Trusted Block as a Service: Towards Sensitive Applications on the Cloud

Jianan Hao Wentong Cai
Parallel and Distributed Computing Centre, School of Computer Engineering
Nanyang Technological University, Singapore
haojn@pmail.ntu.edu.sg, ASWTCAI@ntu.edu.sg

***Abstract*—Cloud computing grows rapidly as today's advanced information technology. However, by allowing outsourcing computation on the Cloud, users risk of disclosing privacy and obtaining forged results. These potential threats block sensitive applications to join the Cloud.**

In this paper, we characterize sensitive applications on the Cloud (SAND) problem and define two critical security requirements: confidentiality and verifiability. The former refers to the protection of sensitive programs/data from disclosing to other users or even the Cloud administrators. The latter concerns with user's capability to verify whether computing results are faithfully calculated.

To address SAND, we propose a new Cloud model, Trusted Block as a Service (TBaaS), to provide a confidential and verifiable environment for each sensitive application. TBaaS limits Cloud provider's access of sensitive applications while granting user the ability to verify whether the computation is faithfully carried out. Moreover, it offers high flexibility and low performance overhead.

I. Introduction

The emergence of Cloud computing significantly changes the rules of information technology. By integrating geometry and capacity of resources, a Cloud provider can offer its users flexible environments to execute their own business. The Cloud provides the illusion of unlimited resources to satisfy users' dynamic and scalable requirements. Meanwhile, by ruling out the necessity to establish the infrastructure, users save their investments at initial stage, as well as the time to market for products. The pay-per-use model may further decrease the cost for resource usage. Finally, centralized resources can be efficiently maintained by the Cloud provider and thereby reduces its users' maintenance cost.

Despite the various benefits offered by the Cloud, many potential users do not join the Cloud, or only put their less sensitive applications on the Cloud [4]. The worry mainly comes from data leakage, and service hijacking [6]. For a company, losing core intellectual property may lead to a disaster and hijacked services may result in falsified information which damages company's reputation.

To solve the problem of Sensitive Application oN clouD (SAND), we suggest two security requirements must be satisfied: confidentiality and verifiability. Confidentiality guarantees that users can securely transfer a processing program, parameter and return value to/from the Cloud without risk of disclosing them to another Cloud tenant or other parties. It is important for sensitive applications since a processing program, parameter and return value may contain raw sensitive data. Even the Cloud provider should not be allowed to access these data. The other requirement, verifiability, is also necessary. Verifiability grants that users can verify their results returned from the Cloud, that is, whether it is faithfully computed by the applications they deployed along with the genuine parameter they provided. Moreover, since an application's purpose may vary from one to another, a highly flexible environment is desirable and the performance overhead should be minimized.

To mitigate security issues, some existing Cloud solutions rely on trusted computing [16] to build up a trusted Cloud. The Cloud grants users the ability to verify its Trusted Computing Base (TCB). TCB refers to a set of components critical to system security, e.g., hypervisor and management domain for Xen [5]. Nevertheless, knowing what software stack is running on the Cloud is not enough to provide confidentiality and verifiability. A secure protocol is required to satisfy such requirements. Furthermore, a trusted Cloud only answers what constructs TCB; however, whether components inside TCB can meet their asserted design depends on trustworthiness. A trusted and trustworthy Cloud is necessary for deploying sensitive applications. For example, Xen can be used to build a trusted Cloud, but if Xen hypervisor is buggy, the Cloud cannot be trustworthy and the attackers may utilize the bug to attack sensitive applications. To prove a Cloud is trustworthy, Cloud's TCB should be formally verified via mathematical proof or be inspected manually. Both approaches are sensitive to the size of TCB which can be measured by Lines Of Code (LOC). The smaller the TCB, the higher the possibility to be trustworthy. Unfortunately, most of the existing hypervisors have a large TCB. For instance, hypervisor of Xen 4.0.1 is around 187K LOC (for `x86` platform only) calculated by SLOCCount [18], not including the management domain (Dom0) that includes numerous hardware drivers.

We propose Trusted Block as a Service (TBaaS) to solve SAND problem in this paper. A tiny hypervisor, named SandVisor, is introduced to achieve isolation. The simplified infrastructure significantly reduces the TCB. In addition, we present carefully designed protocols to satisfy security requirements of SAND. Informal analysis of the protocols will be also given. We hope our work can be helpful to release the potential power of Cloud for sensitive applications.

The rest of paper is organized as follows. The next section defines SAND problem and its requirements. Section III investigates technical background. The overview of TBaaS is shown in section IV. Section V illustrates five phases of TBaaS and their protocols. Section VI informally verifies the protocols and Section VII reviews related research work. Finally, Section VIII concludes the paper and discusses future work.

II. Problem Formulation

We will define Sensitive Applications oN clouD (SAND) problem in this section.

We assume the computation is carried out between two parties: Alice and Cloud. Alice refers to an user who is outsourcing a sensitive application on the Cloud. The logic of processing program is abstracted as foo and the parameter/return value of the program is denoted by $input/result$. Cloud is supposed to compute $result = foo(input)$. Generally, the computation can be described in four steps:

1) $Alice \rightarrow Cloud$: foo, $input$
2) $Cloud$: $result = foo(input)$
3) $Cloud \rightarrow Alice$: $result$
4) $Alice$: Verify $result$

Firstly, Alice will tell Cloud foo and $input$ (optionally with encryption). Then Cloud will obtain $result$ by performing $foo(input)$ and send return value $result$ back to Alice (optionally with encryption). Finally, Alice may verify $result$.

Confidentiality: foo and $input$ are sensitive program/data for computing $result = foo(input)$ exclusively. Cloud is responsible for protecting them from disclosure.

Verifiability: Alice must be able to detect whether $result$ returned from Cloud is faithfully computed by foo with $input$ as parameter.

To solve SAND, potential solutions must satisfy the above two security requirements. Confidentiality guarantees Alice can securely deliver the processing program foo and parameter $input$ to the Cloud and obtain return value $result$ without taking the risk of disclosing them to another Cloud tenant or other parties. Even the Cloud administrators should not have rights to access them. The other requirement, verifiability is also necessary for sensitive applications. It grants Alice to verify whether $result$ returned from the Cloud is faithfully computed by the program foo with parameter $input$ she provided.

