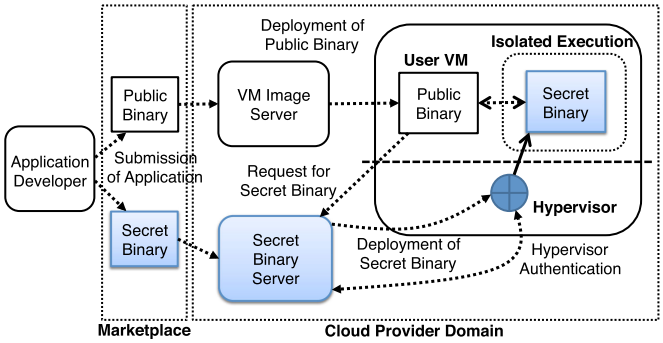

Fig. 2. Overview of the end-to-end confidential execution architecture of CAFE.

which are essentially a set of hypercalls that request the hypervisor to load, unload, and execute a secret binary. The public binaries are packaged in a VM image along with other binary files on which the application depends. When the application is submitted to the marketplace, the public binaries and the secret binary files are submitted separately. The public binaries are linked at build time to a library, called the *CAFE library*. The CAFE library provides a communication layer for the hypervisor and the hypercall interfaces.

Upon submission, the public VM image is transferred to the VM image server that stores and manages VM images as in existing cloud infrastructures. In contrast, the secret binary files are stored in the *secret binary server (SBS)* which works as a secure storage for sensitive application logic. Both the VM image server and the SBS are a part of the cloud provider domain linked to user VMs with a dedicated high bandwidth connection.

Purchase and Deployment of Applications. Cloud users use the cloud marketplace to find the applications for their needs. Once an application is purchased by a cloud user in the marketplace, the cloud provider lets the cloud user create a VM using the corresponding VM image that includes the public binaries of the purchased application. On the other hand, the secret binaries stored in the SBS are not delivered to the user VM when the VM is created. Instead, they are transparently delivered to the hypervisor through a secure channel when the binaries are requested for use.

Execution of an Application. When the cloud user runs the purchased application in the VM, the application requests the hypervisor to load the secret binaries that it needs for execution via a hypercall. The hypervisor, in turn, communicates with the SBS to prove the authenticity of itself and the user's license for the application. More specifically, the hypervisor and the SBS exchange the session key to establish a secure channel, and the SBS attests the integrity of the hypervisor leveraging the Trusted Platform Module (TPM) to ensure that it is communicating with the genuine hypervisor. After that, the SBS transfers the secret binaries after encrypting them using the session key shared with the hypervisor. Upon receiving them, the hypervisor decrypts and loads the secret binaries in a secure runtime environment which is isolated from the user VM.

Packet transmission is handled by the CAFE library to minimize the TCB of the hypervisor. This library transmits and receives all packets between the hypervisor and the SBS instead of the hypervisor. The packets are moved in and out of the hypervisor via hypercall.

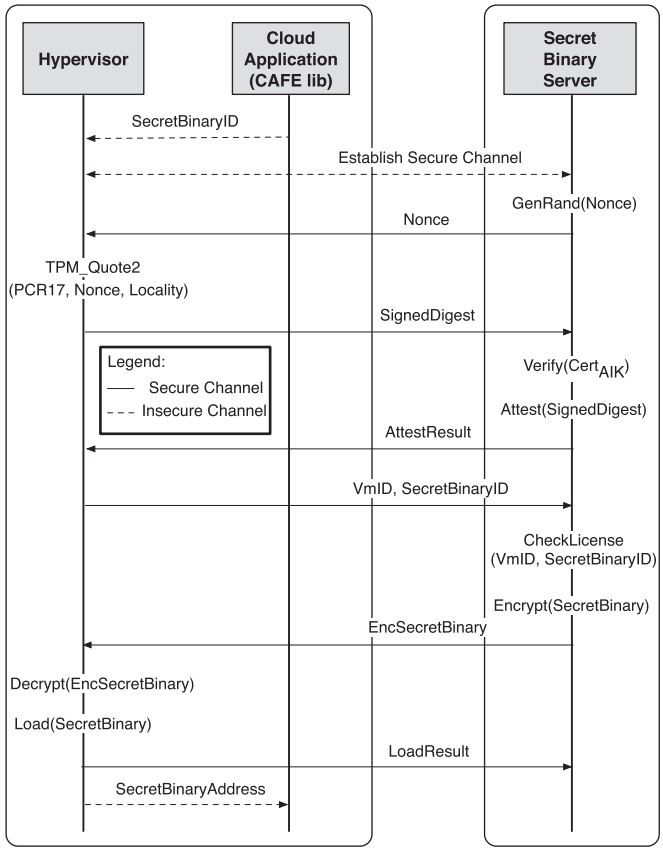

Fig. 3. Secret binary deployment protocol.

Designation of Sensitive Code. Security sensitive code has the key logic of an application requiring a high degree of protection. Typically this is a small portion of the entire application code. Therefore, the cost to achieve confidentiality is generally amortized in the overall performance of programs. Our evaluation in Section 5 will confirm that this high secrecy property can be achieved with minimal overhead.

In the following sections, we present the details of each component involved in the execution of secret binaries: (1) the secure deployment of secret binaries, (2) proving the trustworthiness of the hypervisor, (3) the secure loading of secret binaries, and (4) the runtime protection of secret binary codes.

3.2 Life-Cycle of a Secret Binary

In this section, we present our deployment protocol of secret binaries, designed to defend against attacks based on the adversarial model in Section 2.2.

Fig. 3 depicts our protocol for the deployment of secret binaries. The protocol consists of the following steps.

1) A cloud application developer implements public binaries and secret binaries separately and submits them to the cloud marketplace.
2) A cloud user purchases a cloud application in the cloud marketplace and gets a VM including the purchased cloud application.
3) The purchased cloud application requests the use of the secret binary from the hypervisor using the CAFE library.

4) The SBS and the hypervisor exchange a session key to establish a secure channel.
5) The SBS verifies the trustworthiness of the hypervisor via remote attestation.
6) The hypervisor requests the SBS to send the secret binary requested by the cloud application.
7) The SBS verifies whether the cloud user associated with the VM has a license to use the requested application.
8) The SBS transmits the secret binary image to the hypervisor if the user VM has a valid license for the application.
9) The hypervisor decrypts and loads the secret binary in the confidential execution environment isolated from the user VM.
10) The hypervisor responds to the application with the address of the secret binary loaded in the confidential execution environment.
11) When the public binary calls secret functions, the function calls trap to the hypervisor. The hypervisor handles the traps and runs the secret functions.
12) Prior to the termination of the purchased application, the public binary initiates unloading of the secret binary via the CAFE library and the hypervisor withdraws the protected memory of the secret binary.

Creation of Secret Binaries. Cloud application developers determine which part of application logic needs to be confidential and separated into two groups, the public binary and secret binary groups.

The public binary is a part of the cloud application which can be open to cloud tenants. When the cloud application developers build the public binary, the CAFE library is linked to the application. For the secure transmission of the secret binary, the hypervisor and the SBS establish a secure channel over the network. Since the hypervisor does not contain the functionality of the packet transmission to minimize TCB, the CAFE library performs it on behalf of the hypervisor. The packets are moved into and out of the hypervisor via hypercall. The CAFE library can be viewed as a thin network layer on top of the hypervisor—it does not contain any security-critical logic.

The secret binary is a set of code and data that composes the security-sensitive functionalities. CAFE generates the secret binary in form of the shared library because the secret binary is loaded at execution time in a user VM.

Secure Channel Establishment. When an application requests a secret binary from the hypervisor, a secure channel is established between the hypervisor and the SBS for tamper-resistant communication. The secure channel protects the communication between the hypervisor and the SBS against network-based attacks. The details of the secure channel establishment are as follows.

