Health Database Confidential Compute

Matheu Campbell, Denzel Farmer, Isaac Trost

May 14, 2025

Abstract :

Modern cloud computing services promise clients access to large-scale compute and storage, shielding them from the complexities of distributed systems. As cloud infrastructure continues to scale, attracting client's with larger and more sensitive workloads, chip designers, architects, and kernel developers are increasingly incentivized to develop cloud systems which are secure not only from external attackers, but even from low-level internal software. This motivated the development of trusted execution environments (TEE) with a focus on isolating a virtual machine and its guest OS from its host OS, other virtual machines, and the whole system's hypervisor. Intel's Trusted Domain Execution (TDX) is one such implementation, which guarantees VM isolation even in the case of a compromised hypervisor. This project examines the architecture of a TDX-protected database application hosted on Google's Cloud Compute platform, comparing its performance to a baseline unprotected version.

1 Introduction

Hardware-based isolation mechanisms are deeply integrated into an overall system's design, implementing protections at the levels of registers, caches, and instruction pipelines. This method makes hardware protections difficult to subvert, as fault injection would require overriding physically etched structures. These isolation mechanisms vary in scope and architectural basis, but this project will focus on intel's Trusted Domain Execution (TDX), which aims to secure entire virtual machines hosted in cloud environments using memory encryption, page table extension. TDX-enabled execution environments can launch hardened VMs called Trusted Domains (TDs). Additionally, the Trusted Domain module enforces secure boot and exit protocols to maintain confidentiality even as the guest OS and VM undergo state changes. A primary feature of the TDX framework is the ability to generate a TDX quote, a cryptographic certificate that verifies software in a particular environment is indeed running within a trusted domain. To verify the authenticity of a given quote, it must be submitted to an attestation service, either hosted locally or through Intel's official servers, which can guarantee the authenticity of a provided quote.

To study the effects of the increased overhead of TDX on program execution, we will implement a secure SQLite database which stores medical records and hosts HTTPS endpoints for reading and writing. We will run the database from a software container running within a protected machine and compare its performance on several metrics of throughput and latency to those of the same container running on an unprotected virtual machine.

2 Background

2.1 TDX Protections

Intel's TDX's hardware extensions enforce isolation and memory integrity at several levels. The Intel TDX-module is a software module responsible for managing trusted domains. It is run by the CPU in a secure mode as a peer VMM, and maintains integrity during VM entry and exit.

At the memory level, TDX protects by encrypting each cache line as well as main memory itself. TDs maintain distinct TD memory and shared memory systems, and private memory is encrypted using a ephemeral keys assigned on a per-TD basis. The CPU prevents the TD from executing code found in shared memory by inducing page faults on attempts to access code or data stored in shared memory. The Intel-TDX-module maintains a record of physical address metadata to prevent remapping of pages into those of other secure TDX modules. Finally, the TLB associates each translation with its source TD to maintain address integrity.

For most of the security features offered, there is an expected cost in overhead incurred due to the need to decrypt all cache and pointer data, perform extra integrity checks during page walks, and maintain strict isolation, even disallowing certain types of speculative execution.

2.2 Quote Generation

A key feature of TDX is its quote generation and remote attestation capability, which offers a relying party confidence that software is running on a genuine Intel TDX system. Full verification requires quote generation and quote verification phases.

Upon request, a TD will generate a report, through a series of requests to subsystems. This process involves the TDX-module, a software module responsible for managing trusted domains, and the TD-quoting enclave, an architectural enclave responsible for performing signing. The resulting report from an attestation request may contain:

1. Relevant security data measured directly by the TDX-module.
2. TD measurements provided by the Intel-TDX module.
3. Other details about unmeasured state provided by the Intel-TDX module.
4. Security version numbers of all elements in the TDX-TCB provided by the CPU hardware directly.

In general, the quote generation flow proceeds as follows:

1. A relying party makes a request for a quote (the signed result of remote attestation).
2. The TD requests the TDX-module for a report.
3. The TDX-module requests the CPU for a report structure, which is generated by a special instruction that generates a cryptographically hardware-bound report. This report includes the requested information.
4. The TD receives the report, and hands it off to the VMM, which passes it to the TD-quoting enclave for signing us-

ing an ECDBSA-based attestation key which represents the Intel-TDX-TCB version.
5. The signed report, or quote, is passed back to the relying party.

Once a quote is generated, the relying party can verify its authenticity using a verification service.

2.3 Quote Verification

Intel provides infrastructure to certify TD-quoting enclaves with a chain backed ultimately by Intel. The basis of this chain is the Provisioning-Certification Enclave, which authenticates requests from a TD-quoting enclave and issue a corresponding certificate that identifies the quoting enclave and the attestation key. This certificate is further signed by a device/TCB-specific key which is published by Intel. This allows for 3rd-party verification and custom verification software.

3 Architecture

3.1 Secure Database Architecture

3.2 FitHealth Database

Our TDX-enabled FitHealth database was deployed on a Google general purpose c3-standard-4 instance (4 vCPUs, 16GB memory) with TDX support with an Ubuntu 24.04 image. The standard database was deployed on the same type of machine, but not in the TDX TrustZone.

