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

Chung Hwan Kim[†], Sungjin Park[‡¶], Junghwan Rhee[§],
Jong-Jin Won[‡], Taisook Han[¶], Dongyan Xu[†]
[†]Purdue University, [‡]The Attached Institute of ETRI, [§]NEC Laboratories America, [¶]KAIST
[†]{chungkim,dxu}@cs.purdue.edu, [‡]{taiji,wonjj}@ensec.re.kr,
[§]rhee@nec-labs.com, [¶]taisook@kaist.ac.kr

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.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

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 [20]. 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.

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.

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

Existing approaches [16, 15, 10] leveraging virtualization-based memory protection 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.

As a new alternative, we propose a system named Cloud Application Function Enclaving (CAFE) to address these

ASIA CCS '15, April 14 - 17, 2015, Singapore, Singapore

http://dx.doi.org/10.1145/2714576.2714594.

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.

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.

This paper is structured as follows: Section 2 presents the adversary model. The design is presented in Section 3. The evaluation of CAFE is presented in Section 4. Related work is discussed in Section 5, and Section 6 concludes this paper.

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 the kernel, drivers, libraries, and applications in the user and kernel mode running in the user VM. 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 using file I/O. 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, using a debugging tool or by injecting malicious code into its memory the attacker can obtain the binary code from the runtime memory of the target processes.

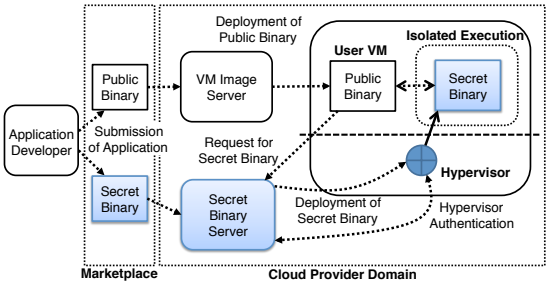

Figure 1: Overview of CAFE.

Access to network: Another attack method to obtain the binary would be using the network layer. We assume that the attacker can eavesdrop and modify network traffic between cloud provider servers and cloud user VMs. Therefore, if a binary file is transferred in a plain form, the attacker may be able to obtain it from network packets.

3. DESIGN

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 this goal 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 [11, 13, 14, 10, 16, 15]. The overview of the CAFE architecture is illustrated in Figure 1.

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 APIs of CAFE, 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 VM image that contains the public binaries, and the secret binary files are submitted separately. 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)*, 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 purchase applications from the cloud marketplace and then the cloud provider lets the cloud user create a VM using the corresponding VM image that includes the public binaries of the purchased application. The secret binaries, on the other hand, are not delivered this time. Instead, they are delivered to the hypervisor through a secure channel when the binaries are requested for use.

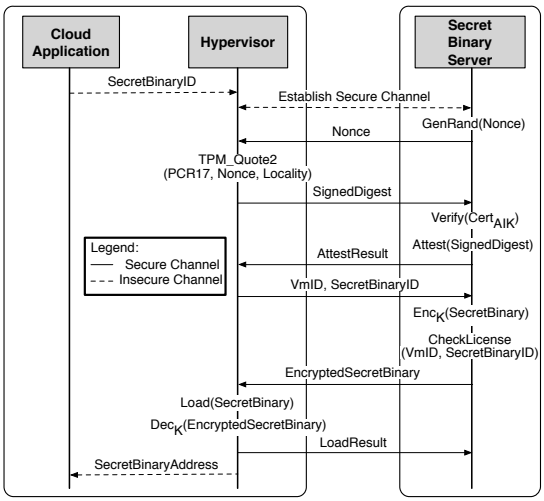

Figure 2: Secret binary deployment protocol.

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 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. 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 the genuineness of hypervisor. After that, the SBS transfers the secret binaries encrypted 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 completely isolated from the user VM.

Designation of sensitive code: Security sensitive code has the key logic of an application requiring a high degree of protection. Since this is typically a small portion of the entire application code, the cost to achieve confidentiality is generally amortized in the overall performance of programs. Our evaluation in Section 4 will confirm that this high secrecy property can be achieved with minimal overhead.

3.1 Secure Authentication and Deployment of Secret Binary

Figure 2 depicts our deployment protocol of secret binaries, designed to defend against attacks.

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 details of the secure channel establishment are as follows.

Among several candidate key exchange algorithms (i.e., Diffie-Hellan and RSA), we choose a variant of Transport Layer Security (TLS) [3] 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 an 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`) 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 [12]. The hypervisor performs `TPM_Quote2` and transfers the resulted digest (`SignedDigest`) to the SBS which verifies the integrity of the hypervisor by matching the received digest with the certificate ($\texttt{Cert}_{AIK}$) from the private CA.

