1-2023

T-counter: Trustworthy and efficient CPU resource measurement using SGX in the cloud

Chuntao DONG
Peking University

Qingni SHEN
Peking University

Xuhua DING
Singapore Management University, xhding@smu.edu.sg

Daoqing YU
Peking University

Wu LUO
Peking University

See next page for additional authors

Citation

DONG, Chuntao; SHEN, Qingni; DING, Xuhua; YU, Daoqing; LUO, Wu; WU, Pengfei; and WU, Zhonghai. T-counter: Trustworthy and efficient CPU resource measurement using SGX in the cloud. (2023). *IEEE Transactions on Dependable and Secure Computing*. 20, (1), 867-885.
Available at: https://ink.library.smu.edu.sg/sis_research/6968

Published in IEEE Transactions on Dependable and Secure Comuting, 2022, DOI: 10.1109/TDSC.2022.3145814

T-Counter: Trustworthy and Efficient CPU Resource Measurement using SGX in the Cloud

Chuntao Dong, Qingni Shen, Xuhua Ding, Daoqing Yu, Wu Luo, Pengfei Wu, and Zhonghai Wu

Abstract—As cloud services have become popular, and their adoption is growing, consumers are becoming more concerned about the cost of cloud services. Cloud Service Providers (CSPs) generally use a pay-per-use billing scheme in the cloud services model: consumers use resources as they needed and are billed for their resource usage. However, CSPs are untrusted and privileged; they have full control of the entire operating system (OS) and may tamper with bills to cheat consumers. So, how to provide a trusted solution that can keep track of and verify the consumers' resource usage has been a challenging problem. In this paper, we propose a T-Counter framework based on Intel SGX. The T-Counter allows applications to construct a trusted solution to measure its CPU usage by itself in cloud computing. These constructed applications are instrumented with counters in basic blocks and added three components in trusted parts to count instructions and defend against malicious CSPs' manipulations. We propose two algorithms which selectively instrument counters in the CFG. T-Counter is implemented as an extension of the LLVM framework and integrated with the SGX SDK. Theoretical analyses and evaluations show that T-Counter can effectively measure CPU usage and defend against malicious CSPs' manipulations.

Index Terms—Resource Measurement, Instruction Accounting, SGX, Cloud Services, Control Flow Graph (CFG)

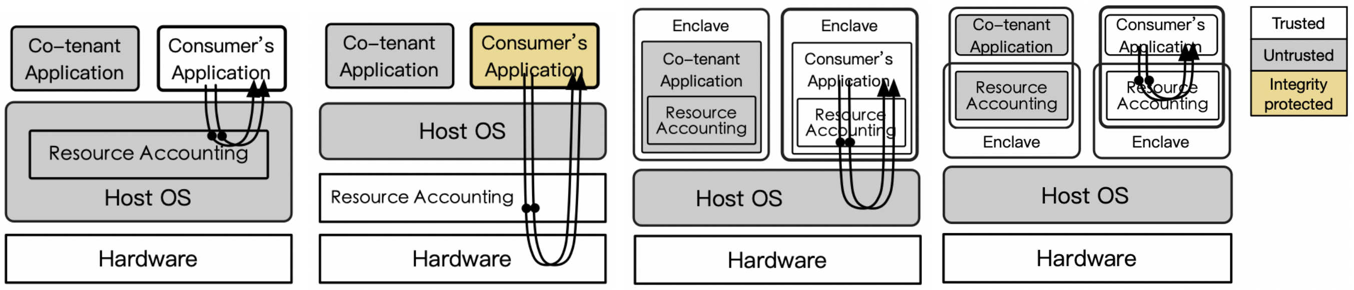
1 Introduction

Cloud computing adoption continues to overgrow nowadays. Cloud Services Providers, such as Amazon Web Service [1], Google Cloud [2], and Microsoft Azure [3], have become popular. CSPs generally use a pay-per-use billing scheme in the pay-as-you-go pricing model [4] [5] [6] [7]: consumers use resources as needed and are billed for their resources usage by the end of an agreed-upon period, and consumers only need to pay for resource charges based on used services (memory, CPU and bandwidth, etc.). From the perspective of economic benefits, consumers do not have to maintain dedicated infrastructure. They can save a lot of equipment maintenance costs, and only need to pay according to resource usage. Cost is a significant driver for most consumers to adopt cloud services. As cloud adoption grows, organizations and consumers focus on governing charges for cloud services.

However, the CSPs could be adversarial and untrusted from the perspective of consumers. They can not verify the bills provided by CSPs. CSPs have the motivation to make a false report to get more benefits, usually with an exaggerated service time, resulting in more payment from consumers. Even these CSPs adopting the pricing model for their Service Level Agreements (SLAs) can not prove that they have never cheated consumers on the bills as the bills lack detailed records of resource usage. In this case, the consumers may want to require the service usage logs to verify the billed charges by checking the total service time, and the number of running software or platforms, etc. But all these behaviors of billing are under the control of these CSPs. We consider a privileged attacker who may tamper with the service time or resource usage logs to cheat consumers on the bills. Besides, the billing systems of CSPs may provide wrong bills due to system bugs or attacks. Jellinek et al. show that cloud billing systems have bugs that cause over-charging or free CPU time for consumers in their study [8]. So, providing a trusted manner to keep track of and verify the consumers' cloud service usage is critical for CSPs and consumers.

There have been several attempts to construct trustworthy solutions that measure CPU resource usage, but they have severe limitations. For example, THEMIS [9] uses a cloud notary authority to oversee resource usage. ALIBI [10] uses a trusted observer at the highest privilege level underneath the providers' platform software to account for resource usage of customer instances. VeriCount [11] can measure resource usage of programs running in SGX enclaves. It instruments ecalls to read and store the starting time using the SGX trusted time function provided by the SGX SDK [12]. However, Vericount's mechanism for measuring computation time could be arbitrarily inflated by an adversarial CSP by manipulating the system's clock. Fritz et al. proposed S-FaaS [13], a set of resource measurement mechanisms that securely measure computation time based on a trusted clock [14] [15] inside an enclave. But, S-FaaS can only measure CPU usage of the applications running in the enclave, does not support measure applications that are running outside of the enclave. These efforts do not implement trusted resource measurement perfectly.

- *C. Dong, Q. Shen, D. Yu, and Z. Wu are with School of Software and Microelectronics, Peking University, Beijing 102600, China. E-mail: chuntaodong@pku.edu.cn, qingnishen@ss.pku.edu.cn, yudaoqing@pku.edu.cn, wuzh@ss.pku.edu.cn*
- *X Ding is with School of Information Systems, Singapore Management University, Singapore 188065. E-mail: xhding@smu.edu.sg*
- *W Luo is with School of Electronics Engineering and Computer Science, Peking University, Beijing 102600, China. E-mail: lwyeluo@pku.edu.cn*
- *P. Wu is with School of Computing, National University of Singapore, Singapore 117417. E-mail: pfwu@nus.edu.sg*
- *Z. Wu is also with National Engineering Research Center for Software Engineering, Peking University, Beijing 100084, China*
- *Q. Shen is corresponding author.*



(a) Current approach (b) Observer approach (c) Self-Measurement approach (d) T-Counter

Fig. 1: Four approaches: a) Current approach, b) Observer approach, c) Self-measurement approach, and d) T-Counter.

These solutions need to rely on cloud server (see Fig 1a), a third party [9], the privilege level software [10] (see Fig 1b), or untrusted system services [11] (see Fig 1c), they solve the problem of trusted accounting with threefold or in limited scenarios. For example, VeriCount [11] still needs to rely on the system service (system clock), not strictly self-measurement, see Fig. 1c. Besides, these self-measurement solutions [11] [13] that are based on trusted technology (e.g., SGX) have many limitations; we summarize as follows. (1) *Security requirements.* Not all applications have strong security requirements. Many measured applications only have weak security requirements and do not need to use SGX originally. Therefore, it is not reasonable to run whole applications in the enclave just for measuring resource usage. (2) *Performance overhead.* A pivotal obstacle to building self-measurement solutions is performance overhead. The whole application running in the enclave will increase enclave memory access and lead to frequent ecalls/ocalls. Enclave memory access will bring high-performance overhead [16], and frequent ecalls/ocalls also carry unacceptable overhead [17]. (3)*TCB Size.* Legacy applications entirety run inside enclaves will results in a large trusted computing base (TCB) [18] [19]. A big TCB will not only bring high-performance overhead but also cause the application in the enclave to face more security risks [20] [21]. (4) *Application modification.* The self-measurement solution needs to rewrite the applications to SGX applications and added measurement codes; this requires consumers to have relevant development experience.

Because the current self-measurement schemes have the above limitations, we want to build: *a self-measurement solution that measures an application's CPU usage by itself and not relays on system services, even if the application runs outside the enclave* . We also use the SGX enclave to ensure the confidentiality of the measurement result and defend against the adversary to tamper with accounting result by manipulating the value or control flow graph of the application. Our solution uses ecalls and switchless calls provided by SGX SDK [12] to realize efficient instruction counting. The advent of the Intel SGX enclave for shielded execution has become popular. We assume that the untrusted cloud is willing to provide an SGX enclave to consumers.

1.1 Our Contributions

In this paper, we seek to address the above challenges by proposing a framework for building a trusted CPU usage measurement solution, named **T-Counter**, which can measure CPU usage verifiably and efficiently by an application itself in an untrusted cloud, as shown in Fig 1d. Our proposal measures the CPU usage of a program by accounting instructions that have been executed. We design a *trusted counter* in the trusted space (enclave) to count instructions, and instrument some **counters** (*CALL* instructions) at the basic block level in the program to call the *trusted counter* to record counters and count instructions. The main contributions of the paper are summarized as follows.

- *Universal Solution to Trusted CPU Usage Measurement:* We propose a universal CPU usage solution, the T-Counter framework, which can measure the CPU usage of a program running outside the enclave in a manner that can be trusted and verified by consumers. To address the above four challenges, we correspondingly adopt the following designs. (1) The T-Counter architecture is that the program runs outside the enclave, and the trusted counter runs in the enclave (in Section 3.3). The *trusted counter* running in the *enclave* [22] is to against the adversary to tamper with the account result. The counter-value table stores the counters' value, and the counter-sequences table stores the sequences of counters (in the paths of the CFG) in the enclave against the adversary's tamper (in Section 5.2). (2) To achieve the best performance, we reduce the number of ecalls as much as possible by selectively instrumenting counters in the CFG, avoid context switch by using switchless call (in Section 7.2), and optimize loops using hotspot strategy (in Section 4.3). (3) The application runs outside the enclave, and only trusted components such as the trusted counter run in the enclave. The design makes the TCB minimized. (4) With T-Counter's help, users can easily instrument counters in their programs with minimal modification and build the measurement code in the enclave automatically.
- *Selective and Optimal Instrumentation for Low Overhead:* Based on T-Counter's designed architecture, a key obstacle is how to instrument counters in as few blocks as possible and achieve instruction counting. Instrumenting counters in every basic block will cause high-performance overhead. We need to selectively instrument a subset of all basic blocks in the CFG and instrument counters in selected basic blocks to account for instructions. We define five types of basic blocks in the CFG and analyze that only two types of basic blocks need to be instrumented counters, then design two algorithms of instrumenting counters in the CFG (in Section 4)). According to our analysis, we design a **Base-**

Instrumentation algorithm that instrument counters in two type of basic blocks: exit blocks and multi-links predessors, named **Base-Ins** (in Section 4.1). Based on **Base-Ins**, we propose an **Opti**mization-**Ins**trumentation algorithm that instrument less counter by analyzing and comparing the weights of paths, and further reduce counters via counter optimization strategy, named **Opti-Ins** (in Section 4.2). In addition to the two algorithms, we also proposed a series of optimization strategies to optimize our solution further. Such as (1) using switchless calls to reduce SGX context switch, (2) moving counters out of loops, and (3) randomly ecalls at a hotspot (in Section 4.3).

• *Theoretical analysis of T-Counter:* Based on the designed architecture, components, and algorithms of T-Counter, we conduct theoretical analysis on the algorithm correctness, security property, performance overhead and accuracy of T-Counter (in Section 6)). We show that (1) the counters are instrumented in the CFG correctly, and the values of these counters are calculated accurately in theoretical; (2) the trusted counter and verifier can defend against attacks to counters and ensure the integrity of the instructions counting; (3) the hotspot scheme can achieve the lowest latency and overhead in theoretical, measurement code constructed by T-Counter bounds the latency and overhead.