Note that practical solutions to SAND should not rely on morality of the Cloud provider. As indicated in [6], when the potential reward becomes sufficiently attractive, Cloud employees may become malicious insiders. We suggest solutions should only depend on high assurance knowledge, e.g., well-studied hardware-based technologies.

Besides security, we also require solutions to provide high flexibility and performance. The former provides the freedom to deploy various applications on the Cloud; the latter means less performance overhead as measured by execution efficiency margin between non-Cloud and Cloud.

III. Background

In this section, we will review technical background related to our work.

A. Cryptographic Primitives

Cryptography is the critical fundamental for security systems. In this paper, one-time pad, public encryption, digital signature, hash function and nonce will be used.

One-time pad is an encryption method. Each bit from the plaintext is encrypted by eXclusive-OR (XOR) with a bit from a secret random key (pad) of the same length, returning a ciphertext. If the pad is truly random, kept secret and never reused, it is infeasible to decrypt the ciphertext without knowing the pad.

Public encryption and digital signature are both based on public-key cryptography. Essentially, each key pair consists of two parts, public key and private key. The former will be exposed to public and the latter should be kept by its owner. Plaintext encrypted by a public key can only be recovered by the corresponding private key and vice versa. For encryption, Alice can encrypt a message by Bob's public key so only Bob can reveal it by his private key. For signature, Bob can digitally sign a message by his private key so Alice can verify the signature by Bob's public key, knowing the signature is made by Bob.

Hash function is a well-defined deterministic procedure to produce a fixed digest (hash value) by given a message with arbitrary length. It guarantees that it is infeasible to recover a message by given its digest. It also guarantees that it is infeasible to find two messages that share an identical hash value.

A nonce is an one-time random number used to mitigate replay attacks. Nonces are generated as session tokens to distinguish messages sent at different time. Receiver thereby can verify the nonce to detect replayed messages.

B. Virtualization

Virtualization is a key technology for Cloud computing to multiplex resources among users. Generally, virtualization involves two parts, hypervisor and virtual machines (VMs). A hypervisor is responsible for managing physical hardware and providing virtual environments to VMs. Isolation and communication are also offered by the hypervisor. Traditional `x86` platform does not satisfy Popek and Goldberg's virtualization requirements [13]. Thereby, emulation or dynamic binary translation [3] is used to build the hypervisor. However, it leads to a large code base and decreased performance. It also lacks of ability to block illegal DMA requests.

To improve this situation, Hardware-assist Virtual Machine (HVM) is applied to the latest `x86` platform to satisfy the virtualization requirements. Architectural supports are made to facilitate hypervisor design. A hypervisor can simply set up a series of sensitive events and transfer control to a VM. When an event occurs, the hypervisor will automatically regain control to do necessary operations and resume the VM again. As a result, the code base of hypervisor can be reduced. Moreover, the instructions inside VMs are running natively, thus resulting in reduced performance overhead. Besides that, HVM offers new features to prohibit DMA attacks. It can mark a region of memory as "no DMA" so any DMA-capable devices cannot access it.

To facilitate communication between VMs and hypervisor, HVM provides hypercall interface. A VM can invoke hypercall instructions to issue requests to hypervisor.

C. Trusted Computing

Trusted computing is a technology promoted by the Trusted Computing Group (TCG [16]). The terminology *trust* is defined as: *An entity can be trusted if it always behaves in the expected manner for the intended purpose.*

In particular, TCG has specified Trusted Platform Module (TPM [17]), a low-cost secure chip. It can be used to extend trusted region from trust root to necessary components by measurement. The trust root can be initialized by Dynamic Root of Trust for Measurement (DRTM [1], [9]) via invoking a special instruction with a hypervisor as parameter. DRTM will reset processors and chipset to a trusted/determined status. Next, DRTM will calculate hash value of the hypervisor (measurement) and store it into a well-known Platform Configuration Register (PCR) which is inside TPM's protected region. The register (pcr_{drtm} in this paper) will be locked and the hypervisor will gain the control. Therefore, pcr_{drtm} can be referred to as evidence of first execution component after DRTM. Later, other PCRs can be used to store measurements of additional components.

Furthermore, we can invoke a TPM command to generate key pairs with protection. The public part of the key will be returned in plaintext while the private part will be protected under a specified PCR value. This protection is called "seal" in TCG's terminology. For example, one can generate a key pair and seal it to $pcr_{drtm} = 1234$, resulting in a message containing the plain public part and the encrypted private part plus sealing information. To use the key, one should firstly load the key into TPM by passing the message. TPM internally can recover the original private part and know the key is sealed to $pcr_{drtm} = 1234$. A handle will be returned as a reference to the key. When one requests decryption or signature by the key, TPM will first check whether current pcr_{drtm} equals to 1234. The operation can be performed only if the check is passed. It further hints the hypervisor presented now is the one presented when the key was generated.

Above operations are all executed locally, to convince a remote party that a key is sealed to a certain PCR value, TPM offers reporting function. Each TPM is shipped with an unique Endorsement Key (EK) to certify it is a genuine TPM. To mitigate privacy issues, Attestation Identity Key (AIK), as an alias of EK, will be used to prove a message is generated from a genuine TPM. Once a key is loaded, a TPM command can be invoked to generate a certificate for the key. The certificate essentially contains the public part of the key and sealing information (e.g., $pcr_{drtm} = 1234$). An AIK will be employed to sign the certificate. Verifier then can check its validation by using AIK's public part. The loaded key can be released by unloading operation if not used.

D. Trustworthy Cloud

Currently, trusted computing becomes the corner stone to build a trusted Cloud. Users are able to know what software is running on the Cloud by measuring TCB.

In general, TCB refers to a set of hardware and software components that are critical to the security. If a vulnerability occurs in components inside TCB, security properties cannot be satisfied. On the contrary, bugs outside TCB will not harm the security.

When a trusted Cloud is built, we can know how TCB is constructed; however, whether a component inside TCB can meet its design is still unclear. For example, a hypervisor (e.g., Xen hypervisor) is supposed to achieve isolation between VMs. However, even knowing the hypervisor is faithfully running, we cannot infer whether the implementation of Xen hypervisor can correctly realize isolation as it asserts. The term to describe a component's behavior follows its design is called "trustworthiness". To convince users the Cloud is secure to deploy sensitive applications, it must be trusted and trustworthy.