Among several candidate key exchange algorithms (i.e., Diffie-Hellman and RSA), we choose a variant of Transport Layer Security (TLS) [4] as our handshake protocol and RSA as our key exchange algorithm. TLS allows two parties of the secure channel (i.e., the SBS and the hypervisor) to authenticate each other with the other's certificate. The certificate authority (CA) guarantees the certificates of both sides; thereby, they can securely authenticate each other. Unlike the standard handshake protocol of TLS, CAFE leverages the TPM to generate an RSA key pair of the hypervisor and a *pre-secret* which is used to derive shared secrets such as an encryption key, an initial vector (IV), and a HMAC key. With the use of the TPM, we provide a level of security higher than the standard TLS.

CAFE generates a RSA key pair inside the TPM and wraps it with the TPM's Storage Root Key (SRK). The SRK is a unique, non-migratable 2048-bit RSA key and is guaranteed to always be present in the TPM. Due to these features, a key wrapped by the SRK can only be used in the machine on which the same TPM is placed. Therefore, even in the case that the attacker acquires a wrapped RSA key pair, he cannot unwrap it without the same TPM used to generate and wrap it.

Remote Attestation of the Hypervisor. After establishing a secure channel, the SBS generates a nonce (`Nonce`), which is a true random number generated by the TPM, and sends it to the hypervisor. The nonce is used as a parameter of the `TPM_Quote2` operation, a TPM operation used for integrity measurement [23]. The hypervisor performs `TPM_Quote2` and transfers the resulted digest (`SignedDigest`) to the SBS. The SBS verifies the integrity of the hypervisor by matching the received digest with the certificate ($\texttt{Cert}_{AIK}$) from the privacy CA. The detailed process of the remote attestation is presented in Section 3.3.

Verification of Application Licenses. Upon successful attestation of the hypervisor, the cloud application sends the VM ID and the secret binary ID (respectively `VmID` and `SecretBinaryID` in Fig. 3) to the SBS requesting the transmission of the encrypted secret binary image. A VM ID is the unique identifier of a VM managed internally by the cloud infrastructure. A secret binary ID is the unique identifier of a secret binary determined upon the submission of the application to the cloud infrastructure, and it is known to both the SBS and the application that uses the secret binary. The cloud infrastructure maintains the association between a VM ID and a user, and what licenses the user has (e.g., types of VM, cloud applications, etc.) for billing purposes. Based on this information, the SBS determines whether the user associated with the VM ID has a valid license for the application of the secret binary ID. If the license is valid, the SBS proceeds to the transmission, otherwise it refuses the request.

Transmission of Secret Binaries. The hypervisor and the SBS share the same encryption key, IV, and HMAC key after establishing the secure channel. The SBS encrypts the relocation, the secret code, and the secret data sections in a secret binary with the encryption key and the IV using the HMAC message authentication of the encrypted secret binaries with the shared HMAC key. The SBS and the hypervisor establish a new secure channel with the derived shared secrets at every request of a secret binary, and the secret binary is encrypted using the session key. Therefore, the deployment protocol can resist brute-force attacks on the secure channel and the secret binary encryption.

Loading of Secret Binaries. The encrypted secret binaries are stored in the user VM's disk after the transmission. The original contents of the secret binaries remain encrypted in the user VM's disk throughout the all steps of the secret binary loading and execution. The confidentiality of the secret binaries in the disk are as strong as the encryption algorithm that the SBS uses. The hypervisor decrypts and

loads secret binaries in the confidential execution environment that is isolated from the user VM. The integrity of the secret binaries is checked using the shared HMAC key with the SBS before the loading. The details of the secret binary loading are described in Section 3.5. Finally, the result of the loading is sent back to the SBS, and the application receives the address of the secret binary.

Even though attackers manage to acquire encrypted secret binaries from the VM's file system, the encrypted secret binaries are cryptographically protected because the keys are stored in the memory where only the hypervisor can access.

Storing secret binaries from the SBS in the hypervisor's file system could be another option. This design would require high level functionalities such as a network stack for the communication with the SBS and a file system support to store secret binaries. This choice will lead to the increase of the code size of the hypervisor, which is the TCB of CAFE. Therefore, we did not pursue this direction in our approach.

Invocation of Secret Functions. After loading of secret binaries, secret functions are inaccessible from the guest VMs. The secret functions, however, must be able to be invoked by the public functions for the application to run despite the boundary between the two groups of binaries. The hypervisor performs the following steps to serve the secret function invocation.

1) Any secret function call from the public binaries traps into the hypervisor.
2) The hypervisor marshals input parameters from the calling public function into the secret function and calls the secret function.
3) The secret function serves its own security-sensitive features.
4) After executing the secret function call, the hypervisor unmarshals returned values from the secret function to the calling public function.
5) The calling public function obtains the returned values and is resumed.

Unloading of Secret Binaries. When the cloud user terminates the purchased cloud application, the public binaries initiate unloading of the secret binaries prior to its termination. At first, the hypervisor initializes the memory area where the secret binary is placed in order to protect the security-sensitive codes from the access to this memory area after unloading of the secret binaries. Next, the hypervisor merges the memory of the secret binary into the user VM and the purchased cloud application is terminated.

3.3 Verifying Trustworthiness of Hypervisor

Remote Attestation of the Hypervisor. Since the hypervisor is part of the trusted computing base of CAFE, the SBS must ensure that the hypervisor is trustworthy prior to sending the secret binaries. In order to perform the attestation of the hypervisor at startup time, we utilize the hardware technology, the Dynamic Root of Trust for Measurement (DRTM). Modern processors with the virtualization technology (e.g., AMD SVM and Intel TXT) and the TPM enabled support DRTM [11], [26]. DRTM provides a dynamic means to launch a hypervisor (or an OS) after checking the integrity of the code using the TPM. We leverage the DRTM technology to measure the hypervisor. More specifically, DRTM extends the PCR17 (i.e., a TPM register designated for DRTM) with the measurement value (i.e., the cryptographic hash) of the hypervisor when the hypervisor is launched. To securely transfer the measurement value to the verifier (in our case, the SBS), the TPM provides cryptographic operations, such as `TPM_Quote` and `TPM_Quote2`, designed to provide cryptographic reporting of the PCR values. These operations use a RSA private key to sign a measurement value that specifies the current value of a chosen PCR and externally supplied data such as the nonce generated by the SBS. CAFE leverages the `TPM_Quote2` with a specific locality (locality 2) preoccupied by the hypervisor at startup time. Any later attempts in the user VM that try to use the locality are trapped to and blocked by the hypervisor. The semantic of the `TPM_Quote2` operation is as follows:

$$Sign\,(\,SHA1\,(\,PCRs,\,nonce,\,locality\,),\,PR_{AIK}\,)$$

With the remote attestation using `TPM_Quote2`, we can guarantee that the SBS performs the session key exchange with the genuine hypervisor that has not been compromised.

3.4 Runtime Protection of Secret Binary

In order to ensure that the sensitive application logic is not exposed, it is important to protect the secret binary code throughout the entire stages including authentication, deployment, loading, and execution of the secret binary. To achieve this, the secret binary image must remain encrypted anytime in the user VM during deployment until it is loaded in an isolated memory for execution. More specifically, the confidentiality of the secret binary should be maintained while the code is being transferred over the network, on the disk, and in the memory of the user VM. We showed in Section 3.2 that the secret binary deployment protocol securely transfers secret binaries through the network in the cloud provider domain. In this section, we describe how CAFE protects secret binaries on disk and memory.