Verification of application licenses: Upon successful attestation of the hypervisor, the cloud application sends the VM ID and the secret binary ID 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 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 SBS encrypts 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 a different session key (`K` in Figure 2) 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: After the transmission, the secret binaries remain encrypted in the user VM's disk throughout the all steps of the secret binary loading and execution. 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. Finally, the result of the loading is sent back to the SBS, and the application receives the address of the secret binary.

3.2 Runtime Protection of Secret Binary

In this section, we describe how CAFE protects secret binaries on disk and memory.

Protection of the secret binary on disk: The secret binary file on disk is cryptographically protected by an

encryption algorithm. Among various available encryption algorithms, we choose AES-256 in the CBC mode to encrypt and decrypt secret binaries. 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 3.1) 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. 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 loaded into memory, it is protected by 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 and AMD Rapid Virtualization Indexing. CAFE uses this technique to load the secret binary into the memory and create a secure execution environment isolated from the user VM.

Specifically before the secret binary is loaded, there is one set of nested page tables (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 which enforce the exclusive access by the secret binary 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 [16, 15]. Compared to the existing work, CAFE ensures that the sensitive application logic is inaccessible from the user VM at any time.

3.3 Secure Hypervisor Loading of Secret Binary

A program's execution is performed through low level operations such as allocation of system resources, program loading and linking, and the bootstrapping of the program. These operations are typically performed by the high privileged software layer in the OS or system level libraries. In our setting, the user VM including its OS 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.

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, the location in the virtual memory of the process and the symbols (e.g., exported global variables and functions) are determined dynamically by the *loader*. Thus, any instructions of the program that refer to the symbols must be updated with the addresses determined at runtime 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 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 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: The application requests the hypervisor to load the secret binary using the APIs provided CAFE. Then the hypervisor decrypts the secret binary into the memory isolated from the user VM. Since the OS loader is untrusted, 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. 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.

4. EVALUATION

CAFE consists of three major components: the hypervisor, the secret binary server (SBS), and the user level APIs that the cloud applications use to send requests to the hypervisor via hypercall.

The hypervisor is implemented on top of eXtensible and Modular Hypervisor Framework (XMHF) [22] which is used in several related work [22, 15, 23]. 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.

We use two separate machines for our experiments for the hypervisor and the SBS. Both machines are equipped with an AMD Turion II P520 2.30GHz processor, 4GB RAM, and a 256GB SSD, and run the 32-bit version of Ubuntu 12.04. The virtual machines and the SBS are connected to a 1Gb/s network.

4.1 Use Cases

We demonstrate that various types of application code can be protected by CAFE using six 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. 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. Our 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. The first column presents three categories of program logic: decision-making logic, cryptographic operations, and data processing work-

Application Category	Program Name	Binary Name	Code info			Runtime info			
			Protected code	$\|L\|$	$\|F\|$	$\|C\|$	$\|D\|$	$\|R\|$	Overhead
Decision-making logic	NGINX	`nginx-access`	Access module	169	1	4	44	19	1.90%
	Sendmail	`sendmail-filter`	Mail Filter (Milter)	106	1	52	52	559	2.81%
Cryptographic operations	Google Authenticator	`gauth-otp`	One-time passcode generation	102	1	8	44	16	2.52%
	EncFS	`encfs-aria`	ARIA block encryption/decryption	220	3	24	48	346	900.13%
Data processing workload	MapReduce	`mapreduce-kmeans`	k-means clustering	173	1	12	44	380	8.04%
	Hadoop	`hadoop-wcount`	Word counting	180	2	4	44	32	5.82%

Table 1: Use cases of confidential execution of secret cloud application binaries.

load. The following columns show 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 secret functions ($|F|$) and runtime characteristics of the secret binaries ($|C|$: code section size, $|D|$: data section size, $|R|$: relocation table size, overhead). 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 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.

Cryptographic operations: If the OS is compromised, cryptographic operations are no longer safe because OS can look into their runtime states which can potentially be used to infer their operations. We show several use cases of well known cloud applications. We port the Google Authenticator Pluggable Authentication Module (PAM), `gauth-otp`, that protects the passcode generating code, and EncFS which is a file system with the block-level encryption. We selected a EncFS, `encfs-aria`, that uses the ARIA cipher [4]. The encryption and decryption algorithms along with the key initialization function are protected.

Data processing workload: Some security sensitive code may involve intensive computation. We use two parallel data processing algorithms running on a parallel computing framework to show the support of this category. `mapreduce-kmeans` is an implementation of k-means clustering based on Phoenix [19], a shared-memory and C language based MapReduce framework. The protected code partitions n-dimensional integer points into a number of clusters. Lastly, `hadoop-wcount` is an 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 (average 1.18%). Compared to the total LoC of the entire program, it is a small portion of the program.