• *Implementation and Evaluation of T-Counter:* Except for comprehensive theoretical analysis we implement T-Counter in LLVM passes and conduct empirical experiments to measure the performance of test cases (in Section 7)). T-Counter is implemented as an extension of the LLVM [23] framework and integrated with the Intel Linux SGX SDK [12]. We compare the performance in different parameters, such as runtime and message sending frequency. We also compare the performance overhead of T-Counter with some existing trusted resource usage measurement systems, such as S-Faas [13] and VeriCount [11]. The evaluation shows the runtime performance overhead of the T-Counter is almost less than 8.52% and the accuracy error of the T-Counter is almost less than 10.86 %. The results indicate that the efficiency of T-Counter coincides with our theoretical analyses. T-Counter can accurately and efficiently account CPU usage.

1.2 Related work

Many efforts have been made to design trusted resource accounting schemes. According to techniques, these efforts can be classified into three classes as follows.

• **Trusted billing.** Current cloud services in the industry all used the unconditional trust model for billing, see Fig. 1a, the early works in academia focus on the third party trust model. For example, THEMIS [9] uses a cloud notary authority as a trusted third party to oversee the resource consumption, making future resolutions of dispute more acceptable and objective. The common feature of these trusted billing proposals is that a third-party supervises resource allocation or billing; the third party is threefold. Many studies have proposed a billing scheme that does not rely on third parties. For example, Chen et al. proposed a mutually verifiable billing system Bitbill [24], a public trust model to ensure natural and intuitive verification of billable events in the cloud, leveraging a Bitcoin-like mechanism.

• **Verifiable Resource Accounting.** In addition to the trusted billing system, many works have proposed resource usage accounting schemes. These solutions are most related to our design and also based on trusted technology. We summary these works as verifiable resource accounting. Based on scenarios, verifiable resource accounting can be divided into three categories, as shown in Fig. 1b,1c, 1d . ALIBI [10] uses nested virtualization to place a trusted observer at the highest privilege level underneath the providers' platform software and all customer instances. ALIBI can be divided into observer approach, see Fig 1b. Sekar et al. also adopted a similar "observer" approach in their approach [26]. VeriCount [11] measures resource usage of programs by instrumenting ecalls to read and store the starting time and ending time using the trusted time function provided by the SGX SDK [12]. However, VeriCount's mechanism could be arbitrarily inflated by an adversarial service provider by manipulating the system's clock. VeriCount can be divided into a self-measurement approach, see Fig.1c. Fritz et al. proposed S-FaaS [13], which measures computation time based on a trusted clock [14] inside an enclave. However, it can just measure the compute time used by the enclave. S-FaaS can also be divided into a self-measurement approach. AccTEE [25] also uses a self-measurement approach that leverages enclave for isolation and instrumentation for resource accounting. The two schemes' common feature is building a trustworthy self-measurement scheme based on the enclave. However, they only can measure the resource usage of applications that running in the enclave. T-Counter can measure CPU resource usage of applications running outside the enclave, see Fig 1d. We summarize the TCB of existing trusted resource accounting schemes in Table 1.

TABLE 1: Summary of the TCB of Existing Trusted Resource Accounting Schemes[†]

Schemes	TP	Bitcoin-like	SS	Privilege	SGX	IP	Drawbacks
THEMIS [9]	✓	∗	∗	∗	∗	∗	TP/threefold
Bitbill [24]	∗	✓	∗	∗	∗	∗	TP
ALIBI [10]	∗	∗	∗	✓	∗	∗	Privilege
Vericount [11]	∗	∗	✓	∗	✓	∗	SS/in enclave
S-FaaS [13]	∗	∗	✓	∗	✓	∗	SS/in enclave
AccTCC [25]	∗	∗	∗	∗	✓	✓	in enclave
TCounter	∗	∗	∗	∗	✓	✓	-

([†]In this table, "**TP**" is short for "Third Party", "**SS**" is short for "System Service", "**SGX**" is short for "Software Guard Extension", "**IP**" is short for "Integrity Protection", the label "✓" shows the scheme is based on this kind of technology to against the adversary. Otherwise, it is set "∗.")

• **Instructions counting.** Our approach measures CPU resource usage by accounting instructions. The following works also adopted a similar approach. Zhang et al. [27] propose Resource Efficient Mining (REM) to count the number of instructions executed within the enclave, which is used to perform arbitrary useful computations. REM uses a customized tool chain that reserves a register for instruction counting and instruments the start of each basic block with an instruction to increase the count by the number of instructions in the basic block. Oleksenko et al. [14] use a rough count of instructions to estimate the frequency of AEX events. Similarly to our approach, these proposals achieved their goals through accounting instructions, and instructions accounting is sufficiently accurate for their purpose. The difference is, T-Counter does not need to count instructions in each block and only need to instrument measurement instructions to a few blocks (in Section 4).

TABLE 2: Summary of Notations and Descriptions

Notations	Descriptions
$G=(N,E)$	A control flow graph
V_i	A basic block in the CFG
(V_i, V_j)	An edge between two basic blocks in the CFG
P	A path or pathlet from one basic block to another basic block in the CFG
W	The weight of a basic block or a path/pathlet in the CFG.
C	A counter instrumented in the CFG.
S	The number of instructions that have been executed.

1.3 Organization

The remainder of this paper is organized as follows. In the next section, we introduce some necessary preliminaries of this paper. In Section 3, we describe motivation and challenges, threat model, and attacks on instructions accounting and architecture of T-Counter. Next, we detail two instrumentation algorithms for instrumenting counters, strategies to optimize hotspot and method to instrument counters in Section 4. In Section 5, we discuss the detailed design of the T-Counter and modular construction of a measurement scheme. We demonstrate the correctness and security of T-Counter and estimate the efficiency in Section 6. In Section 7, we perform some experiments and evaluate the performance of the T-Counter. We discuss some limitations of T-Counter in Section 8. Finally, we conclude Section 9.

2 Preliminaries

We introduce some preliminaries in this section. In Table 2, we summarize some helpful notations used in T-Counter.

2.1 Intel Software Guard Extensions

Intel Software Guard Extensions (SGX) [22] is an extension to x86 architecture, which enables user-level code to create trusted execution environments named **enclaves** [28] [29]. SGX allows applications to ensure confidentiality and integrity against all other software on the platform, including privileged system software [30].

Enclave switches. Enclave switches occur whenever the CPU enters or exits an enclave, they can be triggered by application developers in two ways, ECalls and OCalls. Ecalls are trusted function calls from the outside of the enclave to the inside, whereas OCalls are the opposite [31]. Ecalls and ocalls are considered edge functions, as they cause execution to cross security boundaries. Several security checks need to be performed on the call's parameters for the boundary-crossing to be secure, particularly when they are pointers. In the T-Counter, counters instrumented in the program call the trusted counter in the enclave via ECalls.

Performance overhead. SGX incurs a performance overhead when executing enclave code: (1) ecalls and ocalls come at a high cost for security reasons. Weisse et al. [32] measured enclave transitions of ecalls and ocalls in the order of 8,600 to 14,000 cycles, depending on cache hit or miss; (2) enclave code also pays the penalty for writes to memory and cache misses because the MEE must encrypt and decrypt cache lines. As described in work [17], if the accessed memory with L3 cache misses, there is a performance overhead of up to 12 $\times$; and (3) applications whose memory requirements exceed the EPC size(128MB) must swap pages between the EPC and unprotected DRAM. The eviction of EPC pages is costly because they must be encrypted and integrity-protected before being copied to external DRAM. This leads to an overhead of three orders of magnitude [17]. These results show that a design should reduce enclave transitions and access to enclave memory for performance reasons. In our design, we try not to use enclave memory and ecalls and avoid triggering page swaps and enclave transitions as much as possible.

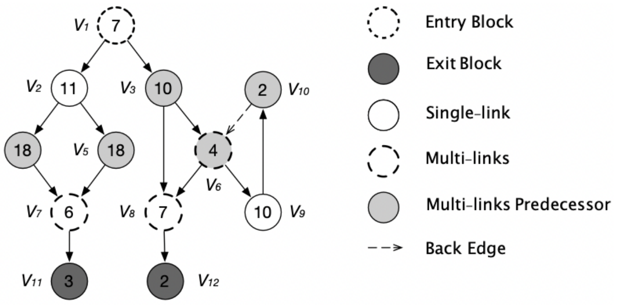


Fig. 2: A sample example of CFG

2.2 Control Flow Graph

A Control Flow Graph (CFG) is the graphical representation of a program that captures all possible flows through the program. It is a directed graph in which the nodes represent basic blocks, and the edges represent control flow paths, G=(V, E). The simplest unit of control flow in a program is a basic block. A basic block is a linear sequence of program instructions that always execute together. Each node V corresponds to a basic block. Each edge (V_i, V_j) corresponds to a possible transfer of control from block V_i to block V_j. Control flow always enters a basic block at its entry point (the first instruction executed) and exits at its exit point (the last instruction executed). Control flow cannot exit or halt inside the basic block except at its exit point. A Path is a sequence of nodes on the CFG, including an entry node and an exit node. CFG is represented differently for all statements and loops.

Definition. We classify the basic blocks in the CFG into five categories and define the five types of basic blocks as follows. We also give examples of each type of basic block in Fig. 2.

- *Entry Block*: A basic block with indegree is 0 in the CFG, V_1 is the entry block.
- *Exit Block*: A basic block with outdegree is 0 in the CFG, V_{11} and V_{12} are exit blocks.
- *Single-link*: A basic block with indegree is 1 in the CFG, V_2, V_3, V_4, V_5, V_9 and V_{10} are multi-links predecessor.
- *Multi-links*: A basic block with indegree larger than 1 in the CFG, V_{11} and V_6, V_7 and V_8 are multi-links.
- *Multi-links predecessor*: A basic block that has an edge to a multi-links in the CFG, V_3, V_4, V_5, V_6 and V_{10} are Multi-links predecessor.

We also use other concepts to describe our approach; the definitions and examples are as follows.

- *Back edge*: An edge (u, v) such that v is ancestor of edge u but not part of DFS (Depth First Search) tree. The edge (V_{10}, V_6) is a back edge.
- *Path*: A Path is a sequence of node on the CFG, including an entry node and an exit node, (V_1, V_3, V_8, V_{12}) is a path.
- *Pathlet*: A pathlet is path segment that consists of a subsequence of nodes along the path, (V_1, V_3)is a pathlet.
- *Weight*: Weight is the number of instructions in a basic block or the sum of all basic blocks' instructions in a path or pathlet, the weight of (V_1 is 7, and the weight of pathlet (V_1, V_3) is 17.
- *Counter*: Counter is the instruction measurement code added to the basic block in the CFG.

2.3 LLVM

The LLVM [23] compiler infrastructure project is a set of compiler and toolchain technologies, which can be used to develop a front end for any programming language and a back end for any instruction set architecture [33]. LLVM is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes [34]. The LLVM pass framework is an important part of the LLVM system, LLVM passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations [35]. LLVM also can make the CFG of every function explicit in the representation. The Clang project provides a language front-end and tooling infrastructure for languages in the C language family for the LLVM project [36]. Our work extends the LLVM pass framework to implement algorithms for instrumenting counters in the program.

3 Overview

In this section, we define the adversary model (in Section 3.1), outline the design goals (in Section 3.2), then we overview the architecture of T-Counter (in Section 3.3).

3.1 Threat Model

Consumers only trust the enclave provided by the cloud in our threat model, and the *trusted counter* is running inside the enclave. The instrumented program with counters is running outside the enclave. The malicious CSPs' purpose is to obtain fees other than that he deserves. CSPs have full control of all resources of the cloud. Therefore, CSPs may exploit all means to achieve their goal. We make the following assumption on a malicious CSP(adversary):

Rational and Cautious Adversary. *CSP is a rational adversary, in the sense that its attacks do not cost more CPU cycles than it "saves" from cheating in the program's execution. And a CSP is also a cautious adversary, in the sense that its attacks should not attract consumers' attention.*

We argue that the assumption is reasonable since a CSP's ultimate goal is on the financial advantages. It has no benefits if the attack costs more than the gains. CSP's primary goal is to maximize benefit rather than steal the information or hamper the user's program's execution. And such attacks may easily attract consumers' attention and damage CSP's reputation. It is unwise for the CSP. We also assume that untrusted CSPs are willing to provide a trusted execution environment to consumers' programs. We trust the processor and assume the adversary cannot physically open the processor package and extract secrets or corrupt the state inside the processor. SGX Side-channel [37], [38], [39] is not within the scope of this paper. DoS attacks are also beyond the paper's coverage, as a malicious OS can easily do so by not scheduling the shielded execution.