Protection of the Secret Binary on Disk. First, the secret binary file on disk is cryptographically protected by an encryption algorithm. Unlike existing work that obfuscates binary code to make reverse-engineering more difficult [21], [30], [32], CAFE aims to prevent the theft of sensitive application logic by completely encrypting the binary code using cryptography. The security of the binary content is determined by the strength of the encryption algorithm and the protection of the encryption key. Among various available encryption algorithms, we choose AES-256 in the CBC mode to encrypt and decrypt secret binaries. Any stronger cryptographic algorithm will improve the strength of the encryption making an attack more difficult. The SBS performs the encryption of the binary on demand when a request for a binary is received from an application. The session key obtained from the session channel establishment (Section 3.2) is used for encryption to prevent brute-force attempts that target the transmission of the secret binary. The application uses an API provided by CAFE to load the secret binary and the API is responsible for receiving the encrypted secret binary from the SBS and storing it on disk. The secret binary file remains encrypted until it is verified and loaded into an isolated memory maintained by the

hypervisor. Therefore, any attempts to reverse-engineer the decrypted content of the binary file on disk fail.

Protection of the Secret Binary in Memory. Once the secret binary is delivered to the user VM, the integrity of the secret binary is verified by the hypervisor using the HMAC key shared with the SBS. After that, the secret binary is loaded into the isolated memory for an execution. The memory isolation is implemented leveraging the hardware-assisted memory virtualization technology.

Several types of hardware-support for memory virtualization are readily available in modern commodity processors such as Intel Extended Page Tables (EPT) and AMD Rapid Virtualization Indexing (RVI) which can be used to implement CAFE. When any of these virtualization support is enabled, the memory management unit (MMU) accesses both the guest page tables and the nested page tables (NPTs) to translate the guest linear addresses to guest physical addresses, and the guest physical addresses to machine physical addresses respectively.

CAFE uses this technique to load the secret binary into the memory and create a secure execution environment isolated from the user VM. Before the secret binary is loaded, there is one set of NPTs that is maintained by the hypervisor to map all guest physical addresses to machine physical addresses. When the binary is loaded, the NPTs are split into two exclusive sets: the public NPTs and the secret NPTs. The public NPTs contain the page entries for all memory blocks used by the user VM except those used by the secret binary, whereas the secret NPTs contain the page entries for the secret binary only. The hypervisor ensures that the user VM uses the public NPTs while the public binary is running. Thus, any memory access to the secret binary during the execution of the public binary is blocked by the MMU. On the other hand, when the application executes the secret binary the hypervisor switches the public NPTs with the secret NPTs. The secret NPTs enforce that the secret binary can only access the memory that is exclusively assigned to it providing strong isolation between the secret binary and the user VM.

Previous work leveraging memory virtualization primarily focus on the runtime isolation of the program binary in memory [33], [34]. Compared to the existing work, CAFE ensures that the sensitive application logic is inaccessible from the user VM at any time. The binary code remains encrypted both on the disk and in the memory of the user VM, and the secure execution environment runs the decrypted code in complete isolation from the user VM.

3.5 Secure Hypervisor Loading of Secret Binary

A program's execution is performed through several operations such as allocation of system resources (e.g., memory pages, stack, heap), loading the program image into the memory, linking library code, and bootstrapping the program code. These operations are typically performed by the high privileged software layer in the operating system or system level libraries, to manage such resources. In our setting, the user VM including its operating system is untrusted. Therefore, CAFE has its own mechanism for such operations to ensure the confidentiality of the program throughout its execution. In this section, we describe how this execution environment is established.

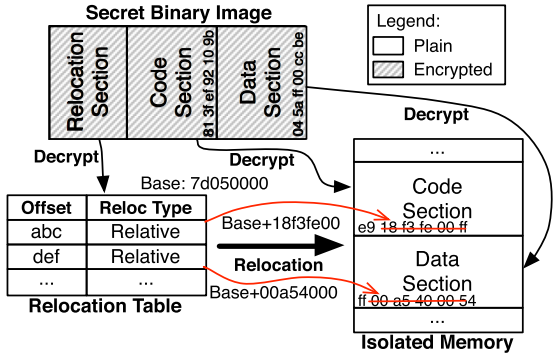

Fig. 4. Hypervisor loading of a secret binary.

Loading of an Executable Binary. Program code typically consists of multiple binary codes: the main executable and a number of shared libraries linked to the main executable. When a binary is loaded, its location in the virtual memory of the process is determined dynamically by the *loader*. In addition to the location of the binary image, the addresses of the symbols (e.g., exported global variables and functions) in the binary are also selected at runtime. Thus, any instructions of the program that refer to the symbols in the binary must be updated with the addresses determined at run time and this process is known as *relocation*. When the executable file is built, the compiler constructs a relocation table that includes the information necessary for the relocation. The information includes the locations of the instruction operands to be patched, the location of the symbols in the binary, and how the loader is expected to perform relocation (i.e., the relocation type). This table is used by the OS loader when the executable file is loaded or when the relocation is necessary in a lazy manner depending on the configuration.

Loading by the Hypervisor. In order to provide the confidential execution of an application, the secret binaries are encrypted and sent to a user VM via a secure channel in CAFE. When the application wants to use a secret binary, it requests the hypervisor to load the secret binary using the CAFE library. Then the hypervisor decrypts the secret binary into the memory isolated from the user VM. The challenge here is that the loader in the user OS does not have an access to the decrypted secret binary in the isolated memory; thus, it cannot perform relocation on it. On the other hand, if the secret binary is decrypted in the memory that the OS loader can access to, the untrusted OS can obtain the decrypted binary content during the loading process.

To solve this challenge, we use the hypervisor to perform the loading, the decryption, and the relocation of the secret binary. When the binary is loaded into the isolated memory, the hypervisor performs the decryption. After that it performs the relocation of the binary. Fig. 4 demonstrates how relocation is performed by the hypervisor. The relocation on the public binaries is performed by the OS loader as usual, but the relocation on the secret binaries is delegated from the loader to the hypervisor. When the application requests CAFE to load a secret binary, it locates the relocation section of the secret binary, and the encrypted relocation information is sent to the hypervisor via a hypercall. When the hypervisor receives the hypercall, it decrypts the relocation information. If decryption is successful, then it performs relocation on all sections of the secret binary including the

TABLE 1
Use Cases of Confidential Execution of Secret Cloud Application Binaries

Application category	Program name	Binary name	Code info			Runtime info		
			Protected code	$\|L\|$	$\|F\|$	$\|C\|$	$\|D\|$	$\|R\|$
Decision-making logic	NGINX	`nginx-access`	Access module	169	1	4	44	19
	Sendmail	`sendmail-filter`	Mail Filter (Milter)	106	1	52	52	559
	Apache HTTP Server	`httpd-firewall`	Web Application Firewall (WAF)	104	1	52	52	556
Cryptographic operations	Google Authenticator	`gauth-otp`	One-time passcode generation	102	1	8	44	16
	EncFS	`encfs-aria`	ARIA block encryption/decryption	220	3	24	48	346
	MariaDB (server)	`mariadb-aes`	AES tuple encryption/decryption	163	4	12	52	169
Data processing workload	MapReduce	`MapReduce-kmeans`	k-means clustering	173	1	12	44	380
	MPI	`mpi-mmult`	Matrix-matrix multiplication	149	1	4	44	20
	Hadoop	`hadoop-wcount`	Word counting	180	2	4	44	32

$|L|$: LoC added for porting, $|F|$: the number of exported secret functions, $|C|$: Code section size (KB), $|D|$: Data section size (KB), $|R|$: Relocation table size.

code and data sections. In addition to the code section, relocation is also performed on the data section because the symbols that hold memory addresses (e.g., global pointer variables) must also be updated in memory.

Position Independent Code. While relocation is the most common and the traditional way to ensure correct code and data references, there is another approach to resolve the issue, namely Position Independent Code (PIC) [24]. PIC solves the relocation problem using a Global Offset Table (GOT) and a Procedure Linkage Table (PLT). A GOT is a table where each entry contains the address of a target symbol in memory. The code instruction referring to a target symbol uses this table to obtain its dynamic address; thus, avoiding the need for relocation. PLT is similarly used for function addresses. However, GOT does not completely avoid relocation because the entries of GOT are still subject to relocation to contain the accurate symbol addresses determined at runtime. CAFE supports the loading and execution of both relocation-based and PIC-based executables.

