Early Draft: Evaluating Software-Based Replay Attack Mitigations in Intel SGXv2

Micah Brody
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
mbrody@nd.edu

I. Introduction

The growth of cloud computing in recent years has created a significant security question: How does one protect sensitive data while it is being processed by a third-party provider? Traditional data protection methods, such as encrypting data at rest (on disk) and in transit (over the network), do not protect data while it is in use (in memory and CPU registers). This leaves a critical vulnerability, as a compromised or malicious cloud provider could potentially access sensitive information.

Intel® Software Guard Extensions (SGX) is Intel's implementation of a Trusted Execution Environment (TEE), first introduced with the 2015 Skylake microarchitecture [1]. SGX allows a developer to create a protected area of execution in memory called an "enclave." The contents of the enclave are encrypted in main memory and are only decrypted inside the CPU package, making them inaccessible to any other software on the system, including the operating system (OS) or virtual machine manager (VMM). Communication with the enclave is restricted to a well-defined interface via special instructions (ECALLs and OCALLs). Furthermore, SGX provides a critical feature known as *remote attestation*, a cryptographic protocol that allows a remote party to verify that a specific application is running within a genuine SGX enclave on a legitimate Intel CPU.

The original SGX architecture (SGXv1) employed a hardware-based Merkle tree structure, known as the integrity tree. This allows SGX to cryptographically verify that enclave memory had not been tampered with during execution [1]. This mechanism provided strong guarantees against both modification and replay attacks, where an adversary attempts to restore old but valid memory states to compromise system integrity.

However, in recent years, Intel® SGX has undergone a significant strategic evolution. The initial generation was later deprecated on Intel's client-side Core processors. The second generation, SGXv2, is focused on server-grade platforms, such as the Intel® Xeon® Scalable processor family. This modern architecture features key changes, most notably a significant expansion of the Enclave Page Cache (EPC) size, which was achieved by removing the integrity tree and its associated protection against certain replay attacks [2].

While this change reduces computational overhead and enables larger enclave sizes, it fundamentally alters SGX's security guarantees, and makes SGX vulnerable to new attacks. Thus, Intel changed the security model of SGX, saying "Such attacks are outside the scope of the boundary of protection," [3]. This presents a weaker security model, as memory encryption alone cannot prevent an attacker from manipulating encrypted memory through substitution, replay, or remapping attacks.

Recent research has demonstrated the practical exploitability of this vulnerability. The Battering RAM attack, which costs less than $50 to implement, successfully compromises Intel SGXv2 by targeting memory integrity weaknesses [4]. These developments raise new questions about the security of SGXv2 deployments in production environments, particularly for applications handling sensitive data such as cryptographic keys, financial transactions, or confidential machine learning models.

Given that practical attacks like Battering RAM demonstrate the exploitability of SGXv2's weakened integrity guarantees, there is a need to evaluate whether software-based mitigations can provide adequate protection.

Research Questions. This project addresses three fundamental questions:

1) How exploitable are replay attacks against real-world SGXv2 applications?
2) What software-based mitigation strategies exist in the literature?
3) What are the security guarantees and performance of current software-based mitigations?

This project will address these questions in two parts. First, it will establish a practical baseline by developing a proof-of-concept replay attack against a representative SGXv2 application to demonstrate the vulnerability's real-world exploitability. Second, it will conduct a systematic analysis of software-based mitigation strategies in current literature. This will include an evaluation of the security guarantees and performance characteristics for each approach.

II. Related Work

A. Memory Integrity Attacks on SGX

Several attacks, categorically known as *replay attacks*, have demonstrated the exploitability of memory integrity vulnerabilities in SGX enclaves. PlunderVolt [5] exploits CPU voltage manipulation to induce controlled faults in SGX computations, compromising the integrity of enclave execution. SGX-Bomb [6] leverages Rowhammer-based bit flips to violate memory integrity guarantees. Most recently, the Battering RAM attack [4] demonstrates practical memory integrity attacks against Intel SGX using a low-cost hardware interposer (less than $50), showing that physical memory manipulation remains a viable attack vector even in modern TEE implementations. While there are attacks such as wiretap [7] that primarily focus on breaking the confidentiality of data within SGX, these integrity-focused attacks are particularly relevant to SGXv2, which removed hardware-based integrity protections. However, it is worth noting that integrity violations frequently serve as stepping stones to confidentiality breaches [8]. For example, an attacker who can replay old enclave states might restore cryptographic keys to previously known values, manipulate control flow to leak sensitive data, or force the enclave into a vulnerable state that exposes secrets. Comprehensive surveys [8], [9] document this broader attack landscape, emphasizing that integrity and confidentiality protection are not independent concerns but heavily linked requirements.