We reasonably assume the adversaries attack capability: (1) can easily and directly modify the bills. (2) can make a false report indirectly by tampering with the resource usage logs, manipulating service time, modifying the computing environment, etc. (3) may tamper consumers' program covertly, avoiding triggering program errors to be quickly awarded by consumers. Therefore, we assume that adversaries can tamper with consumers' programs statically but will not fiddle programs while running because this can easily trigger program errors. Specifically, the adversaries can modify constants or manipulate control flows of programs finely to avoid errors and achieve its purpose. Liu and Ding also have identified ways in which a CSP can damage the integrity of resource metering in their work [40].

3.2 Design goals

We define several desirable properties to securely and credibly measure a program's CPU usage based on the threat model.

C1: Trusted and verifiable resource accounting. The approach should ensure that a program's resource usage is accounted for in a trustworthy and verifiable manner; the consumers must trust the accounting mechanism. It requires that no attacker interferes with the accounting mechanism, e.g., manipulate the accounting process to their advantage.

C2: Accounting integrity and confidentiality. To against the CSPs tamper the accounting, the approach should ensure the integrity and confidentiality of the accounting process and result. In our threat model, the accounting result is protected in the enclave. However, the program instrumented with counters is out of the enclave, the adversary can manipulate the accounting process. As far as we know, protecting the integrity of accounting outside of the enclave without other technologies is an unsolved challenge.

C3: Execution Integrity. Note that CSPs may tamper with the control flow of consumers' programs. Execution integrity is to ensure that consumers can detect attacks on the program control flow. This task is more challenging than accounting integrity because it runs out of the enclave without any protection. The proof-of-work schemes have been proposed in previous work [41], but it does not apply to our scenario. Our goal is not to ensure the integrity of control flow but to ensure that adversary tampering with the control flow will not affect the measurement results.

C4: Low Performance Overhead and minimal modifications. The challenge is to achieve practical and verifiable resource accounting with a relatively low performance overhead and minimal modifications to existing applications.

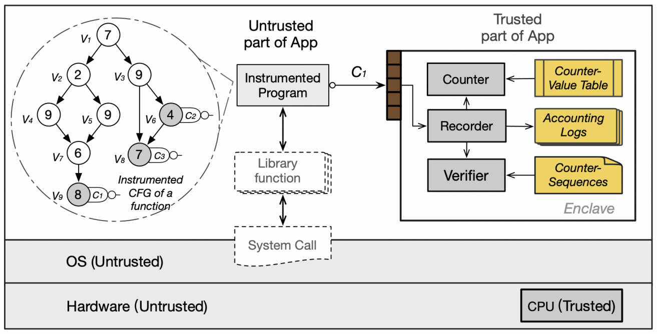


Fig. 3: Architecture Overview.

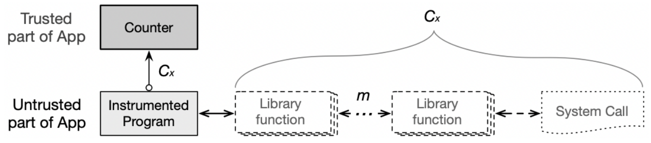


Fig. 4: Measurement of Library Functions and System Calls.

3.3 Architecture Overview

This section gives the architecture overview of the T-Counter (see Fig. 3). In the T-Counter, a program that runs in an untrusted cloud can measure its CPU usage by accounting instructions by the trusted part of itself. T-Counter mainly consists of two parts: an *instrumented program with counters* and a *trusted counter* that runs in the enclave. The trusted counter counts instructions by instrumented counters at the basic-block level in the program.

Instrumented Program with Counters. T-Counter accounts for a program's instructions based on counters instrumented in the compilation process. Based on these instrumented counters, T-Counter can record the path through the CFG and account instructions in the path. However, the instrumented program with counters runs outside the enclave, and the adversary can tamper with the program arbitrarily. If the value of counters is directly instrumented in the program, they are exposed to the adversary and easily manipulated. Therefore, we number all instrumented counters and put the specific value of these counters in the enclave for protection. Such an approach cannot resist all tampering attacks, but it can significantly reduce the adversary's attack surface. Moreover, T-Counter can perform integrity verification (see Section 5) in the enclave based on these counters' numbers. Another question is why we choose the way of instruction counting for CPU resource measurement. Compared to instructions, CPU cycles may be a more accurate metric to measure CPU usage. However, measuring CPU usage by accounting cycles is vulnerable to the manipulation of malicious OS (adversary). Therefore, instruction counting is the best method for securely and reliably estimating the CPU usage in our threat model.

A program's execution time comes from three parts: the application (written by the developer), the library functions it invokes, and the system calls that may be triggered (see Fig. 3). The library functions and system call are controlled by the underlying OS and are easily attacked. To ensure the integrity and improve the accuracy of the measurement, we propose methods for measuring library functions and system calls.

Library Functions. An alternative to measuring library functions is to statically compile the application and library functions that the application invokes together. We can measure library functions' CPU usage like the application by instrumenting counters in the library functions. Statically compiling the application can avoid using libraries provided by the OS and reduce the attack surface. However, static compilation changes the traditional program compilation and execution process. And, instrumenting more counters will lead to the T-Counter face more attacks (as analyzed in Section 5.1.) and increase the performance overhead of the T-Counter. Another alternative to measuring library functions is to measure them statically, as the cost of calling library functions is often relatively fixed. We can directly instrument a counter in the basic block where the library function invoked to measure the library function's cost (include library functions that the library function invokes). The counter's value can be calculated by static statistics of the instructions' number of library functions or by experiment measurement. We adopt the statical measurement method based on experiments in the T-Counter (see Section 7.6).

System Calls. The untrusted OS completely controls the system calls. The program cannot control the system calls nor measure them. Therefore, we also adopt a static measurement method for measuring the cost of system calls. We can measure the average cost of processing each type of system call and instrument counters to the basic blocks where the system call exists; the counter's value is the cost required to execute the system call. T-Counter can identify system calls by their names in the *system call table* during the compilation phase. During the program's running, the application was written by the developer seldom directly triggers system calls, usually through the library function indirectly triggers system calls. Therefore, in the design of the T-Counter, we measure the cost of library functions, and the system calls that the library function triggers together by experiments, as shown in Fig. 4), which can reduce the attack surface available to the adversary and the overhead.

Trusted Part of Program. The trusted part of the program does instruction counting and integrity protection. The trusted part is protected in the enclave against the adversary to tamper the accounting progress and result. The trusted mainly consists of three components: trusted counter, recorder, and verifier. (1) The **trusted counter** counts instructions that have been executed according to counters instrumented in the program. When the trusted counter receives a counter number from the program, it will search the *Counter-value table* to find the corresponding value of the counter, then add the instructions number of the counter to the total instructions number S; (2) To against adversary tampers counters' number, the **verifier** verifies the correctness of these counters' number by checking the order in which the counters are called. T-Counter uses a *Counter-Sequences Table* to store the sequence of counters. If a malicious OS modifies the counter sequence optionally, the T-Counter will be aware of the malicious behavior and send alerts to consumers. (3) The **recorder** records the sequences of counters in the paths in the CFG and generates accounting logs for online and offline verification.

4 Instrumented Program with Counters

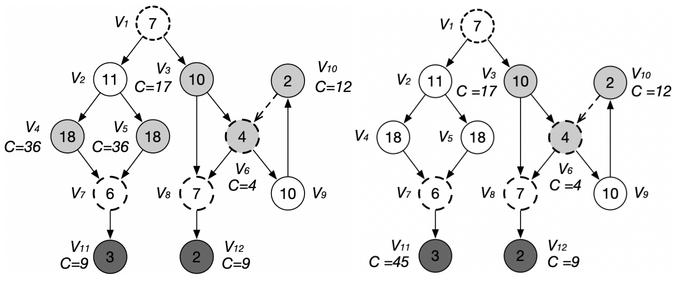
In this section, we introduce the algorithms for selectively instrumenting counters in the program. A key obstacle to building an universal CPU measurement solution is overhead; frequent ecalls will cause too high-performance overhead in T-Counter. So, T-Counter selectively instruments counters in a subset of basic blocks. We produce two algorithms: **Base Ins**trumentation algorithm and **Optimization Ins**trumentation algorithm, named **Base-Ins** and **Opti-Ins**. We implement two algorithms in LLVM passes.

4.1 Base Instrumentation Algorithm

We classify basic blocks in the CFG into five types, as described in Section 2.2. By analyzing the structure of CFG, we find that two types of basic blocks can break down the CFG into pathlets. The two types of basic blocks are (1) the exit basic block; and (2) predecessors of multi-links, which contains a particular case, the predecessors of the back edge's end node. Instrumenting counters in these two types of basic blocks can completely record the program's execution path no matter how the program is executed.

Exit Block and Multi-links Predecessor Instrumentation. The end nodes of back edges can be treated as special multi-links. Then, instrumenting counters in all exit basic blocks and multi-links predecessors of a CFG can fully record a program's execution path and count instructions with these instrumented counters; we call this algorithm as Base-Ins (The proof is shown in Section 6.1). Compared to instrument counters in each basic block, Base-Ins instruments less counter in the CFG. The work [14] also adopted a similar instrumentation scheme like Base-Ins. According to the Base-Ins algorithm, T-Counter instruments counters in all exit basic blocks and multi-links predecessors and calculate these counters' values as follows. The value of a counter C_x is a path's or a pathlet's weight that ends with counter C_x and start from the last counter C_l in the path. The value of C_x includes the weight of the basic block that C_x instrumented in and excludes the weight of the basic block that C_l instrumented in. If there is no counter before C_x, then the value counter C_x is the path's weight, or the pathlet ends with counter C_x, and start from the entry block. The value of C_x includes the weight of the block C_x that instrumented in and the entry block's weight. For the back edges, the marking process is similar due to the end node of a back edge must be a multi-links. In summary, the exit blocks and the predecessors of multi-links (includes the end node of the back edge) need to be instrumented counters.

We illustrate an example of Base-Ins in Fig. 5a. First, Base-Ins traversals G(V, E) and record the indegree and outdegree of all the nodes, then find entry block V_1 and all the exit blocks: V_{11} and V_{12}. These exit blocks need to be marked. Next, the Base-Ins algorithm processes V_{11} and V_{12} respectively to search multi-links. For the exit block V_{11}, we find multi-links V_7, and its predessors V_4 and V_5. For the exit block V_{12}, we find multi-links V_8 and V_6, and their predessors V_3, V_6 and V_{10}. For the back edge (V_{10}, V_6), we mark predecessors of node V_6: V_3, V_{10}. In summary, we need to instrument 7 counters in V_4, V_5, V_{11}, V_3, V_6, V_{10} and V_{12}. These counters' value is 36, 36, 9, 17, 4, 12, and 9, respectively.



(a) Base Instrumentation (b) Optimization Instrumentation

Fig. 5: An example of Base Instrumentation Algorithm and Optimization Instrumentation Algorithm.

4.2 Optimization Instrumentation Algorithm

Note the design goal of our approach; we are concerned about how many instructions are executed when the program ends with an exit block. As our analysis, not all exit basic blocks and predecessors of multi-links need to be instrumented counters. For example, if weights of paths that end with a multi-links are similar, the multi-links' predecessors do not need to be instrumented counters. So, we proposed the Opti_Ins algorithm.

For an exit block V_{exit}, (1) If there is only one path ends with V_{exit}, or there are several paths end with V_{exit} and the weights of these paths is similar, we only need to instrument counters in V_{exit}, and other basic blocks in the paths that end with V_{exit} are not needed to be instrumented counter. Because no matter how the program is executed, as long as the program exit from V_{exit}, the number of executed instructions is similar. (2) Otherwise, there are several paths end with V_{exit} and weights of these paths end with V_{exit} is not similar. In this case, we need to analyze further which basic blocks in the paths end with V_{exit} need to be instrumented counters. There are paths with different weights from entry block to exit block because there are multi-links in the paths. Multi-links means that multiple paths reach it. The predecessors of the multi-links can indicate which path the program goes through. But not all predecessors of multi-links are needed to instrument a counter. Similarly, If the weight of paths from entry to the multi-links is not similar, these multi-links' predecessors need to be instrumented counters. Instrumenting counters in the multi-links predecessors can indicate the pathlets and count instructions before they visit the multi-links.