Achieving trustworthiness is non-trivial. Conceptually, formal verification is principally the only method to make a proof of trustworthiness; however, the mathematical proof is costly. A study [8] indicates industry estimate for CC EAL6 certification (semiformal verification) is $10K per Line of Code (LOC). An alternative way is to perform manual inspection of source code. Both approaches are sensitive to the size of TCB which is typically estimated by LOC. In other words, reducing TCB size is necessary to build up a trustworthy Cloud.

For an end application on Cloud, its TCB can be divided into two parts–service provided by the Cloud and software (and platform) provided by user. Both Cloud-provided and user-provided TCBs must be reduced.

IV. Trusted Block as a Service

We propose Trusted Block as a Service (TBaaS) as a solution to SAND problem. The core idea is to provide a trusted block for each user to perform the sensitive application with security guarantees. Meanwhile, TCB will be significantly reduced.

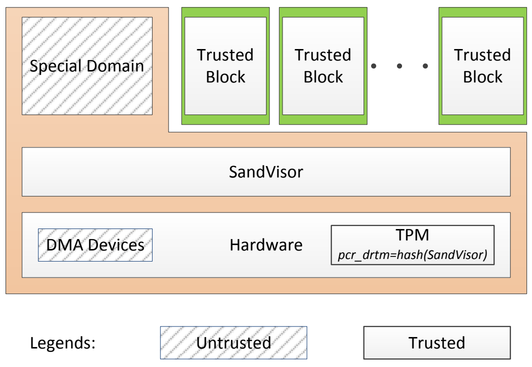

Figure 1. Trusted Block as a Service Architecture

The architecture of TBaaS is shown in Figure 1. The components provided by Cloud provider are hardware, SandVisor and a special domain while users provide sensitive applications in the trusted blocks. The isolation among different users is guaranteed by SandVisor, that is, our proposed tiny hypervisor whose code base is much smaller compared to general hypervisors. The sensitive application will be executed in the block whose environment consists of only virtual processors, memory and essential interfaces. TPM will be dominated by SandVisor exclusively. The special domain will manage devices, e.g., network adapter.

We will introduce principal features of TBaaS in this section and protocol details will be described in the next section.

A. Reduced TCB

As mentioned in Section 3, a trusted Cloud without trustworthiness is impractical. TBaaS introduces many methods to reduce TCB.

First of all, a tiny hypervisor called SandVisor is employed in TBaaS. It provides isolation and scheduling among VMs like a normal hypervisor. Different from the existing hypervisors, the SandVisor has very limited functionalities, thus resulting in a reduced code base. Only the most necessary hardware, such as processors and memory, will be virtualized to execute $foo(input)$. Benefit from HVM, the SandVisor can fully virtualize them without inflating its size. The physical TPM is protected and only accessible to the SandVisor. Other devices, such as network adapter, will be managed by a special domain instead, offering ability to communicate with Alice. Unlike Dom0 in Xen, the special domain in TBaaS is outside each trusted block's TCB. Any bug in the special domain will not break security requirements of confidentiality and verifiability.

Next, to further reduce user-provided TCB, SandVisor hides unnecessary architectural details for the trusted blocks. For example, identity mapping page table can be set up in advance; system description tables can be initialized with default values; external interrupts can be disabled and exceptions (e.g. divided by zero) can be handled by the SandVisor. The processing program foo now can run on a ready-to-use context thus eliminating the demand of Operating System (OS), which in turn dramatically reduces user-provided TCB.

B. Confidential Execution

As one requirement of SAND, TBaaS is able to securely deliver processing program foo and parameter $input$ to a trusted block and obtain return value $result$ without disclosing them to the special domain, other trusted blocks or even Cloud administrators. In other words, only Alice knows what is running on the Cloud. Cloud provider or a third party is thereby hard to violate user's privacy.

To this end, Alice and her trusted block should share a secret by establishing a secure channel via encrypted messages. However, how to generate the secret and safely distribute it to both parties without disclosing will be a challenge. Traditionally, Diffie-Hellman [7] protocol is widely used to exchange secret. However, this method suffers man-in-the-middle (MITM) attack.

To address this issue, we utilize new features provided by the trusted computing, i.e., creating a sealed key for each block. SandVisor is responsible for distinguishing each user and its block, selecting corresponding key for secure delivery. The private part will be sealed to pcr_{drtm} and a certificate will be issued to certify the sealed information. After verifing the certificate, Alice knows the key is sealed to SandVisor, thus can employ public encryption to securely transfer foo and $input$ to SandVisor. To encrypt $result$ without generating more keys, one-time pad is used.

C. Verifiable Result

As the other security requirement of SAND, it is necessary to know whether the Cloud is carrying out $result = foo(input)$ faithfully.

Our solution is to construct a digital digest of foo, $input$ and $result$. Since Alice owns the knowledge of genuine foo and $input$, as well as $result$ recovered by one-time pad, she can verify the digest by herself. If correct, Alice can safely conclude the computation is faithfully performed by the SandVisor.

D. High Flexibility and Low Performance Overhead

TBaaS is flexible since it sets a few limitations for foo. In general, any non-privileged instruction sequence can be used. Runtime library if required can be embedded in foo too. For applications that require network connection, Alice may only put the most sensitive part, e.g., authentication module, on the TBaaS Cloud and execute other parts on a second traditional Cloud with external communication enabled.

Every instruction of sensitive applications will run natively without emulation or translation, thus leading to low performance overhead. Additionally, removing OS also saves memory that can be used by trust blocks to run sensitive applications.

