Execution Integrity without Implicit Trust of System Software

Ahmed Awad and Brad Karp
University College London (UCL)

Abstract

When trusted application code in a TEE computes over results produced by an untrusted kernel and hypervisor [1, 2], it is difficult at best to reason about the secrecy and integrity properties achieved by the overall ensemble—to establish, despite the wide breadth of the Linux system call interface, that in-enclave code is immune to Iago attacks [3]. In this paper, we argue that an attractive use case for TEEs is *tamper-proof audit:* the TEE executes a trusted observer (TO) that allows efficient offline validation that application code running outside the TEE has executed as expected. We describe a TO design that inherently does not require any trust of system call results (and thus of the kernel or hypervisor), and DOG, a prototype TO implementation for Intel SGX that upholds application execution integrity, even for applications that do not fit within today's SGX virtual memory limits, and incurs modest execution overhead.

1 Introduction

The advent of Trusted Execution Environments (TEEs), such as Intel SGX, has spurred the exploration of how to deploy application code on a server in an untrusted environment (such as at a cloud provider). In these scenarios, typically the user deploying the application does not trust the operating system or hypervisor on the machine where the application code will run, yet wants secrecy and/or integrity guarantees for the application's execution. TEE hardware, typically embedded within the CPU, provides means for validating the integrity of application code at launch, and launching the application into a secure *enclave* virtual memory region whose contents are hardware-isolated from all other code executing on the same machine, including the OS and hypervisor.

Designs for TEE-secured application deployment to date have tended to place the application (in whole or in part) in an enclave [1, 2]. In this approach, the deploying user trusts all code within the enclave, and desires not to trust any of the OS or hypervisor code.[1] Two difficulties arise when applying TEEs in this way.

First, it is difficult to know in practice whether application code that consumes results from untrusted code (e.g., system call return values, as determined by the untrusted kernel and hypervisor) will produce correct results. Checkoway and Shacham catalog a broad range of *Iago attacks* that untrusted kernels can mount on trusted application code running above them; avenues of attack include system calls that manipulate virtual memory, conduct I/O, and provide access to hardware-generated entropy and time [3]. Some of the more tantalizing uses of TEEs specifically target placing *legacy* application code in enclaves [1, 2]. For these designs, it is difficult for the deploying user to know what assumptions the application's original developer made about the correctness of system call return values. Indeed, the original developer may not have been conscious of such assumptions. Descriptions of adaptations of legacy code to run within SGX enclaves typically include claims that the SGX deployment's designers thought hard about how the application code uses system calls, or even about the semantics of the system call APIs available to the application, and added trusted *shielding* code to validate OS system call return values [1, 2], but these assurances amount to asking the user to trust that this complex auditing and shielding activity has been done exhaustively and entirely accurately.

Second, any limits on the size of an enclave's virtual memory constrain application size. For example, current Intel SGX hardware limits the enclave page cache (EPC) to 93 application-accessible MB [4]. Even if future hardware loosens this constraint, access to integrity-protected RAM will remain fundamentally expensive because of the costs of updating Merkle trees [4].

In this paper, in light of the two above challenges, we reconsider how most effectively to use a TEE to uphold the integrity of execution of application code in an untrusted cloud environment. We observe that a "sweet spot" for the use of TEEs is the deployment of a *trusted observer* (TO), a compact software entity that permits *tamper-resistant audit* of application code that runs *outside* the TEE, where an adversary can tamper with the application's execution. This approach abandons secrecy for the deployed application, but provides robust integrity. In particular, it thwarts Iago attacks by design because it inherently does not trust the results of any system call.

SysTEX '19, October 27, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6888-9/19/10…$15.00
https://doi.org/10.1145/3342559.3365337

[1] Throughout this paper, we exclude denial-of-service attacks from consideration; TEEs do not prevent the OS or hypervisor from killing or refusing to schedule a task running within an enclave.

