VERICOUNT: Verifiable Resource Accounting Using Hardware and Software Isolation

Shruti Tople   Soyeon Park[⋆]   Min Suk Kang   Prateek Saxena

National University of Singapore
{shruti90, kangms, prateeks} @comp.nus.edu.sg

[⋆]Georgia Tech
spark720@gatech.edu

Abstract. In cloud computing, where clients are billed based on the consumed resources for outsourced tasks, both the cloud providers and the clients have the incentive to manipulate claims about resource usage. Both desire an accurate and verifiable resource accounting system, which is neutral and can be trusted to refute any disputes. In this work, we present VERICOUNT — a verifiable resource accounting system coupled with refutable billing support for Linux container-based applications. To protect VERICOUNT logic, we propose a novel approach called *self-accounting* that combines hardware-based isolation guarantees from trusted computing mechanisms and software fault isolation techniques. The self-accounting engine in VERICOUNT leverages security features present in trusted computing solutions, such as Intel SGX, to measure user CPU time, memory, I/O bytes and network bandwidth while simultaneously detecting resource usage inflation attacks. We claim three main results. First, VERICOUNT incurs an average performance overhead of 3.62% and 16.03% over non-accounting but SGX-compatible applications in hardware and simulation mode respectively. Next, it contributes only an additional 542 lines of code to the trusted computing base. Lastly, it generates highly accurate, fine-grained resource accounting, with no discernible difference to the resource measuring tool available with the OS.

1 Introduction

Verifiable resource accounting is a security primitive that checks whether the measured resource accounting (e.g., CPU cycles, memory, network bandwidth, or I/O resources) of cloud computing infrastructure is accurate for an outsourced computing task. In today's "pay-as-you-use" model of cloud computing, where clients are billed based on the usage of the computing resources, verifiable resource accounting is increasingly desirable [12]. As the accounting result determines the final bill, both a cloud provider and a client have strong motivation to manipulate the results in favor of their economic interests; e.g., cloud providers overcharge clients or clients try to be undercharged. This demands for a *refutable* billing system where denying charges is possible based on a decision from a neutral backdrop. In the last decade, serious concerns have been raised

[⋆] Research done when visiting National University of Singapore about the billing problem in an untrusted cloud model [12, 15, 34, 26]. Several attacks such as mis-attribution of resources, false accounting, tampering execution to increase resource utilization have been demonstrated in presence of a malicious operating system (OS) [22, 32, 38, 31]. This is particularly hazardous for widely accepted container-based virtualization techniques, such as Docker [3] where resources are shared at finer granularity than virtual machines and thus accurate accounting is more challenging. Worse yet, even cloud providers seem to be struggling to implement accurate fine grained accounting and safe billing systems. Research has shown bugs in EC2 that lead to free CPU time and over-charging for storage in Rackspace [18]. Given the indisputable necessity of fairness in billing, we ask *whether it is to build a refutable billing system for cloud computing that allows significant security assurance?*

Currently, many OSes offer resource accounting features (e.g., `cgroups`); yet, such OS-based resource accounting mechanisms in commercial clouds (e.g., Amazon EC2 [2]) are not ideal due to their large TCB and attack surface. We discuss a class of attacks called *resource usage inflation* that a malicious OS can perpetrate to overcharge clients in Section 3.1. Research has demonstrated isolating resource accounting from the untrusted OS [28, 13]. In particular, Alibi [13] utilizes nested virtualization and Trusted Platform Modules (TPMs) to implement an observer placed at the hypervisor layer. However, Alibi includes a huge TCB (entire Linux kernel and KVM) for accounting. In this paper, we present VERICOUNT, a verifiable resource accounting system that accounts for four major computing resources used for executing outsourced computing tasks within secure containers (e.g., [10, 17, 30]). These containers ensure secure execution of applications assuming trusted computing solutions. VERICOUNT guarantees untampered resource accounting operations while allowing the clients and the cloud providers to explicitly establish a pre-agreed policy (e.g., maximum recoveries from crashes) for the execution and reports any violation of the policy. This eventually enables a *refutable billing* model which is a desirable feature in today's cloud computing.

At its core, VERICOUNT aims to implement strong isolation of the resource accounting logic from *both* the underlying OS and the client-submitted applications. However, it is challenging because the accounting logic is easily dependent either on the OS or the client applications based on where it is placed in the system. In VERICOUNT, we address this system dependency problem by combining both *hardware and software isolation* techniques. We isolate the OS and other privilege code using hardware isolation supported by trusted computing mechanisms (such as Intel SGX [1]), and implement sandboxing mechanisms for untrusted client applications. First, we show that a novel system architecture, which we call *self-accounting*, coupled with an execution policy enforcement provides strong independence of VERICOUNT's accounting logic from the underlying OS. Self-accounting lets the client application perform its own resource accounting efficiently *within* the same isolated memory region during its execution and thereby identify false accounting. Second, to ensure that the accounting logic is independent of untrusted client applications, VERICOUNT sandboxes the client-code using software fault isolation [33, 19, 14, 24]. A second challenge is to design an efficient yet accurate self-accounting approach. Basic approach of generating huge execution logs at runtime and verifying them later are expensive in terms of performance and verification effort [16]. To address this, we investigate an alternate way to effectively detect attacks that manipulate resource usage. We explore several trusted features that recent SGX-enabled CPUs support and leverage them to design accurate resource accounting system. Our solution exhibits desirable properties such as low performance overhead, low verification effort and a small trusted computing base (TCB).

System & Results. We build a proof-of-concept implementation for our design and evaluate it on SPEC CPUINT 2006 Benchmarks and H2O web server. Our prototype adds 542 source lines of code (SLoC) to the TCB and is simple enough to be formally verified later. We observe that VERICOUNT-enabled applications incur an average performance overhead of $\mathbf{3.62\%}$ and $\mathbf{16.03\%}$ as compared to non-accounting SGX applications in hardware and simulation mode respectively.

Contributions. We outline our main contribution below:

– *Self-Accounting* - Our novel approach of self-accounting lets each application account for its own resources, while employing hardware and software isolation.
– VERICOUNT *System* - VERICOUNT system consists of a compiler, a static verifier and a post-execution analyzer to guarantee verifiable accounting and refutable billing.
– *Evaluation* - We evaluate our prototype of VERICOUNT for performance overhead and accuracy of SPEC CPUINT 2006 Benchmarks and H2O web-server.

2 Problem Definition

Hosting containerized (docker-based) applications on the cloud is gaining popularity. Securing such applications is shown to be possible using trusted computing mechanism [10]. In this work, we support an additional primitive of verifiable resource accounting.

