GuaranTEE: Introducing Control-Flow Attestation for Trusted Execution Environments

Mathias Morbitzer  
Fraunhofer AISEC

Benedikt Kopf  
Fraunhofer AISEC

Philipp Zieris  
Fraunhofer AISEC

Abstract—Many cloud providers offer Trusted Execution Environments (TEEs) to protect critical data and processes from high privileged adversaries. Unfortunately, TEEs can only be attested at launch. To also enable attestation during run-time, we present GuaranTEE. GuaranTEE uses control-flow attestation to ensure the integrity of a service running within a TEE. To protect the attesting code from a potentially compromised service, we place it in a separate TEE. Additionally, the TEEs guard both the service and the attestation from malicious cloud providers. To reduce the overhead resulting from the use of two TEEs, we securely cache collected information and perform the attestation in parallel to executing the service. The detailed performance evaluation of our prototype based on Intel SGX in Microsoft Azure demonstrates that GuaranTEE provides a practical solution for cloud users focused on protecting the integrity of their data and processes at run-time.

Index Terms—Trusted Execution Environments, Control-Flow Attestation, Intel SGX

I. INTRODUCTION

To address concerns when relocating infrastructure into the cloud [1], the majority of cloud providers offer Trusted Execution Environments (TEEs) [2], [3], [4], [5], [6] that protect data and processes against malicious administrators. However, TEEs cannot detect the exploitation of vulnerabilities within their code. This can lead to control-flow attacks, which have already been used in the past to attack TEEs [7], [8], [9]. Control-Flow Attestation (CFA) detects such attacks by recording the control flow of a program and comparing it to a set of previously determined legal control flows. Yet, most previous work on CFA focuses on embedded systems and their specific challenges such as limited resources. In comparison, cloud environments present different challenges, such as a possibly malicious cloud provider.

With our work, we aim to close this gap of CFA designs lacking adaptation to cloud environments. To be precise, we present GuaranTEE, a solution that attests the control flow of a service running in the cloud, while shielding the service and the attestation from a malicious cloud provider. GuaranTEE adapts CFA to cloud-based services by introducing a control-flow verifier to the cloud. We place this verifier in a separate TEE to protect the attestation from a high privileged adversary in control of the cloud environment. With this design, GuaranTEE is capable of protecting the integrity of the service from malicious users and malicious cloud administrators alike. To demonstrate the practicability of GuaranTEE, we describe our prototype\(^1\)\(^2\), which we implemented using Intel Software Guard Extensions (SGX) and evaluated in Microsoft Azure. In summary, we make the following contributions:

- We present GuaranTEE, a design targeted for cloud environments which adds CFA to TEEs, allowing us to detect attacks that modify the TEE's control flow.
- We show the practicability of our design by presenting a prototype based on the widespread TEE Intel SGX.
- We describe our extensions to the LLVM compiler framework which allow for the easy deployment of Intel SGX enclaves protected with GuaranTEE.
- We provide a detailed performance evaluation of our prototype in Microsoft Azure using a signing service as example.

II. THREAT MODEL

We focus on cloud environments providing a virtual machine with full administrative privileges. This virtual machine runs the service we aim to protect, our target. As with most services, we have to assume that the target’s users are potentially malicious. By exploiting vulnerabilities in the target’s code, a user can gain full control over the target’s memory at a specific point in time. Being in control over the memory, the attacker can perform control-flow attacks [7], [8], [9]. As we are hosting the target in a cloud environment, we further consider the possibility of a malicious cloud provider controlling the software and hardware infrastructure. Having control over the infrastructure, the malicious cloud provider can extract and modify data and processes within our virtual machine. The attacker can either achieve this goal by using software-based [10] or hardware-based methods [11], [12], [13]. Yet, we do not consider advanced physical attacks such as tapping buses or dismantling the CPU. In other words, we assume the same threat model for a malicious cloud provider as TEEs such as Intel SGX [14].

III. DESIGN

GuaranTEE’s main goal is to detect control-flow attacks in untrusted cloud environments, posing several challenges to overcome. Here, we identify those challenges and present our respective solutions that form the design of GuaranTEE.