DAG Instrumentation Algorithm. We can decompose a CFG can into a DAG and several back edges. We first analyze how to selectively mark nodes (basic blocks that need to be instrumented counters) in the DAG. The pseudocode for marking DAG is given in Algorithm 1. The input of the algorithm is a DAG G(V, E), a set of exit nodes exit_node[] of G, and an array *visit[]* to label whether a node has been visited, the initial value of elements in *visit[]* is false. The algorithm's output is a set of marked nodes mark[V] in the DAG; the initial value of elements in mark[V] is false. The algorithm processes each exit node respectively, for each unvisited exit node V_{exit} in exit_node[]: The algorithm first marks V_{exit} and sets the V_{exit} visited, then take reverse depth-first search from V_{exit}. The pseudocode of reverse

Algotithm 1 The Algorithm of markDAG(G)

```
Input: $G(V,E)$, $S_{exit}$ = {$\nu$ | $\nu \in V(G) \wedge \nu$.outdegree = 0 }
Output: $A\ set\ of\ marked\ nodes\ in\ G(V,E) : S_{mark}$
 1: for each unvisited $\nu \in S_{exit}$ do
 2:     $S_{mark} \leftarrow S_{mark} \cup \{ \nu \}$
 3:     $\nu$.visit ← true
 4:     Reverse_DFS($\nu$)
 5: end for
```

Algotithm 2 The Algorithm of Reverse_DFS(ν)

```
Input: $G(V,E)$, $entry$
Output: $A\ set\ of\ marked\ nodes\ of\ G(V,E) : S_{mark}$
 1: //Find all paths that start from entry and end with $\nu$
 2: $S_{path}$ = { $path|path.start = entry\ and\ path.end = \nu$ }
 3: //Compute the weight of all paths in $S_{path}$
 4: $S_{weight}$ = { $p.weight$ | $path\ p \in S_{path}$ }
 5: if |$S_{path}$ | $> 1$ && max($S_{weight}$) $\neq$ min($S_{weight}$) then
 6:     if $\nu.indegree = 1$ then
 7:         Reverse_DFS($\nu$.pre)
 8:     else
 9:         $S_{pre}$ = { $\mu$ | $\mu \in V(G) \wedge <\mu,\nu> \in E(G)$ }
10:         for each unvisited vertex $\mu \in S_{pre}$ do
11:             $S_{mark} \leftarrow S_{mark} \cup \{ \mu \}$
12:             $\mu$.visit ← true
13:             Reverse_DFS($\mu$)
14:         end for
15:     end if
16: else
17:     $\nu$.visit ← true
18:     return
19: end if
```

Algotithm 3 The Algorithm of markCFG(G)

```
Input: $G(V,E)$, $S_{exit}$ = {$\nu|\nu \in V(G) \wedge \nu$.outdegree = 0 }
Output: $A\ set\ of\ marked\ nodes : S_{mark}$
 1: Find all back edges in G(V, E)
 2: $S_{back}$= $\{<\mu,\nu> \mid <\mu,\nu> \in E(G) \wedge \exists path\ p : p.start = \nu\ and\ p.end = \mu \}$
 3: for each backedge $<\mu,\nu> \in S_{back}$ do
 4:     E(G) ← E(G) / { $<\mu,\nu>$ }
 5:     $S_{mark} \leftarrow S_{mark} \cup \{ \mu \} \cup \{ \omega|\omega \in V(G) \wedge <\omega,\nu> \in E(G) \}$
 6:     $\nu$.indegree ← $\nu$.indegree - 1
 7:     $\mu$.outdegree ← $\mu$.outdegree - 1
 8:     if $\mu.outdegree$ = 0 then
 9:         $S_{exit} \leftarrow S_{exit} \cup \{ \mu \}$
10:     end if
11: end for
12: After delete all backedges, G is a DAG
13: markDAG(G)
```

depth-first search (DFS) is given in Algorithm 2. The input of the algorithm is G(V, E), indegree of all nodes in the G(V, E) *indegree[V]*, weight of all nodes *weight[V]*, entry node entry_node, and a array *visit[V]* to label whether a node in the G(V, E) has been visited, the initial value of elements in *visit[V]* is false. For a node V, the algorithm first determines whether the reverse search has ended. If V is the entry_node, then the search end. Otherwise, our algorithm needs to determine whether to continue to search for reversely. It depends on the number of paths that start from the entry node and ends with node V: N and the weights of these N paths. If N is one, or N greater than one and the weights of N paths are similar, we don't need to search reversely from node V and the search ends. Otherwise, take a reverse depth-first search from the node V. Our algorithm searches reversely until finding a multi-links node U. For each previous node V_p of node U, mark V_p and set it has been visited, and then take a reverse depth search from it.

We illustrate an example of optimization instrumentation algorithm mark in Fig. 5b (exclude the back edge (V_{10}, V_6)). First, our algorithm traversals G(V, E) and find entry block V_1 and all the exit blocks: V_{11} and V_{12}, and the two exit blocks need to be marked. Next, our algorithm also processes V_{11} and V_{12} respectively to determine whether a reverse search is required. For the exit block V_{11}, there are two paths end with it: P_1(V_1, V_2, V_4, V_7, V_{11}) and P_2(V_1, V_2, V_5, V_7, V_{11}). Due to the weight of P_1 and P_2 is the same, so we end the reverse search from V_{10}. For the exit block V_{12}, there are also two paths end with it: P_3(V_1, V_2, V_4, V_9, V_{11}) and P_4(V_1, V_2, V_4, V_7, V_9, V_{11}). Because the weight of P_3 and P_4 is different, so search reversely from V_{12} until finding the first multi-links V_8. Then T-Counter process V_3 and V_6 respectively, take V_3 as an example, first mark V_3, and search reversely from the V_3. Because the number of paths that end with V_3 is one, so search ends. In general, V_3, V_6, V_{10}, V_{11}, and V_{12} need to be marked.

CFG Instrumentation Algorithm. We have introduced the algorithm for marking DAG. We can decompose a CFG to a DAG and several back edges and process the back edges and the DAG, respectively. Pseudocode for marking CFG is given in Algorithm 3. The input of the algorithm is digraph G(V, E), indegree of all the nodes in G: *indegree[]*, outdegree of all the nodes in G: *outdegree[]*, and a set of exit nodes exit_node[]. The algorithm's output is a set of marked nodes mark[V] in the CFG; the initial value of elements in mark[V] is false. Our algorithm first traversal G(V, E) and find all the back edges E[G] in G(V, E). For each back edge (V_i, V_j), we first mark the node V_i and all other previous nodes of V_j. These paths that end with V_j can be distinguished by V_i and the other previous nodes of V_j. Then, we delete the back edge (V_i, V_j) from E(G) and update the indegree of V_j and the outdegree of V_i, the indegree of V_j, and the outdegree of V_i need minus 1. If the outdegree of V_i becomes 0 after minus 1, we add V_i to the array of exit nodes exit_node[] because V_i becomes a new exit node due to deleting the back edge (V_i, V_j). After we delete all back edges in G(V, E), G(V, E) has become a DAG. At last, we mark the G(V, E) using the algorithm of marking DAG (Algorithm 1).

We illustrate an example of marking CFG in Fig. 5b. There is one back edge in the CFG: (V_{10}, V_6). For the back edge (V_{10}, V_6), we delete it from G(V, E) and mark V_{10} and the other previous nodes of V_6, and update the indegree V_6 as 1 of and the outdegree of V_{10} as 0. Because the outdegree V_{10} of is 0, so V_{10} is an exit node now, we add it to exit_node[]. After, we delete the back edge (V_{10}, V_6), G(V, E) has become a DAG. Then we mark the DAG using algorithm 1. The mark result is V_3, V_6, V_{10}, V_{11} and V_{12}.

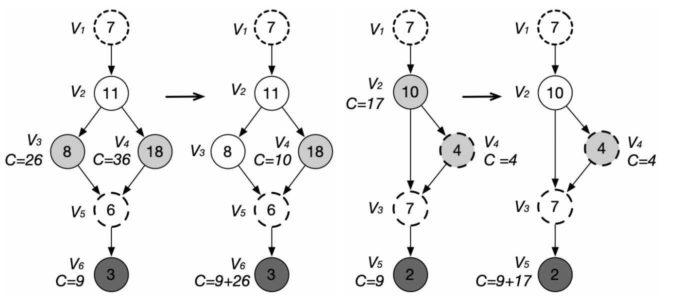


Fig. 6: An example of Counter Number Optimization.

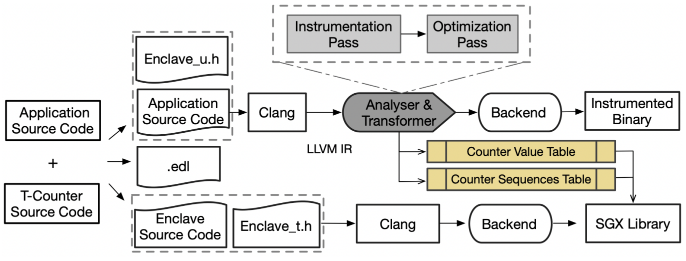


Fig. 7: Instrumenting counters and Integrating T-Counter with SGX SDK.

Counter Value and Optimization. We need to instrument counters in the marked nodes. The value of counter C_x is the weight of the path that ends with C_x and start from the last counter C_l. The value includes the weight of the block C_x instrumented in and excludes the weight of the block C_l instrumented in. If there is no counter before C_x, then the value counter C_x is the weight of a path ending with counter C_x and starts from the entry block. The value includes the weight of the block C_x that instrumented in and the entry block's weight.

Based on our marking algorithms, we further give two optimization strategies for instrumenting counters. For a multi-links V_m, not all predecessors of V_m need to be instrumented counters. We can reduce some counters by modifying the counter's value to make the program run through as few counters as possible. As shown in Fig. 6, there are two specific cases. (1) In Fig. 6a, the multi-links V_5 has two predecessors V_3 and V_4, and V_3 and V_4 are not the predecessor to each other. In such a case, we can instrument counters in V_4, set the counter's value as w_4-w_3, and add w_4-w_3 to these exit nodes that V_5 ends. The instrumented result is as shown in the subfigure. If a multi-links has more than two predecessors, we can omit counters at the predecessors with the same weight and the largest number. (2) In the Fig. 6b, the multi-links V_3 also has two predecessors V_2 and V_4, and V_2 is the predecessor of V_4. In this case, we can instrument counters in the successor nodes of them, so we instrument counter in the V_4 with weight W_4, and add W_2 to these exit nodes that V_3 ends.

4.3 Hotspot Optimization Strategies

According to the 90/10 law, 90% of the execution time is spent in 10% of the code [42]. Most of the execution time is spent in loops. A loop corresponds to a back edge in CFG. As our instrumentation algorithm design, each iteration of the loop needs at least an *Ecall* to record the CPU usage, which will lead to high performance overhead. Because there are always dozens of instructions in a loop, the overhead of an Ecall is thousands of cycles, much more than the number of instructions in the loop. Spend thousands of cycles to record dozens of instructions usage is unwise. As our analysis, the program call counter for each iteration of the loop will lead to high overhead. We propose two solutions to solve this problem: (1) Moving counters out of loops, or (2) Randomly Ecall at the hotspot to avoid calling the counter for each iteration of the loop.

Strategy 1: Moving counters out of loops. If the iteration of the loop is a constant, the number of instructions in each iteration is definite. We can instrument counter only once after exiting the loop, instead of instrumenting the instruction counter in each iteration. The counter's value is the number of instructions in the loop's body times the number of iterations. This strategy can only apply to control-flow independent instructions inside the loop body. If a loop variable is identified, we need to use Strategy 2.

Strategy 2: Randomly Ecall at the hotspot. Assuming that iterations of a loop is n, sum of instructions of each iteration is c. We can use a random function produces number x between 0 and m-1. If x equals 0, the counter will call the trusted counter to account; the recorded value is $m \times c$. Otherwise, the counter will not call the trusted counter. It means that each iteration of the loop with $1/m$ probability to call the trusted counter. For each iteration of loop, the expectation of instruction number is $(1/m) \times m \times c$, it equals c. A confusing problem is how to select the m; it is a balance of performance and accuracy. According to experiments, when m is less than 100, the overhead is still unacceptable. When m greater than 10000, the error of accuracy is greater than 5% sometimes, it is unsatisfactory. So we advise m's suitable range is from 100 to 10000, and we set m as 1000 in our experiments. Attackers could try to exploit this optimization by tampering with random number intervals and target. T-Counter against this by allowing an ecall to get a random interval (min, max) and target t before execute the loop. In this way, attackers cannot tamper with (min ,max) and t.