2.1 Threat Model

Our is the first work to consider two different adversary models simultaneously: a malicious cloud provider and a malicious client. Both adversaries have strong motivation to manipulate the resource accounting information in favor of their economic interests. A malicious provider receives a task of executing an application A from a client and aims to manipulate the resource usage summary R_A, to *increase* the final bill $\mathcal{B}_A$ of the task; i.e., overcharging the client. We consider that the malicious provider has full control over the operating system (OS), which allows the attacker to access any system resource that the OS controls and break any security mechanism that rely on it (e.g., process isolation, access control in reference monitor, shown in Section 3.1). At the same time, a malicious client aims to manipulate the resource usage summary R_A to *decrease* the bill $\mathcal{B}_A$; i.e., being undercharged. We consider that the malicious client has full control over the application code that are submitted for the outsourced computation. Both the provider and the client must trust VERICOUNT components and SGX.

Scope. We consider that the two adversaries would not collude, as they have contradictory goals. Moreover, the execution of A is strictly constrained to the input provided by the client. This restriction is necessary for verifiable accounting since it is in general impossible to define the notion of correct resource accounting between a client and a provider when applications expect to run with arbitrary inputs due to the undecidable problem. We do not consider denial-of-service attacks caused by arbitrary inputs to the application A; i.e., VERICOUNT does not detect resource usage manipulation if the cloud provider can generate valid inputs to the application.

2.2 Problem Statement

Verifiable resource accounting has three protocol steps between a client and a provider. First, a client and a provider agree on an *execution policy* $\phi = (\mathtt{p}, \mathtt{c}, \mathtt{t})$, where ϕ is a tuple of three parameters: a per-resource pricing scheme ($\mathtt{p}$), a crash recovery limit ($\mathtt{c}$), and the maximum OS response time ($\mathtt{t}$). The per-resource pricing scheme $\mathtt{p}$ includes the unit price for CPU, memory, I/O operations and network bandwidth usage. The crash recovery limit $\mathtt{c}$ is an integer number that permits the cloud provider to recover a crashed instance without informing the client. The maximum OS response time $\mathtt{t}$ is the time that an OS requires to respond to a service request from the application. Second, the client sends an application A along with an authenticated input I for the execution.

After the provider executes A, the accounting logic generates the resource usage summary R_A and the final bill $\mathcal{B}_A$ and sends them to the client for verification. $\mathcal{B}_A$ is calculated with the knowledge of R_A and ϕ. We seek the following security properties.
a) **Isolation from compromised OS.** A compromised OS cannot interfere with the resource accounting operation for any client-submitted application.
b) **Isolation from malicious client application.** A maliciously generated client application cannot tamper with the resource accounting information.
c) **Verifiable execution policy.** At the end of outsourced computation, a client and a provider can efficiently check any violation of the pre-agreed execution policy; i.e., confirm whether $\mathcal{B}_A \leftarrow (\phi, R_A)$.

Moreover, our verifiable resource accounting offers three desirable properties:
1) *Low performance overhead.* A VeriCount-enabled application should incur low performance overhead. The advantage of verifiable resource utilization should not unacceptably slow down as compared to the original application.
2) *Low verification overhead.* Verifying resource utilization should not require a client or a cloud provider to spend large resources for either repeating the outsourced execution or accessing huge logs of execution process [16]. This is critical for clients who outsource their computations to a remote cloud due to their insufficient local resources.
3) *Small TCB.* The solution should have small trusted software base, beyond that is implied by use of SGX, to avoid bugs that are present in large software.

Assumptions. We assume cloud providers support SGX-enabled CPUs and SGX guarantees are preserved throughout the execution lifetime. We assume that all hardware chip-sets are not malicious and do not contain backdoor that would violate the isolation properties of our verifiable resource accounting [37]. We assume no side-channels in the hardware architecture of the cloud provider. Attacks exploiting side-channels are outside the scope of the present work [35, 29, 27, 20].

3 Baseline Approaches and Attacks

Previous solutions have proposed using an external observer for resource accounting [13]. We discuss a baseline with a similar approach and discuss attacks on it.

Baseline Solution. A straightforward approach for verifiable resource accounting is to isolate the resource accounting engine from the underlying OS. Figure 1 (b) shows the design of such a baseline approach. One can use any trusted computing mechanism such as TPM or Intel SGX and port the resource accounting engine to a secure container using an existing system [10, 30]. Compared to the existing resource accounting

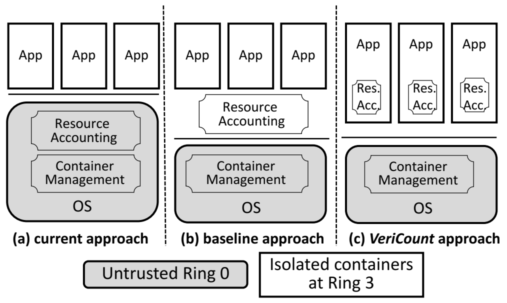

Fig. 1: Three isolation approaches: a) current approach, b) baseline approach, and c) VERICOUNT approach. The shaded region is untrusted components and non-shaded is isolated container.

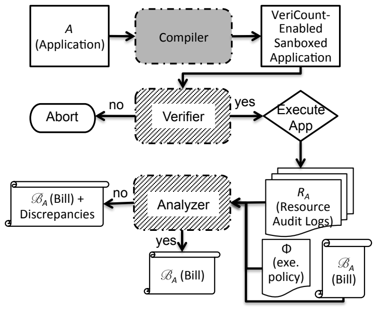

Fig. 2: Workflow of VERICOUNT system. The verifier and the analyzer (hatched) are trusted whereas the compiler (shaded) is untrusted.

architecture in Figure 1 (a), where the accounting engine resides in the underlying OS, the accounting engine in Figure 1 (b) is isolated and acts as an external observer and accounts the resource utilization for each secure containerized application. Compromising the OS and gaining privilege access does not enable the attacker to directly tamper with the accounting information. This baseline solution ensures accurate attribution of resource utilization as it eliminates any direct method of attacking the accounting system. However, we show that there still exists indirect dependency, which we call the execution dependency, on the underlying OS. The adversary can easily influence the execution operations to inflate the resource consumption.

3.1 Resource Usage Inflation Attacks

Although the baseline places the accounting engine within a secure container, the adversary can increase the resource usage of containers in the absence of support for refuting spurious charges. We discuss these resource usage inflation (RUI) attacks below.

Invoking Multiple Container Instances. The underlying OS is responsible for launching the container with the application on request from a client. Although the client requests to launch a single instance of the application, the OS can execute multiple instances of the same container. This results in inflated resource consumption corresponding to client's container. The accounting system incorrectly attributes the resources utilized by the unrequested instances to the client. In the absence of verifiable accounting, the client and the cloud provider have no way to refute disputed claims.