The app itself used node.js, with a SQLCipher database attached. This database is built on SQLite, and transparently encrypts all of your data with AES, using a given key.

It has a single endpoint: `/record`. You can GET from that endpoint, for example with `/record/user_id1`, and it will send back the full entry for user 1 in JSON format. If you post to that address, it will parse the body of your request as JSON and add it to the database.

We use nginx as a proxy to handle TLS encryption and further protect the backend.

On the start of the secure container, the secure box will attempt to generate a TDX report. Under normal conditions, this will fail if it is not a valid TDX trust environment. The Intel documentation for verifying a TDX quote from within a docker container is spotty at best, so we decided the best route was to verify the quote in the outer VM, and have it refuse to launch the docker container if the quote does not pass. This does invalidate the startup time metric for the containers, and the container itself does not generate or check a quote. On the other hand, the security assurances are the same, as the outer container also has access to the docker image, and thus the encrypted database, as well as Google Secrets Manager to decrypt said database.

When the VM sees that it is in a valid TDX enclave, it will start the docker container, and the docker container can pull down the db key and start running.

3.3 Quote Generation/Verification

For quote generation, the secure FitHealth Database uses the TDX guest driver to request a quote directly from the TD-quote enclave. For verification, the secure FitHealth Database uses a locally hosted quote verification service in combination with a toy implementation of Verification Custom Software (VCS) called SSS, provided by Intel, each of which is run out of its own Docker container.

3.4 Quote Verification Service

The responsibility of the quote verification service is to receive a quote and return a report that contains the quote status (either OK or with a known error) along with metadata about the report itself. Once the report is issued, the QVS passes it to the VCS for signing in order to verify to the initial requester, in this case our FitHealth service, that the QVS itself is not compromised.

3.5 Simple Signing Service (a basic VCS)

Our simple signing service signs the report received by the VCS and returns the signed report to the original requester for verification.

Note that the QVS-SSS architecture would not be secure in a real environment. First, the QVS and VCS communicate via mutual TLS based on self-signed certificates. A more robust implementation would use a CA to prove authenticity. Secondly, the verification of the signed report is not performed in our basic implementation. Only a naive check that the isvQuoteStatus body of the QVS report shows "OK" is checked, exposing the FitHealth Database to a compromised QVS. For the purposes of benchmarking, these differences are irrelevant.

General architecture and control flow are shown during attestation are shown in the following figure. Blocks marked with an X are run in Docker containers.

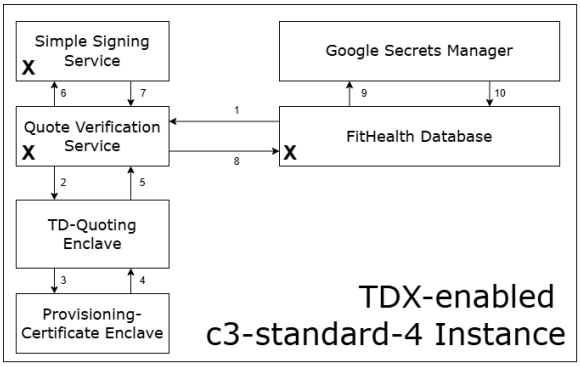

Figure 1. Secure FitHealth Architecture

4 Methods

4.1 Testing Database Performance

To evaluate the performance and scalability of our secure record service, we developed a custom asynchronous benchmarking tool in Python, testing/test_speed.py, using aiohttp and asyncio for high-concurrency HTTP workloads. The methodology is as follows:

Initial Data Load: Prior to mixed workload testing, the tool optionally performs an initial data load phase, controlled by a command-line flag. In this phase, a configurable number of unique records (default: 10,000) are inserted into the service using concurrent HTTP POST requests. Concurrency is limited by a semaphore to avoid overwhelming the backend, and each request is timed to measure write latency. We setteled on 200 threads constantly sending for the reading and writing phase, as that seemed to fully

saturate the backend while avoiding having it fall behind and drop connections. QSLlite is a fairly slow database, especially for writes, as it has a global write lock. Thus, with too many threads acting on the device at the same time, the node program would drop connections with nignx, causing errors and missed inputs on the fronted. We say this in fairly similar numbers for both the TDX and non-TDX versions.

Mixed Workload Phase: After the initial load, the tool executes a sustained mixed workload for a configurable duration (default: 120 seconds). During this phase, up to a fixed number of concurrent requests (default: 200) are maintained. Each in-flight request is randomly chosen to be either a read (HTTP GET, 90% probability) or a write (HTTP POST, 10% probability) operation, targeting randomly selected user IDs from the preloaded dataset. This simulates a realistic operational environment with a read-heavy access pattern.

Latency and Throughput Measurement: For every request, the tool records the operation type, HTTP status, and end-to-end latency. Latency statistics are computed separately for reads and writes, including average, minimum, maximum, median, and tail latencies (P90, P95, P99). Throughput is reported for both the initial load and mixed workload phases, as well as overall.