B. Software-Based Integrity Protection Mechanisms

In the absence of hardware integrity protection, several software-based approaches have been proposed to maintain data integrity in trusted execution environments.

Trusted monotonic counters provide a mechanism for detecting replay attacks by maintaining a strictly increasing value that cannot be rolled back [10]. Any attempt to restore an old state will be detected because the counter value will not match the expected sequence. Sarmenta et al. [11] introduced virtual monotonic counters that use a Trusted Platform Module (TPM) to implement a large number of logical counters with minimal physical hardware support. More recent work has explored cloud-based monotonic counter services [12] that provide rollback protection for distributed TEE applications.

Authenticated data structures (ADS) enable untrusted parties to prove properties about data integrity without revealing the entire dataset [13], [14]. One common ADS is a Merkle tree, a data structure in which a cryptographic hash tree allows verification of individual elements against a trusted root hash. While SGXv1 used hardware-based Merkle trees for memory integrity verification, software-based Merkle trees can serve as a possible alternative for protecting application data in SGXv2.

Some approaches rely on application-specific integrity checks using message authentication codes (MACs) or digital signatures to verify data authenticity. However, standalone cryptographic primitives are insufficient for replay protection, as attackers can substitute old-but-valid authenticated states. ROTE (Rollback Protection for Trusted Execution) [15] addresses this limitation through a distributed architecture. Rather than relying on a single platform's monotonic counter (which an attacker with physical access could reset), ROTE enlists multiple SGX-enabled processors to collectively maintain rollback protection. Each enclave periodically commits its state to a quorum of remote attestation servers, which verify and timestamp the updates. During recovery or verification, the enclave contacts this distributed service to ensure it has not been rolled back to an earlier state. This approach trades local hardware guarantees for distributed protocol guarantees, making it particularly relevant for SGXv2 environments where hardware integrity trees have been removed. However, ROTE's reliance on network connectivity and distributed consensus introduces new assumptions about availability and network security that must be carefully evaluated.

C. Research Gap and Contribution

While prior work has extensively documented attacks on SGXv1 and proposed various software-based integrity protection mechanisms, a critical gap remains in understanding their applicability to SGXv2. Most existing replay attack mitigation research assumes the presence of hardware integrity guarantees [8], which are absent in SGXv2's architecture. Furthermore, while attacks like Battering RAM [4] demonstrate the exploitability of SGXv2's weakened security model, there has been no systematic evaluation of whether software-based countermeasures can provide equivalent protection.

This work differs from existing research in three key ways. First, it focuses specifically on SGXv2's unique threat model, where hardware integrity protection has been deliberately removed. Second, rather than proposing new mitigation techniques, this project will present a comparative analysis of existing software-based approaches to evaluate their security guarantees and limitations in the absence of hardware support. Third, this project will provide a practical demonstration of replay attacks against SGXv2 to establish a concrete baseline for evaluating mitigation effectiveness. This work aims to bridge the gap between attack demonstrations and mitigation proposals by systematically analyzing whether software-only solutions can compensate for SGXv2's architectural changes.

III. Methodology

To address the research questions, this project employs a two-part methodology that combines practical demonstration with systematic analysis.

The first component is an empirical evaluation designed to confirm the real-world exploitability of replay attacks against SGXv2 (RQ1). We implemented a proof-of-concept attack targeting a representative stateful application developed for this project.[1] To make the concept of state rollback tangible, the application simulates a simple wallet service, managing an account balance and transaction count. This demonstration shows how a malicious service provider can manipulate the application's persistent state by replaying old, encrypted data to a newly instantiated enclave. The full details of the application architecture, the specific vulnerability exploited, and the attack execution are presented in Section IV.

The second component addresses the landscape of potential defenses (RQ2, RQ3) through a comparative analysis of software-based mitigation strategies found in the literature.

[1]The source code for the proof-of-concept is available at https://github.com/M21121/SGX-replay-attack-demo.

We analyze three primary categories of countermeasures: monotonic counter-based approaches, authenticated data structures, and distributed verification protocols. Each category is evaluated against a consistent framework to assess its security assumptions, performance overhead, and practical limitations. This comprehensive analysis is detailed in Section V. Together, these two components provide a holistic view of both the threat and the potential solutions for replay attacks in modern SGX environments.