The high-level intuition underlying this property—and that makes a TO a good fit for deployment in a TEE—is that TO code executing in the TEE need only accurately record the application's inputs and outputs. As we discuss in Section 2, prior work shows how to efficiently audit a server's execution offline given such a trace [18]. A key technical challenge unaddressed by prior work is how to obtain this accurate trace in the cloud setting, where the deployer of the application does not trust the cloud provider's infrastructure. To do so, we interpose a TEE-protected TO between the client and application. The TO's system call invocations are almost exclusively for I/O. Our central insight is that because TLS integrity-protects client-TO communication, and the TO's TLS code executes within the TEE, TLS itself will flag any Iago attacks that tamper with the TO's payload-transferring network I/O system calls, and hence the TO need not trust the results of these system calls. Moreover, on today's Intel SGX platform, placing the application entirely outside the TEE avoids imposing today's highly constrained 93 MB EPC limit on application code and data.

In what follows, we describe the design of a TO for TEEs that eliminates trust in the system software and allows tamper-resistant audit of an application's execution on an untrusted server; describe DOG, a TO implementation for Intel SGX; and provide experimental evidence that DOG incurs acceptable performance overhead on current SGX hardware.

2 Problem and Design

The systems research community has studied how to verify outsourced (untrusted) executions given knowledge of the inputs and outputs to/from those executions [11, 18, 20]. At a very high level, these designs give a *verifier* a *trace* of the outsourced server's ground-truth inputs and outputs, and a set of untrusted *reports* produced by the server, e.g., that describe non-deterministic events during execution, such as the interleaving of concurrent threads. The verifier uses these data to audit whether the ground-truth trace corresponds to a faithful execution of the outsourced server.

We leverage recent work on Orochi [18], which relies on record-replay [7–9] of the untrusted server application (SA). A central challenge in making this approach work in practice is obtaining the ground-truth input/output trace for the verifier's use. While Orochi offers various options for producing this trace (e.g., packet capture or a separate proxy server at an edge network's egress link, in settings where all clients are on a single, trusted edge network), none of these proposals fits the cloud scenario, where an untrusted server in a data center serves clients at diverse Internet attachment points, such that the only "choke point" for traffic to/from the SA is in that same data center. Naively placing an Orochi trusted proxy in the same data center avoids backhauling a copy of all the SA's communication across the wide area, but that arrangement would mean co-locating two servers and trusting one (running an Orochi proxy) but not the other (running the SA), when both servers process the same potentially malicious input from the wide-area Internet. In the remainder of this paper, we show how to solve this ground-truth trace production problem in the cloud setting.[2]

We introduce a *trusted observer* (TO) that runs in a Trusted Execution Environment (TEE) within the untrusted server. The TO ensures the integrity of the request/response trace generated for a server application (SA), as Orochi's verifier requires a ground-truth trace. Let us first define trace integrity. A trace consists of a series of entries, each containing either a client's request or a server's response, and numbered with a monotonically increasing index. At audit time, we consider a trace's integrity preserved if the following properties hold:

1. The TO has seen all requests received by the SA and all responses received by clients up to the moment when the trace is retrieved for auditing.
2. A client's request cannot be modified in transit before the TO records it.
3. Similarly, once the TO has seen a response to a given client, the response cannot be modified before it reaches the client.
4. When a trace is retrieved for auditing, all entries with indices in the range $(i, k]$ are present in the trace, where i is the index of the last entry previously audited and k is the greatest index generated by the TO.
5. Any modification of the entries in the persisted trace will be detected at audit time.

A separate offline verifier that runs on hardware trusted by the SA's deployer uses the trace and knowledge about the SA to carry out audit.

2.1 Threat Model

We assume an adversary capable of compromising privileged software (i.e, the OS and hypervisor) executing on the untrusted machine, as well as any hardware in the machine other than the CPU. We further assume that she can present arbitrary network packets to the system and can control sources of randomness other than the `RDRAND` instruction on modern Intel SGX CPUs. Finally, we trust the development environment, compilation toolchain, Intel attestation infrastructure, and SGX libraries.