4.4 Selectively Instrumenting with LLVM Pass

we implement our algorithms as LLVM IR optimization passes. Our extended Clang instruments basic blocks that are marked and add instructions that *call* the trusted counter in the enclave to measure the instructions. The detailed work progress of LLVM is shown in Fig. 7. The LLVM takes the application source code as input and compiles the source code to IR. Our instrumentation pass analyzes and instrument counters in the IR code and generates the counter-value table and counter-sequences table, respectively. LLVM backend compiles the IR to instrumented binary. The counter-value table and counter-sequences table are used to verify the correctness and integrity of counting by counter and verifier. T-Counter library code is compiled using the clang compiler to an object file. Fig. 8 shows a snippet of instrumented assembly code. The enclave is finally signed by an Intel SGX signing tool, creating an application Enclave.

```
loc_18E3:
lea       rax, global_eid   #added by T-Counter
movsd     xmm0, [rbp+var_20]
movsd     xmm1, [rbp+var_40]
mov       ecx, [rbp+var_C]
shl       ecx, 1
sub       ecx, 1
cvtsi2sd  xmm2, ecx
divsd     xmm1, xmm2
subsd     xmm0, xmm1
movsd     [rbp+var_20], xmm0
mov       rdi, [rax]
lea       rsi, a_z5cmainv   ;"_Z5cmainv"
mov       edx,0Ch
call      ecall_add        #added by T-Counter
```

Fig. 8: A snippet of assembly code instrumented with the T-Counter. Global_eid is the ID of the enclave that the trusted part of the application runs in; the call instruction to call the enclave is incremented at the ending of the basic block.

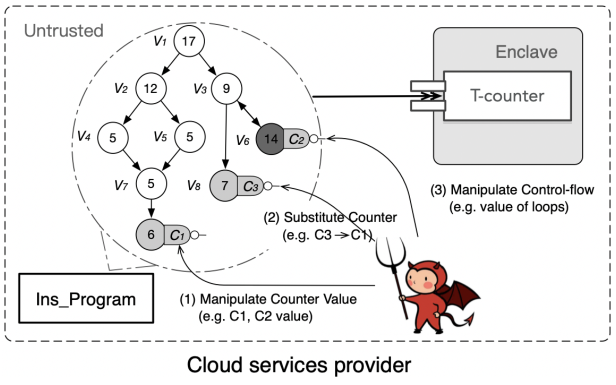


Fig. 9: Three specific types of attacks against T-Counter.

5 Trusted Counter in Enclave

In this section, we discuss several types of attacks against T-Counter, detail the components of the trusted part of the program, and introduce the protection mechanism of accounting integrity and confidentiality.

5.1 Attacks on Instructions Accounting

As described in the threat model (in Section 3.1), the attackers can modify the program's constants outside the enclave. As the program runs outside the enclave, system services are under the control of the malicious OS. We can not measure the CPU usage based on any services provided by OS. It is a challenging problem to ensure that instructions are counted correctly, even in the presence of malicious OS. So, as our design, we account for instructions by the application itself without any system services. More challenging is that the application is also under the control of the malicious OS. Therefore, the T-Counter must defend against direct attacks on the values of instrumented counters. We will discuss several types of attacks against resource metering in this section. Specifically, based on the assumption of the adversary's attack capability, we consider three specific types of attacks against reliably accounting of T-Counter: 1) counter manipulation attacks; 2) counter substitution attacks; and 3) control-flow manipulation attacks, see Fig. 9. These attacks may tamper with the measurement result.

A1: Counter manipulation. Malicious servers may modify the value of counters instrumented in the program. And this will result in an incorrect measurement result. If we instrument the counter's value in the basic block directly, the malicious server can modify the value, resulting in larger measurement results. So, we number counters and store the specific value of the corresponding counter in the enclave. It can reduce the adversary's attack surface.

A2: Counter substitution. As described in A1, if an adversary cannot directly tamper with the counters' value, it may tamper with measurement results by substituting the value of counters with values of the other counters instrumented in the program. And this will also result in an incorrect calculation result. For example, a malicious CSP can substitute the counter C1 with C2, see Fig. 9.

A3: Control-flow manipulation. Programs are running outside the enclave; a malicious CSP may statically modify code(such as constants) to hijack the program's control flow to tamper with the measurement result. For example, a malicious CSP may modify the registers that control the flow so that the process takes a much longer execution path to finish, such as reducing the number of loops. Our scheme saves sequences of counters in the enclave and checks the program's control flow based on these sequences to protect the control flow. We will detail this in 5.2

5.2 Components of Trusted Part.

In this section, we detail components of the trusted part. According to the threat model and attacks on instructions accounting (in Section 3.2 and Section 5.2), we design three components and three data structures in T-Counter: trusted counter, recorder and verifier, counter-value table, counter-sequences table and accounting logs. The details of these components are as follows.

• **Trusted Counter.** The function of the trusted counter is to account for the instructions. The counter receives a counter and searches the counter-value table to find the counter's relevant value, then account for the instructions number that has been executed. The ***Counter-Value Table*** is used to stores the value of counters instrumented in the program. As described in attack A1, the attacker can modify the program's constants outside the enclave. If the value of counters is directly instrumented in the program, it is directly exposed to the attacker and easily manipulated. In our approach, the ”values” instrumented in the program are just the counters' number; the real values of counters are stored in the enclave.

• **Verifier.** As the counters' values are stored in the enclave, the adversary can no longer arbitrarily tamper with the accounting result. But, the adversary still can modify the number of counters to interfere with instructions accounting. The adversary can attack by substituting the counter number with other numbers in the counter-value table (A2). We can define a series of rules for checking the counters' correctness, such as the order of counters called. The verifier uses the counter-sequences table to verify the counters' correctness against A2 and verify the integrity of program control flow against control-flow manipulation (A3). If a malicious OS modifies the counter sequences optionally, the verifier will be aware of the malicious behavior and send an alert to the consumer. The correctness check dramatically reduces the attack surface against T-Counter. We will detail the verification strategies in Section

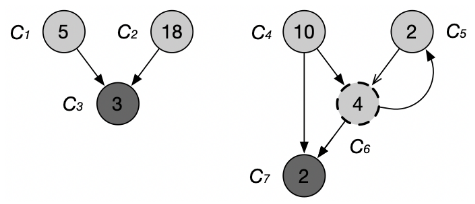


Fig. 10: The forest of counter sequences.

5.3. The ***Counter-Sequences Table*** is used to store the forests of counter sequences instrumented in the program. For example, the sequence $< C_1, C_3 >$ and $< C_2, C_3 >$ shown in the Fig. 10. The counter-sequences table and counter-value table are generated by LLVM in the compilation progress and integrated with the trusted part, as described in Section 4.4.

• **Recorder.** The recorder is used to record the counter number sent from the untrusted part for online verification and to generate accounting logs for offline verification. The ***Accounting Logs*** are generated by the recorder; these logs can be used to offline verification and solve future resolutions between CSPs and consumers.

5.3 Verification Strategies for Accounting Integrity

As we described in attacks on counters, the attacker can not modify the value of a counter anymore due to the value stored in the enclave. The "value" instrumented in the counter is the number of the counter. Generally, more than one instrumented counters in a CFG, and each counter's value is often different. The adversary can also get the value of these counters via calculation. In this case, the adversary can replace the counters' number with the numbers of other counters with enormous value. To defend against this type of attack, we define the counter-sequences table and design the verifier to verify the counter sequence. Only storing the counters' sequences than holding the entire CFG can significantly save storage space and demonstrate the control flow faster. We give an example in Fig. 10. For example, when the sequence of counters passed during program execution is $< C_1, C_3 >$, the adversary can replace one or both of the two counters. Suppose the adversary individually replaces counter C_1 with counter from C_3 to C_7, or individually replace counter C_3 with counter from C_1 to C_7. In that case, the verifier will notice the malicious behavior, as the counter sequence can't match any counter sequence in the counter-sequences table. But if the adversary modify $< C_1, C_3 >$ as $< C_2, C_3 >$, The verifier cannot notice the malicious behavior because the counter sequence $< C_2, C_3 >$is also legal in the counter-sequences table. The verifier the adversary can also replace the sequence $< C_1, C_3 >$ as $< C_4, C_7 >$, the verifier even can't aware of the modification directly. But these counter sequences will be recorded and checked out by the offline verification because the recorded sequence is conflicting with the counters sequences that the program executes typically. In general, our approach can resist most of the adversary's substitution attacks to the counters offline but cannot wholly resist all of them in time online. Our design significantly increased the difficulty and cost of the adversary's attacks and reduced the impact of the adversary's attack.

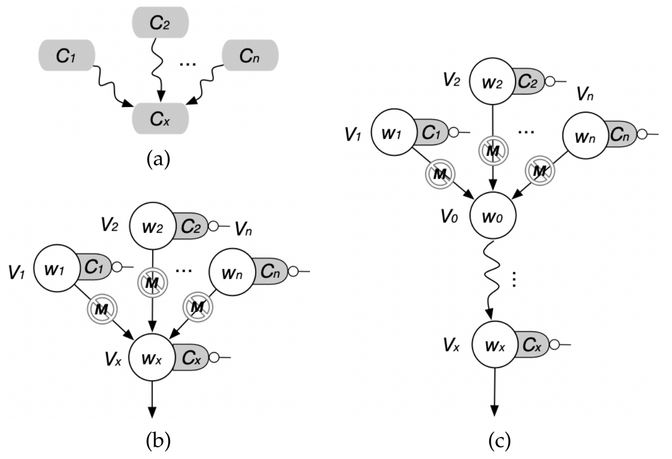


Fig. 11: Proof of Base Instrumentation Algorithm

6 Theoretical Analysis

In this section, we first give the correctness analysis of the proposed T-Counter (in Section 6.1) and then demonstrate its security(in Section 6.2). Last, we quantitatively analyze some user cases' efficiency in terms of latency and runtime to demonstrate the optimal solution (in Section 6.3).

6.1 Correctness Analysis

In this section, we give the correctness analysis of instrumentation algorithms. For a counter C instrumented in the CFG, we need to prove the pathlets between counters that directly link to C and C with the same weight, see Fig. **??**.

Proof. Suppose there is a counter C_x in the Marked CFG. (1) If there are no counters before C_x in the path, C_x is the first counter in the path from the entry node, then the value of C_x is the weight of the pathlet end with C_x. (2) If there are counters before C_x, the number of counters that link to C_x directly through different pathlets must be more than one (without counter optimization), as the design of our algorithms. It means that there are multiple pathlets P_1, P_1, ..., P_N, start with these counters and end with counter C_x. We need to prove that these pathlets has the same weight. The counters omitted by the counter optimization strategy can be regarded as a counter with a value of zero. The weights of these pathlets that start from the entry node and end with C_x must be different from our algorithms' design. Because the weights of these pathlets are the same, there will be no counter before C_x. As these pathlets' weight is different, we need to search reversely from C_x until finding one node V_m, which indegree larger than 1. We assume the node that C_x instrumented in is V_x. There are two cases: 1) one case is that the indegree of V_x is larger than 1, V_x is the V_m, see Fig. 11b, then the value of C_x is the weight of C_x. 2) another case is that the V_x's indegree is one, there exists a node V_0 before V_x, and it's the indegree larger than one, and the degree of all the nodes that between V_0 and V_x is one. In other words, there exists only one path V_0 and V_x, V_0 is the V_m, see Fig. 11c. Therefore, these pathlets start from V_0 and end with C_0 consist of the same nodes, so the weights of these pathlets are the same. In general, the nodes between V_x and V_m (maybe V_x itself) are the same under any condition.

6.2 Security Analysis

In this section, we give the security analysis of our T-Counter framework. We first analyze the measurement confidentiality of the T-Counter. Then, we prove that T-Counter can realize verification. Finally, we demonstrate that T-Counter can also realize execution integrity.

Measurement confidentiality. In our design, a trusted counter is running in the enclave to ensure the confidentiality of measurement results against the malicious CSP, as explained in Section 3.3. And, the values of counters are also protected in the enclave instead of directly instrumented in the program that runs outside enclave. We design the counter-value table to store the value of counters in the enclave. The malicious CSP can not get the specific values of counters and damage the measurement process's integrity. Therefore, even a malicious CSP cannot damage the accounting integrity of the process and result. In a word, T-Counter can protect the integrity of resource measurement.

Measurement Verifiability As we described in Section 5.1, the adversary can not tamper with the "values" of counters that instrumented in the program directly because these values are just numbers of these counters. The adversary can only substitute counters' numbers with other legal counter numbers stored in the counter-value table. To defend against such type of attack, we design the verifier and verification strategies to verify the counters' number by checking sequences of received counters' number. The counter-sequences table can contain the relationship between counters instrumented in the CFG of a function and the relationship between counters instrumented in the CFG of different functions. As we analyzed in Section 5.3, the verifier can detect most substitution attacks online and detect all substitution attacks offline with the counter sequence corresponding to the execution result. In general, T-Counter can verify the process and result of the measurement.