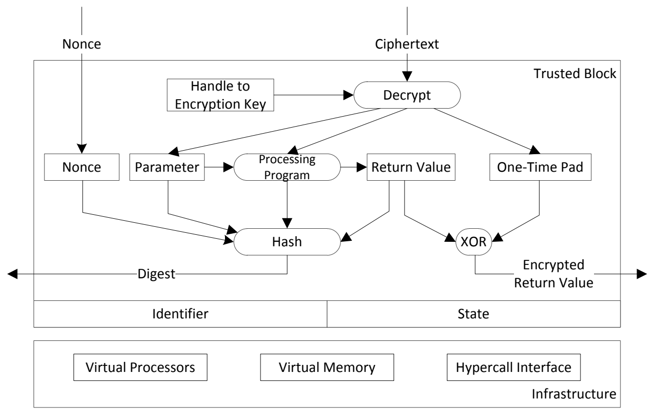

Figure 2. Trusted Block Architecture

TABLE I
NOTATIONS OF DATA AND OPERATOR

Symbol	Content
n	A nonce
otp	An one-time pad
k_x	Public part of a key pair x
k_x^{-1}	Private part of a key pair x
h_x	A handle to key x
$data_1 \| data_2$	Concatenation of $data_1$ and $data_2$
$data_1 \oplus data_2$	$data_1$ XOR $data_2$
$state_f$	State of the trusted block whose identifier is f
pcr_m	A PCR register. In this paper, pcr_{drtm} refers to the special PCR register that measures SandVisor. Its value will be $hash(SandVisor)$ after DRTM.
$hash(data)$	Digital digest of $data$
$sealedkey(k_x^{-1}, pcr_m)$	An encrypted private key k_x^{-1} which is sealed to pcr_m.
$keycert(k_x, pcr_m, n)$	Certificate of key x, stating its public part is k_x and its private part is sealed to pcr_m where n is a nonce.
$ciphertext(k_x, data)$	$data$ encrypted by k_x
$signature(k_x^{-1}, data)$	$data$ signed by k_x^{-1}

TABLE II
NOTATIONS OF COMMANDS

Symbol	Content
TPM_CreateEncKey(pcr_m)	will create a key pair x for encryption and seal its private part k_x^{-1} to pcr_m. Return k_x and $sealedkey(k_x^{-1}, pcr_m)$.
TPM_CertifyKey(h_x, n, h_a)	will generate a certificate signed by AIK K_a^{-1} to prove key pair x is created with certain sealing information where n is a nonce. Return $keycert(k_x, pcr_m, n)$ and $signature(k_a^{-1}, keycert(k_x, pcr_m, n))$.
TPM_LoadKey($k_x, sealedkey(k_x^{-1}, pcr_m)$)	will load key pair x to TPM. Return its handle h_x.
TPM_UnloadKey(h_x)	will unload key pair x from TPM.
CMD_Encrypt($k_x, data$)	will encrypt $data$ by k_x. Return $ciphertext(k_x, data)$.
TPM_Decrypt($h_x, ciphertext(k_x, data)$)	will decrypt $ciphertext(k_x, data)$ by k_x^{-1} referred to by h_x if current PCR value matches sealing information of k_x^{-1}. Return $data$.
CMD_Verify($k_x, signature(k_x^{-1}, data)$)	will verify whether $signature(k_x^{-1}, data)$ is signed by k_x^{-1}.

TABLE III
INITIAL KNOWLEDGE OF TBI

TPM	k_{aik}^{-1}	Private part of AIK aik
SandVisor	h_{aik}	Handle to AIK aik
Alice	k_{aik}	Public part of AIK aik
	$hash(SandVisor)$	SandVisor's digest
	foo	Processing program

V. DESIGN RATIONALE

In this section, we will present the design of TBaaS in detail. To facilitate demonstration, we introduce several notations in Table I and Table II for data, operations and commands respectively. The commands with prefix TPM_ will have equivalent TPM hardware implementations while the others can be completed by software.

In general, five phases are involved: TBaaS bootstrapping, trusted block initialization (TBI), processing program installation (PPI), sensitive application execution (SAE) and return value fetch (RVF).

Figure 2 shows trusted block architecture. Every block will be created by a TBI request. The state illustrates block's current condition and updates according to the state machine as shown in Figure 3. The identifier is used to distinguish a trusted block from others, generally equal to $hash(foo)$. The encryption key will be created for secure delivery in TBI phase. In PPI phase, processing program foo will be securely installed in the block. In SAE phase, parameter $input$ and one-time pad otp will be confidentially delivered to the block as well. Then, return value $result$ can be obtained by executing $foo(input)$. In RVF phase, otp will be used to encrypt return value while a digital digest will summarize foo, $input$, $result$ and nonce. Finally, Alice can recover $result$ and verify the digest. She may further input another parameter for next computation.

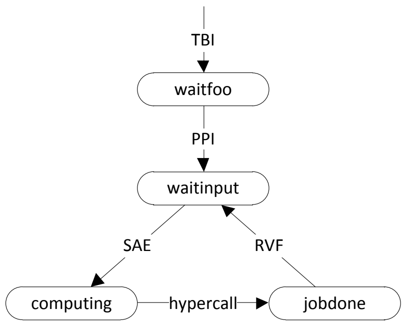

Figure 3. Finite State Machine of Trusted Block

We will describe the details of each phase. For simplicity, only the main steps will be explained in this paper.

TABLE IV
PROTOCOL OF TRUSTED BLOCK INITIALIZATION (TBI)

	Component	Content
1)	A→SV	Request of trusted block initialization, n_{a_1}, $hash(foo)$
2)	SV	Assign a trusted block with $hash(foo)$ as identifier. Calculate $hash(hash(foo)\|\|n_{a_1})$.
3)	SV→TPM	TPM_CreateEncKey(pcr_{drtm})
4)	SV←TPM	$k_{enc_{foo}}$, $sealedkey(k^{-1}_{enc_{foo}}, pcr_{drtm})$
5)	SV→TPM	TPM_LoadKey($k_{enc_{foo}}, sealedkey(k^{-1}_{enc_{foo}}, pcr_{drtm})$)
6)	SV←TPM	$h_{enc_{foo}}$
7)	SV→TPM	TPM_CertifyKey($h_{enc_{foo}}, hash(hash(foo)\|\|n_{a_1}), h_{aik}$)
8)	SV←TPM	$keycert(k_{enc_{foo}}, pcr_{drtm}, hash(hash(foo)\|\|n_{a_1}))$ $signature(k^{-1}_{aik}, keycert(k_{enc_{foo}}, pcr_{drtm}, hash(hash(foo)\|\|n_{a_1})))$
9)	SV→TPM	TPM_UnloadKey($h_{enc_{foo}}$)
10)	SV	$state_{foo} = waitfoo$
11)	A←SV	$keycert(k_{enc_{foo}}, pcr_{drtm}, hash(hash(foo)\|\|n_{a_1}))$ $signature(k^{-1}_{aik}, keycert(k_{enc_{foo}}, pcr_{drtm}, hash(hash(foo)\|\|n_{a_1})))$
12)	A	Assert CMD_Verify($k_{aik}, signature(k^{-1}_{aik}, keycert(k_{enc_{foo}}, pcr_{drtm}, hash(hash(foo)\|\|n_{a_1})))$) succeeds. Assert $pcr_{drtm} = hash(SandVisor)$. Assert $hash(hash(foo)\|\|n_{a_1})$ is correct.