Replaying Inputs. The OS can replay the given input and increase the utilization of resources for the particular container. Note that the adversary cannot generate new set of valid inputs and hence is limited to replaying existing inputs arbitrary number of times. This inflates the resource consumption causing overcharging.

Arbitrary Halts. The application may experience unexpected crashes during its execution and the client is supposed to be informed about it so that she can request for starting a new execution. The malicious OS can exploit this property to silently crash an instance and restart it arbitrary number of times. The OS can forcefully halt the execution of the container before completion. The accounting engine being unaware of the malicious OS intention accounts the resources utilized for all the crashed instances and bills the client for the inflated resource usage.

Slowing Down OS Service. The application depends on the OS for several services like system calls, interrupts and others. If the user is charged based on the total time that the container is up and running then the OS may maliciously delay to execute the requested service. Thus, increasing the amount of time utilized by the particular container [22].

3.2 Towards Self-Accounting

All these RUI attacks demonstrate that the malicious OS has several ways to increase resource consumption even when the accounting engine is isolated as shown in Figure 1 (b). Note that the baseline approach places the accounting engine outside every application and thus cannot detect such malicious execution strategies by the OS. One can consider establishing inter-container communication channels between applications and the isolated accounting engine to address the RUI attacks; e.g., every I/O operation, user-kernel context switch or network usage is performed via the isolated resource accounting engine. Such an approach would incur prohibitively large overhead due to continuous IPC involved. Thus, we propose a novel *self-accounting* approach where each accounting engine runs alongside the client application within the container as shown in Figure 1 (c). That is, the accounting engine is tied with the atomic execution unit, which is the application itself, removing the execution dependency on the OS. The operation of the accounting engine is always executed with its application and thus the malicious execution strategies presented above cannot be effective. Thus, shifting the accounting engine from Ring O to Ring 3 removes the execution dependency from the OS and provides protection against an adversary perpetrating RUI attacks.

Client-Code Dependency Attacks. The self accounting design choice, however, creates another system dependency, which we call the client-code dependency. It makes the trusted accounting logic susceptible to attacks from a malicious client trying to undercharge itself. First, the client may not use the prescribed procedure for enabling verifiable resource accounting and hence result in undercharging. Second, the client can embed subtle vulnerabilities to exploit during runtime and tamper the accounting engine data. Since the accounting engine and client-code share the same memory space (see Figure 1 (c)), the malicious application can tamper the accounting data to decrease its resource consumption. This demands isolating the accounting engine from client's application as well. To protect against dynamic attacks during runtime, we sandbox the untrusted application (explained in detail in Section 5.2). To address the compile-time threat, we statically verify the correctness of the client application.

3.3 Basic Self-Accounting Approach

One way to realize the self-accounting approach is to let the accounting engine log sufficient information related to each run of the application. First, to prevent execution replays or invocation of multiple instances, the accounting engine calculates and records a measurement or hash of the executing application before initiating its execution. Note that same application generates same measurement every time. Thus observing the frequency with which the same measurement appears in the resource consumption logs lets us detect execution replays. Next, to detect spurious executions due to input replay, the accounting engine records the hash of all the inputs of the application. Observing all the records of hashed inputs along with the application measurement helps in detecting executions with replayed inputs. Further, to detect arbitrary crashes, the logs are generated on the fly throughout the execution. Any arbitrary crash results in partial records of the hashed inputs. Presence of such partial records in resource logs lets us detect whether the same application is halted more than the pre-agreed crash recovery limit. A simple post-execution analyzer running in trusted environment either at the client or cloud provider side can perform analysis of these logs and detect the occurrence of RUI attacks [16]. The post-execution analyzer can further generate valid bill based on legitimate resource utilization and pre-agreed policy.

Inefficiency. Although the above basic solution detects RUI attacks, it demands continuous hashing and logging operations, thus making it highly inefficient. The accounting engine computes a hash of every input and invokes a system call to write it to the resource consumption logs. The accounting engine easily becomes a bottleneck for the executing application and incurs a non-negligible performance overhead. Our experimental evaluation confirm that performance overhead of an application is directly proportional to the number of system calls performed during its execution (see Section 6.1). Moreover, it generates huge logs that need to be processed at either the client or the cloud provider. This violates our desirable property of low verification overhead. In this work, we investigate the problem of designing a significantly more efficient verifiable accounting system than the basic approach. To this end, we advocate the novel use of SGX features to design an efficient solution thwarting RUI attacks.

4 Our Design

VERICOUNT comprises of three components: a compiler, a static verifier, and a post-execution analyzer. Figure 2 shows the workflow of our VERICOUNT system. The hatched components (verifier and analyzer) run in an untampered environment.

4.1 Overview

Compiler. VERICOUNT provides its own compiler that transforms client's application to support resource accounting based on our accounting library that executes with the application. The compiler inserts APIs in the application to invoke the accounting engine. This VERICOUNT-enabled application generates encrypted and integrity-protected resource consumption logs at the end of the execution. To eliminate client-code dependency and isolate the accounting engine from the client's application, VERICOUNT compiler sandboxes the application and protects the accounting engine data. Further, to remove trust from the underlying OS, the transformed application executes in a trusted execution environment in the cloud. Note that the compiler itself is executed in a potentially-malicious (thus *untrusted*) client platform.

Static Verifier. Clients may not use prescribed compiler with the intention to reduce charges. To detect against such static compile-time misbehavior, the verifier runs in a trusted environment and lets the cloud provider validate the correctness of the client-submitted, transformed application. On successful verification, the provider launches the application; otherwise, the execution is aborted. It verifies these properties:
a) *Correctness* - The VERICOUNT-enabled application has all the API calls to the accounting engine at appropriate interfaces in the application.

b) *Safety* - The application code does not access the memory region of the accounting engine; i.e., it verifies the sandboxing of the application code.
c) *Integrity* - It verifies the accounting engine integrity embedded in the application.

Post-Execution Analyzer. The trusted post-execution analyzer takes the resource consumption logs, pre-agreed execution policy $\phi = (\mathtt{p}, \mathtt{c}, \mathtt{t})$, and the cloud provider generated bill $\mathcal{B}_A$ as inputs and verifies whether the bill $\mathcal{B}_A$ adheres to the execution policy ϕ. The analyzer outputs a *yes* if the bill correctly reflects the resource consumption as per ϕ. Otherwise, it outputs *no* along with a discrepancy report in the bill.