3.6 Formal Verification of Protocol

To verify the safety of our protocol for the deployment of secret binaries, we use a formal verification tool, ProVerif [16] on our protocol algorithm which consists of the secure channel establishment, the remote attestation, and the transportation of the secret binaries. Since the secure channel establishment is based on a variant of TLS, we assume that the verification of the channel is performed by TLS. ProVerif can handle many cryptographic primitives like public key cryptography and an unbounded number of sessions of protocol. Based on the formal verification, we confirm that an attacker cannot acquire the secret binaries transmitted over the network.

4 Implementation

CAFE consists of three major components: the hypervisor, the SBS, and the user level APIs that the cloud applications use to send requests to the hypervisor.

First, the hypervisor is implemented on top of eXtensible and Modular Hypervisor Framework (XMHF) [49] which is used in several related work [33], [49], [50]. XMHF provides a general framework for building a DRTM-based hypervisor. We implement the authentication/verification layer that interacts with the SBS and the loading and unloading mechanism for secret binaries that involve the hypervisor level relocation.

Second, the SBS is a server that interacts with the hypervisor to receive the requests of secret binaries and send the encrypted binaries along with their authentication records. This component is implemented using OpenSSL to perform certificate verifications and cipher operations.

Third, the user level APIs are essentially a set of hypercalls that the cloud applications issue to request the hypervisor to load, unload, and execute secret binaries, as well as for input and output data marshaling.

We use two machines for our experiments for the hypervisor and the SBS. Both machines are equipped with an AMD Turion II P520 2.30 GHz processor, 4 GB RAM, and a 256 GB SSD, and run the 32-bit version of Ubuntu 12.04. The virtual machines and the SBS are connected to a 1 Gb/s network. We wrote 5,743 lines of code (LoC) which include all three components beyond the base systems that we used.

5 Evaluation

5.1 Use Cases

We demonstrate that various types of application code can be protected by CAFE using 9 applications grouped into three distinct categories (as listed in Table 1). The applications are selected based on their popularity in real-world usages and cloud marketplaces. Many of these applications are available in AWS Marketplaces at the time of writing this paper. We use the source code of these applications to slice out example sensitive code that is compiled into secret binaries. The chosen program logic may not be "confidential" in real world, but they are selected to simulate the developers' efforts to take the benefit of confidential execution of CAFE. In our evaluation, only 100-220 LoC are added for the separation of the secret binaries. Despite our use cases mainly address tenants applications, we would like to note that our techniques would be also applicable to cloud providers applications such as accounting logics and management functions of services.

This experiment shows the applicability of CAFE to various types of application software to verify that similar program logic can be executed confidentially. Table 1 shows the details of the applications that run on CAFE with confidential execution. In the first column, the table has three categories of applications in the type of protected program

TABLE 2
Runtime Characteristics of Secret Binaries

Binary name	$\|I\|$	$\|O\|$	Input scale factor (N)
`nginx-access`	$12N + 12$	4	Number of ALLOW/DENY rules in server configuration
`sendmail-filter`	N	4	Length of sender's email address
`httpd-firewall`	N	4	Length of requested URI string
`gauth-otp`	$5(N + 7)/8$	4	Length of authentication token (default: 90)
`encfs-aria`	N	N	Size of block to encrypt or decrypt (default: 256)
`mariadb-aes`	$N + 64$	N	Size of block to encrypt or decrypt (default: 16)
`MapReduce-kmeans`	$4N/M + 4M + 12$	$4N/M + 4$	Number of n-dimensional points, M: Number of clusters
`mpi-mmult`	$8N + 12$	$4N$	Number of elements in each of two-dimensional input matrices
`hadoop-wcount`	N	4	Size of text file that contains input words to count

Input parameter and output data sizes are on a per-function call basis. $|I|$: input parameter size (bytes), $|O|$: output data size (bytes).

logic: decision-making logic, cryptographic operations, and data processing workload. In the following columns, Table 1 shows the program information (program name, binary name), the description of the protected program logic (Protected Code), the number of lines of code added as porting attempts ($|L|$), the number of exported secret functions ($|F|$) and runtime characteristics of the secret binaries ($|C|$: code section size, $|D|$: data section size, $|R|$: relocation table size). Note the exported secret functions ($|F|$) mean the interface functions that are exported to the non-secret (public) binary code so that they can be called from outside. Within the secret binary there are many more finer-grained functions, but we mainly focus on the exported functions because they are more relevant to the evaluation. The characteristics of the applications in the three categories are as follows.

Decision-Making Logic. Application code in this category determines the behaviors of the application. For example, the access module, `nginx-access`, of NGINX decides whether the web server allows or denies an incoming web request based on the ALLOW/DENY list in the web server configuration. Another application, the mail filter module, `sendmail-filter`, of the sendmail server analyzes the content of an outgoing mail and decides whether to send out the mail or not. Specifically the ported code examines the header of an input mail and finds whether the sender's address is illegal using a regular expression. In addition, we port the Web Application Firewall (WAF) module, `httpd-firewall`, in the Apache HTTP Server. This module watches for HTTP requests and responses to allow or block them based on their contents. Particularly this module analyzes the GET parameters of a HTTP request using a regular expression to detect a potential SQL injection attack.

Cryptographic Operations. In general, the strength of cryptographic operations are determined by their keys and algorithms. However, if the OS is compromised, such operations are no longer safe because high privileged software can look into the runtime states of the cryptographic functions, cipher modes, or the algorithms of closed cryptographic functions. This can impact the confidentiality of cryptographic operations of cloud users as well as the cloud providers via techniques like reverse engineering. CAFE can make cryptographic code on cloud applications confidential to the entire user VMs. We show several use cases of well known cloud applications. Google Authenticator includes a time-based one-time passcode generator in the form of OpenSSH's Pluggable Authentication Module (PAM) for two-factor user authentication. We port the Google Authenticator PAM, `gauth-otp`, that protects the passcode generating code. Another example is EncFS which is a file system with the block-level encryption. We selected a particular version of EncFS, `encfs-aria`, that uses the ARIA cipher [6] for the file block encryption and decryption. The encryption and decryption algorithms along with the key initialization function are protected. MariaDB is a SQL server that aims to replace the MySQL DBMS with enhanced performance and increased productivity in the enterprise and the cloud. We port the AES encryption and decryption functions of MariaDB, `mariadb-aes`, to demonstrate that the confidentiality of such crypto logic can be protected.

Data Processing Workload. Some security sensitive code may involve intensive computation on a certain amount of data. We use three parallel data processing algorithms as examples (k-means clustering, matrix-matrix multiplication, and word counting), each running on a parallel computing framework to show the support of this category by CAFE. `MapReduce-kmeans` is an implementation of k-means clustering based on Pheonix [40], a shared-memory and C language based MapReduce framework. The protected code partitions n-dimensional integer points into a number of clusters. `mpi-mmult` is a matrix-matrix multiplication algorithm running on MPI (OpenMPI). It takes two two-dimensional arrays as inputs and produces another 2-D array as a result of the multiplication. Lastly, `hadoop-wcount` is a simple algorithm based on Apache Hadoop which analyzes an input text file and outputs the total number of distinct English words.

The number of lines of code added to the applications for conversion depends on the amount of the confidential code of the application. In our use cases, it ranges 100-220. Compared to the total LoC of the entire program, it is a small portion of the program (average 1.18 percent for the applications we evaluate).

The performance of a secret binary relies on the execution frequency of the secret functions and the amount of the input and output data marshaled each time when the code is executed. Table 2 shows the runtime characteristics of the secret binaries protected by CAFE. The sizes of the input data to the secret code ($|I|$) and the output data ($|O|$) from the secret code are mainly determined by the input scale factor (N), which depends on the size of input to the application, such as the configuration of the application and the input data that the application receives. The details of N is presented in the fourth column.