IV. Attack Demonstration

As outlined in Section III, this section details the practical demonstration of a replay attack against an SGXv2 enclave to provide a concrete baseline for the exploitability of such vulnerabilities (RQ1). We designed and implemented a proof-of-concept application and used it to execute a state rollback attack that successfully bypasses the enclave's in-memory integrity protections. This section is structured as follows: First, we describe the architecture of our proof-of-concept application. Second, we analyze the specific vulnerability in its state-handling logic that makes the replay attack possible. Finally, we present the step-by-step execution of the attack and its successful outcome.

A. System Implementation

The application consists of three main components:

- *Client (`client.cpp`):* A command-line tool that allows a user to interact with the service. It sends commands such as deposit, withdraw, and query to the server over a UNIX socket.
- *Server (`server.cpp`):* The untrusted portion of the host application that creates and manages the SGX enclave. It listens for client connections, forwards commands to the enclave via ECALLs, and is responsible for persisting the enclave's encrypted state to the filesystem.
- *Enclave (`Enclave.cpp`):* The trusted component that manages the sensitive application state. The core data structure, `enclave_state_t`, holds the user's account balance and a transaction counter. To protect against simple replay attacks, the enclave also maintains a volatile, in-memory monotonic counter, `g_runtime_counter`, which demonstrates a naive attempt at replay protection. This will be shown to be inefficient.

State persistence is handled via an OCALL, `ocall_save_encrypted_state`, where the enclave encrypts its `enclave_state_t` along with the current `g_runtime_counter` and passes the ciphertext to the server to be written to a file, `enclave_state.bin`.

This design models real-world use cases for SGX such as secure credential storage, encrypted database operations, or state-dependent cryptographic protocols where replay attacks could compromise security.

The proof-of-concept was developed on a workstation equipped with an Intel® Xeon® Gold 5412U processor supporting SGXv2. All development utilized the official Intel® SGX SDK for Linux.

B. Vulnerability Analysis

The enclave implements a seemingly protective mechanism using a volatile, in-memory monotonic counter (`g_runtime_counter`) that increments with each state-changing operation. This counter is intended to detect replay attacks by verifying that any loaded state has a counter value consistent with the enclave's execution history. However, this approach suffers from a fundamental flaw that renders it ineffective against realistic replay attacks.

The replay detection logic, shown in Listing 1, attempts to prevent state rollback during enclave execution:

```
// Inside ecall_load_state() in Enclave.cpp
if (g_initialized && decrypted.saved_counter <=
    g_runtime_counter) {
    // Replay detected during runtime!
    return 1; // Replay attack detected
}
```

Listing 1. The replay check in `ecall_load_state`.

Vulnerability: Unprotected Counter State

The critical vulnerability lies in the counter's lack of protection against manipulation. The `g_runtime_counter` variable resides in the enclave's memory with no mechanism to verify its integrity or freshness. Without trusted, tamper-resistant storage to ensure the counter represents the true execution history, it can be manipulated through two primary attack vectors:

- **Reset on restart**: When the enclave is destroyed during program termination or system restart, the counter value is lost and re-initialized to zero in the new enclave instance.
- **Memory-based rollback**: An attacker with physical access can restore old memory snapshots that include both the application state and the counter itself, with no hardware mechanism to detect the manipulation.

This vulnerability enables two primary attack scenarios:

Persistent State Replay: A malicious host can manipulate the enclave's persistent storage between program executions. The attacker:

1) Captures a copy of the encrypted state file at a favorable point
2) Allows the legitimate user to advance the state
3) Terminates the enclave, destroying the in-memory counter
4) Replaces the current state file with the previously captured version
5) Restarts the enclave, which accepts the old state as valid because the new counter (0) is less than the saved counter

This scenario, which our proof-of-concept demonstrates in Section IV-C, exploits the counter reset that occurs between enclave sessions.

Memory-Based Replay: A more sophisticated attacker with physical access can perform memory-level manipulation without relying on file system operations. As demonstrated by the Battering RAM attack [4], an attacker can:

1) Capture encrypted memory snapshots during enclave execution
2) Manipulate the memory controller or use hardware interposers to restore old memory states
3) Replay entire memory pages, including both application state and the counter itself

In this scenario, the attacker restores both the application state *and* the counter to their previous values simultaneously, making the replayed state appear internally consistent and bypassing the protection mechanism entirely.