A. TBaaS Bootstrapping

This phase is to establish the foundation of a TBaaS Cloud by trusted computing and virtualization, preparing essential environments for potential users. The phase is performed only once after power up.

It starts by DRTM to initiate the trust. Processors and chipset will be reset to a trusted state. SandVisor will be loaded into a special memory zone marked with "no DMA". A special PCR pcr_{drtm} (in practice, it would be a combination of specific PCRs) will be employed to measure SandVisor.

$$pcr_{drtm} = hash(SandVisor)$$

Once SandVisor is in charge, it will enable virtualization with proper settings. The physical TPM will be protected and only SandVisor can access it. The special domain will be loaded to manage other devices. The physical memory will be reserved as several regions for isolation: SandVisor, special domain and trusted blocks. The "no DMA" memory zone should be adjusted to cover SandVisor and trusted blocks.

Afterwards, a TBaaS Cloud has been established with SandVisor as hypervisor. The special domain is ready to response user requests.

B. Trusted Block Initialization

This phase aims to initialize a trusted block for a user, e.g., Alice. The block identifier will be set and an encryption key will be generated and sealed to SandVisor. Alice will verify whether the key is generated properly. The protocol involves three parties: Alice (A), SandVisor (SV) and TPM. Their initial knowledge is shown in Table III.

The TBI protocol is described in Table IV. It is important to note that in case a message is corrupted and an assertion fails, the protocol will abort immediately. The communications between SandVisor and Alice are actually performed via the special domain. Only these messages can be seen by untrusted parties. We will prove disclosing or even modifying these messages will not break secuirty requirements in Section VI.

TABLE V
INITIAL KNOWLEDGE OF PPI, SAE AND RVF

TPM	$k^{-1}_{enc_{foo}}$	Private part of encryption key for the block whose identifier is foo
SandVisor	$h_{enc_{foo}}$	Handle to enc_{foo}
Alice	foo	Processing program
	$input$	Parameter for foo
	otp	One-time pad for encryption of $result$

Alice firstly requests to initialize a trusted block with $hash(foo)$ as identifier (step 1). She also sends a nonce n_{a_1} to mitigate replay attack. SandVisor will then assign a trusted block to Alice and set its identifier to $hash(foo)$, as well as calculating $hash(hash(foo)||n_{a_1})$ (step 2). It will also perform necessary operations to configure its infrastructure. In particular, a hypercall, namely `JOBDONE`, will be registered to the SandVisor as a signal for completion of computation.

Next, SandVisor generates a key enc_{foo} as encryption key, along with its certificate (steps 3 to 9). It invokes a TPM command TPM_CreateEncKey to create a key pair for encryption and seal its private part to pcr_{drtm} which essentially measures SandVisor (steps 3 and 4). A certificate will be issued by invoking TPM_CertifyKey, proving the key is sealed to pcr_{drtm} (steps 5 to 8). This certificate will be signed by k^{-1}_{aik} with $hash(hash(foo)||n_{a_1})$ as nonce. After that, handle to the key will be released (step 9).

When Alice receives the key and its certificate from SandVisor, she will verify whether the certificate is signed by the genuine TPM as she expects (step 12). Moreover, she will check whether the PCR value the key sealed to is matching with her expected value $hash(SandVisor)$. She will also verify whether the nonce in certificate is $hash(hash(foo)||n_{a_1})$. If all correct, she can conclude the encryption key enc_{foo} is sealed to SandVisor, and can be used exclusively to encrypt messages to the trusted block whose identifier is $hash(foo)$ exclusively.

TABLE VI
PROTOCOL OF PROCESSING PROGRAM INSTALLATION (PPI)

	Component	Content
1)	A	CMD_Encrypt($k_{enc_{foo}}, foo$) returns $ciphertext(k_{enc_{foo}}, foo)$
2)	A→SV	Request of processing program installation, $ciphertext(k_{enc_{foo}}, foo)$, $hash(foo)$
3)	SV	Locate the block whose identifier is $hash(foo)$
		Assert $state_{foo} = waitfoo$
4)	SV→TPM	TPM_LoadKey($k_{enc_{foo}}, sealedkey(k^{-1}_{enc_{foo}}, pcr_{drtm})$)
5)	SV←TPM	$h_{enc_{foo}}$
6)	SV→TPM	TPM_Decrypt($h_{enc_{foo}}, ciphertext(k_{enc_{foo}}, foo)$)
7)	SV←TPM	foo
8)	SV→TPM	TPM_UnloadKey($h_{enc_{foo}}$)
9)	SV	Assert $hash(foo)$ equals to identifier of the block.
		SandVisor loads foo as processing program in the trusted block located above.
		$state_{foo} = waitinput$

TABLE VII
PROTOCOL OF SENSITIVE APPLICATION EXECUTION (SAE)

	Component	Content
1)	A→SV	Request of sensitive application execution
2)	A←SV	n_{sv_1}
3)	A	CMD_Encrypt($k_{enc_{foo}}, input$) returns $ciphertext(k_{enc_{foo}}, input)$.
		CMD_Encrypt($k_{enc_{foo}}, otp$) returns $ciphertext(k_{enc_{foo}}, otp)$.
4)	A→SV	$ciphertext(k_{enc_{foo}}, input)$, $ciphertext(k_{enc_{foo}}, otp)$
		$hash(foo)$, $hash(foo \| n_{sv_1})$, $hash(foo \| input)$, $hash(foo \| otp)$
5)	SV	Locate the block whose identifier is $hash(foo)$.
		Assert $state_{foo} = waitinput$.
		Assert $hash(foo \| n_{sv_1})$ is correct.
6)	SV→TPM	TPM_LoadKey($k_{enc_{foo}}, sealedkey(k^{-1}_{enc_{foo}}, pcr_{drtm})$)
7)	SV←TPM	$h_{enc_{foo}}$
8)	SV→TPM	TPM_Decrypt($h_{enc_{foo}}, ciphertext(k_{enc_{foo}}, input)$)
9)	SV←TPM	$input$
10)	SV→TPM	TPM_Decrypt($h_{enc_{foo}}, ciphertext(k_{enc_{foo}}, otp)$)
11)	SV←TPM	otp
12)	SV→TPM	TPM_UnloadKey($h_{enc_{foo}}$)
13)	SV	Assert $hash(foo \| input)$ is correct.
		Set $input$ as parameter in the trusted block located above.
		Assert $hash(foo \| otp)$ is correct.
		Set otp as one-time pad in the trusted block located above.
14)	SV→TB	Transfer control to the trusted block.
15)	TB	Calculate $result = foo(input)$.
	TB	Set $result$ as a return value.
16)	SV←TB	Invoke hypercall `JOBDONE`.
17)	SV	$state_{foo} = jobdone$