Execution Integrity The instrumented counters can be used not only for accounting resource usage but also to verify the control-flow integrity (CFI) [43] of the program. As our analysis in Section 3.1, the malicious CSPs have many ways to hijack the program's control flow in our threat model. Such as non-control-data attacks [44] to manipulate variables that affect the program's control flow, jump-oriented programming attacks [45] exploit indirect jump and call instructions at the end of each invoked code sequence, et. CFI is one of the main directions in the research field of runtime exploits mitigation. It can ensure that a program only follows a valid path in the program's CFG. Many works have been done to make CFI be an efficient and effective mitigation technology [46] [47]. But, CFI does not cover non-control-data attacks [44], leading to the execution of unauthorized but valid CFG paths. T-Counter can attest to the execution path of a program running outside the enclave. The counters instrumented in the CFG divides CFG into several pathlets and can indicate which pathlets the program goes through during execution. And the sequences stored in the counter-sequences table to verify the correctness of control flow. Execution integrity here is to detect but not defend against the behaviors that the adversary damages measurement integrity. C-FLAT [48] is a similar attestation scheme in the embedded system.

TABLE 3: Performance Estimation of Six Schemes‡

Schemes	Runtime	Average Overhead
Native	T_P	-
Baseline	$T_B = T_P + T_{ini} + T_{des}$	$O_B = T_{ini} + T_{des}$
Base-Ins	$T_B + N_{base} \times T_{ecall}$	$O_B + \underline{N_{base} \times T_{ecall}}$
Opti-Ins	$T_B + N_{opti} \times T_{ecall}$	$O_B + \underline{N_{opti} \times T_{ecall}}$
Swtichless	$T_B + N_{opti} \times T_{switch}$	$O_B + \underline{N_{opti} \times T_{switch}}$
Hotspot	$T_B + N_{opti} \times T_{switch}$ - $N_{hots} \times (N_{rand}\text{-}1) \times T_{switch}$ $+T_{rand} \times N_{hots}$	$O_B + N_{opti} \times T_{switch}$ $-N_{hots} \times (N_{rand}\text{-}1) \times T_{switch}$ $+T_{rand} \times N_{hots}$

(‡In this table, "T_P" is the runtime of application itself, "T_B" is the runtime of Baseline scheme, "T_{ini}" and "T_{des}" are the runtime of SGX enclave initialization and destruction respectively, "O_B" is the overhead of Baseline scheme, T_{ecall} is the runtime of one ECall (include one OCall), N_{base} is executed counters based on the Base-Ins algorithm, N_{opti} is executed counters based on the Opti-Ins algorithm, T_{switch} is the runtime of one ECall without context switch, N_{rand} is the random number chosen in hotspot scheme, N_{hots} is the number of hotspot in hotspot scheme,T_{rand} is the runtime of random function runs one time.)

6.3 Efficiency Analysis

This section quantitatively estimates average latency and overhead for seven applications in six schemes: native, baseline, base-ins, opti-ins, switchless, hotpots. These six schemes are defined as follows, and their estimated results are revealed in Table 3.

- **Native and Baseline**: We first run the unmodified application in an environment without enclave configuration. We measure the runtime of the unmodified application as a **native** reference for experimental evaluation. Then, we run the unmodified application in an environment with enclave configuration, which mainly includes the enclave's initialization and destruction but does not include ecalls. We measure the runtime of applications that run in this configuration as **baseline** scheme. The overhead of baseline compared to native is T_{ini}+T_{des}.
- **Base-Ins Scheme and Opti-Ins Scheme.** We implemented two instrumentation algorithms as two instrumentation schemes: **base-ins scheme** and **opti-ins scheme**, and evaluated the performance of the two schemes, respectively. The Base-Ins scheme instrumenting ecalls according to the Base-Ins algorithm, and the Opti-Ins scheme instrumenting ecalls to the application according to the Opti-Ins algorithm. Then, we measured the runtime of two schemes, respectively. The overhead of these two schemes is the number of executed counters times T_{ecall} compared to baseline.
- **Switchless Optimization and Hotpots Optimization.** As we described in Section 2.1, the strong security offered by SGX does not come for free; the overhead of an ecall is over 8000 CPU cycles, which is 50X more expensive than that of a system call [32]. Due to the high overhead after instrumented ecalls, we adopted a series of optimization strategies. The optimization strategy is divided into two parts to achieve and evaluate. We first achieve **swithless optimization** scheme that adopted "switchless call" [49] (we detail it in Section 7.2). The optimization runtime is $N_{opti} \times (T_{ecall}\text{-}T_{switch})$ compared to opti-ins scheme. Based on the opti-ins scheme. Based on the swithless scheme, we use the hotspot strategy to achieve **hotpots optimization** scheme that further optimize the loops. Similarly, we measured runtime of two schemes respectively. The optimization runtime is $T_{switch} \times (N_{opti}\text{-}N_{hots} \times (N_{rand}\text{-}1))$-$T_{rand} \times N_{hots}$ compared to switchless scheme.

In the next section, we will perform some experiments to compare these schemes' performance in reality.

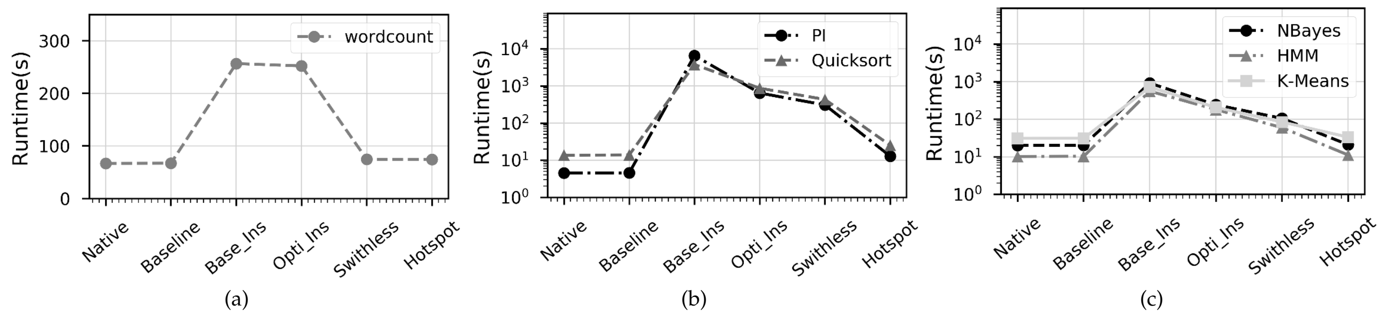


Fig. 12: Runtime comparisons of six schemes of six applications..

6.4 Accuracy Analysis

In this section, we give the accuracy analysis of the T-Counter. Due to the CPU's internal design, the number of instructions cannot be completely equal to the CPU cycles overhead. Different instructions require different cycles to execute. Although we can measure a list of cycles required for the execution of all instructions, and use the list in the path analysis in T-Counter. AccTEE adopted such a solution [25]. But the cost of instructions depend on CPU architecture, CPU generation and the runtime. In fact, even we measure the cost of each instruction in detail, we can't get accurate results, so we did not adopt AccTEE's method in our design. We directly use the average cost of all instructions as the weight and multiply the weight by the number of instructions as the final CPU overhead. Such a strategy can not only greatly reduce the difficulty of path analysis but also fully guarantee the accuracy of the measurement. The specific experimental results are shown in Section 7.6.

7 Implementation and Evaluation

To validate the efficiency estimation presented in Section 6.3, we design several experiments to evaluate T-Counter's performance in seven use cases. The T-Counter framework is implemented by extending the LLVM [23] and programmed in C programming language with about 850 LOC. We benchmark constructed SGX applications on Dell Optiplex 7050 laptop, which is equipped with a 2.60GHz Intel I7-7700 Skylake CPU with four cores and supports SGX extensions, 16GB RAM, and 1TB disk. The size of EPC was the default, 128MB. The operating system was a Ubuntu 18.04 LTS operating system with Linux kernel version 3.19.0. In our experiments, we use the SGX Software Development Kit (SDK) in version 2.7.1. We evaluate the runtime of 7 applications in the six schemes. The average runtime overhead of the two instrumentation algorithm is 3.82 (wordcount) - 1441.62(PI) times and 3.75 (wordcount) -141.33 (PI) times, respectively. The average optimization of 2 optimization schemes are 50.32 % (Quicksort) - 70.45 (wordcount)% (the random number is 1000) and 0 (wordcount) - 95.82% (PI).

7.1 Integration T-Counter with SGX.

The T-Counter is designed independently of the software development environment. For demonstration purposes, we integrate T-Counter with the official Linux SGX SDK [12]. The workflow for integrating T-Counter with the official Linux SGX SDK (see Fig. 6). Following the standard use case of Intel SGX described by the SDK, these applications' source code is separated into an application part and an enclave part. The separation is enabled by the SGX SDK with an *edl* file, which is manually created by the developer and specifies which files and functions are to be compiled into which parts. With the help of an SDK-provided tool called edger8r, two header files are generated to help the two parts interact with each other [12]. The standard SDK compiles the application source code using the GCC compiler. To enable program analysis and instrumentation, we replace the compiler for the source code with Clang [36]. The standard SDK compiles the enclave source code using the GCC compiler. We leave this part unchanged; the compiled binary is the SGX library loaded into the enclave.

7.2 Asynchronous Call via Switchless Calls

The high-performance overhead incurred by enclave switches is especially problematic, where ecalls are called in a high frequency to call the trusted part inside the enclave. To minimize the performance overhead incurred by ecalls/ocalls, several previous works [16] [32] [50] propose techniques that share the same core idea: caller threads send the requests of ecalls/ocalls into shared untrusted buffers, from which the requests are received and processed asynchronously by worker threads. These approaches make ecalls/ocalls does not trigger enclave switches. Intel team calls this type of technique *Switchless Calls* and adopted it in the work [49]. Switchless calls provide a speedup over the default interface. The Intel SGX SDK [12] has provided the switchless calls from version 2.2. We adopt switchless calls in our switchless scheme and hotspot scheme.

7.3 Use cases.

We implement six representative use cases of computation intensive in C/C++ programming language, including three micro benchmarks: wordcount [51], PI and Quicksort, 3 machine learning applications: **N**aive Bayes (NBayes), K-Means and **H**idden **M**arkov **M**odel (HMM), and or select a web server application: Thttpd [52].

WordCount. WordCount is a distributed system Hadoop MapReduce application. It is a simple application that counts the number of occurrences of each word in a given input set [51]. We implement the *map* function in the C programming language and take it as a use case. The size of the input set is 860M.

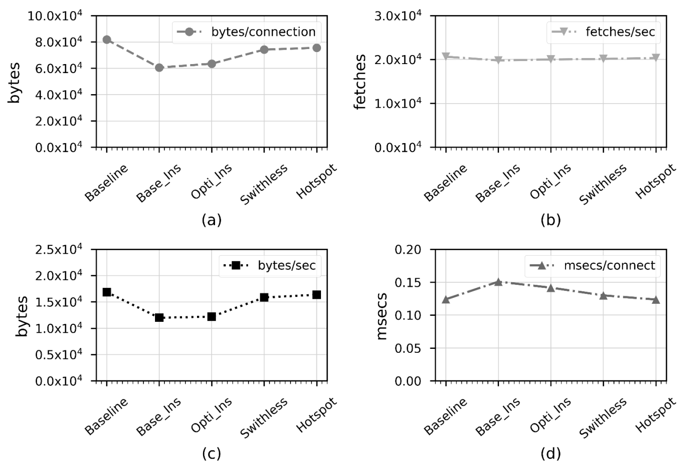


Fig. 13: Performance comparisons of Thttpd in 5 different schemes. (a) Mbytes/connection, (b) fetches/sec, (c)bytes/sec and (d) msecs/connect in different schemes.

Benchmarks. Quicksort is a commonly used algorithm for sorting. PI is an algorithm to approximate π with arbitrary accuracy (repeat times). Quicksort and PI have commonly used microbenchmarks. We implement them in the C programming language as benchmarks to evaluate our schemes. The accuracy of PI as 9×10^5 in our evaluation.

Machine Learning (ML). NBayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naive) independence assumptions between the features. K-Means clustering is a method of vector quantization that aims to partition n observations into k clusters. Each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. HMM is a probabilistic system designed to model a sequence due to a markovian process that cannot be observed. We implement three applications in the C++ language.