Under this threat model, an adversary can mount *Iago attacks* [3]. Iago attacks exploit an application's trust in the OS's interface to subvert the application's execution. For example, older versions of Apache used the values returned from `getpid` and `time` to seed SSL/TLS's entropy pool. A malicious actor who compromised the OS could thus replay

[2] We inherit two of Orochi's scope limitations: Orochi only applies to SAs that do not retrieve values from other servers as part of computing a response to a request, and it may not always be able to detect tampering with an SA whose code invokes a function that returns a non-deterministic value. We elide much of Orochi's design, particularly of its verifier, as it is unchanged in our context; we focus only on ground-truth trace generation in the cloud, and refer the reader to Tan et al. [18] for more on Orochi.

a previous SSL/TLS session by returning the same pid and value to Apache as those used for that session [3].

Prior SGX software frameworks for securing legacy applications [1, 2] do not fully address the legacy application's trust of the operating system.[3] A developer must take on the onerous task of analyzing each system call invocation in an application to ensure that maliciously constructed OS results do not compromise application execution integrity.

The TO, on the other hand, inherently uses only a small subset of the system call interface. And while it relies on the OS for communication between itself and clients/servers, it uses authenticated, integrity-protected channels to prevent a compromised OS from tampering with any of this communication without detection. An adversary may still stop the TO from communicating with clients or the SA, but that does not compromise the integrity of communication.

We rely on Intel's remote attestation mechanism to verify that the microcode of the CPU in the untrusted machine implements mitigations for Spectre variants 2 and 4,[4] Microarchitectural Data Sampling (MDS), and L1 terminal fault (L1TF). During the remote attestation procedure, we also confirm that hyperthreading (HT) is disabled, for complete mitigation of MDS and to prevent L1TF attacks from leaking enclave secrets through the L1 cache [12].

The remaining well-known side-channel attacks against SGX are cache-timing and paging-based attacks. We disable HT so that an adversary cannot carry out cache-timing attacks that target the L1 or L2 cache. Similarly, paging-based attacks that rely on flushing the TLB require HT to work [21]. We do not address the remaining SGX side-channel attacks that exfiltrate enclave state by frequently interrupting an enclave's execution. Current software mitigations for these attacks impose a non-trivial performance penalty [10, 16]. Our design, however, is not tied to SGX and can be used on other TEE platforms. Sanctum and Keystone [5, 13], for example, mitigate these side-channel attacks in hardware.

Finally, like prior TEE-based designs, our design does not prevent denial of service (DoS, e.g., descheduling the application). While DoS of the TO can cause DoS of the SA, it cannot prevent detection of tampering with the results of the SA's execution.

2.2 System Architecture

Figure 1 depicts the use of a TO to enable audit of execution integrity of an untrusted SA. The top-level entities are:

- the untrusted SA;
- the TO, running inside a TEE, which records a trace of all communication between clients and the SA to untrusted local storage, and persists cryptographic hashes of the trace to a trusted store;

[3]For example, a developer attempting to secure the aforementioned version of Apache would have to avoid using `getpid` as a seed for the PRNG.

[4]We compile the TO with retpolines to avoid using performance-degrading indirect branch restricted speculation (IBRS).

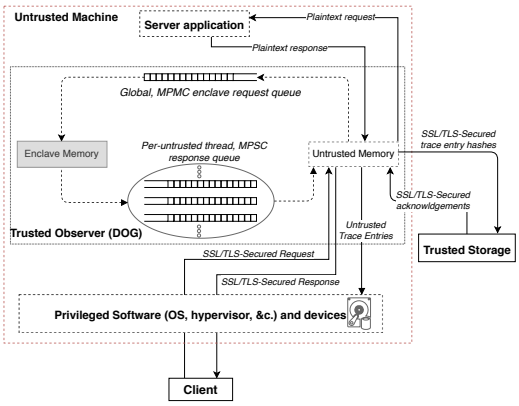

Figure 1. System architecture. Dashed boxes indicate untrusted entities; the dotted box is the TO (DOG); solid boxes are trusted entities. Dashed arrows show flow of control information; solid arrows show data flow.

- a *trusted store* that receives trace hashes from the TO over TLS, and persists them for retrieval at audit time;
- clients of the SA, proxied via the TO over TLS connections terminated within the TO; and
- a verifier (not shown in Figure 1), which takes the trace and hashes the TO generates, as well as the untrusted reports recorded directly by the TO, and determines whether the trace corresponds to a faithful execution of the SA.