\(^1\)https://github.com/Fraunhofer-AISEC/guarantee-llvm/
\(^2\)https://github.com/Fraunhofer-AISEC/guarantee-code/
Before the VerifyTEE can detect control-flow attacks, we require an offline phase in which the analyzer learns the target’s Control-Flow Graph (CFG). Afterwards, in the online phase, users may send requests to the target (\textcircled{1}). While the target is processing a request, the trampoline records the control flow (\textcircled{9}). Next, the trampoline forwards the information to the analyzer in the VerifyTEE (\textcircled{8}). The forwarding prevents an attacker in control of the target or even the entire ProveTEE from tampering with the control-flow analysis. Having received the information from the trampoline, the analyzer compares the collected control flow against the target’s CFG and stores differences in the attestation log (\textcircled{9}). Finally, the service owner can retrieve the attestation log from the VerifyTEE (\textcircled{8}) to check the target’s control flows.

**B. Offline Phase**

In the offline phase, we instrument the target to regularly pass control to the trampoline. This enables the trampoline to record the target’s executed control flow (\textcircled{9}) and to provide this information to the analyzer (\textcircled{8}). As we perform the instrumentation automatically during compilation, we do not require any source code modifications. Instead, the developer must only annotate where the attestation of the target’s control flow starts and ends.

Once the target is fully instrumented, we calculate the target’s CFG by sending requests to the target until it has executed every control flow path at least once. To ensure that no malicious paths are followed, we perform this step in a controlled, offline environment without user interaction. To ensure all of the target’s legal control flows are collected, GuardTEE aims to protect targets with reduced complexity: so-called microservices, which are frequently deployed in cloud environments. Another difficulty for the creation of the CFG is Address Space Layout Randomization (ASLR) [23]. ASLR causes the target to use different virtual addresses at every execution, preventing us from using addresses to identify the endpoints of edges in the CFG. Instead, we identify the endpoints by assigning them unique IDs, leaving the identification unaffected by ever-changing virtual addresses.

**C. Online Phase**

During the online phase, the target hosted in the ProveTEE waits for requests from potentially malicious users (\textcircled{1}). While the target processes the request, the trampoline records all IDs along the executed CFG path (\textcircled{9}). Yet, the trampoline itself is not responsible for processing the IDs. This is due to the fact that it runs in the ProveTEE, next to the target. Hence, an attacker gaining control over the target could also tamper with the trampoline. To ensure that such an attack does not influence the CFA, the trampoline only forwards the collected IDs to the analyzer (\textcircled{8}). For the verification, the analyzer in the VerifyTEE uses the received IDs to reconstruct the target’s control flow. By then comparing the control flow against the target’s CFG created in the offline phase, the analyzer can detect control-flow attacks. After the control flow is verified, the analyzer stores detected violations in

---

**A. Overview**

In cloud environments, the target to be attested will likely be a service, such as a signing service for health certificates. These services are deployed within the European Union to sign the so-called Digital Green Certificates providing proof of vaccination against or recovery from COVID-19 or a negative test result [15]. To avoid forging of health certificates, the service’s private key must be equally protected from a malicious cloud provider and the service’s users. GuardTEE protects the target signing service against both types of adversaries: By using TEEs, GuardTEE prevents an administrator from accessing the service’s private key, and by deploying CFA, it detects users trying to compromise the control flow.

Specifically, GuardTEE introduces a control-flow verifier that attests the target’s control flow at run-time. This is in contrast to previous designs, in which the user attests the service’s remote execution [16], [17], [18], [19]. Instead, assigning the verification to the service owner has multiple advantages. First, it leaves the interface between the user and the service unchanged, making GuardTEE transparent to the user. Second, it enables us to perform CFA without requiring any information from the user, for example to ensure the freshness of the attestation [16], [17], [18], [19], [20], [21], [22]. Third, it allows us to execute the attestation time-independent from the communication between user and service.