4.2 Background on SGX

Intel SGX supports creating hardware isolated execution environment called enclaves that execute at Ring 3. Such execution is termed as *enclaved-execution*. Enclave code and data pages reside in a hardware protected memory region during execution called Enclaved Page Cache (EPC). The hardware protects a compromised OS or any other process from tampering pages in EPC. For more details, readers can refer to Intel SGX Manual [1]. SGX introduces `OCALLs` to call functions that reside outside the enclave from within an enclave and `ECALLs` for vice versa. Hence, to invoke OS services, existing system perform `OCALLs` to access filesystem, network and I/O services [30]. We describe the important primitives supported in the SGX platform.
1) *Remote attestation* - SGX allows to attest enclave code such that any remote entity can verify the integrity of the code and authenticity of the executing hardware.
2) *True random number* - In SGX, the use of `rand` and `srand` functions in the C/C++ library within an enclave is disabled as these are susceptible to bias. Instead, SGX supports `sgx_read_rand` API that generates a true random number using the `RDRAND` instruction directly from the hardware and returns it to the enclave.
3) *Monotonic counter* - SGX supports creating a limited number of monotonic counters (MC) for each enclave. Monotonic counters are shared among enclaves that have the same code. On creating a MC, it gets written to the non-volatile memory in the platform. The `sgx_create_monotonic_counter` returns a UUID and a value.
4) *Trusted elapsed time* - The function call to `sgx_get_trusted_time` returns the current time from a reference point. The difference between the returned time of two calls gives the trusted elapsed time between two events from the same reference point.

4.3 Protection against Malicious Provider

Self-Accounting. VERICOUNT places the resource accounting engine alongside a secure enclaved application that guarantees tamper-resistant accounting of resources used within the enclave against a compromised OS. Self-accounting enables fine-grained accounting of resources utilized during the application execution. Moreover, it offers transparent method to report to clients about exact operations and executions invoked by the underlying OS. Along with enclaved execution, remote attestation allows client to verify correct execution of their application on the cloud provider's platform. Enclaved execution combined with remote-attestation enables VERICOUNT to move the accounting engine from Ring 0 to Ring 3.

Preventing Replay Attacks. After remote attestation of enclaved application, the client establishes a secure channel with the enclave to provision encrypted and integrity protected inputs [10, 1]. To prevent replays, instead of hashing and logging every input to the application, we utilize the support for creating monotonic counters and true random number in SGX. In VERICOUNT system, the accounting engine registers a UUID corresponding to a monotonic counter using `sgx_create_monotonic_counter` API for the application. This UUID is sent to the client over the secure channel and acts as a hardware identity of the enclave. If the client legitimately wants to create multiple instances of the same enclaved application, she requests for multiple UUIDs. VERICOUNT appends these UUIDs and there value in the final bill, thereby allowing the client to validate the resource consumption details. Since the UUID is accessible only from within the enclave and is securely transferred to the client, an adversary cannot learn this value. Moreover, since all enclaves with the same measurement share common set of monotonic counters on same machine, different instances of the same application cannot have the same UUID. Therefore, invocation of an unrequested application instance results in a new UUID which is unknown to the client. Thus, the client can match the registered UUID of its application enclave with the UUID present in the final bill to detect execution replay attacks. As monotonic counters are written to non-volatile memory, they provide rollback protection from platform reboots as well. VERICOUNT can benefit from any additional security from recent solutions (e.g., ROTE [23]), however, we do not encapsulate them in our current design. Along with the UUID, the enclaved container uses the trusted randomness primitive to generate a random nonce corresponding to this enclave. This random nonce is sent to the client along with the UUID. To ensure the freshness of data, the client is enforced to append this random nonce to every authenticated-and-encrypted input data. To legitimately send multiple copies of the same input, the client increments the random nonce each time and appends with the input. Since this random value is generated from hardware and kept secret, the adversary cannot learn it. This prevents the attacker from generating copies of the input to inflate resource usage.

Preventing Arbitrary-halt Attacks. Every time the enclave halts or exits, the accounting engine seals the monotonic counter UUID and value using `sgx_seal_data` [1]. On invoking the enclave instance again, it unseals the monotonic counter UUID and value using `sgx_unseal_data` and verifies the value using `sgx_read_monotonic_counter`. On successful verification, it increments the value using `sgx_increment_monotonic_counter`. Thus, if the adversary tries to arbitrarily halt and restart the execution to inflate resource usage, the counter corresponding to the UUID value increases and the final value is reported in the bill. Based on the monotonic counter value, the client can detect whether the application is invoked for more than the requested number of executions. The c value in ϕ specifies an upper bound for recovery of crashed instances allowed to the provider. This also applies to crashes that occur due to bugs in client's application and are required to be restarted legitimately. The policy agreement between the client and cloud provider captures both these cases. Thus, the use of monotonic counters helps in designing a simple solution and brings transparency with respect to the enclave invocations. Note that a forced system shutdown that does not allow a clean enclave exit results in an incomplete resource log. Such incomplete logs allows us to detect if an enclave process is killed arbitrarily.

Detecting Slow-OS Attacks. A malicious OS can slowdown the kernel mode of operation to overcharge clients. VERICOUNT enables fine grained accounting by measuring the CPU time spent in an enclave and outside the enclave separately. The `sgx_get_trusted_time` API is invoked just before switching from enclave to non-enclave mode and when it switches back. Hence, we support specifying an upper bound for time spent outside the enclave necessary for replying to any system level request from the user. If the OS exceeds this time, then it indicates that the adversary intentionally delays the response to increase the total time and thereby overcharging client. In VERICOUNT, we use `t` value in ϕ as the upper bound for delay in OS response for all system calls. Ideally, the billing model should charge the container for only CPU time within an enclave and provide a fixed charge for OS services. VERICOUNT supports both these models and can be decided in the execution policy between a client and a cloud provider.

4.4 Protection Against Malicious Client

Self-accounting allows a client-submitted application to run in the same enclave and thus the client application may overwrite the accounting engine's data at runtime. To remove this client-code runtime dependency, VERICOUNT compiler sandboxes the application from the trusted accounting engine. Worse yet, a client may not use the prescribed protection mechanism or modify the compiler to ignore the sandboxing and API insertion logic entirely. The client's code may exclude executing APIs which invoke the resource accounting engine. There are three possible approaches to address this: (1) To enforce the clients to submit their source codes and compile them in a trusted environment; (2) To assume all the clients are equipped with a hardware-based trusted execution environment (e.g., SGX) to compile their applications in their local machine; and (3) To statically verify the client-compiled, sandboxed applications in a trusted environment at the cloud provider. Although all these approaches are technically feasible, we choose the third approach as the former two approaches impose too strong requirements for the clients in practice. VERICOUNT's trusted static verifier checks that the client-application satisfies a set of rules correctly to perform accurate resource accounting. The static verifier is responsible for ensuring correct sandboxing of unsafe instructions, appropriate API insertion and integrity of accounting engine. The cloud provider rejects the execution of the application if the verifier fails.