The numbered trace integrity requirements at the start of Section 2 match those the original Orochi verifier needs from the original Orochi trusted proxy [18], and dictate the TO's design, with the exception of property (5), which is new to our cloud context because the TO writes its trace on untrusted storage. To satisfy property (1), we instantiate the TO as a reverse proxy within a TEE on the same machine as the SA, through which all client-SA communication passes. Interposing the TO between clients and the SA increases latency. We measure this cost in Section 4.

Maintaining properties (2) and (3) requires authenticated, integrity-protected communication between clients and the TO, so that the TO can detect any malicious modification of an inbound client request, and the client can detect any modification of a server response between the TO and the client. Clients communicate with the TO using TLS. The TO must therefore prevent an adversary from obtaining the TO's long-term TLS private key or ephemeral session keys, so that client-TO channels are not compromised. Executing the TO in a TEE protects this sensitive TLS state.

A TLS session key [15] is derived from random values generated during the TLS handshake. The TO within the TEE must derive these random values without invoking untrusted sources of randomness such as Linux's `/dev/urandom`. Instead the TO's TLS implementation uses the user-level `RDRAND` instruction, which generates cryptographically secure pseudo-random numbers without kernel involvement.

Properties (4) and (5) require trusted persistent storage. Local storage on the machine where the SA runs offers high

bandwidth and low latency, but is untrusted. The TO stores the full trace on untrusted local storage. To allow detection of modified trace entries at audit, the TO appends a cryptographic hash over each trace entry to a network-attached trusted store within the data center.[5] To ensure property (4), trace freshness, the TO also maintains a monotonically increasing counter for the current index within the trace, which it persists to the same remote trusted store upon every trace append. Using this fresh counter value from the trusted store, the verifier can ensure that the trace retrieved from untrusted local storage has the correct number of entries, and has not been reverted to a stale, prefix version.[6] To ensure that at audit the verifier always has a fresh value of this counter, upon receiving a new request or response to record, the TO always persists the incremented counter value and hash of the request or response to the trusted store (i.e., awaits a successful response from the trusted store) *before* writing the corresponding trace entry to local storage, only after which it releases the corresponding request to the SA or response to the client. We measure the performance implications of this synchronous use of the trusted store in Section 4. The TO and trusted store communicate using TLS.[7]

Finally, recall that the verifier needs untrusted reports (concurrent event information) to correctly replay concurrency within the SA. This is visible only within the SA itself, which directly records it to untrusted local disk. We refer the interested reader to Tan et al. [18] for the details of how the verifier uses these reports.

3 Implementation

Figure 1 includes details of the Delegated Observer Gateway (DOG[8]), an implementation of the TO for Intel SGX hardware. DOG consists of 6159 lines of untrusted (extra-enclave) and 3589 lines of trusted (in-enclave) C++, excluding library code. The enclave further includes code for the referenced portions of the SGX libstdc++ and OpenSSL libraries, and a lock-free queue data structure (described below). DOG uses a 256-bit Blake2b cryptographic hash.

DOG's TLS stack resides within an SGX enclave. As SGX prohibits enclave programs from making system calls, DOG's enclave code delegates calls to `read, write, bind,` etc. to an extra-enclave untrusted I/O module. Because the OS is untrusted (as is DOG's user-level I/O module itself), the execution of system calls may be malicious—i.e., the system call may take arbitrary action deviating from the requested operation, and the system call's return value may be arbitrary.