TABLE VIII
PROTOCOL OF RETURN VALUE FETCH (RVF)

	Component	Content
1)	A→SV	Request of return value fetch
2)	A←SV	n_{sv_2}
3)	A→SV	$hash(foo)$, $hash(foo \| n_{sv_2})$, n_{a_2}
4)	SV	Locate the block whose identifier is $hash(foo)$
		Assert $state_{foo} = jobdone$
		Assert $hash(foo \| n_{sv_2})$ is correct
5)	SV	Obtain $result$ from return value in the block located above.
6)	A←SV	$result \oplus otp$, $hash(result \| foo \| input \| n_{a_2})$
7)	SV	$state_{foo} = waitinput$
8)	A	Recover $result = (result \oplus otp) \oplus otp$.
		Assert $hash(result \| foo \| input \| n_{a_2})$ is correct.

C. Processing Program Installation

This phase is going to confidentially install processing program foo to the trusted block. The protocol is detailed in Table VI with initial knowledge in Table V.

Since the private part of encryption key enc_{foo} is sealed to SandVisor, Alice can deliver sensitive messages by this key. She will encrypt processing program foo by $k_{enc_{foo}}$ and send the ciphertext to SandVisor without risks of disclosure.

After receives the request of PPI, the SandVisor will try to locate the block whose identifier is $hash(foo)$ and assert

TABLE IX
VULNERABLE MESSAGES

Phase	Step	Message
TBI	1)A→SV	Request of TBI, n_{a_1}, $hash(foo)$
	11)A←SV	$keycert(k_{enc_{foo}}, pcr_{drtm}, hash(hash(foo) \Vert n_{a_1}))$, $signature(k_{aik}^{-1}, keycert(\ldots))$
PPI	2)A→SV	Request of PPI, $ciphertext(k_{enc_{foo}}, foo)$, $hash(foo)$
SAE	1)A→SV	Request of SAE
	2)A←SV	n_{sv_1}
	4)A→SV	$ciphertext(k_{enc_{foo}}, input)$, $ciphertext(k_{enc_{foo}}, otp)$, $hash(foo)$, $hash(foo \Vert n_{sv_1})$, $hash(foo \Vert input)$, $hash(foo \Vert otp)$
RVF	1)A→SV	Request of RVF
	2)A←SV	n_{sv_2}
	3)A→SV	$hash(foo)$, $hash(foo \Vert n_{sv_2})$, n_{a_2}
	6)A←SV	$result \oplus otp$, $hash(result \Vert foo \Vert input \Vert n_{a_2})$

current state of the block is $waitfoo$ (steps 1 to 3). If passed, the SandVisor will load $k_{enc_{foo}}^{-1}$ to recover processing program foo from the ciphertext (steps 4 to 8).

Once foo is revealed, SandVisor will calculate $hash(foo)$ locally. If it matches block's identifier, SandVisor will load foo as processing program in the block and update $state_{foo}$ to $waitinput$ (step 9).

D. Sensitive Application Execution

This phase aims to confidentially deliver parameter $input$ and one-time pad otp to the trusted block and perform calculation $result = foo(input)$. The trusted block (TB) will be involved in the interactive protocol as shown in Table VII.

Alice sends a request of SAE to SandVisor and gets a nonce back (steps 1 and 2). Using the same method employed in PPI, she will encrypt $input$ and otp by $k_{enc_{foo}}$, and send the ciphertexts along with $hash(foo)$, $hash(foo \| input)$ and $hash(foo \| otp)$ to SandVisor (steps 3 and 4). SandVisor will locate the trusted block and verify its state (step 5). SandVisor will further load $k_{enc_{foo}}^{-1}$ to recover $input$ and otp from the ciphertexts (steps 6 to 11). If $hash(foo \| input)$ and $hash(foo \| otp)$ are correct, $input$ and otp will be set as parameter and on-time pad (step 13). After that, computation of $result = foo(input)$ will start (steps 14 and 15).

When foo finishes computation, it will invoke a hypercall named `JOBDONE` (step 16). As a result, SandVisor will regain control and update the state to $jobdone$ (step 17).

E. Return Value Fetch

The last phase is for Alice to fetch and verify the return value $result$. The protocol is described in Table VIII.

Similarly, Alice sends a request of RVF and receives a nonce (steps 1 and 2). She will send $hash(foo)$ as identifier, digests of foo and nonce received, and a new nonce n_{a2} (step 3). SandVisor will then locate the trusted block and check its state (step 4). If passed, $result$ will be fetched from the trusted block (step 5) and encrypted by one-time pad otp (step 6). In the meantime, a digital digest $hash(result \| foo \| input \| n_{a_2})$ will be also generated (step 6). The state of the block will be set back to $waitinput$ again (step 7).

When $result \oplus otp$ and the digest arrive, Alice can recover $result$ by XOR with otp and verify the digest (step 8). If passed, she can conclude the computation is carried out faithfully.

VI. SECURITY ANALYSIS

Since security is our top concern, we will provide informal verification of the protocols in this section. Basic assumptions will be given and two security requirements, i.e., confidentiality and verifiability will be proved. In particular, we summarize all messages exposed to potential attackers who can be classified into passive attackers and active attackers. The former can eavesdrop these messages only while the latter can alter them. We will analyze passive and active attacks separately and prove neither can break the security requirements.