Figure 1 depicts the generic design of GuardTEE. The dashed line indicates the virtual machine offered by the cloud provider. Within this virtual machine, we create two TEEs, the ProveTEE and the VerifyTEE. Both are owned by the service owner and mutually attest each other using the launch-time attestation mechanism provided by the TEE. Among the two TEEs, the ProveTEE is responsible for hosting the target. Furthermore, it also contains the trampoline which collects information about the target’s control flow before the flow gets executed. In comparison, the VerifyTEE hosts the analyzer, responsible for verifying the control flow collected by the trampoline. After verification, the analyzer stores control-flow violations in the *attestation log* also located in the VerifyTEE.

---

**Figure 1:** The generic design of GuardTEE. By using two TEEs, we prevent malicious cloud providers from extracting or modifying critical data and malicious users in control of the target from influencing CFA results.
the attestation log (3). Finally, the service owner fetches the attestation log from the VerifyTEE (3), which contains all requests that altered the target’s control flow.

Note that as the trampoline does not provide an interface to the outside world, the attacker will first have to compromise the instrumented target to be able to take control over the trampoline. This compromise of the target causes the trampoline to transfer IDs indicating the attack to the analyzer in the VerifyTEE. Hence, the analyzer can detect the breach before the attacker can influence the trampoline.

IV. IMPLEMENTATION

Based on our design, we implemented the GuaranTEE prototype using Intel SGX. The basic building blocks of our prototype are the same as in the design (Figure 1), except for an additional shared memory region we use to exchange data between the ProveTEE and the VerifyTEE. This region allows the TEEs to exchange information without requiring any context switches, thereby reducing performance overhead. To create the shared memory region, we use the fact that SGX TEEs share the virtual address space with their Host Application (HA) [24]. Specifically, as HAs can launch multiple TEEs [25], we use the HA to launch both the ProveTEE and the VerifyTEE, causing them to share the same virtual address space [26]. This shared address space enables us to allocate a memory region which both the ProveTEE and the VerifyTEE can access, allowing them to exchange collected IDs. Note that while both TEEs can read and write the shared memory, they cannot access each other’s memory.

In the implementation, each basic block in the target corresponds to a node in the CFG. The blocks are connected via edges, the endpoints of which are recorded by the trampoline. To be precise, the target calls the trampoline each time before entering or exiting a basic block, providing the ID of the respective endpoint. The trampoline stores this ID in an ID batch, which acts as a cache.

A. Caching IDs

Figure 2 depicts how we are able to securely cache the IDs by combining them with the calculation of a new key for every data exchange and tracking the number of exchanged ID batches. To ensure the integrity of the cached IDs, we use a hash chain. The root of this hash chain is a secret the service owner provides to both the ProveTEE and the VerifyTEE. When the trampoline caches an ID in the ID batch, we update the hash in the ProveTEE, \( \text{hash}_p \), with the new ID. If the ID batch is full, we combine both with the current value of \( \text{hash}_p \) to form a single message, which we then encrypt and store in the queue located in the shared memory region. After the VerifyTEE reads and decrypts the message from the queue, it adds all IDs to its own hash, \( \text{hash}_v \). As we initialize both \( \text{hash}_p \) and \( \text{hash}_v \) with the same secret and add the same IDs to both hashes, \( \text{hash}_p \) should equal the value of \( \text{hash}_v \) transmitted with the batch. Additionally, the VerifyTEE regularly acknowledges the receipt of the ID batches. On the other side, the ProveTEE, having stored a certain number of ID batches into the queue, will wait for an acknowledgment before continuing to execute the target.

As we exchange the ID batch via shared memory, it is also accessible to other, untrusted entities such as the HA or the operating system. This means that a malicious cloud provider could inspect and modify transferred ID batches. To protect the batches from such attacks, we ensure their confidentiality and integrity. For confidentiality, we encrypt all messages stored in the shared memory. In detail, we provide both TEEs with an initial key, \( K_{\text{Keyinit}} \), which they use as input to a Key Derivation Function (KDF) to calculate an encryption key. After encrypting the message, the key serves as input to another KDF to produce a new encryption key for the next message. This allows us to encrypt each message with a different key, not only ensuring the confidentiality of ID batches and their hash values, but also preventing an attacker in control of the ProveTEE from decrypting previous messages.