4.2 Performance of Confidential Execution of Cloud Applications

We present the overhead of the applications for confidential execution in CAFE (Table 1). We calculate the overhead by comparing the performance of the original version with the modified version with confidential protection. 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. 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. To measure the overhead of `nginx-access`, we have the Apache Benchmark issue 10K transactions per trial. 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]. The overhead comparing the number of requests per second is 1.9%.

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 evaluation shows that the overhead is only 2.81%.

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. The overhead for confidential execution is as trivial as 2.52%.

The overhead of `encfs-aria` is measured using the IOzone Filesystem Benchmark [7]. We use the average throughput (KB/sec) of each process writing a 512KB file using a 4KB 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. In typical real-world cloud applications such workload is unusual especially when an encrypted disk is used, thereby we expect the overhead in the realistic setting to be much lower.

`mapreduce-kmeans` is configured to partition 8,192 two-dimensional integer points (64 MB) into 4,096 clusters. The application is about 8% slower with the confidentiality support than the original application, both given the same input. `hadoop-wcount` is run with an input text file that contains 10,000 words (108 KB). We use the CPU time spent during the two phases as the unit of the comparison. The overhead introduced by CAFE for this binary is 5.8%.

4.3 Performance Impact to Applications without Protection

We use the XMHF hypervisor framework [22] 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).

5. RELATED WORK

Flicker [16] and TrustVisor [15] 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 [10] provides cloaking for general purpose legacy unmodified applications and untrusted kernel. Related work [9, 18] have shown that a malicious kernel is able to compromise the protected OS even with the protection schemes by Overshadow. 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.

Software vendors have been working hard to protect their code from reverse engineering and software piracy. There has been a large body of work on obfuscators [14, 17, 21, 13, 5, 6, 8] which make disassembly hard. While the code obfuscation techniques can impede the analysis of code, they are not designed to provide the complete secrecy of executable code because obfuscated code may still retain code semantics. Unlike such solutions, CAFE provides full confidentiality by cryptographically encrypting the binary code and running the decrypted code in an isolated environment.

6. 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 OS of the tenant VM is compromised. We present its evaluation on a number of applications commonly offered in cloud marketplaces showing the effectiveness and practicality of CAFE.

7. REFERENCES

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

[2] Mstone. `http://mstone.sourceforge.net/`.

[3] The Transport Layer Security (TLS) Protocol Version 1.2. `http://tools.ietf.org/html/rfc5246`.

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

[5] Themida, 2010. `http://www.oreans.com`.

[6] VMProtect, 2010. `http://vmpsoft.com/products/vmprotect/`.

[7] IOzone Filesystem Benchmark, Feb. 2013. `http://www.iozone.org/`.

[8] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg. Truly-Protect: An Efficient VM-Based Software Protection. *IEEE Systems Journal*, 7(3):455–466, Sept. 2013.

[9] S. Checkoway and H. Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 253–264, New York, NY, USA, 2013. ACM.

[10] 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 *Proceedings of ASPLOS'08*, New York, NY, USA, 2008.

[11] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. *Technical Report 148 Department of Computer Science University of Auckland July*, page 36, 1997.

[12] David Challener, Kent Yoder, Ryan Catherman, David Safford, Leendert Van Doorn. *A Practical Guide to Trusted Computing*. IBM Press, 2007.

[13] B. Lee, Y. Kim, and J. Kim. binOb+: A framework for potent and stealthy binary obfuscation. In *Proceedings of ASIACCS'10*, New York, NY, USA, 2010.

[14] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of CCS'03*, New York, NY, USA, 2003.

[15] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of SP'10*, DC, USA, 2010.

[16] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the Eurosys'08*, pages 315–328, New York, NY, USA, 2008.

[17] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary Obfuscation Using Signals. In *Proceedings of USENIX Security'07*, Berkeley, CA, USA, 2007.

[18] D. R. K. Ports and T. Garfinkel. Towards application security on untrusted operating systems. In *Proceedings of the 3rd Conference on Hot Topics in Security*, HOTSEC'08, pages 1:1–1:7, Berkeley, CA, USA, 2008. USENIX Association.

[19] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of HPCA'07*, Washington, DC, USA, 2007.

[20] Rob van der Meulen, Janessa Rivera. Gartner Says Worldwide Public Cloud Services Market to Total $131 Billion. `http://www.gartner.com/newsroom/id/2352816`.

[21] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. *Proceedings of NDSS'08*, 2008.

[22] 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 *Proceedings of SP'13*, pages 430–444, DC, USA, 2013.

[23] Vasudevan, Amit and Parno, Bryan and Qu, Ning and Gligor, Virgil D and Perrig, Adrian. Lockdown: Towards a Safe and Practical Architecture for Security Applications on Commodity Platforms. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, 2012.