4.5 Measuring Resources

CPU Elapsed Time. The billing metric for CPU differ for every cloud provider. While Amazon EC2 charges in hours, Google Compute Engine is moving towards a more granular accounting and charges per minute. VERICOUNT measures the time at the fine granularity of per second [4, 2]. To perform fine-grained accounting at Ring 3, we use the trusted elapsed time feature of SGX hardware that supports calculating time with a precision of seconds. The accounting engine invokes `sgx_get_trusted_time` function on a switch from enclave mode to non-enclave mode and vice versa. Finally, before exiting the enclave, the engine records final value at the end of the execution. As per VERICOUNT design, the *user time* corresponds to the time spent for execution within the enclaved region. The execution time spend in untrusted region outside enclave at the user level and in the OS are accounted towards *service time*. We consider this as a valid design since a compromised OS can always tamper the execution of non-isolated application code executing outside the enclaved environment to delay the execution.

Memory. For SGX CPUs, BIOS allocates a certain region called processor reserved memory (PRM) of sizes of 32, 64 or 128 MB [1]. The underlying CPU reserves a part of this PRM as EPC memory. Enclave pages are loaded in EPC which is a hardware protected memory region. Thus, the billing of memory resource is restricted to the allowed physical memory region by SGX. VERICOUNT currently does not calculate the allocated memory from within the enclave. However, SGX2 instructions support reporting page faults to the enclave [25]. VERICOUNT can utilize this feature to correctly report the number of page faults during the execution of an application.

Network Bandwidth. Cloud providers charge bandwidth usage based on the amount of bytes transferred over the network. To calculate network bandwidth, VERICOUNT records the inputs and outputs from the enclave via network system calls. All the data that is sent over the network is accounted towards the bandwidth utilization for the application. As the accounting is performed within the enclave, it ensures that the bandwidth accounting is correctly attributed to the application and avoids mis-accounting.

I/O Resources. VERICOUNT records the I/O bytes that are read/written using system calls. Due to the restriction of SGX on direct invocation of system calls, applications use `OCALLs` to request I/O service from the underlying OS. A VERICOUNT-enabled application invokes the accounting engine to account the I/O bytes after every such `OCALL`. The engine considers only the actual data passed as arguments to these system calls. Hence, if the OS responds with fewer or more bytes than requested, VERICOUNT fairly charges for the actual I/O bytes read / written. This design guarantees that only the I/O operations made from within the enclave are accounted to the particular enclave.

5 Implementation

VERICOUNT consists of library, a compiler, a static verifier and a post-execution analyzer. It relies on existing proposals to port legacy applications to enclaves [10, 30].

5.1 VERICOUNT Library

VERICOUNT library contains functions to compute resources utilized during application execution. It accounts for user and service time, I/O bytes, total I/O calls and network bandwidth. We implement the VERICOUNT accounting engine as a statically linkable `C` library `libvericount.a`. The application, accounting library and SGX libraries are linked together to create the trusted enclave file. The accounting library invokes the `sgx_create_pse_session` function to start the trusted platform service at the start of enclave execution. Once a session is started, the library invokes appropriate calls to the trusted runtime functions to get elapsed time and monotonic counter.

API Insertion. We implement our VERICOUNT compiler as a pass in LLVM v3.8.1. The compiler inserts APIs to invoke resource accounting logic in VERICOUNT library. At the entry of an `Ecall` function inside an enclave, it inserts `vericount_init_user_time()` which starts the counter for accounting user time. The compiler inserts `vericount_init_service_time()` before every `Ocall` from within the enclave which essentially stops the user time counter and starts the counter for service time. After return of every `Ocall`, `vericount_end_service_time()` API is inserted which stops the service time counter and starts the user time counter again. Finally, before end of every `Ecall` function in the enclaved application, the compiler inserts `vericount_end_user_time()` which calculates the final user time. All these APIs invoke the trusted time function to account for elapsed time and add them to corresponding counter value. For accounting I/O bytes and network bandwidth, the VERICOUNT compiler inserts `vericount_io_bytes()` and `vericount_net_bytes()` APIs after the return of every `Ocall` to libc function related to I/O such as `fread, fwrite, fgets, fputs`, and others and network, such as `send, recv`. The library accounts the total usage based on their arguments and return values.

Output Logs. A simple way to log resource consumption is to ensure every API invocation logs the usage to the output. However, this incurs overhead since a write operation requires an `Ocall` that performs context switch from trusted to untrusted region. Thus, in our implementation, we choose to begin accounting on enclave entry and write resource consumption logs only before the enclave executes the `EEXIT` instruction. The `vericount_init_user_time()` API logs the enclave UUID to the output file and marks the start of the accounting procedure and the `vericount_end_user_time()` API logs the accounted usage to the output. The total enclave memory required to hold the accounting information is as small as storing 4 counters (user_time, service_time, io_bytes and net_bytes). We enable a provision for cloud provider's to send a user signal to fetch the resource consumption logs on demand or at a timer expiration.

5.2 Sandboxing within Enclaves

While sandboxing is not a new idea, implementing it in enclaves involves a number of non-trivial challenges. For e.g., enclaved applications have specific limitations about execution, such as no system calls making use of existing sandboxing tools impossible for enclaved applications [36]. Hence, we implement our sandbox logic compatible with Linux SGX SDK based on standard SFI techniques [33, 19].

Fault Domain Isolation. We divide the enclave virtual address space into two regions: application memory and VERICOUNT memory. The VERICOUNT memory is the region that contains the code and data of the accounting library and SDK trusted libraries. The application memory is a shadow memory or SFI section that is created to confine the application's code and data. We use portable SFI techniques to implement the application memory [19]. We add our own `malloc` function alongside the `malloc` function in SDK to create a separate heap in SFI section for application variables. The VERICOUNT compiler instruments all `malloc` calls in the application to invoke our added `malloc` function. The library code continues to use the SDK `malloc` that allocates memory in the default section. We add sandboxing instructions to all unsafe instructions in the application memory as per standard SFI sandboxing rules [33].