While DOG is not intended to prevent DoS, it must uphold trace integrity. Because DOG invokes only a narrow set of system calls for disk and network I/O, and uses TLS for all communication with clients and the trusted store, *no behavior by any of these few system calls can violate trace integrity without detection.* Any tampering with data sent between a client and DOG or between DOG and the trusted store will be detected by the recipient because these communications are TLS-protected. An adversary may tamper with communication between DOG and the SA, which is not over TLS, but such tampering is equivalent to tampering with the SA itself, which is untrusted, and will be detected at audit time if the tampering causes divergence between the responses in the trace logged by DOG and the responses generated during re-execution for audit. An adversary may tamper with trace data written by DOG to local storage, but such tampering will be detected at audit time by validating the trace from local storage with the hashes stored by the trusted store. As discussed in Section 2.2, the inclusion of a monotonically increasing counter in the data stored by the trusted store allows validation of the trace's freshness at audit time. In summary, all I/O conducted by DOG either is integrity-protected by TLS executing within the TEE, at the client, or at the trusted store; or is untrusted (and validated at audit time).

When the SA's deployer launches DOG she uses SGX's remote attestation procedure to provision DOG with a TLS private key. DOG then uses SGX's sealing primitive to persist the key to untrusted disk in encrypted, integrity-protected form.[9] The enclave code containing the private key must expose an interface that allows DOG to:

- complete TLS handshakes with clients and the trusted store,
- encrypt and authenticate payloads destined for clients and the trusted store,
- and authenticate and decrypt clients' payloads destined for the SA.

DOG incorporates several performance optimizations. To avoid the overhead of `EENTER` and `EEXIT` transitions, DOG allocates threads that enter the enclave and remain there permanently, and a separate pool of untrusted I/O threads that never enter the enclave. It uses asynchronous, shared-memory communication between enclave and non-enclave threads,

[5] One may build a network-attached trusted store in which one trusts only a small piece of hardware in an otherwise untrusted server [22]; see Section 5.

[6] Trace rollback can thwart detection of unfaithful SA execution at audit time: an adversary can simply provide an earlier trace that ends before unfaithful execution occurred. While Intel SGX supports *sealing* of persistent data to disk, naive sealing is subject to rollback attacks, and Intel's support for on-CPU persistent monotonic counters cannot keep pace with realistic storage write rates [14]. TPM chips also provide persistent monotonic counters, but suffer from similar update-rate limits [14]. Matetic et al. explore replicating a monotonic counter across distributed enclaves [14], but in our problem setting, a malicious cloud provider could reset the counter by resetting all counter replica enclave programs.

[7] The TO's binary includes the TLS public key of the trusted store, used to authenticate the trusted store. If the key in the binary is modified, the CPU will not launch the enclave.

[8] So named as DOG itself executes faithfully.

[9] SGX's sealing mechanism allows developers to void previously sealed data by updating the application's enclave version number. This requires recompilation, but it is suitable for ensuring freshness of an infrequently updated secret, such as a TLS private key.

after FlexSC's approach to eliminating system call trap overhead [17]. DOG also batches requests and responses between enclave and non-enclave threads, to reduce context-switch overhead. Requests from untrusted I/O threads to enclave threads pass through a lock-free multi-producer, multi-consumer queue data structure [6]; responses pass through a (per-untrusted-thread) lock-free multi-producer, single-consumer (MPSC) queue [6], both modified to avoid in-enclave memory allocation (to comply with current SGX toolchain restrictions).

4 Evaluation

To assess and understand DOG's costs, we consider the following questions:

- Does DOG achieve trusted logging of dynamic-content servers at acceptable end-to-end throughput and latency?
- To what extent does SGX limit DOG's performance (vs. the inherent cost of logging itself)?

4.1 Experimental Setup

We ran DOG and the SA on a Dell XPS 8930 with a 6-core, SGX-enabled Intel i7 8700 CPU clocked at 3.20 GHz, 16 GB of DDR4 RAM, and a 256 GB SSD.[10] Our load generator ran on a Dell C640 with 2 12-core Intel Xeon 5118 CPUs clocked at 2.3 GHz, 64 GB of DDR4 RAM, and a 1 TB SSD. The trusted store ran on an identically provisioned Dell C640. All machines were connected at 10 Gbps. Unless otherwise stated, communication between the load generator and DOG used the ECDHE-AES-128-GCM-SHA-256 TLS 1.2 ciphersuite.