To ensure the integrity of messages, we encrypt them using AES-GCM [27]. AES-GCM requires an Initialization Vector.
(IV), which we need to synchronize between the trampoline and the analyzer to ensure correct encryption and decryption. We achieve this by managing an independent 64-bit counter, BatchNo, in both the ProveTEE and the VerifyTEE. After each encryption or decryption, we increase the respective counter to keep both counters synchronized.

B. Instrumentation

For the instrumentation of the target, we rely on the LLVM compiler framework in version 11. Our instrumentation records the target’s control flow before execution, allowing us to construct the full CFG. For this, we record the endpoints of the CFG’s edges: the exit from a basic block and the entry into another basic block, both of which we identify with a unique ID. We perform our instrumentation in two different phases of the LLVM compilation process: the IR optimization and the backend.

In our IR pass, we add calls to the trampoline at the beginning and the end of each basic block, and before and after direct function calls. In comparison, the backend pass adds calls to the trampoline before and after every indirect function call, before every indirect jump, and before every return instruction. Additionally, on indirect calls, indirect jumps, and returns we XOR the ID with the offset between the current instruction pointer and the jump destination to detect modifications of the jump address. This safeguard allows us to detect jumps from instrumented into uninstrumented code. Specifically, it causes a modification of the jump destination to also change the recorded ID, allowing us to detect the modification even if it points the execution to uninstrumented code. To be able to access the instruction pointer, we perform the instrumentation of indirect branches in the backend pass.

C. Execution

To minimize the impact of our control-flow attestation, we use the high processing power of cloud systems and provision our virtual machine with multiple vCPUs, thus supporting parallel processing of different tasks. To fully capitalize on this ability, we distribute the tasks between two threads. Within the prover thread, the target processes incoming requests (8) and executes the trampoline (9), which caches recorded IDs and stores full ID batches in the queue (9). In comparison, the verifer thread reads the ID batches from the queue and forwards the contained IDs to the analyzer (8), which verifiers the target’s control flow against the CFG. When the analyzer is finished, the verify thread stores detected control-flow violations in the attestation log (9).

To fetch the attestation log, the service owner confirms the integrity of the VerifyTEE using remote attestation [28], [29], [30]. The attestation process also establishes a TLS connection, thereby creating an encrypted channel directly into the VerifyTEE [31]. This channel permits the service owner to securely retrieve the attestation log.

V. EVALUATION

For our evaluation, we used the Intel SGX SDK version 2.15 to create a microservice for signing health certificates (Section III-A). Our signing service receives the SHA256 hash of a newly created health certificate, signs it with its private 2048-bit RSA key, and returns the signature. The service handles requests via a secure connection, which we establish based on code from the SGX-OpenSSL project [32]. In total, our signing service consists of 4,533 instructions plus libraries such as OpenSSL, and the trampoline of 47,542 instructions. To perform our evaluation, we instrumented the signing service with our LLVM passes and ran it as ProveTEE’s target in the Microsoft Azure cloud on a standard DC4s_v2 machine with an Intel Xeon E-2288G CPU, four vCPUs, and 16 GiB of memory. Within the virtual machine, we were running the default Ubuntu 18.04 provided by Azure with kernel version 5.4.0-1089-azure and the Linux SGX DCAP driver in version v1.41. Using this setup, we measured the processing times of the individual components (8 – 9) and the real-world performance overhead.

A. Component Performance Evaluation

Using our signing service, we started by evaluating the different components of GuaranTEE. To evaluate the throughput of the instrumented target (9), we simulated 100,000 client requests. While processing the requests, the target called the trampoline every 101.29 ns on average. We call this frequency the trampoline call frequency.

Next, we determined the throughput of the queue used to transfer IDs from the trampoline to the analyzer (9), which depends on two variables: the ID batch size and the feedback frequency. The feedback frequency defines the number of ID batches the trampoline sends to the analyzer before waiting for an acknowledgment. A frequency of 1 means that the trampoline waits for feedback from the analyzer after every single ID batch. In comparison, using a frequency of 1,000, the trampoline only waits for feedback after having transferred 1,000 ID batches.