Static Verifier. The VERICOUNT static verifier checks a set of rules in a disassembled executable of a VERICOUNT-enabled application to ensure its correctness, safety, and integrity before executing it. We include all standard SFI checks to ensure instruction safety, control flow and trusted memory protection against attacks that exploit indirect jumps, code-reuse attacks and others [24, 14]. In addition, all call instructions to `sgx_ocall` are immediately followed by VERICOUNT APIs and there are no jump instructions between them. This ensures that the client-code cannot bypass the accounting logic when invoking an `Ocall`. Moreover, the call to `vericount_init_user_time()` and `vericount_init_end_time()` are the first and last instructions of every `Ecall` function in the enclave.

Post-Execution Analyzer. The post-execution analyzer executes within an enclave and takes the execution policy ϕ, resource consumption logs, and the bill from the cloud provider. The post-execution analyzer first checks that every UUID entry and its value is followed by resource consumption details in the output log. Next, the analyzer computes charges based on resource consumed and $\phi = (\mathtt{p}, \mathtt{c}, \mathtt{t})$ and compares it with the cloud provider's bill. The analyzer outputs the difference, if any, between VERICOUNT computed and cloud provider's bill. The enclaved post-execution analyzer cryptographically signs this final bill to be verified by the client. The client can refute provider's charges based on the output of VERICOUNT.

TCB Size. We measure the size of our TCB that includes the accounting engine, the verifier, the analyzer and trusted libraries from Intel SGX SDK using `CLOC` tool. The application code along with glibc libraries are not a part of our TCB. Our VERICOUNT compiler consists of 872 SLoC, which is outside of our TCB well. The accounting engine library consists of 230 lines of `C` code. The verifier and the analyzer contribute 180 and 132 SLoC respectively. Thus, VERICOUNT contributes only **542** SLoC to the total TCB beyond the trusted SDK libraries of 80 K SLoC. The total TCB is orders of magnitude smaller than any privileged software which consists of millions of LoC.

6 Evaluation

We evaluate our system on a Lenovo Thinkpad T460s with Ubuntu Desktop-14.04-LTS 64bits and Intel Core i7-6600U CPU running at 2.60GHz $\times$ 4 with 4 MB cache and 12 GB of RAM. We use open source version of Intel SGX SDKv1.8 available for Linux systems [6]. We perform our evaluation with two goals a) To evaluate the performance overhead of VERICOUNT application as compared to non-accounting secure (enclaved) applications b) To evaluate the accuracy of VERICOUNT accounting engine.

Selection of Benchmark. To evaluate the effectiveness of VERICOUNT, we select standard SPEC CPUINT 2006 v1.2 benchmarks [9] and H2O web server which is an optimized HTTP server [5]. We do *not* claim contribution in porting our benchmarks to execute on SGX CPUs, which by itself is a hard problem [11, 10, 30]. At present, support for creating secure enclaved containers and the corresponding libraries is not available for public use. Hence, we use a recently proposed and open source Panoply system that supports executing legacy applications on SGX-enabled CPUs [8]. Yet, VERICOUNT system is general and compatible with any other enclaved execution system.

Evaluation Methodology. VERICOUNT system extends the guarantees of enclaved applications to support verifiable accounting. We use enclaved application executed using Panoply libraries as our base for comparison throughout our evaluation. We calculate the overall execution time of our benchmarks using `time` command in Linux. Each measurement is averaged over 5 runs. All benchmarks are compiled using Clang v3.8.1. We do not include time for compiling and verifying our applications as these are offline operations and do not incur any overhead. We divide our experiments into three sets.

Benchmarks	**User Time (s)**		**Service Time (s)**		**VeriCount I/O Operations**		**VeriCount**
	OS	**VeriCount**	**OS**	**VeriCount**	**I/O Bytes**	**I/O Ocalls**	**Total Ocalls**
mcf	20.05	20	1.01	0	2360827	192458	192462
bzip2	31.83	31	1.00	0	653190	32	34
astar	77.99	78	0.99	0	3146654	68	92
hmmer	127.31	127	1.02	0	16633	766	1265
h264ref	11.96	11	1.02	1	969975	360	1646052
libquantum	5.15	5	1.03	0	261	7	8
sjeng	229.40	229	1.00	0	17211	1601	4587
gobmk	0.91	0	1.10	0	11876	10024	10412
gcc	18.21	17	1.04	1	1744658	827474	827494
H2O (10000 req.) file size=10 KB	16.12	5	0.81	13	102400000	10000	122073

Table 1: Table reporting OS and VERICOUNT accounted user and service time, VERICOUNT accounted I/O bytes, no. of I/O `Ocalls` and total no. of `Ocalls` for our benchmarks.

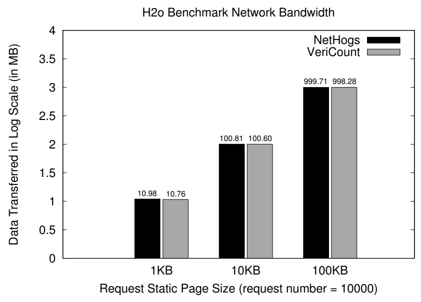

Fig. 3: Comparison of data bytes transferred using VERICOUNT and NetHogs. VERICOUNT accounting differs by 0.66% as compared to NetHogs.

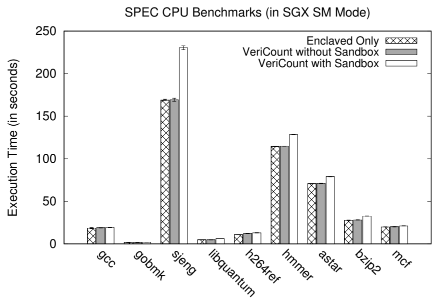

Fig. 4: Execution time of a) Enclaved only b) VERICOUNT without sandbox and c) VERICOUNT with sandbox in simulation mode.

1) Simulation Mode. We first evaluate all benchmarks in simulation mode to observe the performance overhead due to VERICOUNT design and implementation. To understand the overhead breakdown, we measure the overall execution time for enclaved only, VERICOUNT without sandbox, and VERICOUNT with sandbox applications.

2) Hardware Mode. Further, to understand the overhead of using SGX, we perform experiments in hardware mode. We compare the execution time of VERICOUNT and non-VERICOUNT enclaved applications in hardware mode.

3) Resource Measurements. To evaluate the accuracy of our accounting engine, we measure the user time, service time, I/O bytes, and network bandwidth using VERICOUNT and compare them to resource accounting tools from the OS.