4.2 DOG's Performance for Dynamic Content

For these experiments we used Mediawiki as the SA. Mediawiki is an open source PHP application that allows users to jointly edit pages containing information about a given topic. We used publicly available Wikipedia access traces [19] to generate a constant-rate HTTP request load to measure the end-to-end throughput and response latency of:

- unmodified Mediawiki (MW) executing alone,
- Mediawiki modified by Tan et al. (MW-L) to capture enough information for auditing with Orochi [18],
- and DOG proxying Mediawiki-L (DOG +MW-L).

Note that unlike DOG, MW and MW-L omit TLS and instead communicate in cleartext with clients.

We expect DOG to modestly reduce MW-L's performance, as the logging operations DOG performs are cheap relative to the operations required to generate dynamic content (fetching content from the database, rendering the page based on a template, etc.). Indeed, as Table 1 illustrates, DOG imposes an overhead of 6.4% on MW-L's throughput, and increases 50^{th}, 90^{th}, and 99^{th} percentile response latency by ~5%, ~2%, and ~4% respectively. We conclude that DOG's overhead is acceptable for dynamic content.

[10] In 2018, among available machines with an SGX-compatible BIOS, this was the one with the most cores.

Configuration	Throughput (MB/s)	Latency (ms)		
		50%	90%	99%
MW (HTTP)	4.1	187.2	280.0	290.5
MW-L (HTTP)	3.6	202.6	356.4	370.2
DOG + MW-L	3.4	212.2	370.1	386.4

Table 1. Latency comparison for varied MW configurations.

4.3 The Price of Logging and SGX

To expose the costs of logging and SGX, we now consider nginx serving static content as the SA. While not the target use case for DOG, this workload reveals DOG's worst-case performance penalty, when the DOG proxy's operations are more expensive than a minimal SA's. Unless otherwise specified, hyperthreading is disabled in all experiments. We measure end-to-end throughput of:

- nginx executing alone with hyperthreading enabled (HT) and disabled,
- DOG proxying nginx (DOG) with HT enabled (HT) and disabled,
- a variant of DOG that does not use SGX (DOG No SGX),

We expect a greater relative performance penalty for DOG proxying a static-content SA vs. a dynamic-content SA. A static-content server takes approximately 0.7 μs to decrypt a 100-byte request, a negligible amount of time to process it (with the requested page in the server's cache), and 0.3 μs to encrypt the response, for a total of 1 μs. DOG adds 1.6 μs to compute the hashes for both the request and the response, and at least 1.2 μs for each enclave crossing (one for the request and one for the response). The resulting total is 3.8 μs, almost a 300% increase in time to return a response (ignoring additional time spent in the kernel to send the request from DOG to the SA, and to send the response from the SA to DOG). This estimate doesn't account for cross-request concurrency, but does show that for an extreme, minimal SA, the worst-case added compute would be significant.

Let us now consider DOG's effect on response latency. For an offered load of 200K requests per second, DOG increases 50^{th} percentile latency by ~60% (from 1.0 μs to 1.6 μs), 90^{th} percentile by ~40% (from 1.6 μs to 2.3 μs), and 99^{th} percentile by 25% (from 2.3 μs to 2.9 μs). As DOG won't forward a given response/request to its destination until it receives an acknowledgement that the corresponding hash is persisted within the trusted store, it adds at least 1 RTT of delay to response latency.

Figure 2 shows average throughput vs. offered load where all requests are for a 100-byte HTML page. Comparing nginx and DOG reveals that DOG imposes a ~32% throughput penalty over baseline nginx. This penalty decreases as response size increases, and the rate of request processing becomes limited by available bandwidth.

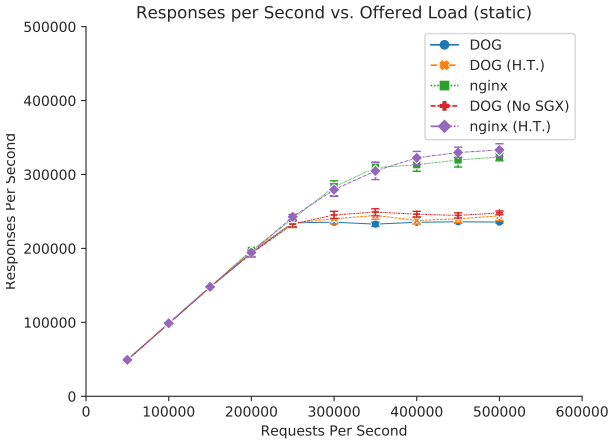

Figure 2. Responses per second vs. offered load for various nginx configurations. Responses are ~500 bytes (400 bytes of HTTP header and 100 of HTML). Comparing DOG (HT) vs. DOG and nginx vs. nginx (HT), we see that disabling HT costs approximately 3%. As logical cores share functional units, for this compute-intensive workload (mostly AES encryption), we don't expect HT to significantly improve performance.