A. Assumptions

We list several assumptions here as basis of the following proofs.

Assumption 1. *It is computationally infeasible to break cryptographic system. In particular, one cannot recover k_x^{-1} by given k_x; and one cannot find data such that $h = hash(data)$ by given h and $hash$.*

Assumption 2. *SandVisor is trustworthy. If SandVisor is executed as hypervisor, design properties, such as isolation, can be guaranteed; and protocols in all phases can be faithfully carried out.*

Assumption 3. *If $pcr_{drtm} = hash(SandVisor)$ appears on a genuine TPM, one can conclude DRTM is faithfully performed and the hypervisor is SandVisor.*

Assumption 4. *If a key certificate is signed by k_{aik}^{-1}, one can conclude the key is generated by a genuine TPM which holds k_{aik}^{-1}. Its private part is sealed to the PCR that the certificate asserts.*

Assumption 5. *Alice will faithfully perform the protocols.*

B. Summary of Vulnerable Messages

As mentioned in last section, only the messages transferred between Alice and SandVisor are vulnerable to potential attacks. We summarize such messages in Table IX. In the following sections, we use Phase-Step to denote a message group or operations. For example, SAE-1 refers to the message "Request of SAE".

C. Passive Attack

For passive attack, the adversary (Eve) knows all messages in Table IX but cannot modify any of them. Apparently, if every message is unchanged, Alice can go through the protocol thus successively verifying $result$. Thereby, Eve cannot break the requirement of verifiability. For confidentiality, Eve may try to recover foo, $input$ and/or $result$ from these messages.

First, requests of protocols (TBI-1, PPI-2, SAE-1 and RVF-1), nonces (TBI-1, SAE-2, RVF-2 and RVF-3) and signature

(TBI-11) are totally irrelevant to secrets. Based on Assumption 1, the adversary cannot recover original message by given hash value (TBI-1, PPI-2, SAE-4, RVF-3 and RVF-6). To recover foo from $ciphertext(k_{enc_{foo}}, foo)$ (PPI-2), Eve has to know $k^{-1}_{enc_{foo}}$ which cannot be revealed from $k_{enc_{foo}}$ (TBI-11). Likewise, Eve cannot recover $input$ or otp in SAE-4. Moreover, recovering $result$ from $result \oplus otp$ in RVF-6 requires knowledge of otp which is unknown to Eve.

Consequently, passive attack cannot violate requirements of confidentiality and verifiability.

D. Active Attack

For active attack, the adversary (Mallet) is able to modify the messages before it reaches its destination.

First of all, Mallet cannot modify the structure of messages; otherwise protocols will be aborted. He also has to survive assertions to avoid detections.

For TBI, forging the signature of key certificate (TBI-11) requires knowledge of $k^{-1}_{enc_{foo}}$ which cannot be acquired. Therefore, Mallet cannot alter $k_{enc_{foo}}$, pcr_{drtm} and/or n_{a_1} in the key certificate or Alice can detect the integrity of certificate is broken in TBI-12. He cannot modify n_{a_1} in TBI-1 either; otherwise Alice can detect the nonce in certificate is inconsistent with her own in TBI-12. If he forges a processing program bar and modifies $hash(foo)$ to $hash(bar)$, Alice will receive $keycert(k_{enc_{bar}}, pcr_{drtm}, hash(hash(bar)\|n_{a_1}))$ and the check on $hash(hash(bar)\|n_{a_1})$ will be failed. He may construct n_{m_1} and $hash(bar)$ such that $hash(hash(foo)\|n_{a_1}) = hash(hash(bar)\|n_{m_1})$ to pass the assertion. However, this violates Assumption 1. As a result, Mallet cannot replace any message in TBI without being detected. The identifier of the trusted block is set as $hash(foo)$.

For PPI, Mallet cannot modify $hash(foo)$; otherwise SandVisor will not locate Alice's trusted block and thus the protocol will be aborted. If he alters $ciphertext(k_{enc_{foo}}, foo)$ to $ciphertext(k_{enc_{foo}}, bar)$, SandVisor will recover bar in PPI-7 and assertion in PPI-9 will be failed since $hash(foo) \neq hash(bar)$. No message in PPI can be changed without being detected and genuine foo will be installed.

Similarly, $ciphertext(k_{enc_{foo}}, input)$ and $ciphertext(k_{enc_{foo}}, otp)$ in SAE-4 cannot be altered. Without knowledge of foo, Mallet cannot construct forged n_{sv_1} and $hash(foo\|n_{sv_1})$ to pass assertion in SAE-5, or construct forged $hash(foo\|input)$ and $hash(foo\|otp)$ to pass assertion in SAE-13. Again, he cannot alter $hash(foo)$ which is used to locate Alice's trusted block. In other words, no message in SAE can be modified without being detected and genuine $input$ and otp are set inside the trusted block. $result$ can be faithfully computed with foo as processing program and $input$ as parameter.

For the last phase RVF, forged n_{sv_2} in RVF-2 and/or $hash(foo\|n_{sv_2})$ in RVF-3 again cannot pass assertion in RVF-4. Changing $hash(foo)$ will result in abortion of protocol since SandVisor cannot locate the trusted block. If Mallet forges n_{a_2} in RVF-3, $result \oplus otp$ and $hash(result\|foo\|input\|n_{a_2})$ in RVF-6 to n_{m_2}, $forgedresultotp$ and $forgedhash$ to pass assertion in RVF-8, equation $hash((forgedresultotp \oplus otp)\|foo\|input\|n_{m_2}) = forgedhash$ must be satisfied. However, this requires knowledge of otp, foo and $input$ that are all unknown to Mallet. Therefore, no message can be replaced without being detected.

In sum, any modification of message can be detected in different phases. It prohibits possible active attacks and requirements of confidentiality and verifiability are satisfied.

VII. Related Work

How existing research has the potential to mitigate concerns on controlling data in the Cloud is analyzed in [4]. It mentions major corporations only put their less sensitive data in the Cloud because of the lack of control. Moreover, it indicates the core issue that prevents the execution of potential applications on the Cloud is that Cloud providers have some control of the users' data. The authors conclude that current control methods do not adequately address Cloud computing's needs. To improve the situation, they propose to extend control measures from the user to the Cloud through trusted computing and applied cryptographic approaches, in order to limit Cloud provider's access of data. But, this survey mainly focuses on macro level and does not give concrete solutions.