6.1 Performance Overhead

Simulation Mode Overhead. Figure 4 shows the execution time of VERICOUNT without and with sandbox for our benchmarks as compared to enclaved benchmarks in simulation mode. We observe that VERICOUNT without sandbox benchmarks incur an average overhead of only **2.28%** as compared to non-accountable enclaved applications. Thus, the accounting engine contributes a small overhead to perform resource accounting. This relatively small overhead of VERICOUNT's resource accounting engine (without sandbox) suggests that a trusted application (which does not tamper with the accounting engine) can exclude the sandbox logic and enjoy the low performance overhead. The maximum overhead in VERICOUNT applications without sandbox is for

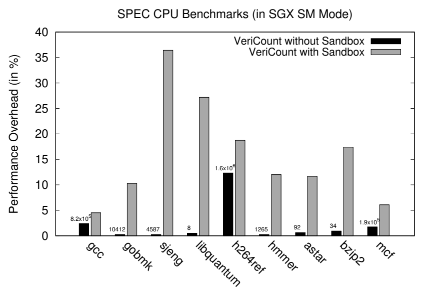

Fig. 5: Performance overhead of VERICOUNT with and without sandbox application over enclaved only applications in simulation mode.

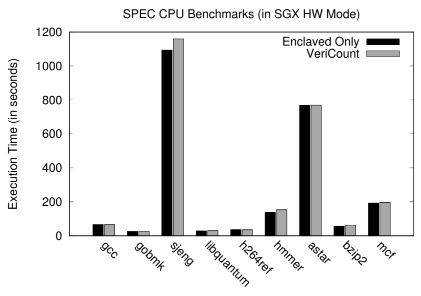

Fig. 6: Execution time of VERICOUNT applications and enclaved only applications in hardware mode.

applications with large number of `Ocalls`, thereby causing higher number of invocations to accounting engine. Figure 5 shows the performance overhead where `h264ref` benchmark invokes 1.6×10^5 `Ocalls` and hence incurs maximum overhead of 12.3%. We observe that VERICOUNT with sandbox benchmarks incur an average overhead of $\mathbf{16.03}\%$. This shows that the sandboxing logic in VERICOUNT contributes to major portion of the overall overhead. The overhead is directly proportional to the number of sandboxing instructions added to the application. Applications with higher number of sandboxing instructions (e.g., `sjeng`) show a higher overhead.

Hardware Mode Overhead. To get the estimate of VERICOUNT overhead in hardware mode, we compare the execution time of VERICOUNT and enclaved only applications (shown in Figure 6). VERICOUNT-enabled applications incur an average overhead of $\mathbf{3.62}\%$ as compared to enclaved only applications in hardware mode. 3.62% approximately captures the overall overhead (i.e., sandboxing and accounting engine) of VERICOUNT design in hardware mode. Thus, we observe that the performance overhead due to VERICOUNT is less in hardware mode than in simulation mode. Figure 4 and Figure 6 show that porting enclaved applications from simulation to hardware mode increases the execution time by a large margin which essentially hides the overhead due to VERICOUNT. Since enclaved application in HW mode take longer to finish than in simulation mode (the denominator increases), it reduces the overall overhead.

6.2 Resource Utilization Measurement

User-Service Time. The `time` command provides user and kernel time along with the overall execution time. We compare the user time measured by VERICOUNT with that of the OS service. Column 2 in Table 1 shows that the VERICOUNT calculated user time differs from the OS user time within a fraction of second for the SPEC CPUINT 2006 benchmarks. One exception is the H2O web server, where the VERICOUNT user time does not match the OS time. This is because the web server spends most of its time waiting for requests in the untrusted library outside the enclave while VERICOUNT guarantees are scoped only within the enclaved applications. Column 3 in Table 1 shows the difference between VERICOUNT service time and OS accounted kernel time. VERICOUNT's service time includes the kernel time and waiting time of untrusted library residing outside the enclave. This results in a difference between OS kernel time and

VERICOUNT's service time. Arguably, service time should not be considered when accounting the user resources. Or, if client and cloud provider agree, they can decide an upper bound of `t` in policy ϕ or use fixed pricing for service time.

I/O operations. VERICOUNT calculates the I/O bytes, I/O calls and total number of `Ocalls` invoked by the application (shown in Column 4, 5, and 6 in Table 1). As there is no precise OS supported tool to measure I/O operations executed within the enclave, we confirm the correctness of our accounting engine using the `strace` and `ltrace` commands. We use `ltrace` command to verify the number of I/O `Ocalls` and total number of `Ocalls` which invoke the glibc library functions in the untrusted region. We use the `strace` command to calculate total bytes read & written after `open(app.signed.so)` call i.e., once the control switches to the enclave. We observe that VERICOUNT accurately accounts the I/O bytes for all our benchmarks. For eg., the I/O bytes and `Ocalls` for H2O web server is exactly 102400000 Bytes and 10000 for 10000 requests of 10 KB file size. As all the values match, we do not report the OS generated values in Table 1.

Network Bandwidth. We calculate the data transferred over the network by VERICOUNT-enabled H2O web server and compare it to `NetHogs` tool available in most Linux distributions [7]. We observe that VERICOUNT accounted network bytes differ from NetHogs results on an average by 0.66% (shown in Figure 3). We were unable to determine the cause for the extra data traffic accounted by NetHogs but speculate it to be due to TLS handshakes before the actual response is sent over the network. We use `h2load` performance measurement tool to generate workload of 10000 requests for static web pages of size of 1 KB, 10 KB and 100 KB.

7 Related Work

Issues in Cloud Accounting and Billing. Previous work like Bouchenak et.al [12], Fernandes et. al [15], Xiao et.al [34] have discussed the importance of verifying resource consumption, accountability and billing. The key message is that users benefit from the ability to reason about the operations at the server. VERICOUNT realizes this idea and provides a refutable billing model for both users and cloud providers. Jellinek et.al [18] perform a study of billing systems in current cloud systems such as Amazon EC2, Google Compute Engine, Rackspace and others. Their results show that cloud billing systems have bugs that cause over-charging or free CPU time for users. VERICOUNT solve this issue with a verifiable accounting engine and a refutable billing primitive.

Solutions for Verifiable Resource Accounting. Sekar and Maniatis proposed the first practical design for resource accounting based on nested virtualization and TPMs [28]. They propose the idea of an observer placed at the hypervisor layer which accounts for the resources utilized by guest virtual machines. Alibi is a system based on this solution extending Turtles nested virtualization framework [13]. In contrast to this, VERICOUNT uses a self-accounting approach ensuring effective protection from resource usage inflation attacks. Moreover, their implementation based on Linux kernel and KVM incurs a huge TCB which we avoid in our solution. A second line of research uses execution logs but suffers from similar inefficiency problems as in our baseline approach. Haeberlen et.al propose accountable virtual machines that generate logs during execution that are sent to user [16]. A user can replay a "good" known execution and identify discrepancies in the server logs to detect maliciousness. This solution is hard to use where resource-constrained clients do not have enough bandwidth to download huge logs.

Combining Trusted Computing and Sandboxing. Previous work has proposed the idea of two-way protection using trusted computing and sandboxing approaches for different reasons. MiniBox [21] provides the first sandbox mechanism for native code in platform-as-a-service cloud computing. Though MiniBox uses TrustVisor and NaCl, the core idea applies to SGX as well. Ryoan uses a similar idea to create distributed sandbox for computing on secret data[17]. In VERICOUNT we use this idea to protect the execution of accounting engine from both the client-code and the underlying OS.

8 Conclusion

VERICOUNT achieves a verifiable resource accounting with a refutable billing mechanism for Linux containerized applications with Intel SGX support with low overhead.

9 Acknowledgements

We thank the anonymous reviewers of this paper for their helpful feedback. We also thank Zhenkai Liang, Shweta Shinde, and Loi Luu for useful feedback on an early version of the paper. This research was partially supported by a grant from Singapore Ministry of Education Academic Research Fund Tier 1 (R-252-000-624-133), and the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

References

1. Software Guard Extensions Programming Reference. `software.intel.com/sites/default/files/329298-001.pdf` (Sept 2013)
2. Amazon ec2 container service. `https://aws.amazon.com/ecs/` (2017)
3. Docker. `https://www.docker.com/` (2017)
4. Google container engine (gke). `https://cloud.google.com/` (2017)
5. H2o web server. `https://github.com/h2o/h2o` (2017)
6. Intel sgx linux sdk. `https://github.com/01org/linux-sgx` (2017)
7. NetHogs. `https://github.com/raboof/nethogs` (2017)
8. Panoply source code. `https://shwetasshinde24.github.io/Panoply/` (2017)
9. Spec cpu2006 benchmarks. `https://www.spec.org/cpu2006/` (2017)
10. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., OKeeffe, D., Stillwell, M.L., et al.: Scone: Secure linux containers with intel sgx. In: 12th USENIX Symp. Operating Systems Design and Implementation (2016)
11. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. ACM Transactions on Computer Systems (TOCS) 33(3), 8 (2015)
12. Bouchenak, S., Chockler, G., Chockler, H., Gheorghe, G., Santos, N., Shraer, A.: Verifying cloud services: present and future. ACM SIGOPS operating systems review (2013)
13. Chen, C., Maniatis, P., Perrig, A., Vasudevan, A., Sekar, V.: Towards verifiable resource accounting for outsourced computation. In: VEE (2013)
14. Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., Necula, G.C.: Xfi: Software guards for system address spaces. In: OSDI (2006)

15. Fernandes, D.A., Soares, L.F., Gomes, J.V., Freire, M.M., Inácio, P.R.: Security issues in cloud environments: a survey. International Journal of Information Security (2014)
16. Haeberlen, A., Aditya, P., Rodrigues, R., Druschel, P.: Accountable virtual machines. In: OSDI. pp. 119–134 (2010)
17. Hunt, T., Zhu, Z., Xu, Y., Peter, S., Witchel, E.: Ryoan: a distributed sandbox for untrusted computation on secret data. In: OSDI (2016)
18. Jellinek, R., Zhai, Y., Ristenpart, T., Swift, M.: A day late and a dollar short: The case for research on cloud billing systems. In: HotCloud (2014)
19. Kroll, J.A., Stewart, G., Appel, A.W.: Portable software fault isolation. In: Computer Security Foundations Symposium (CSF), 2014 IEEE 27th. pp. 18–32. IEEE (2014)
20. Lee, S., Shih, M.W., Gera, P., Kim, T., Kim, H., Peinado, M.: Inferring fine-grained control flow inside sgx enclaves with branch shadowing. arXiv preprint arXiv:1611.06952 (2016)
21. Li, Y., McCune, J.M., Newsome, J., Perrig, A., Baker, B., Drewry, W.: Minibox: A two-way sandbox for x86 native code. In: USENIX Annual Technical Conference (2014)
22. Liu, M., Ding, X.: On trustworthiness of cpu usage metering and accounting. In: IEEE 30th International Conference on Distributed Computing Systems Workshops (2010)
23. Matetic, S., Kostiainen, K., Dhar, A., Sommer, D., Ahmed, M., Gervais, A., Juels, A., Capkun, S.: Rote: Rollback protection for trusted execution. In: Usenix Security (2017)
24. McCamant, S., Morrisett, G.: Evaluating sfi for a cisc architecture. In: Usenix Security ’06
25. McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., Rozas, C.: Intel&reg; software guard extensions (intel&reg; sgx) support for dynamic memory management inside an enclave. HASP 2016
26. Mihoob, A., Molina-Jimenez, C., Shrivastava, S.: A case for consumer–centric resource accounting models. In: IEEE 3rd International Conference on Cloud Computing (2010)
27. Ming-Wei-Shih, Lee, S., Kim, T., Peinado, M.: T-sgx: Eradicating controlled-channel attacks against enclave programs. In: NDSS 2017
28. Sekar, V., Maniatis, P.: Verifiable resource accounting for cloud computing services. In: ACM workshop on Cloud computing security workshop (2011)
29. Shinde, S., Chua, Z.L., Narayanan, V., Saxena, P.: Preventing page faults from telling your secrets. In: ASIACCS (2016)
30. Shinde, S., Le Tien, D., Tople, S., Saxena, P.: Panoply: Low-tcb linux applications with sgx enclaves. In: NDSS (2017)
31. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Secretly monopolizing the cpu without superuser privileges. In: USENIX security. vol. 7, pp. 1–18 (2007)
32. Varadarajan, V., Kooburat, T., Farley, B., Ristenpart, T., Swift, M.M.: Resource-freeing attacks: improve your cloud performance (at your neighbor’s expense). In: CCS (2012)
33. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: ACM SIGOPS Operating Systems Review. pp. 203–216. ACM (1994)
34. Xiao, Z., Xiao, Y.: Security and privacy in cloud computing. IEEE Communications Surveys & Tutorials 15(2), 843–859 (2013)
35. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: IEEE Symposium on Security and Privacy (SP), 2015
36. Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native client: A sandbox for portable, untrusted x86 native code. In: Security and Privacy, 2009 30th IEEE Symposium on. pp. 79–93. IEEE (2009)
37. Zhang, F., Zhang, H.: Sok: A study of using hardware-assisted isolated execution environments for security. In: Proceedings of the Hardware and Architectural Support for Security and Privacy 2016. HASP 2016 (2016)
38. Zhou, F., Goel, M., Desnoyers, P., Sundaram, R.: Scheduler vulnerabilities and coordinated attacks in cloud computing. Journal of Computer Security 21(4), 533–559 (2013)