Root Cause: Absence of Memory Integrity Verification

The fundamental issue is SGXv2's removal of the hardware integrity tree that existed in SGXv1. In the original SGX architecture, a Merkle tree structure cryptographically verified the integrity and freshness of enclave memory on every access [1]. This hardware mechanism could detect if memory had been tampered with or rolled back to a previous state.

SGXv2 eliminated this integrity tree to reduce overhead and enable larger enclave sizes [2]. As a result, there is no hardware-level verification that the memory contents represent the most recent valid state. An attacker with physical access can restore old memory snapshots, and the CPU has no mechanism to detect this manipulation.

While trusted non-volatile storage (such as Intel SGX Platform Services counters [10], TPM counters [11], or distributed verification services [15]) can be adapted to prevent the demonstrated reboot attack, they do not fully address memory-based replay attacks where the counter itself is part of the rolled-back state.

Our demonstration focuses on the persistent state replay scenario because it clearly illustrates the core vulnerability: the inability to verify state freshness across enclave restarts. The security implications extend to memory-based attacks, which exploit the same fundamental weakness: the absence of hardware-level memory integrity verification. The software-based mitigations analyzed in Section V attempt to address this limitation through various architectural approaches, each with different trade-offs between security guarantees and performance overhead.

C. Experimental Results

As described in Section IV-B, replay attacks can be performed against a running enclave through memory manipulation or microarchitectural side-channels. Our experiment focuses on a persistent state replay attack, where the attacker manipulates saved state between program executions. This scenario provides a clear and reproducible demonstration of the core vulnerability in SGXv2's handling of persistent data. It is worth nothing that the underlying weakness, absence of memory integrity verification, applies equally to memory-based attacks.

The attack was executed in two phases, as automated by our `demo.sh` script:

Phase 1: State Capture and Advancement. First, the legitimate user's actions were simulated to create a history of valid states.

1) *Initial State Creation:* The server was started, and the client application was used to set the initial account balance to $100. The enclave saved this state (balance: $100, counter: 1) to `enclave_state.bin`.
2) *State Capture:* The malicious host operator made a copy of this file, saving it as `old_state_balance_100.bin`. This represents the old state to be used in the replay.
3) *State Advancement:* The user then made a deposit of $75. The enclave's internal state was updated (balance: $175, counter: 2) and saved, overwriting the previous `enclave_state.bin`.
4) *Program Shutdown:* The server was terminated, destroying the enclave and its volatile in-memory counter.

Phase 2: Replay Attack After Restart. Next, the server was restarted under the control of the malicious operator.

1) *Malicious State Replacement:* Between shutting down and restarting the server, the operator replaced the current `enclave_state.bin` (containing $175) with the previously captured `old_state_balance_100.bin`.
2) *Program Restart:* The server was launched again. It created a new enclave instance, which started with its in-memory counter reset to 0.
3) *State Rollback:* The new enclave loaded the state from the malicious file. The replay check passed because the stale state's counter (1) was greater than the enclave's new in-memory counter (0).
4) *Verification:* A final query from the client confirmed that the account balance was $100. The attack was successful, and the user's $75 deposit was effectively erased.

The console output below shows the final query after the attack, confirming the state was rolled back:

```
+--------------------------------------+
|          SECURE BANK ACCOUNT         |
+--------------------------------------+
| Account: John's Account              |
| Balance: $100                        |
| Transactions: 1                      |
+--------------------------------------+
```

V. Comparative Security Analysis of Software-based Mitigation Strategies

Having demonstrated the practical threat of persistent state replay attacks in Section IV, the final component of this research will address the landscape of potential defenses (RQ2, RQ3). This section outlines the methodology for a literature-based analysis of software-based mitigation approaches. The analysis will synthesize findings from existing research to evaluate security guarantees and performance characteristics, structured around three primary categories of countermeasures and a consistent evaluation framework.

A. Mitigation Categories

Our analysis will examine three primary categories of software-based replay protection found in the literature:

1) *Monotonic Counter-Based Approaches*: This category includes solutions that rely on a secure, non-volatile source of monotonicity, such as the Intel SGX Platform Services (PS) counters [10], TPM-backed virtual counters [11], or cloud-based counter services [12].
2) *Authenticated Data Structures*: These approaches use cryptographic accumulators, such as Merkle trees, to create a verifiable history of enclave state transitions, allowing for the detection of forks or rollbacks [13], [14].
3) *Distributed Verification Protocols*: This category covers systems that leverage external, trusted verifiers or a distributed network of peers to attest to the freshness of an enclave's state, such as the ROTE protocol [15].