Figure 3: The transfer time for a single ID from the trampoline to the analyzer (9) is mostly affected by the batch size.
Figure 3 gives an overview of the throughput with different variable combinations. While the X-axis depicts the different batch sizes, the Y-axis indicates the average transfer time per ID in nanoseconds. The four different plots indicate the impact of different feedback frequencies on the transfer time. They show that while a frequency of 1 does decrease the throughput, the differences between the other frequencies are only minor. In comparison, the ID batch size has a significantly higher impact on the throughput. Using a size of 1, we needed 2,098.11 ns on average to transfer a single ID with feedback frequency 1, and 665.17 ns with a frequency of 1,000. Yet, we can drastically reduce the transfer time by increasing the ID batch size. For example, a ID batch size of 10,000 and a feedback frequency of 10 reduces the transfer time to 37.76 ns.

For a better interpretation of these results, the dashed gray line in Figure 3 indicates the trampoline call frequency, which we previously determined to be 101.29 ns. All configurations achieving transfer times below this frequency transfer the IDs faster to the analyzer than the target creates new IDs. With our prototype, we stay above this frequency for ID batch sizes below 10. Yet, a size of 10 combined with a feedback frequency of 100 or 1,000 already transfers the IDs faster than the trampoline call frequency. Additionally, the transfer time stayed below the trampoline call frequency with an ID batch size of at least 50 and a feedback frequency of 10 or 1.

Next, we evaluated the analyzer’s throughput (Θ) by measuring the time required to process the IDs of 100,000 requests. For each request, the analyzer processed around 539 IDs, requiring 7.34 ns per ID on average. This is significantly lower than the target’s trampoline call frequency of 101.29 ns, allowing the analyzer to process incoming IDs faster than they are produced by the target.

### B. Signing Service Performance Evaluation

Next, we evaluated GuaranTEE’s overall overhead on our signing service by first measuring the time required by an uninstrumented service to process an incoming request. When performing 100,000 signing requests, the uninstrumented service required 54.79 µs per request on average. Afterwards, we measured the processing time of the instrumented service. Figure 4 depicts the time the prover thread requires to process a single request on the left and the time the verifier thread requires to validate the control flow on the right. Again, we performed the measurements for different parameter combinations. The X-axes show the different ID batch sizes, while the Y-axes indicate the average processing time per request. Furthermore, the graphs depict the different feedback frequencies.

Similar to the evaluation of the queue, the feedback frequency has only limited impact on the processing time in comparison to the ID batch size. Using a batch size of 1, the prover thread requires between 755.84 and 304.16 µs to process a single request, depending on the feedback frequency. Yet, the graph also shows that a higher ID batch size can drastically reduce this processing time. For example, our default configuration with a size of 10,000 and a feedback frequency of 10 reduced the processing time to 57.28 µs. This represents a 0.05x overhead in comparison to the original response time of 54.79 µs. In Table I, we show the overhead for further configurations. While we recorded a maximum overhead of 12.80x with both the ID batch size and feedback frequency set to 1, we were able to reduce the overhead to as little as 0.05x with different parameter combinations.

Compared to the prover thread, the verifier thread requires between 3,594.44 and 3,554.69 µs to process a single request with an ID batch size of 1. Similar to the prover thread, we can drastically reduce the processing time per request by using a higher ID batch size. Again, with our default configuration we are able to reduce the processing time to 28.46 µs.

### VI. Discussion

#### a) Missing Context-Sensitivity: Our current implementation verifies the edges of the control flow, but does not consider its context. Hence, an attack mixing two different but valid control flows would remain undetected. Future work is required to determine how context-sensitive CFA can be performed with an acceptable performance and memory overhead.

#### b) Manipulation of Cached IDs: We reduce the overhead of transferring IDs between the TEEs by caching collected IDs. To modify or delete these cached IDs, an attacker first needs to compromise the target to gain control over the ProveTEE. Yet, before the target executes a malicious control flow, the trampoline will create at least one ID identifying the compromise, and add it to the hash chain (Section IV). The attacker cannot remove this ID from the chain, as only the current value of hash<sub>p</sub> is accessible and the hash is not reversible. To nevertheless infer its previous values, a malicious cloud provider could attempt to collect previously exchanged ID batches containing earlier values of hash<sub>p</sub>. To protect against this threat, we encrypt the ID batches before sending and use an irreversible KDF to calculate a new encryption key for every exchange. Hence, the attacker will not be able to decrypt previously sent ID batches even when having access to the current encryption key. This ensures the integrity of IDs cached in the ID batch even in the presence of a malicious cloud provider who additionally gained control over the ProveTEE.