TABLE 3
Microbenchmark of Secret Binary Operations

Secret code size	4 KB	16 KB	32 KB	64 KB
Comm. with SBS	1818	1822	1839	1845
Secret binary loading	5.75	6.78	8.16	10.93
Total	1823.75	1828.78	1847.16	1855.93

(a) Secret binary loading overhead (ms) with varying code sizes.

Secret code size	4 KB	16 KB	32 KB	64 KB
Unloading time (ms)	8.21	9.37	10.88	14.06

(b) Secret binary unloading overhead (ms) with varying code sizes.

Number of relocation entries	0	50	100	500	1000
Relocation time (μs)	38	56	59	82	117

(c) Hypervisor-level relocation performance with varying relocation table sizes.

Input data size	0 KB	4 KB	8 KB	16 KB	32 KB
Marshaling time (μs)	1.0	17.6	35.0	73.4	147.6

(d) Parameter marshaling overhead with varying input data sizes.

5.2 Performance of Confidential Execution of Cloud Applications

Microbenchmark. First, we evaluate the overhead of the operations necessary to execute secret binaries in an isolated VM: secret binary loading (Table 3a), secret binary unloading (Table 3b), hypervisor-level relocation (Table 3c), and input data marshaling (Table 3d).

In order to show how the overhead scales with the size of input, we select four representative binary code sizes for loading and unloading and five different sizes of the relocation table for the performance of hypervisor-level relocation and marshaling.

The loading of a secret binary includes the communication between the hypervisor and the SBS through network for the delivery of a binary and the local procedures to load it in the user VM. The communication overhead depends on the bandwidth and traffic between the user VM and the SBS. Typically, rack mounted servers have multiple interface cards for management purposes and the delivery of secret binaries can be performed using a separate connection to minimize the impact from data traffic (e.g., network congestion). In our experiments, we use a Gigabit network for connection without significant background traffic assuming an out-of-the-band connection. The communication overhead dominates the overhead of local secret binary loading as shown in Table 3a. Unloading of the binaries also similarly scales with the size of binaries (Table 3b).

The performance of hypervisor-relocation is proportional to the size of the relocation table for the secret binary, the number of entries in the relocation table, as shown in Table 3c. The relocation of a secret binary is performed by the hypervisor after decrypting the binary content in the isolated memory.

Marshaling parameters to the secret binary code involves copying data from a memory region to another. The marshaled parameters are serialized and copied from the unprotected memory to the protected memory that the secret binary exclusively owns. The marshaling overhead is proportional to the size of the data as shown in Table 3d.

Macrobenchmark. Next, we present the overhead of the applications for confidential execution in CAFE (Table 4). We calculate the overhead by comparing the performance of the original version with the modified version with confidential protection of application logic. The overhead scales with the workload of protected code. We present the detailed information regarding the protected code in Section 5.1. In general, one major source of overhead is VM-level context switches that occur while the secret code is running and during marshaling for input and output data. Specifically, the overhead highly depends on the frequency of secret function calls and the size of the marshaled data for each secret function call, which is determined by application behavior. The complexity of the protected logic has a minor impact on the overhead.

We have evaluated several server programs by setting up the client for benchmarking workload in a separate physical machine in a local network (1 Gb/s). We note that the setup is conservative since cloud data centers are typically connected with a higher bandwidth (e.g., 10 Gb/s).

Table 4 presents our evaluation result and the input workload. To measure the overhead of `nginx-access` and `httpd-firewall`, we have the Apache Benchmark (ab) issue 10,000 transactions per trial with each transaction, which requests one page at a time to the server. Diverse content of each page is simulated with a binary blob filled with randomly-generated bytes. We experimented the average size of the web page in top 100 web sites as of July, 2014 [1] (1.829 MB). The number of requests per second is used as the comparison unit. `httpd-firewall` has higher overhead (26.86 percent) than `nginx-access` (1.90 percent) because the `firewall` module performs regular expression matching to detect an SQL injection attack from the request URIs, which is heavier than the simple IP/domain name matching used in `nginx-access`.

TABLE 4
Application Overhead Induced by CAFE's Confidential Execution

Benchmark name	Overhead (%)	Input workload
`nginx-access`	1.90	Apply 7 ACCESS/DENY rules to 10,000 transaction requests
`httpd-firewall`	26.86	Check the URIs of 10,000 transaction requests for an SQL injection attack using a regular expression
`sendmail-filter`	2.81	Verify the FROM addresses of an email sent repeatedly for 30 seconds using a regular expression
`gauth-otp`	2.52	SSH login/logout attempts using one-time passcodes repeatedly for 30 seconds
`encfs-aria`	900.13	Run the IOzone Filesystem Benchmark writing a 512KB file using a 4KB buffer
`mariadb-aes`	14.57	Handle 10,000 queries, each involving the AES decryption of two binary fields (100 and 200 bytes)
`MapReduce-kmeans`	8.04	Partition 8K two-dimensional integer points (64 MB) into 4K clusters
`mpi-mmult`	4.15	Multiplication of two 1000×1000 integer matrices (3.815 MB)
`hadoop-wcount`	5.82	Word count analysis of a text file containing 10,000 words

We use the Mstone SMTP performance testing tool [2] to measure the overhead of `sendmail-filter`. The test is run for 30 seconds with one client which repeatedly sends an email with the default Mstone email content. The format validity of the sender's email address is checked every time the email is sent by the server. The comparison unit is the number of trials per minute. The evaluation shows that the overhead is only 2.81 percent.

To evaluate `gauth-otp` case, the SSH client repeatedly logins to and logouts from the server for 30 seconds using one-time passcodes generated by secret binary. We measure the number of login and logout attempts per second as the unit of the comparison. The overhead for confidential execution is as trivial as 2.52 percent.

The overhead of `encfs-aria` is measured using the IOzone Filesystem Benchmark [10]. We use the average throughput (KB/sec) of each process writing a 512 KB file using a 4 KB buffer. The benchmark result shows that the application with the confidential execution support is about 9 times slower than with the binary without the support. We note that this benchmarking is a *stress case* where the file system is stressed with very frequent secret function calls, which cause high context switch cost. According to [44], which analyzed the file system workloads in various environments, the amount of data written in the web server is 960 MB per day and one in the NT workstation is 19.3 GB per day. On the other hand, `encfs-aria` in CAFE can write up to 68.96 GB per day (798.2 KB/sec), thereby we expect the overhead in the realistic setting to be much lower.

The `mariadb-aes` secret binary performs the encryption or the decryption of database fields when the DBMS executes a related SQL statement. We measure the performance of the server executing 10,000 SELECT statements, each requesting to decrypt and read two binary fields that are 100 and 200 bytes respectively. Our evaluation focuses on decryption, instead of encryption, because decryption is much more commonly and frequently executed than encryption in real-world database use. The time (in seconds) that takes for the DBMS to execute all of the queries is used as the comparison unit. Every time it is requested the query triggers the execution of the protected decryption logic twice and it is executed intensively for a short time, which results in the relatively large overhead of 14.57 percent.

`MapReduce-kmeans` is configured to partition 8,192 two-dimensional integer points (64 MB) into 4,096 clusters. The size of each dimension is set to 1,000, and two threads perform the clustering in parallel. The algorithm searches the data space for the cluster with the nearest mean for each point, computing the squared distance between every pair of the input points. The application is about 8 percent slower with the confidentiality support than the original application, both given the same input.

For the `mpi-mmult` case, we configure the program to take two 1000×1000 matrices of the integer type whose size is near 3.815 MB, as its input to the multiplication using one process. The protected logic is highly computation-oriented and the overhead introduced by the confidential execution is as low as 4.15 percent.