Error Handling and Reporting: Non-2xx responses and network errors are logged with details for post-test analysis. The tool aggregates and reports error rates, including conflict errors (HTTP 409) and timeouts.

Reproducibility: All test parameters (target IP, port, duration, concurrency, and initial load size) are configurable via command-line arguments, ensuring reproducibility and adaptability to different deployment environments.

This methodology enables rigorous, high-fidelity measurement of the service's performance under realistic, concurrent access patterns, and provides detailed insight into both latency and throughput characteristics.

4.2 Server load testing

To get the server load, we would run this one liner on the server before every load test to get the stats, and stop it once it was done and extract and read the data with

5 Results

Trial Number	Quote Verification Time (s)
Trial 1	0.131
Trial 2	0.128
Trial 3	0.145
Trial 4	0.118
Trial 5	0.124
Average	**0.129**

Table 1
Container launch to quote verification times

5.1 Insecure FitHealth Database Data

All tests were run over a 120 second period with 90% reads, max 100 in-flight.

Metric	Mixed	Reads Only	Writes Only
Average (ops/s)	1498.66	1349.03	149.43

Table 2
Average throughput across 5 trials

Percentile	90%	95%	99%
Read Latency (ms)	92.74	105.63	151.72

Table 3
Average Read Latency Percentiles

Percentile	90%	95%	99%
Write Latency (ms)	101.00	114.96	170.18

Table 4
Average Write Latency Percentiles

5.2 TDX-Secured FitHealth Database Data

All tests were run over a 120 second period with 90% reads, max 100 in-flight.

Metric	Mixed	Reads Only	Writes Only
Average (ops/s)	1327.16	1183.97	132.63

Table 5
Average throughput across 5 trials

Percentile	90%	95%	99%
Read Latency (ms)	159.116	185.81	422.902

Table 6
Average Read Latency Percentiles

Percentile	90%	95%	99%
Write Latency (ms)	144.01	189.39	467.13

Table 7
Average Write Latency Percentiles

Resource Usage results: Two example runs from both the TDX and Non-TDX environments are shown:

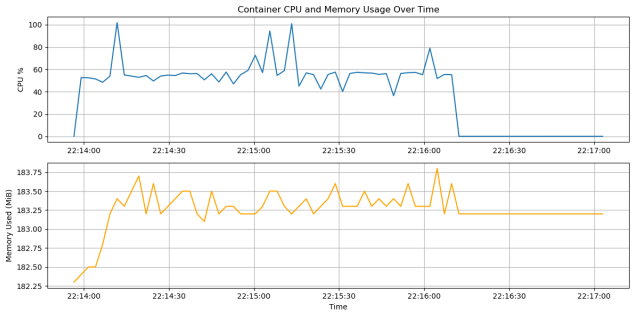

Figure 2. Non TDX run

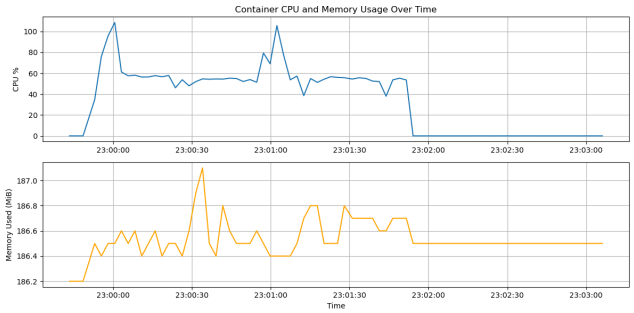

Figure 3. TDX run

6 Discussion

6.1 Verification/Attestation Time

Table 1 shows that the secure database does incur a time penalty from the need to generate and verify a quote. It will take on average 0.129 seconds longer at launch than the unsecured database.

6.2 Throughput Analysis

Across mixed health, reads-only, and writes only, we see a 12.3%, 16.4%, and 12.3% reduction in throughput for the TDX-enabled devices. These lie around the same range and are likely caused by TDX's strict memory encryption requirements for both cache and pinned memory. Over tens of thousands of requests, we clearly see the performance overhead incurred by TDX.

6.3 Write Latency

An even more notable difference can be seen in the upper percentiles of the read and write latencies. For write operations, the latency increases for 90th, 95th, and 99th percentile are 43%, 64.7&, and 174% respectively. The corresponding values for reads are 70%, 78%, and 180%. Again, the increased overhead of having to go through encrypted channels makes cache misses, presumably the rare operations corresponding to the high-latency operations, are more costly.

6.4 CPU and Memory Utilization

In both cases, we see a few distinct spikes in CPU usage, but overall stability around 50%. Memory is fairly stable as well. Seemingly, TDX has no significant impact on CPU load or memory usage, at least for this particular workload.

6.5 Overall Overhead Analysis

While we expect higher overhead as a result of the protections offered by TDX, the incurred penalty is well beyond what is expected. As the execution environments are identical with the sole exception of enabled TDX-features, we suspect this difference to be caused by some combination of network interfacing and hardware configuration that's particular to the design of the app itself. More investigation may reveal a hidden bottleneck or unexpected confounding factor that is exaggerating the measured overhead.