#### c) Malicious HA Interfering with Batch Transfer: The queue located in shared memory enables a malicious cloud provider to access batches stored in the queue. To prevent inspection or modification of the ID batches, we encrypt

| Feedback Frequency | Batch Size 1 | Batch Size 10 | Batch Size 100 | Batch Size 1,000 | Batch Size 10,000 |
|--------------------|-------------|--------------|---------------|-----------------|------------------|
| 1                  | 12.80x      | 1.42x        | 0.23x         | 0.11x           | 0.06x            |
| 10                 | 12.06x      | 1.20x        | 0.13x         | 0.06x           | 0.05x            |
| 100                | 11.34x      | 0.50x        | 0.07x         | 0.06x           | 0.05x            |
| 1,000              | 4.55x       | 0.21x        | 0.06x         | 0.06x           | 0.05x            |
| 10,000             |             |              |               |                 |                  |
them with AES-GCM (Section IV). Yet, an attacker may also attempt to remove an entire ID batch from the queue before it is read by the analyzer. We detect such attacks with the help of the counters in the ProveTEE and VerifyTEE, which are used as IV for the encryption and decryption of ID batches and are updated after every transferred ID batch. Therefore, if the VerifyTEE does not receive an ID batch sent by the ProveTEE, the counters run out of sync and the VerifyTEE detects that it attempted to decrypt the next batch with the wrong IV. Additionally, the feedback frequency allows the ProveTEE to detect a high privileged attacker pausing or stopping the VerifyTEE, as it would notice the VerifyTEE’s missing acknowledgement of previously sent ID batches.

   d) Launching Multiple TEE Instances: To be functional, both TEEs need to be provisioned with the secrets that serve as root for the hash chain ensuring the integrity of the cached IDs (Section IV-A). This enables us to detect a malicious cloud provider crashing the VerifyTEE and afterwards attempting to launch a fresh instance, as this instance would have to be provisioned by the service owner again. Similarly, an attacker could attempt to simultaneously launch multiple instances of the ProveTEE to forward batches containing IDs related to attacks to a different instance as the one accessed by the service owner. Yet, the requirement to provision each TEE with the secret for the hash chain also prevents this attack.

VII. RELATED WORK

The past has seen a number of CFA mechanisms. Yet, most of them either do not apply TEEs or only use them to protect the verification [16], [17], [18], [20], [21], [33], [22], [34]. The only other work that performs run-time attestation of TEEs is by Toffalini et al. [35], which was developed in parallel to our work. In comparison, the authors use a static approach and extract the target’s CFG by using a combination of symbolic execution and static analysis.

Control-Flow Integrity (CFI) mechanisms take a slightly different approach and verify edges before their execution [36], [37], [38]. This requires meta data about the legal edges, which must be stored securely out of an attacker’s reach. Existing CFI mechanisms therefore aim to hide this information, although with limited success [39], [40], [41]. Compared to CFI, we achieve a stronger isolation by storing the data for the verification in a different isolation domain, the VerifyTEE, preventing access by both malicious users and administrators.

VIII. CONCLUSION

In this work, we presented GuaranTEE, a design that allows to perform CFA in TEEs. GuaranTEE verifies the control flow of a target hosted in a TEE by making use of a second TEE. While using TEEs enables us to protect the target and the CFA mechanism from a malicious cloud provider, the separation into two TEEs allows us to protect the CFA mechanism from a potentially compromised target. We implemented our prototype using Intel SGX as TEE and evaluated it in Microsoft Azure. Our evaluation of a signing service showed that by combining a secure caching mechanism with multithreading, we can minimize the overhead in the target’s response time, enabling service owners to ensure the run-time integrity of their services even in cloud environments.

ACKNOWLEDGMENT

This work was partly funded by the Fraunhofer-Cluster of Excellence “Cognitive Internet Technologies” and by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) under Grant No. 13H40V010A. We would also like to thank Alina Weber-Hohengrund for integrating the mechanism for securely fetching the attestation log.