`hadoop-wcount` is run with an input text file that contains 10,000 words (108 KB). The input text is analyzed in two phases. First, each thread takes a distinct line from the text and counts the number of words in the line simultaneously. Second, a new set of threads reduce the intermediate word counts from the first phase to get the total number of words in the text. We use the CPU time spent during the two phases as the unit of the comparison. The phases mainly focus on simple arithmetic operations and memory I/O. The overhead imposed by CAFE for this binary is 5.8 percent.

We note that if these application functions were executed via secure RPC [3] between the hypervisor and the SBS, the overhead would be higher because of the high frequency of the input and output data transmission. In general, the size of input and output data for a program is larger than the size of the code. This is particularly important for data-intensive workload (e.g., big-data applications).

Resource Accounting. Secret functions are executed in the context of the hypervisor. Thus the resource usage of this code (e.g., CPU and memory usage) should be accounted as the resource consumption of a corresponding tenant VM. Cloud providers have various ways to define the resource accounting for confidential execution in CAFE. As in our measurement, the execution frequency of the secret functions and the amount of the input and output data marshaled can be used as a coarse grained measure. Alternatively accurate amount of memory and CPU usage can be also measured with low level techniques.

5.3 Performance Impact to Applications without Protection

We use the XMHF as the base of our implementation for basic hypervisor primitives and DRTM-related code. To evaluate the performance impact we run benchmarks (UnixBench) on CAFE without any secret binaries loaded and compare the results with a vanilla XMHF hypervisor with the basic VM management functionality only. The results confirm that CAFE does not impact unprotected applications in the VM (zero overhead). Due to space constraints, we do not present the data in this paper.

5.4 Porting Effort

In the usage scenario of CAFE, developers for cloud environments configure and modify software to separate the sensitive code, so that its confidentiality could be provided by CAFE. Since the strength of individual developers differ, measuring this efforts would need a user study. As one example, we describe the effort took for porting applications used in evaluation section so that their sensitive code could be separated and protected as a secret binary.

Separated Modules. In most applications of our evaluation, the sensitive code was separated from the rest of code. Thus, it is straightforward to identify and separate the secret binary from the applications. The decision-making logics (`nginx-access`, `sendmail-filter`, `httpd-firewall`), cryptographic operations (`gauth-otp`, `encfs-aria`, `mariadb-aes`), and computation algorithms (`MapReduce-kmeans`, `mpi-mmult`) meet the properties of the self-contained code, porting them into the CAFE framework was a very simple and straightforward task.

Mixed Modules. When the call target of sensitive code is outside of sensitive functions, the developers need to carefully refactor the code so that the whole logic can be

self-contained. `hadoop-wcount` uses classes derived from the HadoopPipe library, which provides C++ APIs such as `Mapper` and `Reducer` that encapsulate many features for supporting the MapReduce framework. Hadoop Pipe uses the well defined classes and functions (e.g., `HadoopPipes::ReduceContext` and `HadoopUtils::splitString`) and the C++ Standard Library (e.g., `vector`) to run a MapReduce job. To protect the algorithm along with such dependent modules, the sensitive code and related parts from the C++ classes are taken and ported to C functions.

Overall porting applications for CAFE only needed between 102 and 220 lines of code to be written as shown in Table 1. The actual effort to understand the code may vary depending on the software's structure, but given a set of benchmark programs in our evaluation, in many programs the sensitive code is organized as a separated module which will make the porting straightforward.

6 Possible Attacks and Defense

Attack of an Illegitimate Hypervisor. A possible attack for a malicious cloud user is to pretend to be a legitimate hypervisor to obtain secret binaries from the SBS. The attack can run in the user or kernel mode in the machine where the legitimate hypervisor is present. In either case, the fake hypervisor has TPM locality 1, different from the genuine hypervisor that has locality 2 which is reserved using the `TPM_Quote2` operation. A fake hypervisor may run in a different physical machine. However, the machine will fail the remote attestation due to unmatched TPM operations. Consequently, the SBS can detect the attack and refuse to send the secret binary.

Attack of an Illegitimate SBS. An attacker can attempt to run his own code on the hypervisor using his own illegitimate SBS. A general attack strategy would be observing how the transportation protocol works by inspecting network packets, and then the attacker can emulate the behavior of the legitimate SBS in a different machine. This attack fails because the fake SBS does not have the private key of the genuine SBS. Specifically, the hypervisor verifies a certificate transmitted from the fake SBS with the certificate from the CA. Since the certificate from the fake SBS is not issued by the CA, the verification fails and the transportation protocol is terminated by the hypervisor.

Stealing the Secret Binary ID or VM ID. In the deployment protocol, the secret binary ID and the VM ID are used to check the license for the loading of the secret binary. An attacker may attempt to steal the IDs and obtain the secret binary using them. However, the license check by the SBS uses the IDs only after the authorization of the attested hypervisor is passed. Therefore, stealing such information is not helpful for the attack to succeed.

Modification of a Secret Binary. An attacker in the user VM may try to manipulate the secret binary sent from the SBS. For example, the attacker may inject his own malicious logic into the code section of the secret binary in the file or memory before the isolation. CAFE prevents this kind of attacks using the cryptographic hash function, *HMAC*. Prior to loading the encrypted secret binary, CAFE verifies the integrity of the secret binary using the HMAC value generated by the SBS. In order to compute the legitimate HMAC value of the manipulated secret binary, the attacker must possess the valid HMAC key. However, because the valid HMAC key is shared only by the hypervisor and the SBS via secure channel, the attacker is not able compute the legitimate HMAC value. Upon the failure of the integrity verification, the loading of the manipulated secret binary is rejected by the hypervisor.

Man-in-the-Middle Attack (MITMA). An attacker may attempt an MITMA to derive the same shared secrets with the SBS that manages the secret binaries. In order to be successful, the attacker must impersonate both the hypervisor and the SBS and relay the communication in a machine where the attacker can eavesdrop the relevant network packets. To prevent this attack, the hypervisor and the SBS authenticate each other using the certificate of the other party. We have verified that our deployment protocol is invulnerable to MITMA using a formal verification tool examining the possible facets of the attack (Section 3.6).

7 Related Work

Secure Execution Environment. Flicker [34] and TrustVisor [33] provide an infrastructure for executing security-sensitive code in isolated memory based on the remote attestation of binary code. However, they primarily focus on blocking user VM's accesses to the application code in memory only while the memory isolation is enabled at runtime. This design may compromise code confidentiality because attackers in the VM may obtain a copy of the application code from the file system or memory during the deployment before the protection is enabled. In contrast, CAFE protects the confidentiality of the binaries in an end-to-end manner for the entire lifetime of the deployed software.

Overshadow [20] provides cloaking for general purpose legacy unmodified applications and untrusted kernel. However, related work [19], [37] have shown that a malicious kernel is able to compromise the protected OS even with the protection schemes by Overshadow. In contrast, CAFE provides stronger code confidentiality than Overshadow by providing tightly verified and sanitized input and output via marshaling layer, and a constrained scope of sensitive code which in combination significantly reduce the chance of vulnerability. More importantly, supporting a massive number of applications in cloud marketplaces requires several essential functions such as the secure attestation of TCB, key management, and deployment of binaries for which Overshadow is not designed. CAFE provides an end-to-end solution that covers the essential functionalities to enable confidential execution with cloud marketplaces.

AppSec [41] verifies the code integrity of protected applications and dynamic shared objects by imposing the safe loader. However, like Overshadow, it does not verify the integrity of OS so that attackers can exploit the Iago attacks.

Several solutions have been proposed for non-x86 architectures. Santos et al. [45] ported a small trusted language runtime for the .NET Framework into a secure environment protected by ARM TrustZone [13]. Thus, the security-sensitive logic of .NET applications running on top of the runtime environment can be protected by ARM TrustZone. Koeberl et al. proposed TrustLite [28], a secure execution environment for tiny embedded devices. It modifies the memory protection unit to manage a range of the trusted code section.