B. Evaluation Framework

To provide a structured comparison, each mitigation category will be assessed according to the following criteria:

- *Security Guarantees*: What specific replay attack scenarios does the mitigation prevent? What are its core security assumptions and trust requirements (e.g., trusted hardware, trusted third parties)?
- *Performance Characteristics*: What is the computational, storage, and network overhead as reported in the literature? How does the approach impact enclave startup time and state transition latency?
- *Deployment Constraints*: What are the external dependencies (e.g., network connectivity, specialized hardware)? How does the approach scale with state complexity and transaction frequency?
- *Limitations and Vulnerabilities*: Under what conditions can the mitigation be bypassed? What are the practical barriers to adoption for developers?

VI. Conclusion and Future Work

This early draft has presented the foundational work for evaluating software-based replay attack mitigations in SGXv2. We have successfully designed, implemented, and executed a proof-of-concept persistent state replay attack against a representative stateful enclave. This practical demonstration provides a concrete baseline for the vulnerability, confirming the real-world exploitability of such attacks and addressing our first research question.

Future work will focus on completing the comparative analysis of mitigation strategies, following the evaluation framework detailed in Section V. By systematically assessing monotonic counter-based approaches, authenticated data structures, and distributed verification protocols, we will provide a comprehensive answer to our remaining research questions. The final paper will synthesize these findings to offer a holistic view of the threat landscape and provide actionable recommendations for developers building secure applications with SGX.

References

[1] V. Costan and S. Devadas, "Intel sgx explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[2] The Linux Kernel Developers, "Software guard extensions (sgx)," https://docs.kernel.org/arch/x86/sgx.html, 2023, accessed: 2025-10-21.

[3] Intel Corporation, "More information on encrypted memory frameworks," Intel Developer Articles, Oct. 2024, accessed: 2025-10-22. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/news/more-information-encrypted-memory-frameworks.html

[4] J. De Meulemeester, I. Verbauwhede, D. Oswald, and J. Van Bulck, "Battering ram: Low-cost interposer attacks on confidential computing via dynamic memory aliasing," in *47th IEEE Symposium on Security and Privacy (S&P)*, 2026.

[5] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel SGX," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1466–1482.

[6] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking down the processor via Rowhammer attack," in *Proceedings of the 2017 Workshop on System Software for Trusted Execution (SysTEX)*. ACM, 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3152701.3152709

[7] A. Seto, O. K. Duran, S. Amer, J. Chuang, S. van Schaik, D. Genkin, and C. Garman, "Wiretap: Breaking server sgx via dram bus interposition," in *2025 SIGSAC Conference on Computer and Communications Security (CCS '25)*. Association for Computing Machinery, 2025. [Online]. Available: https://wiretap.fail

[8] A. Nilsson, P. N. Bideh, and J. Brorsson, "Security vulnerabilities of SGX and countermeasures: A survey," *ACM Computing Surveys*, vol. 54, no. 6, 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3456631

[9] J. Gascón, L. Giner, C. Canales, and L. Lemus, "A survey on side-channel attacks and countermeasures on intel sgx," in *2024 IEEE International Conference on Consumer Electronics (ICCE)*, 2024, pp. 1–6.

[10] Intel Corporation, "Trusted time and monotonic counters with Intel Software Guard Extensions platform services," Intel Developer Documentation, 2024, accessed: 2025-10-22. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/671564/trusted-time-and-monotonic-counters-with-intel-software-guard-extensions-platform-s html

[11] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas, "Virtual monotonic counters and count-limited objects using a TPM without a trusted OS," in *Proceedings of the First ACM Workshop on Scalable Trusted Computing (STC)*. ACM, 2006, pp. 27–42. [Online]. Available: https://people.csail.mit.edu/devadas/pubs/ccs-stc06.pdf

[12] S. Matetic *et al.*, "VMCaaS: Virtual monotonic counters as a service," in *Proceedings of the 17th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2024.

[13] A. Miller, M. Hicks, J. Katz, and E. Shi, "Authenticated data structures, generically," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2014, pp. 411–423. [Online]. Available: https://soc1024.ece.illinois.edu/gpads/gpads-full.pdf

[14] R. Tamassia and N. Triandopoulos, "Authenticated data structures," in *Algorithms and Theory of Computation Handbook*. Chapman and Hall/CRC, 2010, pp. 9–1–9–26.

[15] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1289–1306. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic