Distributed Cheat-Proof Games

Maan Meher

CS4821 - MSci Final Year Project

Supervisor: Dan O'Keeffe

Department of Computer Science

Royal Holloway, University of London

The increasing demand for scalable and secure multiplayer video games has highlighted the inherent limitations of traditional client-server architectures. These architectures rely heavily on centralized servers to manage game logic, which, while effective in mitigating cheating, presents critical challenges in scalability, server load, and operational costs【4】. Additionally, the reliance on cloud infrastructure, such as Amazon EC2, introduces potential security vulnerabilities, including data breaches and malicious activities by cloud service providers【5】【6】.

This project aims to leverage Intel Software Guard Extensions (SGX) to develop a distributed cheat-resistant game architecture that addresses these limitations. Intel SGX is a hardware-based security feature that provides trusted execution environments, known as enclaves, which ensure the confidentiality and integrity of computations even when the operating system is compromised【11】. SGX enclaves enable user-level and operating system code to define protected memory regions that are encrypted and decrypted dynamically by the CPU【12】. While SGX has promising applications in secure remote computation, secure web browsing, and digital rights management (DRM), it also has inherent limitations and vulnerabilities, including susceptibility to side-channel attacks such as PLATYPUS, Rowhammer, and Spectre-like threats【12】【14】.

The motivation for this project stems from the significant negative impact that cheating has on user experience in multiplayer games. Cheating not only undermines the integrity of gameplay but also erodes user trust and satisfaction, ultimately harming long-term engagement and retention. The frustration of losing a game to a cheater and deranking from Platinum to Gold, for example, can have a severe detrimental effect on player morale. Existing anti-cheat solutions, such as Riot Vanguard, often lead to system instability, including blue screens or freezes after updates, while EasyAntiCheat has been exploited through remote code execution (RCE) vulnerabilities【4】. Additionally, kernel-level anti-cheat systems have introduced new risks, such as system instability and potential privacy concerns【13】. These limitations underscore the need for an advanced security solution that effectively safeguards game integrity while enhancing user experience.

This project involves partitioning a real-world game (e.g., ioquake3)【9】 by executing critical game components within SGX enclaves on client machines. This distributed approach enables computational tasks to be securely offloaded to client systems, alleviating the pressure on centralized servers and mitigating latency issues. Additionally, the distributed approach inherently provides greater scalability, as more clients contribute computational power.

Offloading these tasks enhances scalability, reduces server load, and minimizes bandwidth usage. Furthermore, secure communication protocols, such as Transport Layer Security (TLS), will be integrated to ensure data confidentiality and integrity during transmission between the server and clients. The use of SGX enclaves not only prevents tampering with the game logic but also ensures that even if a client device is compromised, the critical game computations remain secure.

Introduction to Intel SGX

Intel Software Guard Extensions (SGX) represents a landmark in architectural innovation, designed to address security concerns in computational environments prone to compromise. SGX enables the creation of isolated execution environments known as *enclaves*. These enclaves ensure the confidentiality and integrity of sensitive computations and data, even in the presence of malicious privileged software. By leveraging hardware-based isolation and cryptographic mechanisms, SGX empowers developers to implement secure solutions for diverse applications, ranging from cloud computing to multi-party computations【¹】【²】【³】.

Hardware Foundations of SGX

1. Enclave Isolation

SGX enforces enclave isolation through a synergy of hardware-enforced access controls and cryptographic protections. The CPU mediates access to enclave resources by consulting the Enclave Page Cache Metadata (EPCM), which tracks permissions for memory pages allocated to enclaves. Unauthorized attempts to access these pages are blocked at the hardware level, ensuring robust isolation【¹】【²】【³】.

2. Processor Reserved Memory (PRM)

The Processor Reserved Memory (PRM) forms the cornerstone of SGX's secure memory model. Reserved exclusively for enclaves, this protected memory region is inaccessible to the OS, hypervisors, and other system-level software. The PRM's boundaries are enforced by the CPU, preventing any direct memory access (DMA) attacks【¹】【²】.

3. Enclave Page Cache (EPC)

The EPC is a dedicated section within the PRM, designed to store enclave code and data in 4 KB pages. The CPU enforces fine-grained access controls over these pages, leveraging EPCM metadata to validate the legitimacy of access requests. Additionally, the EPC supports memory integrity and confidentiality by encrypting and validating pages swapped to DRAM【¹】【²】.

4. Memory Encryption Engine (MEE)

The MEE plays a critical role in securing data as it transitions between the processor and DRAM. By employing AES-based encryption, the MEE safeguards enclave data from physical memory attacks. The encryption keys, generated and managed within the CPU, are inaccessible to external entities, ensuring that data integrity remains uncompromised【²】【³】.

Principles of Trusted Execution Environments (TEEs)

1. Definition and Role

A Trusted Execution Environment (TEE) represents a secure processing enclave within a system, offering assurances for code authenticity, runtime integrity, and data confidentiality. Unlike standard execution environments, TEEs mitigate threats by isolating sensitive operations from potentially malicious system components. This separation is achieved through a combination of hardware protections, cryptographic protocols, and controlled inter-environment communications【33†source】.

2. Key Building Blocks

- **Secure Boot:** TEEs rely on a cryptographic secure boot process to ensure that only authenticated software is loaded into the environment, preventing the execution of malicious code at startup【33†source】.
- **Secure Storage:** Persistent storage within a TEE is encrypted and integrity-protected. Mechanisms like Replay-Protected Memory Blocks (RPMBs) prevent rollback attacks, ensuring that data remains consistent and unaltered【33†source】.
- **Trusted I/O Path:** To protect data integrity during input and output operations, TEEs secure interactions with peripherals such as keyboards and displays. These measures thwart attacks such as keylogging and screen capturing【33†source】.
- **Inter-Environment Communication:** Communication between the TEE and the normal world is facilitated through secure channels, often implemented using Remote Procedure Calls (RPCs). This ensures the confidentiality and integrity of data exchanges【33†source】.
- **Secure Scheduling:** TEEs incorporate secure scheduling mechanisms that balance task execution across trusted and untrusted environments without compromising real-time constraints【33†source】.

Execution Workflow in SGX

1. Enclave Initialization

The lifecycle of an SGX enclave begins with its initialization by untrusted system software. During this phase, the CPU generates a cryptographic measurement of the enclave's loaded code and data, forming the basis for attestation. This measurement, represented as a hash, enables verification of the enclave's integrity and authenticity【1】【2】【3】.

2. Secure Execution

Enclaves execute in user mode (Ring 3), isolated from privileged software operating in Ring 0. Transitions between enclave and non-enclave execution states are managed via Asynchronous Enclave Exits (AEX), ensuring that the enclave's state remains secure during context switches and interruptions【1】【2】.

3. Remote Attestation

SGX's remote attestation protocol provides a mechanism for external entities to verify an enclave's integrity. This process relies on a hardware-protected attestation key, enabling the generation of cryptographic proofs tied to the enclave's measurement. Remote parties can validate these proofs against Intel's certification infrastructure, establishing trust in untrusted environments【1】【2】【3】.

4. Data Protection

All sensitive computations and data handled by the enclave remain encrypted outside its boundaries. Enclave code operates in plaintext only within the confines of the EPC. Interactions with external systems, such as disk I/O or network communication, are safeguarded through encryption and integrity validation【2】.

Advanced Hardware Operations in SGX

1. EPC Management

The EPC's metadata enforces granular access controls for each memory page allocated to an enclave. When memory pages are swapped out to DRAM, they are encrypted to maintain confidentiality. Upon reloading, cryptographic checks ensure that the pages remain unaltered【2】【3】.

2. Memory Access Control

Hardware-enforced access control mechanisms in SGX prevent unauthorized entities, including the OS, from accessing enclave memory. Any violations trigger exceptions, ensuring that malicious software cannot tamper with or observe enclave data【1】【2】.

3. Mitigation of Side-Channel Attacks

SGX incorporates techniques to mitigate common side-channel threats, such as cache timing attacks and speculative execution vulnerabilities. Strategies like memory access obfuscation and noise injection reduce the risk of information leakage【2】【3】.

Applications of SGX

1. Cloud Computing

SGX transforms cloud computing by enabling secure processing of sensitive data on untrusted platforms. Applications include confidential data analysis, secure database queries, and privacy-preserving machine learning【²】【³】.

2. Blockchain

Enclaves play a pivotal role in blockchain ecosystems by securing smart contract execution and ensuring the integrity of cryptographic operations, such as consensus mechanisms and transaction validation【²】【³】.

3. Multi-Party Computations

SGX facilitates collaborative computations among multiple parties by safeguarding the confidentiality and integrity of their respective inputs. This has significant implications for fields such as secure voting and federated learning【³】.

Challenges and Limitations

1. Resource Constraints

The limited size of the EPC poses challenges for memory-intensive applications. Solutions such as SGXv2 address this by introducing dynamic memory allocation, though performance overheads persist【³】.

2. Security Limitations

Despite its robust design, SGX remains vulnerable to sophisticated attacks, such as microarchitectural side channels and rollback exploits. Continued research is essential to address these emerging threats【²】【³】.

3. Trust Dependencies

The reliance on Intel’s root of trust creates potential risks related to vendor lock-in and centralization of authority. This dependency highlights the need for alternative implementations and open standards【¹】【³】.

Future Directions and Enhancements

1. Enhancing Security

Innovative cryptographic protocols and architectural improvements are needed to mitigate advanced threats such as speculative execution vulnerabilities and memory replay attacks【²】【³】.

2. Expanding Use Cases

SGX's integration into emerging domains, such as the Internet of Things (IoT), autonomous systems, and privacy-focused AI applications, holds promise for revolutionizing secure computing【²】【³】.

3. Standardization and Openness

The development of open standards and alternative trusted execution frameworks will reduce dependency on proprietary solutions, fostering greater innovation and adoption【³】.

Objectives

The objectives I have achieved so far in this project include:

Understanding SGX Core Concepts:
I successfully integrated SGX-specific operations such as enclave creation and secure function execution via ECALLs.
SGX-provided libraries (sgx_urts for the untrusted runtime system and sgx_trts for the enclave runtime) were utilized to facilitate this.

Demonstrating ECALLs and OCALLs:
I implemented ECALLs to handle sensitive computations like random number generation, damage calculation, and potion effects within the secure enclave.
OCALLs were employed to facilitate interactions between the enclave and the untrusted application, such as logging data and generating seeds.

Developing Practical Simulations:
Two core applications were built to demonstrate secure enclave functionality:

Shot Damage Simulation (App 1): Simulates damage calculations based on random values for different body parts (head, torso, legs).

Potion Simulation (App 2): Simulates the effect of consuming various potions on a player's health, demonstrating dynamic state updates through secure computations.

Proof of Concept for Secure Computations:

By using SGX enclaves for random number generation and game simulations, I have showcased the potential of secure enclaves in real-world scenarios like gaming, where integrity and fairness are critical.

Project Demo Link:

https://youtu.be/x-xiohlHutM

Technical Accomplishments

1. **Enclave Creation**
A cornerstone of my work lies in the successful creation and destruction of SGX enclaves. This involved:

Initialization:
Using sgx_create_enclave to load the TestEnclave.signed.dll file and initialize it with a launch token and enclave ID.
Ensuring compatibility by managing enclave-specific errors and debugging issues during initialization.

Destruction:
Implementing sgx_destroy_enclave to clean up resources after computations.

2. **ECALLs and Secure Computations**
I have demonstrated a mastery of ECALLs, where sensitive computations were securely executed within the enclave. Examples include:

Damage Calculation:
An ECALL (ecall_calculate_damage) computes damage based on body parts (head, torso, legs), with random modifiers securely generated using sgx_read_rand within the enclave.
The modular structure ensures that different computations are contained securely and independently.

Potion Effects:
Another ECALL (ecall_consume_potion) handles potion consumption simulations, modifying health based on the type of potion consumed (e.g., health potions, berserker potions).

State management ensures health boundaries are respected (e.g., health cannot drop below 0 or exceed 1000).

3. **OCALLs for Secure Interaction**
To demonstrate two-way communication between the enclave and the untrusted application:

Logging:
An OCALL (ocall_log_message) allows the enclave to send logs to the untrusted application for debugging or informational purposes.

Random Seed Generation:
Another OCALL (ocall_get_random_seed) enables the enclave to securely obtain seeds for random number generation, ensuring unpredictability and fairness.
These OCALLs highlight how enclaves can rely on the untrusted environment for auxiliary operations while keeping sensitive computations isolated.

4. **Modular Applications**
I have developed two standalone applications (App 1 and App 2), each showcasing a different use case of SGX:

App 1: Shot Damage Simulation:
Computes damage based on random modifiers for head, torso, and legs.
Outputs the damage values to the user, demonstrating secure computation and modular ECALL design.

App 2: Potion Simulation:
Simulates potion consumption and modifies player health accordingly.
Includes health boundary checks and special cases like "YOU DIED!" and "GODLIKE!!!" states, adding realism to the simulation.
Both applications demonstrate the versatility of SGX enclaves in securely handling computations with random elements.

Challenges and Solutions

1. **Header and Library Dependencies**
I initially encountered errors with missing headers (TestEnclave_t.h) and libraries (sgx_urts.dll), which highlighted the need for precise configuration.

Solution:
Verified include paths and explicitly linked libraries in the Visual Studio project settings.

2. **Unresolved External Symbols**

Errors such as unresolved OCALL symbols (ocall_log_message, ocall_get_random_seed) were resolved by:
Implementing dummy OCALLs in the untrusted application.
Declaring OCALLs correctly in the EDL file.
3. **Random Number Generation**
Using rand() for random number generation introduced predictability and lacked security.

Solution:
I incorporated sgx_read_rand to securely generate random values within the enclave.

4. **Character Set and Encoding**
I faced issues with LPCWSTR (wide characters) due to mismatches in character set configurations.

Solution:
Explicitly used LPCSTR or adjusted the project to use multibyte character encoding.

Why I Chose a VM on Linux

1. **Intel SGX Compatibility**
Intel SGX development requires specific hardware and software configurations, which are not universally supported across all platforms. By using a VM running a Linux distribution, I gained access to:

Direct SGX Support:
Linux distributions such as Ubuntu natively support SGX-enabled kernels, allowing me to enable SGX features efficiently.
I configured the VM to load SGX-specific kernel modules, ensuring a functional environment for enclave development.

Flexibility in Configuration:
Using a VM allowed me to modify kernel parameters and test various configurations without affecting my host machine.

2. **Isolation and Portability**
Sandboxed Environment:
A VM provides a controlled, isolated environment for development, reducing the risk of interfering with my primary operating system.
Portability:
The VM's configuration can be exported or replicated, enabling consistent development environments across machines.

3. **Tooling and Debugging**
Linux offers a wealth of development tools, including gdb, make, and SGX-specific SDKs, which are more mature and feature-complete on this platform.
Debugging SGX enclaves in a VM allowed me to capture detailed logs and test edge cases without risking instability on my host system.

Software Engineering Practices

1. **Modular Design**
To ensure scalability and maintainability:

Independent Applications:
I developed separate applications (App 1 for damage calculations, App 2 for potion effects), ensuring that each use case could be tested independently.

Enclave-Driven Architecture:
Sensitive computations were isolated within the SGX enclave, while non-sensitive operations were handled by the untrusted application.

3. **Build Automation**
I leveraged Visual Studio project settings for consistent builds:

Dependency Management:
Explicitly linking SGX libraries and configuring include paths ensured minimal build errors.

4. **Debugging Practices**
To identify and resolve issues efficiently:

Verbose Logging:
I implemented logging both within the enclave (via OCALLs) and in the untrusted application to capture runtime states.

Error Handling:
SGX error codes were handled systematically, with meaningful messages displayed to the user.

5. **Testing and Validation**
Unit Testing:
I tested individual ECALLs and OCALLs to verify their correctness before integrating them into the main application.

Boundary Testing:
Simulations were subjected to edge cases, such as maximum health thresholds and invalid inputs, to ensure robustness.

General Build Practices

1. **Cross-Platform Considerations**
While development primarily took place on Linux, I ensured compatibility with Windows by:
Testing enclave builds on both platforms.
Using tools like MinGW and Visual Studio for cross-compilation.

2. **Character Encoding**
To resolve compatibility issues, I configured my projects to consistently use multibyte character sets (LPCSTR), avoiding conflicts with wide characters.

3. **Documentation**
I maintained detailed documentation throughout the project:

Diary:
A chronological log of achievements and challenges provided insights into my progress.
Code Comments:
Functions and blocks of code were well-commented to explain their purpose and logic.

Key Takeaways and Future Directions

1. **Flexibility and Scalability**:
My modular approach enables the easy addition of new features, such as more potion types or enhanced damage calculations.

2. **Practical Applications**:
These simulations serve as prototypes for secure gaming mechanics, illustrating the feasibility of using SGX enclaves in the gaming industry to prevent cheating.

Future Work:
Incorporating advanced features like encrypted communication between enclaves and untrusted applications.
Expanding the project to include more complex simulations, such as multi-player scenarios or secure leaderboards.
Evaluating the performance overhead introduced by SGX and optimizing enclave design for real-time applications.
Importing actual cherry-picked ioquake3 mechanics into enclaves to be secured.

Future Work for This Term

Integrate SGX-Based Components with Game Communication Protocol

I need to focus on linking the secure enclave computations I've developed, such as damage calculation and potion effects, with a broader game communication framework. This includes designing how the SGX-enforced logic will interact with game mechanics in real time.

Secure Communication Implementation

I must implement a secure communication protocol, likely using Transport Layer Security (TLS), to ensure data integrity and confidentiality during client-server interactions. This involves testing the performance of TLS within the context of multiplayer gaming to minimize latency overhead.

Prepare for ioquake3 Integration

I need to explore and document how ioquake3 mechanics can be integrated into SGX enclaves. This includes identifying specific game features, such as leaderboards or critical gameplay logic, that would benefit from secure execution.

Optimize Current Simulations

The standalone applications I've developed for secure computations, such as damage calculation and potion simulations, need further refinement. This includes improving modularity, reducing memory overhead, and ensuring scalability for real-world use cases.

Future Work for Next Term

Refine Implementation Based on Feedback

I'll revisit the current SGX implementation and simulations to address gaps or areas for improvement highlighted in the interim review. This includes optimizing enclave logic and exploring advanced features such as dynamic memory allocation in SGXv2.

Develop Performance Metrics

I plan to create and implement performance benchmarks to evaluate the cheat-proof architecture's efficiency. This includes measuring the overhead introduced by SGX enclaves and analyzing their impact on real-time gameplay.

Modularize the Codebase

I will restructure my code into a reusable framework following software engineering best practices. This includes ensuring clear separation of functionality, robust documentation, and modularity for potential expansion or integration with other games.

Establish Secure Client-Server Interfaces

I need to build interfaces that enable seamless communication between SGX-enabled clients and standard game servers. This involves incorporating TLS and ensuring compatibility with existing game protocols.

Conduct Comprehensive Evaluations
Toward the end of Term 2, I'll conduct rigorous testing of the SGX-enabled game architecture. This includes security analysis, scalability testing, and performance evaluations under realistic multiplayer gaming conditions.

Integrate ioquake3 Mechanics into SGX
I'll focus on embedding ioquake3-specific gameplay features into the SGX enclaves, such as secure leaderboards or game-state validations. This will demonstrate the practical application of the cheat-proof architecture.

Finalize Documentation and Project Report
I'll document all methodologies, results, and analyses in the final project report. This will involve consolidating Term 1 and Term 2 work into a cohesive narrative, supported by benchmarks and evaluations.

Comparison and Analysis: Progress vs. Plan

Term 1 Progress
My initial project timeline outlined a structured progression, from learning foundational concepts to implementing secure game architecture with Intel SGX. The following sections compare each milestone with actual progress and highlight challenges faced:

Weeks 1–2: Study Intel SGX Background and Distributed Game Architectures

Planned:
Gain a thorough understanding of Intel SGX, focusing on architecture, security features, and applications.
Analyze the vulnerabilities in distributed game architectures to guide secure system design.

Progress:
Studied Intel SGX architecture, including its enclave isolation mechanism, Processor Reserved Memory (PRM), and the Memory Encryption Engine (MEE).
Investigated distributed game architectures, focusing on scalability challenges and potential attack vectors, such as tampering with client-side computations.

Challenges:
The learning curve for SGX was steeper than anticipated, requiring additional time to grasp enclave functionality and cryptographic foundations.
Distributed game architecture analysis revealed a wider scope of vulnerabilities than initially considered, necessitating refinement of focus areas.

Resolution:

Additional time was allocated to studying SGX tutorials and resources, which provided clarity and foundational understanding.
Prioritized addressing key vulnerabilities, such as unauthorized client tampering and data interception.

Weeks 3–4: Experiment with the Intel SGX SDK

Planned:
Gain hands-on experience with the SGX SDK by implementing small-scale tutorial problems.
Demonstrate basic enclave functionality.

Progress:
Successfully set up the Intel SGX development environment on Ubuntu.
Implemented initial tutorial problems, including secure random number generation and basic ECALL-OCALL communication.

Challenges:
Encountered build and runtime errors, such as missing headers and libraries. Debugging consumed significant time.
Initial misconfigurations in the SGX edger8r tool caused delays in generating required files.

Resolution:
Addressed errors by systematically verifying dependencies and configurations.
Documented debugging steps for future reference to streamline similar issues.
Weeks 5–6: Implement Foundational Game Components in SGX Enclaves

Planned:
Begin integrating SGX into a real-world game (e.g., ioquake3).
Implement foundational components, such as secure computation for game mechanics.

Progress:
Developed standalone applications to simulate secure computations, including random damage calculation and potion effects within enclaves.
Modularized ECALLs to ensure separation of functionality for damage and potion simulations.

Challenges:
Compatibility issues with ioquake3 and SGX integration led to a pivot toward building independent simulations.
Memory constraints within the Enclave Page Cache (EPC) limited the complexity of initial implementations.

Resolution:
Focused on developing modular, reusable simulations to lay the groundwork for future integration with ioquake3.

Adjusted implementation scope to optimize for SGX's memory limitations.

Weeks 7–8: Research and Evaluate Secure Communication Protocols

Planned:
Investigate TLS and its applicability to multiplayer gaming.
Design secure communication strategies for client-server interactions.

Progress:
Reviewed literature on TLS and its integration with SGX.
Explored strategies for minimizing latency and ensuring encrypted communication between enclaves and servers.

Challenges:
Balancing the performance overhead of TLS with real-time requirements of multiplayer games remains a work in progress.

Resolution:
Identified existing libraries for efficient TLS integration and scheduled further testing during Term 2.

Weeks 9–10: Integrate SGX Components with Game Communication Protocol

Planned:
Link SGX-based components with the broader game infrastructure.

Progress:
Developed standalone simulations to validate secure enclave computations.
Began exploring ioquake3 mechanics for future SGX integration.

Challenges:
Integration with ioquake3 was deferred due to compatibility and time constraints.

Resolution:
Decoupled game communication protocol research from enclave simulations to maintain project momentum.

Bibliography

1. M. Sabt, M. Achemlal, and A. Bouabdallah, “Trusted Execution Environment: What It Is, and What It Is Not,” IEEE Communications Surveys & Tutorials, vol. 18, no. 3, pp. 1351–1372, 2016.

2. V. Costan and S. Devadas, “Intel SGX Explained,” IACR Cryptology ePrint Archive, 2016.

3. W. Zheng et al., “A Survey of Intel SGX and Its Applications,” Frontiers of Computer Science, vol. 15, no. 3, pp. 153808, 2021.

4. Dexerto. ”Apex Legends pro match ruined as Genburten and ImperialHal get hacked during ALGS.” Available at: https://www.dexerto.com/apex-legends/apex-legends-pro-match-ruined-as-genburten-and-imperialhal-get-hacked-during-algs-2595468/

5. Intel Software Guard Extensions (SGX) - Wikipedia. Available at: https://en.wikipedia.org/wiki/Software_Guard_Extensions

   Valve Anti-Cheat (VAC) System. Available at: https://www.valvesoftware.com/

   GamesRadar+. ”Diablo III error 37 causing a great wailing and gnashing of teeth as players worldwide fail to log in. The solution? Wait a bit.” Available at: https://www.gamesradar.com/

6. IBTimes. ”The Division Servers Struggle As Ubisoft Confirms Login Problems.” Available at: https://www.ibtimes.com/

7. Raaen, K., Grønli, T.-M. ”Latency Thresholds for Usability in Games: A Survey.” NIK 2014. Available at: NIK Conference

8. Howard, E., Cooper, C., Wittie, M. P., Swinford, S., & Yang, Q. (2014). ”Cascading Impact of Lag on User Experience in Multiplayer Games.” NetGames 2014. Available at: ResearchGate

9. Intel®SGX SDK for Linux OS. Available at: https://01.org/intel-softwareguard-extensions/downloads

10. ioquake3. Available at: https://ioquake3.org/

11. Yahyavi, A., Huguenin, K., Gascon-Samson, J., Kienzle, J., & Kemme, B. (2013). ”Watchmen: Scalable Cheat-Resistant Support for Distributed Multi-Player Online Games.” IEEE 33rd International Conference on Distributed Computing Systems. Available at: IEEE

12. Costan, V., Devadas, S. (2016). ”Intel SGX Explained.” IACR Cryptology ePrint Archive, Report 2016/086. Available at: ePrint

13. ”PLATYPUS: With Great Power comes Great Leakage.” Available at: https://platypusattack.com/

14. GitHub Gist. ”Why You Should Reconsider Playing League of Legends: The Risks of Kernel-Level Anti-Cheat Software.” Available at: https://gist.github.com/stdNullPtr/2998eacb71ae925515360410af6f0a32

15. Fortanix. ”Impact of Processor Side-Channel Attacks.” Available at: https://www.fortanix.com/blog/intel-sgx-technology-and-the-impact-of-processor-side-channel-attacks

Diary (compressed from diary.md)

2024-12-01
Expanded the SGX enclave functionality to support potion consumption alongside shot damage simulation.

Added a new ECALL, ecall_consume_potion, to simulate the effects of five different potion types: Health, Damage, Berserkers, Weakness, and Normalcy.
Modified the existing enclave to ensure compatibility with both potion and shot damage simulations while maintaining separation of functionality.
Developed a new application that exclusively focuses on potion consumption, demonstrating modular reuse of the enclave.
Ensured that health boundaries are enforced, with special outputs for critical health conditions like "YOU DIED!" and "GODLIKE!!!".
2024-11-23
Implemented random damage calculation based on body part hit.

Enhanced the damage calculation function to add random modifiers for specific body parts:
Head: 100 base damage + (0–50 random modifier).
Torso: 100 base damage + (0–25 random modifier).
Legs: Flat 100 damage.
Ensured that the random modifiers are strictly non-negative using unsigned integers and proper modulo operations.
Verified functionality through multiple tests, simulating shots to various body parts.
2024-11-18
Implemented random number generation within the SGX enclave.

Replaced the previous string manipulation logic with a secure random number generator (ecall_generate_random).
Utilized the sgx_read_rand function to generate a 32-bit random number securely within the enclave.
Updated the untrusted application to call the new ECALL and display the random number.
2024-11-15
Successfully implemented and tested a functional SGX project.