TABLE 5
Comparison Between CAFE and Intel SGX

	Trusted service			Trusted computing base		Rich OS support		
	Confidential deployment	Access to secure storage	Trusted path	Hardware	Software	Syscall	Ext. call	Whole app execution
CAFE	Secure binary transmission	Trusted access to NVRAM in TPM	Direct access in hypervisor	CPU, TPM, DRAM, buses	Hypervisor, secret binary	×	×	×
Intel SGX	×	Untrusted due to escape of enclave	×	CPU	Architectural enclaves, secret binary	×	OCALL instruction	×

Software Protection with Code Obfuscation. Software vendors have been working hard to protect their code from reverse engineering and software piracy. A popular technique is code obfuscation [21] which transforms binary code into another form that is a functionally identical but more difficult to understand.

There are several packers that have been widely used such as UPX [5] and ASPack [7]. More advanced methods have been proposed as follows. Linn and Debray et al. [32] proposed a method to thwart disassembly using the self-repairing disassembly, the junk insertion, etc. Popov et al. [36] proposed a signal-based obfuscater that changes control transfers into signals. Sharif et al. [46] proposed an automatic transformation of a program based on code encryption that conditionally uses an input value as a key. binOb+ [30] eliminates statistical abnormality of obfuscated binary by inserting statistics-compensation code.

Virtualization-based code obfuscation techniques are known to be strong against reverse engineering due to their abstraction layers. These techniques convert executable code into a non-standard architecture to make the analysis difficult. Themida [8], VMProtect [9], and Truly-Protect [14] are well known examples.

While the code obfuscation techniques can impede the analysis of code, they cannot provide the complete secrecy of executable code because obfuscated code may still retain code semantics. Also, unless the execution environment is isolated from an adversary, the runtime execution pattern can be observed (e.g., through memory) and used to deobfuscate the code [22], [43]. Unlike such solutions, CAFE provides full confidentiality by cryptographically encrypting the binary code and running the decrypted code in an isolated environment from the user VM. Therefore, any software running in the user VM, including the guest OS, cannot access the runtime state of the protected code.

Comparison with Intel SGX. Intel Software Guard Extensions (SGX) [25] is a new set of instructions to build a secure execution environment for user level applications against physical and software attacks. CAFE and SGX both have a goal of *providing a secure enclave for secret code and data*, however there are several technical differences as shown in Table 5.

(1) Trusted Services. Compared with SGX, CAFE offers several trusted services: confidential deployment of secret binaries, trusted storage for sensitive information, and trusted path, which are the problems of SGX and thus addressed by related work [17], [27], [29], [35].

The confidential deployment of secret binaries is one of the primary features of CAFE. Several researchers proposed conceptual designs [27], [29] that enable secure deployment of encrypted code and data in an SGX enclave. For this feature, SGX should be able to perform dynamic memory allocation and change access permissions associated with the enclave memory, but SGX1 does not have these features. As a solution, F. McKeen et al. [35] presented SGX2 that includes dynamic memory management support for enclaves.

Secure storage is a protected, persistent memory area to securely store private data (e.g., cryptographic keys) that can be directly accessed only by a trusted execution environment (TEE). Its access must not involve untrusted OS. CAFE within a TEE can securely access non-volatile RAM (NVRAM) in a TPM as a secure storage with the access control based on locality and PCR values [52]. In contrast, an enclave operates in a user mode (ring 3). To access NVRAM similarly, its control flow has to escape from it. By relying on untrusted OS code, it is challenging for SGX to embed a separate memory medium under the direct control of SGX [38]. The lack of secure storage poses rollback and forking attacks to stateful SGX enclaves. To address this problem, [17] proposed a protocol to defeat the rollback and forking attacks on SGX enclaves.

CAFE provides a trusted path that guarantees the trustworthiness of input and output (I/O) data because the hypervisor has the top priority for processing I/O events. However, SGX does not support the trusted path feature because an enclave running in a user mode cannot preempt I/O events with a higher priority than kernel mode software. To address this problem, S. Weiser et al proposed a framework called SGXIO [51] that securely handles input and output based on a hypervisor, a Trusted Boot enclave, and one more secure I/O drivers.

As a useful scenario for the CAFE's trusted path feature, CAFE can deliver the trusted geolocation of the cloud service to the cloud tenant who may wonder where their cloud services run. CAFE can provide a trusted path regarding the geolocation of a cloud server by directly handling a GPS sensor placed on the cloud server.

(2) Trusted Computing Base (TCB). The TCB is a set of hardware and software components that should not be compromised for maintaining a trusted system. CAFE leverages the DRTM mechanism that relies on the CPU and the TPM, and the hypervisor in CAFE to be run via the DRTM resides in memory. The CPU, the TPM, and memory transfer data via buses. Therefore, they become the hardware TCBs of CAFE. On the other hand, when SGX enclaves' code and data are loaded into memory, the CPU encrypts them with a CPU-specific key and stores encrypted enclaves' code and data into memory. When the CPU needs to run the enclave's code and data, the CPU decrypts them inside the CPU and executes them. Thus, the hardware TCB of SGX is the CPU.

The software TCB of CAFE includes a secret binary and the hypervisor because a compromised hypervisor can leak secret binaries outside the CAFE framework. In SGX, the software TCB includes a secret binary and architectural enclaves (AEs) [25] because compromised AEs can undermine SGX's remote attestation. An AE is a privileged enclave issued by Intel and performs complex operations (e.g., cryptographic operations) instead of hardware. In AEs, the launch enclave and the quoting enclave play an important role so as to remotely attest an application logic running in an SGX enclave. Thus, the AEs must be maintained securely for trustworthy remote attestation.

(3) Rich OS Support. In general porting secure code and data into the TEE is complicated work, as we mentioned in Section 5.4. SGX and CAFE do not feature rich OS supports causing developers' efforts. To ease this situation, several approaches were proposed. Panoply [47] provides an abstraction layer called micron to enable SGX enclave code to call external functions, including system calls and library functions. Haven [15] and Graphene-SGX [18] implement existing library OSes, Drawbridge and Graphene, into an SGX enclave in order to run unmodified legacy applications inside an enclave.

However, these approaches are not without risks due to recent attacks against such efforts. First, J. Lee et al. presented a new attack called Dark-ROP [31] that finds gadgets for the return-oriented programming (ROP) attack without any knowledge of SGX enclave code and data. Authors noted that expanding the TCB and the attack surface (e.g., Haven and Graphene-SGX) for the easy enclave development is not good research direction because the Dark-ROP attack can find more ROP gargets in an enclave. Second, SGX offers an instruction for calls outside the enclave (OCALL) for developers' convenience, which is used by Panoply's micron. However, the Intel SGX developer guide [25] indicates that OCALLs have associated several security risks such as information leak with OCALL, no execution of OCALL, and recursive OCALLs.

We considered this tradeoff between security and support for development. Currently CAFE does not provide rich OS support due to our priority on the trustworthiness of the CAFE framework.

8 Conclusion

The secure distribution and execution of cloud applications is an essential feature to prevent the illegitimate usage of cloud applications and further for the success of the evolving ecosystem of cloud systems. In order to defeat software piracy and reverse engineering of sensitive software logic, we present CAFE which provides the confidential distribution and execution of cloud applications even when the entire guest OS of the tenant virtual machine is compromised. We present its evaluation on 9 applications which are commonly used in data centers and offered in cloud marketplaces showing the effectiveness and practicality of CAFE in cloud marketplaces.