Comparing DOG and DOG (No SGX) reveals that SGX imposes only a 4% performance penalty on DOG's throughput. Since most of the data DOG operates on is stored in non-enclave memory, most of this overhead comes from context switches (rather than EPC integrity overheads). Beyond the optimizations in Section 3, we also pin DOG's threads and nginx's worker processes to distinct CPU cores to reduce the number of context switches.

5 Discussion

We have explored offline audit of the execution integrity of a server application as an attractive "sweet spot" for the use of TEEs. Prior TEE uses that place an application (or part of one) within a TEE in an effort to provide both execution integrity and secrecy are liable to Iago attacks because application code may subtly rely on the results of diverse system calls. Shielding TEE code against system call return values is difficult to get right, given the complexity of application code and the system call interface. By contrast, the TO approach exemplified by DOG targets only integrity, but sidesteps the thorny problem of Iago attacks by narrowly restricting system calls to those whose results are untrusted (and validated at audit time) and those whose results are validated by a TLS implementation executing within a TEE. It also entirely avoids needing to shoehorn an application into a tightly address-space-constrained TEE, in today's SGX implementation. The TO leverages Orochi's approach to verifying an application's execution integrity [18], while extending Orochi to obtain the necessary ground-truth trace in the cloud setting.

While DOG relies on a trusted store for hashes of trace entries, there is evidence such a store can be built without trusting much hardware. Yang et al. describe a secure network-attached storage service that leverages a small piece of trusted hardware in an otherwise untrusted server to provide integrity- and freshness-protected storage [22]. Such a service could serve as a drop-in replacement for the trusted store in our design. Another alternative would be to have the TO write the full trace to *trusted* local storage provided by Yang et al.'s hardware on the same machine where the TO and server application execute.

Acknowledgments

We thank Michael Walfish for extensive discussions that significantly improved the framing and presentation of this work, and the anonymous reviewers for their helpful comments.

References

[1] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, Daniel, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI 2016*.

[2] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI 2014*.

[3] S. Checkoway and H. Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *ASPLOS 2013*.

[4] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016, 2016.

[5] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security 2016*.

[6] C. Desrochers. Concurrent queue. https://github.com/cameron314/concurrentqueue, 2018.

[7] C. Dionne, M. Feeley, and J. Desbien. A taxonomy of distributed debuggers based on execution replay. In *PDPTA 1996*.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI 2002*.

[9] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE 2008*.

[10] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security 2017*.

[11] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *OSDI 2010*.

[12] Intel. Software guidance for security advisories. https://software.intel.com/security-software-guidance/software-guidance, 2019.

[13] D. Lee, D. Kohlbrenner, K. Cheang, C. Rasmussen, K. Laeufer, I. Fang, A. Khosla, C.-C. Tsai, S. Seshia, D. Song, and K. Asanovic. Keystone enclave: An open-source secure enclave for RISC-V. https://keystone-enclave.org/files/keystone-risc-v-summit.pdf, 2018.

[14] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback protection for trusted execution. In *USENIX Security 2017*.

[15] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[16] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS 2017*.

[17] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *OSDI 2010*.

[18] C. Tan, L. Yu, J. B. Leners, and M. Walfish. The efficient server audit problem, deduplicated re-execution, and the web. In *SOSP 2017*.

[19] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[20] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution.

[21] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS 2017*.

[22] H.-J. Yang, V. Costan, N. Zeldovich, and S. Devadas. Authenticated storage using small trusted hardware. In *CCSW 2013*.