Thttpd. Thttpd is a simple, small, portable, fast, and secure HTTP server [52]. It doesn't have many unique features, but it suffices for most web uses; it's about as fast as the best full-featured servers (Apache, NCSA Netscape) has one handy feature that no other server currently has. We use the Thttpd in version 2.25.

7.4 Instrumentation Overhead and Optimization

In this section, we will show the overhead of two instrumentation schemes. We first evaluate the runtimes of these six benchmarks and applications. Then, we evaluate the Thttpd server's overheads, including the bytes/connection, fetches/sec, bytes/sec, and msecs/connect. We repeat each experiment 10 times to obtain the mean value.

Base-Ins Algorithm Overhead Fig. 12 shows the runtime overhead of six applications in four schemes with T-Counter compared to the native and baseline scheme. From Fig. 12, we can observe that runtime of these six applications in the baseline scheme increases about 20ms-210ms compared to the native scheme. It shows that the runtime overhead of enclave initialization and destruction is relatively certain. From Fig. 12a, we can observe that runtime of base_ins scheme of wordcount increases 2.1× compared to baseline. In Fig. 12b, we can observe that runtime of base_ins

TABLE 4: The LoC (Lines of Code) and NoC (Number of instrumented Counters) of six Applications.

Works	Native	Baseline	Base_Ins		Opti_Ins		Switchless		Hotspot	
	LoC	LoC	LoC	NoC	LoC	NoC	LoC	NoC	LoC	NoC
Wordcount	48	52	57	5	56	4	59	4	63	4
PI	59	63	71	8	69	6	72	6	76	6
QuickSort	57	61	70	9	68	7	71	7	77	7
NBayes	322	326	352	26	347	21	350	21	380	15
HMM	626	630	672	42	667	37	670	37	710	37
K-Means	616	620	661	41	656	36	659	36	717	36
Thttpd	9750	9754	10552	798	10552	798	10555	798	10555	798

scheme of PI and Quicksort increases 1441.62× and 271.92× respectively compared to their baseline. The runtime of base_ins scheme of three machine learning applications increases 44.77×, 53.85×, and 23.07× compared to their baseline, as shown in Fig 12c. The performance overhead of the base_ins scheme of these applications ranges from several times to hundreds of times. The base_ins scheme selectively instrumented less than 20% basic blocks in the CFG. If we do not selectively instrument counters by our base_ins algorithm, the performance overhead will be one to two orders of magnitude higher. In general, our base_ins algorithm adds one to three magnitudes of performance overhead compared to the baseline scheme. The high-performance cost is because the cost of the context switch caused by ecalls is much higher than the cost of the program itself.

Opti-Ins Algorithm Overhead and Optimization Fig. 12 shows the runtime of six applications in the opti_ins scheme. We observe that the overhead of the opti_ins scheme ranges from several times to hundreds of times. We also observe a noticeable reduction in overhead of opti_ins scheme of six applications to base_ins scheme. Significantly, the overheads of PI, Quicksort, and three ML applications reduce by nearly an order of magnitude. The overheads of PI and Quicksort reduce 90.2% and 76.94%; the overheads of three machine learning applications reduce 73.48%, 71.19%, and 67.44% separately. The ingenious design of our opti_ins algorithm reduces counters compared to the base_ins algorithm, which significantly reduces the counters (Table 4) triggered by the program during execution.

For the non-computation intensive web application Thttpd, instrumentation has a much smaller impact on its performance (see Fig. 13. In the same settings, compared to the baseline scheme, the bytes transferred in every connection decrease 25.92% and 22.37% in base_ins scheme and opti_ins scheme separately (see Fig. 13a). The fetches processed in every second decrease 4.13% and 3.07% in two schemes separately (Fig 13b), the bytes transferred in every second separately decrease 28.89% and 3.07% in two schemes (see Fig 13c), the delay per connection individually increase 21.46% and 14.14% in two schemes (see Fig 13d).

7.5 Performance Optimization

This section will show the performance optimization of the switchless scheme and hotspot scheme, respectively.

Swithless Optimization. As shown in Fig. 12, the performance of the opti_ins scheme has improved several times compared to the base_ins scheme. And, the number of counters instrumented and executed is challenging to reduce. But, several times or even dozens of times performance

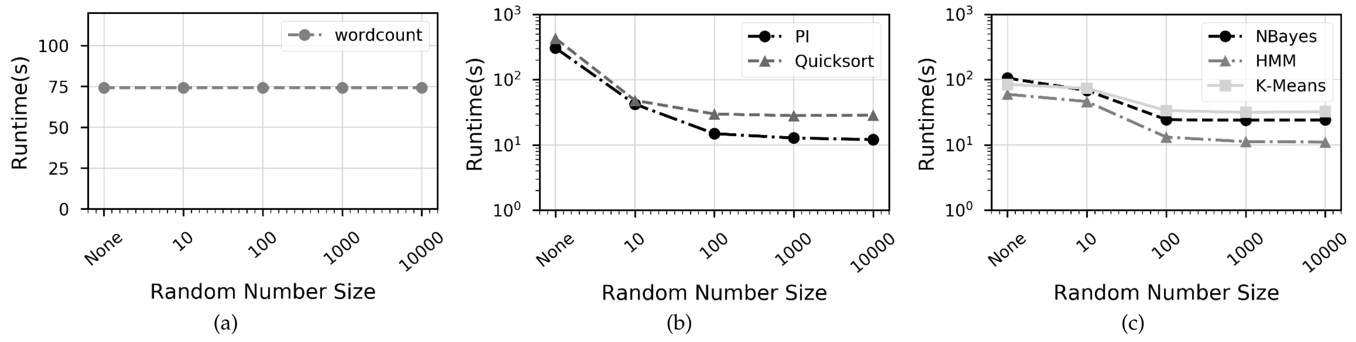


Fig. 14: Runtime comparisons of six applications' hotspot scheme with different number range.

overhead is still unacceptable for consumers. It is mainly due to the context switch caused by ecalls. We adopt the switchless call supported by SGX SDK in the switchless scheme to avoid context switches. From Fig. 12, we also observe a noticeable reduction in overhead of switchless scheme of six applications to opti_ins scheme. The overhead reductions of the switchless scheme of six applications compared to their opti_ins scheme: wordcount is 70.45%, PI and Quicksort are 52.27%, and 50.32%, 56.31%, 67.04%, and 59.01%. In general, the switchless schemes' overhead reduces by more than 50% compared to opti_ins schemes.

Hotspot Optimization. Fig. 12 shows that these applications' hotspot schemes' performance further improves compared to their switchless schemes, the performance improvement of the hotspot schemes are 0.49% (wordcount), 95.82% (PI), 94.15% (Quicksort), 80.06% (NBayes), 81.23% (HMM) and 60.48% (K-Means). And compared to the baseline schemes, the performance overhead of the hotspot schemes is 10.32% (wordcount), 181.76% (PI), 82.24% (Quicksort), 3.39% (NBayes), 8.52% (HMM), and 7.67% (K-Means). In general, three machine learning applications achieve low-performance overhead from 3.39% to 8.52%, and the word count also reaches good performance optimization. PI and Quicksort's lousy performance is because they only have dozens of lines of code; this leads to the runtime of instrumented code is close to the runtime of application code. Such situations only occur in a few cases. Compared with the performance of these existing trusted resource usage measurement systems, such as S-Faas (5.3%/6.3%) [13], VeriCount (3:62%/16:03%) [11] and AccTEE [25] (10%), T-Counter achieve better performance.

As we analyzed in Section 4.3, the existence of the hotspot is an important reason for high-performance overhead. We use a random call to solve the problem and evaluate the runtime of these applications' hotspot schemes with the random number' range of random number m from 10 to 10^4 (see Fig. 14). Fig.14b shows that the performance optimization of PI and Quicksort is getting better obviously with m from 10 to 10^3. But, when m from 10^3 to 10^4, the optimization magnitude of two applications' optimization schemes are almost zero. The performance optimization of three machine learning applications are similar to PI and Quicksort, see Fig. 14c. For the Thttpd, compared to the baseline scheme, the four metrics respectively increase or decrease 9.29%, 2.47%, 6.08% and 4.71% in switchless scheme, and increase or decrease 7.58%, 1.43%, 3.01% and 0.38% in hotspot scheme, see Fig. 13(a)-(d). Instrumentation almost does not affect the performance of the Thttpd after adopting all optimization. So, we set the random number as 10^3 in these applications' hotspot schemes as default. It guarantees both performance optimization and the accuracy(described in Section 7.6) of the resource measurement.

TABLE 5: Instruction Latency and Throughput.

Instruction	Latency	Throughput	Instruction	Latency	Throughput
MOV	1	1	CVTSI2SD	5	1
SUB	1	1	MOVSD	1	1
XOR	1	0.25	DIVSD	14	5
PXOR	1	0.33	MOVAPD	2	0.33
AND	1	0.25	ADDSD	4	0.5
CMP	1	0.25	MULSD	3	0.5
LEA	1	0.5	IMUL	4	1
SAR	1	0.5	TEST	1	0.25
CDQE	1	0.5	UCOMISD	2	1

7.6 Accuracy Evaluation

To accurately measure CPU cost, we need to measure the average cost of instructions. As measured in our experiments and described in Intel document [53], most instructions are executed in 1 cycle (80%), fewer instructions in less than 10 cycles (e.g. `cvtsi2sd` and `addsd`), and especially expensive (e.g. `divsd`) needing more than 10 cycles, as shown in Table 5. So, we simplify the average cost of all instructions to 1 cycle. Then, the CPU cost measured by T-Counter is the product of instructions' number (N) and the CPU frequency (F). Each counter itself needs ten instructions to execute (excluding security check of the enclave etc.). For every library function(include system calls it triggers), we generate code that executes the library function n times. The number of cycles per library function is calculated by dividing the total number of cycles by n (n = 100 in our experiment). These values are then used as weights for T-Counter's library function CPU resource accounting. We adopt Linux command **`htop`** to monitor the CPU time (T) of programs and take the CPU time as reference standards for our experiments. The monitored CPU time refers to the occupancy time of one core.

$$AE = N * weight/(T * F) \quad (1)$$

The accuracy error (AE) is calculated as formula 1. The accuracy evaluation results of six applications are as depicted in Table 6. These applications can be divided into two categories: computation-intensive and I/O-intensive.

TABLE 6: The Accuracy Evaluation of Six Applications.

Works	Work Type	CPUTime	App	Lib+Sys	Instruction	Accuracy(Error)
Wordcount	I/O-intensive	45.214s	0.406	44.808	$1.765*10^{11}$	95.211(-4.789)%
PI	CPU-intensive	4.343	4.257	0.086	$1.974*10^{10}$	110.86(+10.86)%
Quicksort	CPU-intensive	60.006	14.615	45.386	$2.723*10^{11}$	110.68(+10.68)%
NBayes	CPU-intensive	24.436	8.756	15.680	$9.548*10^{10}$	95.301(-4.699)%
HMM	CPU-intensive	10.418	4.856	5.362	$4.067*10^{10}$	95.215(-4.785)%
K-Means	CPU-intensive	35.909	13.565	22.344	$1.413*10^{11}$	95.959(-4.041)%

Wordcount is an I/O-intensive application, and its CPU cost mainly comes from printf operation, which includes library functions and system calls that handle io operations. As our experiment measurement, the total cost per I/O operation is(include library functions and system call) 7500 cycles, and the cost of library functions and system calls are accounted for 99.9%. The measurement accuracy of WordCount is 95.211%, and the accuracy error of WordCount is -4.789%. The measurement accuracy errors of other programs also are less than +10.86%.

Accuracy of Hotspot Optimization. As the Hotspot optimization strategy uses a random function to reduce ecalls, it will affect the accuracy of the measurement. So we evaluate the accuracy of the Hotspot optimization strategy with different parameters. Evaluation results are as depicted in Table 7. The result shows that the larger range of random number has a greater influence on the measurement accuracy, the accuracy of measurement of six applications is decreasing obviously with m from 10 to 10^4, and the accuracy decreasing about 4.279% at worst.

7.7 Binary Size and CPU core Overhead

Binary Size Overhead. The T-Counter does not change the computation logic of the programs of consumers. It just rewrites the application as an SGX application. The original application instrumented with counters is the untrusted part of the SGX application and runs outside the enclave, the added trusted part running inside the enclave to account and verify resource usage. The code added into applications is mainly the ecalls and hotspot randomized codes(exclude enclave edge code). The lines of codes added into applications are shown in Table 4. The instrumented binaries are between 6.4% and 31.25% larger after enabling all optimizations (hotspot).