References

[1] Average Web Page Breaks 1600K, 2014. [Online]. Available: http://www.websiteoptimization.com/speed/tweak/average-web-page/

[2] Mstone, 2014. [Online]. Available: http://mstone.sourceforge.net/

[3] Overview of Secure RPC, 2011. [Online]. Available: http://docs.oracle.com/cd/E23823_01/html/816–4557/auth-2.html

[4] The Transport Layer Security (TLS) Protocol Version 1.2, 2008. [Online]. Available: http://tools.ietf.org/html/rfc5246

[5] UPX: The Ultimate Packer for eXe-cutables, 2007. [Online]. Available: http://upx.sourceforge.net

[6] A Description of the ARIA Encryption Algorithm, 2010. [Online]. Available: http://tools.ietf.org/search/rfc5794

[7] ASPack, 2010. [Online]. Available: http://www.aspack.com

[8] Themida, 2010. [Online]. Available: http://www.oreans.com

[9] VMProtect, 2010. [Online]. Available: http://vmpsoft.com/products/vmprotect/

[10] IOzone Filesystem Benchmark, Feb. 2013. [Online]. Available: http://www.iozone.org/

[11] Advanced Micro Devices. AMD64 architecture programmer's manual: Vol. 2: System programming, 2013. [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf.

[12] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 1st ed. New York, NY, USA: Wiley, 2001.

[13] ARM. ARM Security Technology: Building a Secure System Using TrustZone Technology, 2009. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[14] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg, "Truly-protect: An efficient VM-based software protection," *IEEE Syst. J.*, vol. 7, no. 3, pp. 455–466, Sep. 2013.

[15] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 267–283.

[16] B. Blanchet, M. Abadi, and C. Fournet, "Automated verification of selected equivalences for security protocols," *J. Logic Algebraic Program.*, vol. 75, no. 1, pp. 3–51, Feb. 2008.

[17] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and forking detection for trusted execution environments using lightweight collective memory," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2017, pp. 26–29.

[18] C. Che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *Proc. USENIX Annu. Techn. Conf.*, 2017, pp. 645–658.

[19] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," in *Proc. 18th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2013, pp. 253–264.

[20] 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. 13th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2008, pp. 2–13.

[21] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, University of Auckland, Auckland, Tech. Rep. 148, Jul. 1997.

[22] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 275–284.

[23] D. Challener, K. Yoder, R. Catherman, D. Safford, L. Van Doorn, *A Practical Guide to Trusted Computing*. Indianapolis, IN, USA: IBM Press, 2007.

[24] E. Bendersky, "Position independent code (PIC) in shared libraries." (2011). [Online]. Available: http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-i n-shared-libraries/

[25] Intel Corporation, "Intel(R) software guard extensions SDK developer reference for Linux OS." 2017. [Online]. Available: https://01.org/intel-software-guard-extensions/documentation/intel-sgx- sdk-developer-reference

[26] Intel Corporation. Intel Trusted Execution Technology (Intel TXT) Software Development Guide, 2017. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf.

[27] J. Rutkowska, Thoughts on Intel's upcoming Software Guard Extensions (Part 2), Sep. 2013. [Online]. Available: http://theinvisiblethings.blogspot.kr/2013/09/thoughts-on-intels-upcoming-software.html

[28] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proc. 9th Eur. Conf. Comput. Syst.*, Apr. 2014, Art. no. 10.

[29] Kudelski Group. SGX Secure Enclaves in Practice: Security and Crypto Review, 2016. [Online]. Available: https://www.kudelskisecurity.com/about-us/news/sgx-secure-enclaves-practice-security-and-crypto-review

[30] B. Lee, Y. Kim, and J. Kim, "binOb+: A framework for potent and stealthy binary obfuscation," in *Proc. 5th ACM Symp. Inf., Comput. Commun. Secur.*, 2010, pp. 271–281.
[31] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 523–539.
[32] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. 10th ACM Conf. Comput. Commun. Secur.*, 2003, pp. 290–299.
[33] 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. Secur. Privacy*, 2010, pp. 143–158.
[34] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proc. 3rd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2008, pp. 315–328.
[35] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel®software guard extensions support for dynamic memory management inside an enclave," in *Proc. Hardware Archit. Support Secur. Privacy*, 2016, Art. no. 10.
[36] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *Proc. 16th USENIX Secur. Symp. USENIX Secur. Symp.*, 2007, Art. no. 19.
[37] D. R. K. Ports and T. Garfinkel, "Towards application security on untrusted operating systems," in *Proc. 3rd Conf. Hot Topics Secur.*, 2008, Art. no. 1.
[38] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, "FTPM: A software-only implementation of a TPM chip," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 841–856.
[39] R. Bias, "Amazon's EC2 generating 220M+ annually." (2009). [Online]. Available: http://www.cloudscaling.com/blog/cloud-computing/amazons-ec2-generating -220m-annually/
[40] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*, 2007, pp. 13–24.
[41] J. Ren, Y. Qi, Y. Dai, X. Wang, and Y. Shi, "AppSec: A safe execution environment for security sensitive applications," in *Proc. 11th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, 2015, pp. 187–199.
[42] R. van der Meulen, Janessa Rivera. "Gartner Says Worldwide Public Cloud Services Market to Total $131 Billion." (2013). [Online]. Available: http://www.gartner.com/newsroom/id/2352816
[43] R. Rolles, "Unpacking virtualization obfuscators," in *Proc. 3rd USENIX Conf. Offensive Technol.*, 2009, pp. 1–1.
[44] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proc. USENIX Annu. Techn. Conf.*, 2000, pp. 18–23.
[45] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM trustzone to build a trusted language runtime for mobile applications," in *Proc. 19th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2014, pp. 67–80.
[46] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proc. 15th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2008.
[47] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "PANOPLY: Low-TCB Linux applications with SGX enclaves," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017.
[48] T. Hauser and C. Wenz, "DRM under attack: Weaknesses in existing systems," in *Digital Rights Management*, Berlin, Germany, Springer, 2003.
[49] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an eXtensible and modular hypervisor framework," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 430–444.
[50] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig, "Lockdown: Towards a safe and practical architecture for security applications on commodity platforms," in *Proc. 5th Int. Conf. Trust Trustworthy Comput.*, 2012, pp. 34–54.
[51] S. Weiser and M. Werner, "SGXIO: Generic trusted I/O path for intel SGX," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 261–268.
[52] W. Arthur and D. Challener, *A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security*. New York, NY, USA: Apress, 2015.
[53] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation resilient binary code reuse through trace-oriented programming, in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 487–498.

Sungjin Park received the BS degree from the Inha University, in 2002, the MS degree from the Pohang University of Science and Technology, South Korea, in 2005, and the PhD degree from the Korea Advanced Institute of Science & Technology, in 2017. He is a senior researcher with the Attached Institute of Electronics and Telecommunications Research Institute (ETRI), South Korea. His research interests include network and system security.

Chung Hwan Kim received the BS degree from Sunmoon University, the MS degree from the University of Utah, and the PhD degree in computer science from Purdue University, in 2017. He is currently a researcher in the NEC Laboratories America in Princeton. His research interests include operating systems, system security, and program analysis.

Junghwan Rhee received the BS degree from Korea University, the master's degree from the University of Texas at Austin, and the PhD degree in computer science from Purdue University, in 2011. He is a senior researcher with the NEC Laboratories America in Princeton. His research interests include malware analysis, system security, software debugging, and cloud computing. He is a member of the IEEE.

Jong-Jin Won received the MS and PhD degrees from the Department of Computer Engineering, SungKyunKwan University, South Korea, in 2000 and 2015, respectively. Since 2000, he has been working with the Attached Institute of Electronics and Telecommunications Research Institute (ETRI). His research interests include network and system security.

Taisook Han received the BS degree in electronic engineering from Seoul National University, Korea, in 1976, the MS degree in computer science from KAIST, Korea, in 1978, and the PhD degree in computer science from the University of North Carolina at Chapel Hill, in 1990. He is currently a professor in the School of Computing, KAIST. His current research interests include programming language theory, software safety, and verification of embedded systems. He is a member of the IEEE.

Dongyan Xu received the BS degree in computer science from Zhongshan (Sun Yat-sen) University, in 1994 and the PhD degree from the University of Illinois at Urbana-Champaign, in 2001. He is currently a professor with the Department of Computer Science, Purdue University. His current research focuses on the development of virtualization technologies for computer system security and for cloud computing.