Found a YouTube tutorial that demonstrated the correct setup for an SGX project.
Simplified the enclave functionality by replacing the random number generation with a string manipulation ECALL.
Updated paths to match the project's folder structure, ensuring the enclave file was correctly located.
Verified the enclave's ability to modify a buffer passed from the untrusted application.
2024-11-14
Troubleshooted build and runtime errors in the TestEnclave project.

Corrected the working directory configuration in the Visual Studio project settings.
Addressed missing autogenerated header files by ensuring the sgx_edger8r tool was invoked properly during the build.
Implemented debug output to log the current working directory and identify runtime issues.
2024-11-13
Began setting up a simple SGX project with a TestEnclave.

Created an initial SGX project using the Intel SGX SDK.
Defined a basic ecall_generate_random function to test communication between the enclave and the untrusted application.
Encountered numerous build errors, including missing header files and undefined references.
2024-11-12
Addressed foundational issues and established a functional framework for SGX development.

Resolved enclave build errors by adding a minimal public root ECALL (ecall_dummy).
Successfully created and destroyed an SGX enclave from the untrusted application.
Established communication between the untrusted application and the enclave using a placeholder ECALL.
Validated the build pipeline for both the enclave and the application.
2024-11-08
Built and ran the SampleEnclave project successfully.

Compiled the SampleEnclave project using the SGX plugin in Visual Studio.
Executed the built enclave and verified the expected output.
Confirmed proper SGX functionality and interaction between the application and the enclave.
2024-11-08
Resolved DLL issues and switched to "Simulation" mode.

Identified and imported the required DLLs from the SGX SDK and runtime libraries.
Updated Visual Studio project settings to include the correct paths for the dependencies.
Switched build mode from "Debug" to "Simulation" to address compatibility issues with the SGX plugin.
2024-11-07-cont
Troubleshooting missing DLL errors in Visual Studio with SGX plugin.

Attempted to build SGX-enabled projects in Visual Studio.
Encountered missing DLL issues related to runtime dependencies for the SGX plugin.
Identified the missing files as part of the SGX SDK and runtime.
2024-11-07
Verified SGX activation within QEMU/KVM and installed relevant drivers in Windows.

Confirmed that SGX is properly enabled in the QEMU virtual machine environment.

Installed and configured necessary drivers in a Windows guest environment for compatibility testing.
Validated basic SGX provisioning functionality in the virtualized environment.
2024-11-05
Cross-compiled ioquake3 for Windows.

Installed mingw-w64.
Successfully built Windows executables for both x86 and x86_64 architectures using the make command.
2024-11-04-cont
Built ioquake3 on Linux.

Installed SDL2 development libraries and cloned the ioquake3 repository.
Successfully compiled ioquake3 using the make command.
Verified the build by running the executable on Linux.
Playtested for 3 hours.
2024-11-04
Documenting a week-long break due to illness.

Took a week off from project work to recover from COVID after attending MCM Comic Con.
2024-10-27
Researched SGX support in mainline Linux kernel and QEMU builds.

Investigated configuration requirements for SGX-enabled virtualization.
Explored official documentation and community forums for setup guidelines.
2024-10-26
Debugged SGX provisioning issues and finalized QEMU configurations.

Adjusted QEMU commands to remove provisioning key requirements.
2024-10-25
Built custom QEMU with SGX support enabled.

Compiled QEMU from source with SGX-specific flags.
2024-10-22
Enabled SGX in BIOS and verified functionality.

Configured BIOS settings to enable SGX and Flexible Launch Control.
Confirmed SGX support through kernel logs (dmesg | grep sgx).
2024-10-20
Researched Intel SGX capabilities and compiled custom kernel.

Built Linux kernel 5.13.4 from source to enable SGX virtualization.
Installed SGX DCAP driver for enhanced provisioning support.
2024-10-15

Set up project repository and initial configurations.

Initialized Git repository.
Added .gitignore to exclude unnecessary files.