As mentioned in Section 1, Xen has a large TCB that includes hypervisor and Dom0. Derek G. Murray *et al* [12] propose a method to disaggregate Dom0 in Xen. In their paper, they explain why Dom0 is a part of TCB and how to remove Dom0's user-space code from TCB. In particular, the new partition transfers the VM-building functionality into a small trusted VM, DomB. As a result, TCB only includes kernel, DomB and hypervisor in disaggregated Xen system. The evaluation compares the existing and disaggregated Xen 3.1 system. 920, 15 and 160 KLOC in C, Assembly and Python are removed from TCB while only 9.2 and 0.5 KLOC in C and Assembly are added to TCB for DomB. However, TCB of disaggregated Xen is till too large and therefore, it is arguable to say disaggregated Xen can be trustworthy.

F. John Krautheim *et al* [10] propose a Trusted Virtual Environment Module (TVEM) for rooting trust in Cloud computing. In particular, the virtual trust is a combination of both information owners trust and host platform trust. To achieve it, the authors implement TVEM as a software appliance for TPM virtualization. To support migration, Migratable Storage Key (MSK) is employed to allow virtual TPM transfer from one platform to another. However, TVEM is designed as a new TPM virtualization approach over traditional IaaS, thus depending on a big TCB which makes it impossible to be trustworthy. Moreover, TVEM's dual-root trust relies on social trust (reputation of Cloud provider) to protect information security, which is practically unreliable. Therefore, TVEM cannot be an efficient solution to SAND problem.

Some work focuses on utilizing HVM to alleviate security problems. SecVisor [14] is a tiny hypervisor to maintain kernel integrity. The solution is based on AMD platform to trap transitions between user space and kernel space. Necessary protections are applied to eliminate improper memory access before committing the transition. If combined with Nested Paging Technology (NPT) [2], the total TCB is only 3526 LOC. However, SecVisor requires modified OS to cooperate. BitVisor [15] is a pass-through hypervisor to particularly enhance I/O security, such as ATA device encryption. It will intercept data transferring through ATA host controller and encrypt/decrypt it by Advanced Encryption Standard (AES). The TCB of BitVisor is around 20K LOC. TrustVisor [11] is another hypervisor aiming to protect the execution of sensitive code on legacy systems. It executes security-sensitive code, i.e., Pieces of Application Logic (PAL), in an isolated environment from the legacy system. It also provides a reduced virtual TPM for each PAL to facilitate multiple PAL instances. TrustVisor's TCB is around 6K LOC. This work is similar to our SandVisor but mainly focusing on trusted computing on legacy systems rather than the Cloud computing security. For example, TrustVisor performs cryptographic operations on software because most physical TPMs used on legacy system are slow. However, the cryptographic module (mainly on RSA) introduces 2K-3K LOC. On the contrary, SandVisor executes almost every cryptographic operation on the physical TPM. It is because for Cloud computing, we can reasonably expect a high-performance TPM installed on platform to mitigate the issues.

VIII. Conclusion and Future Work

This paper proposed Trusted Block as a Service (TBaaS) to solve the sensitive application on the Cloud (SAND) problem. The requirements of SAND are defined as confidentiality and verifiability, as well as high flexibility and low performance overhead. TBaaS satisfies these requirements by offering each user a trusted environment to confidentially deploy the sensitive application, along with the ability to verify the computing result. Moreover, the sensitive application can be executed natively without major limitations.

The difference between "trusted" and "trustworthy" is discussed and reducing TCB becomes one critical guideline for TBaaS design. A tiny hypervisor, i.e., SandVisor, is employed to provide simplified infrastructure for sensitive applications execution. Functionality of managing devices such as network adapter is offloaded to a special domain which is outside TCB. Sophisticated protocols for trusted block initialization, processing program installation, sensitive application execution and return value fetch are introduced and informally verified in order to satisfy the security requirements of SAND. We hope TBaaS can be a paradigm to address SAND problem thus releasing the potential power of the Cloud computing.

Currently, a prototype of TBaaS is working in progress. It will be helpful to evaluate the exact TCB size and performance behavior. We will also investigate how to achieve trusted block migration while keeping TCB small.

References

[1] *Secure Virtual Machine Architecture Reference Manual*, AMD, http://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf.

[2] *AMD-V ^{TM}Nested Paging*, AMD, 1976.

[3] S. Bansal and A. Aiken, "Binary translation using peephole superoptimizers."

[4] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina, "Controlling data in the cloud: outsourcing computation without outsourcing control," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, ser. CCSW '09, 2009, pp. 85–90.

[5] "Xen Hypervisor," Citrix Systems, http://www.xen.org/products/xenhyp.html.

[6] *Top Threats to Cloud Computing V1.0*, Cloud Security Alliance, http://www.cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf.

[7] W. Diffie and M. E. Hellman, "New directions in cryptography," 1976.

[8] G. Heiser, "Trusted⇐Trustworthy⇐Proof," Open Kernel Labs and NICTA and University of New South Wales, 2008.

[9] *Intel Trusted Execution Technology Measured Launched Environment Developer's Guide*, Intel, http://download.intel.com/technology/security/downloads/315168.pdf.

[10] F. J. Krautheim, D. S. Phatak, and A. T. Sherman, "Introducing the trusted virtual environment module: a new mechanism for rooting trust in cloud computing," in *Proceedings of the 3rd international conference on Trust and trustworthy computing*, ser. TRUST'10, 2010, pp. 211–227.

[11] 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 the IEEE Symposium on Security and Privacy*, May 2010.

[12] D. G. Murray, G. Milos, and S. Hand, "Improving xen security through disaggregation," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '08, 2008, pp. 151–160.

[13] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974.

[14] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 335–350, October 2007.

[15] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "Bitvisor: a thin hypervisor for enforcing i/o device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '09, 2009, pp. 121–130.

[16] "Trusted Computing Group," Trusted Computing Group, http://www.trustedcomputinggroup.org/.

[17] *Trusted Platform Module Main Speci.cation. Version 1.2, Revision 103*, Trusted Computing Group, 2003.

[18] D. A. Wheeler, "SCLOCCount," http://www.dwheeler.com/sloccount/.