TCB size. To minimize the TCB, we just put the trusted counter, verifier, recorder, and data structures they rely on in the enclave. Excluding the SGX trusted libraries, T-Counter adds less than 350 lines of C++ code in the enclave. Thus, the trusted part of the T-Counter code is easy for security verification.

CPU core Overhead. The switchless call feature of our switchless schemes needs an additional CPU core to process the ecalls. Although the switchless schemes increase the performance overhead, it is wise to add an additional CPU core for processing accounting compared to increasing performance overhead. This is more acceptable to consumers. And one ecall thread can be used to process several worker threads. When the CPU usage of the program is larger, an additional CPU core overhead of the trusted counter is smaller compared with the CPU cores used by the application itself. Intel research team has proposed a scheme to reduce additional CPU core overhead in their work [49].

TABLE 7: The Accuracy Evaluation of Hotspot.

Works	None	Hotpots/10	Hotpots/10^2	Hotpots/10^3	Hotpots/10^4
Wordcount	95.211%	94.984(-0.227)%	94.968(-0.243)%	94.531(-0.68)%	96.147(+0.936)%
PI	110.86%	110.86(–)%	110.86(–)%	110.809(-0.051)%	110.786(-0.07)%
Quicksort	110.68%	110.802(-0.122)%	110.802(-0.122)%	110.802(-0.122)%	110.924(-0.244)%
NBayes	95.301%	95.209(-0.092)%	95.437(+1.362)%	97.573(+2.272)%	98327.(+3.026)%
HMM	95.215%	95.129(-0.086)%	96.227(+1.012)%	98.017(+2.802)%	99.141(+3.926)%
K-Means	95.959%	95.857(-0.102)%	94.737(-1.222)%	99.525(+3.566)%	100.238(+4.279)%

8 Discussions and Future Works

Even though T-Counter provides a security guarantee and achieves efficiency and accurate advantages. The approach can be improved in the future, and we discuss it as follows.

Security Enhancements. The solution we propose in this paper have minimized the attack surface that the adversary can use and make the adversary's attack more expensive than his attack's profit. These efforts guarantee the effectiveness and trustworthiness of our measurement. But it is not perfect against all attacks, and the adversary can still do some damage to resource measurement. We need to resist more attacks and more efficient counter sequence verification methods.

Other resources measurements. In addition to CPU resources, many other important resources need to be measured, such as memory, network, disk, etc. Some works have also proposed schemes to measure the cost of these resources. E.g. S-FaaS [13] measure memory cost by instrumenting the `malloc`, `realloc`, and *free* functions used by the function or interpreter. AccTEE [25] uses the linear memory size for the accounting of memory consumed by the workload. But these schemes can only measure the memory cost inside the enclave. And, measuring I/O overhead also faces the problem. AccTEE uses an additional counter to accumulate how many bytes flow in and out of the enclave through I/O functions, but it cannot measure the operations outside the enclave trustworthy. The malicious OS can manipulate these operations arbitrarily. Achieving trust measurement of resource usage outside the enclave is a challenging problem. For measuring memory and network, we can also use the counter to calculate linear memory for accounting memory cost and calculate bytes flow for accounting I/O overhead. More importantly, we need to focus on solving security challenges in future works.

9 Conclusion

In this paper, we proposed a universal CPU usage measurement framework T-Counter, which allows for the construction of CPU instructions accounting solutions for applications with provable security in the cloud. As far as we know, T-Counter was the first framework and implemented to measure the CPU usage of a program running outside a trusted space in the cloud. We described the threat model of resource measurement and proposed the architecture of the T-Counter. We classified the basic blocks in the CFGs and proposed two counter instrumentation algorithms, named Base-Ins and Opti-Ins. These applications constructed by T-Counter are specified by three components, a trusted counter, a verifier, and a recorder. A concrete application built by these components can measure its CPU usage by itself in a trusted manner and make an adversary unable to tamper counters arbitrarily. Based on the measurement mechanisms, we introduced the importance of switchless. In further, we showed how to optimize instrumentation performance at the hotspot and model the relationship between performance and a random number. Finally, we performed some experiments to show the efficiency and accuracy of T-Counter, and verify the instrumentation algorithms' correctness.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant No.61672062 and 61232005.

REFERENCES

[1] "Amazon web services (aws): Cloud computing services," https://aws.amazon.com/cn/, 2020.

[2] "Google cloud: Cloud computing services," https://cloud.google.com/, 2020.

[3] "Microsoft azure: Cloud computing services," https://azure.microsoft.com/en-us/, 2020.

[4] S. Ibrahim, B. He, and H. Jin, "Towards pay-as-you-consume cloud computing," in *2011 IEEE International Conference on Services Computing*. IEEE, 2011, pp. 370–377.

[5] "Aws ec2 pricing," https://aws.amazon.com/cn/ec2/pricing/, 2020.

[6] "Google cloud pricing," https://cloud.google.com/pricing/, 2020.

[7] "Microsoft azure pricing," https://azure.microsoft.com/en-us/pricing/, 2020.

[8] R. Jellinek, Y. Zhai, T. Ristenpart, and M. Swift, "A day late and a dollar short: The case for research on cloud billing systems," in *6th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, 2014.

[9] K.-W. Park, J. Han, J. Chung, and K. H. Park, "Themis: A mutually verifiable billing system for the cloud computing environment," *IEEE Transactions on Services Computing*, vol. 6, no. 3, pp. 300–313, 2012.

[10] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V. Sekar, "Towards verifiable resource accounting for outsourced computation," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2013, pp. 167–178.

[11] S. Tople, S. Park, M. S. Kang, and P. Saxena, "V eri c ount: Verifiable resource accounting using hardware and software isolation," in *International Conference on Applied Cryptography and Network Security*. Springer, 2018, pp. 657–677.

[12] "Intel software guard extensions sdk for linux," https://01.org/sites/default/files/documentation/, 2020.

[13] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-faas: Trustworthy and accountable function-as-a-service using intel sgx," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019, pp. 185–199.

[14] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjá vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 7–18.

[15] H. Liang and M. Li, "Bring the missing jigsaw back: Trustedclock for sgx enclaves," in *Proceedings of the 11th European Workshop on Systems Security*, 2018, pp. 1–6.

[16] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 238–253.

[17] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "{SCONE}: Secure linux containers with intel {SGX}," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 689–703.

[18] H. Tian, Y. Zhang, C. Xing, and S. Yan, "Sgxkernel: A library operating system optimized for intel sgx," in *Proceedings of the Computing Frontiers Conference*, 2017, pp. 35–44.

[19] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 645–658.

[20] S. C. Misra and V. C. Bhavsar, "Relationships between selected software measures and latent bug-density: Guidelines for improving quality," in *International Conference on Computational Science and Its Applications*. Springer, 2003, pp. 724–732.

[21] V. Y. Shen, T.-J. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying error-prone software—an empirical study," *IEEE Transactions on Software Engineering*, no. 4, pp. 317–324, 1985.

[22] "Intel software guard extensions," https://software.intel.com/en-us/sgx/details, 2020.

[23] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[24] L. Chen and K. Chen, "Bitbill: Scalable, robust, verifiable peer-to-peer billing for cloud computing," in *Proceedings of the 6th USENIX conference on Hot Topics in Cloud Computing*, 2014, pp. 20–20.

[25] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "Acctee: A webassembly-based two-way sandbox for trusted resource accounting," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 123–135.

[26] V. Sekar and P. Maniatis, "Verifiable resource accounting for cloud computing services," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, 2011, pp. 21–26.

[27] F. Zhang, I. Eyal, R. Escriva, A. Juels, and R. Van Renesse, "{REM}: Resource-efficient mining for blockchains," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1427–1444.

[28] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions." *HASP@ ISCA*, vol. 11, no. 10.1145, pp. 2 487 726–2 488 370, 2013.

[29] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative instructions and software model for isolated execution in: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy, hasp'13, 10–1101.. acm, new york," *ACM, Tel-Aviv*, 2013.

[30] V. Costan, I. A. Lebedev, S. Devadas *et al.*, "Secure processors part ii: Intel sgx security analysis and mit sanctum architecture," *Foundations and Trends in Electronic Design Automation*, vol. 11, no. 3, pp. 249–361, 2017.

[31] "Intel sgx developer guide," https://download.01.org/intel-sgx/sgx-linux/2.7.1/docs/IntelSGXDeveloperGuide.pdf, 2020.

[32] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 81–93, 2017.

[33] "The llvm compiler infrastructure," http://llvm.org/, 2020.

[34] "Llvm language reference manual," http://llvm.org/docs/LangRef.html, 2020.

[35] "Writing an llvm pass," https://llvm.org/docs/WritingAnLLVMPass.html, 2020.

[36] "Clang: a c language family frontend for llvm," http://clang.llvm.org, 2020.

[37] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure:{SGX} cache attacks are practical," in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[38] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1041–1056.

[39] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting {SGX} enclaves from practical side-channel attacks," in *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 227–240.

[40] M. Liu and X. Ding, "On trustworthiness of cpu usage metering and accounting," in *2010 IEEE 30th International Conference on Distributed Computing Systems Workshops*. IEEE, 2010, pp. 82–91.

[41] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols," in *Secure information networks*. Springer, 1999, pp. 258–272.

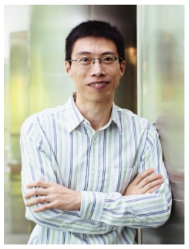
[42] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[43] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[44] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats." in *USENIX Security Symposium*, vol. 5, 2005.
[45] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011, pp. 30–40.
[46] M. Zhang and R. Sekar, "Control flow integrity for {COTS} binaries," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 337–352.
[47] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in {GCC} & {LLVM}," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 941–955.
[48] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.
[49] H. Tian, Q. Zhang, S. Yan, A. Rudnitsky, L. Shacham, R. Yariv, and N. Milshten, "Switchless calls made practical in intel sgx," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018, pp. 22–27.
[50] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 665–678.
[51] "Apache hadoop mapreduce." https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Source_Code, 2020.
[52] "Thttpd." https://acme.com/software/thttpd/, 2020.
[53] "Intel 64 and ia-32 architectures optimization reference." https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html, 2021.

Chuntao Dong received BSc. degree in software engineering from Shandong University, Jinan, China, and the Master degree in software engineering from Peking University, Beijing, China, in 2013 and 2016. He is currently pursuing the Ph.D. degree of Software Engineering in Peking University, Beijing, China. He ever worked as a research assistant with the School of Information Systems, Singapore Management University, Singapore, in 2017-2018. His research interests include system security, big data security, and trusted computing.

Qingni Shen received the Ph.D. degree from Institute of Software Chinese Academy of Sciences, Beijing, China, in 2006. She is currently a professor of School of Software and Microelectronics, Peking University, Beijing, China. Her research interests include OS and virtualization security, privacy preserving in cloud and big data, trusted computing. She is a senior member of CCF, IEEE and ACM member. She is now serving for many international journals and conferences, including Computers & Security, JPDC, TrustCom, SecureComm, ACNS and ever as the general chair of ICICS 2019 and as the steering committee memeber of ICICS since 2019.



Xuhua Ding received the Ph.D. degree from University of Southern California, America, in 2003. He is currently an associate professor of Information Systems in the School of Information Systems, Singapore Management University. His research interests include information systems and technology, cybersecurity, safety and security, security of digital platforms and devices.

Daoqing Yu received the B.Sc. degree in software engineering from Beijing Institute of Technology, Beijing, China, in 2018. He is currently pursuing the Master degree in Peking University, Beijing, China. His research interests include cloud security and system security.

Wu Luo received his BS degrees in telecommunications engineering from Beijing University of Posts and Telecommunication, Beijing, China. He is currently working towards his PhD degree at the School of Electronics Engineering and Computer Science, Peking University. His current research interests include Trusted Computing, Cloud Security and Operating Systems.

Pengfei Wu received BSc. degree in software engineering from Shandong University, Jinan, China, and the PhD degree in software engineering from Peking University, Beijing, China, in 2016 and 2020. Now, he is a research fellow in School of Computing, National University of Singapore, Singapore. He ever worked as a research assistant with the School of Information Systems, Singapore Management University, Singapore, in 2018-2019. His research interests include cloud security, big data security, and applied cryptography.

Zhonghai Wu received the Ph.D. degree from Zhejiang University, Hangzhou, China, in 1997. Previously he worked as a postdoctor and senior research fellow in Institute of Computer Science and Technology, Peking University, Beijing, China. Since then he has been involved both in research and application development of distributed system and multimedia applications. He is currently the executive president of School of Software and Microelectronics, Peking University, Beijing, China. His research interests include context-aware services, embedded software and system, cloud services and security, and open innovation.