A Case for Protecting Computer Games With SGX

Erick Bauman
The University of Texas at Dallas
800 W. Campbell RD
Richardson TX. USA
erick.bauman@utdallas.edu

Zhiqiang Lin
The University of Texas at Dallas
800 W. Campbell RD
Richardson TX. USA
zhiqiang.lin@utdallas.edu

ABSTRACT

The integrity and confidentiality of computer games has long been a concern of game developers, both in preventing players from cheating and from obtaining unlicensed copies of the software. Recently, Intel released SGX, which can provide new security guarantees for software developers to achieve an unprecedented level of software integrity and confidentiality. To explore how SGX can protect a computer game in practice, in this paper we make a first step of exploring new ways to protect the integrity and confidentiality of game code and data, and in doing so we have developed a framework and design principles for integrating games with SGX. We have applied our framework to demonstrate how it can be used to protect a real world computer game.

CCS Concepts

•**Security and privacy** → **Software security engineering;**

Keywords

SGX; Application security; Computer game protection; DRM

1. INTRODUCTION

One prominent example of popular entertainment software is computer games, which provide enjoyment for players and a competitive environment in multiplayer games. The computer game industry continues to grow, and it is one of the largest entertainment industries today with a market value in the tens of billions of dollars. Many computer games are designed to allow multiple players to pit their skills against each other. Popular games have millions of players, and competition may be fierce for the top positions. The best players in popular games can even compete for million-dollar prizes. However, even for more average players, there is an incentive to compete and win against opponents. Because of this, some players are motivated to cheat, and with so many players it is difficult for the game developers to police their game effectively. As games have developed, so have the techniques used to cheat in them [8]. The game code can be reverse-engineered to learn an optimal strategy, values can be altered in memory, or automated tools can be used to give the player abilities that would normally be impossible. All of these give the cheater an unfair advantage, ruining the experience and driving away legitimate players. This can lead to a game vendor suffering large financial losses. Therefore, in order to maintain fairness and provide a high quality experience, it is crucial to protect game software against various cheating techniques.

An important aspect of cheat prevention is anti-reverse engineering. A significant application of anti-reverse engineering is in proprietary software protection, which has been a concern ever since software began to be sold. Historically, software developers have designed a wide variety of DRM (Digital Rights Management) [15] techniques to prevent unauthorized redistribution and protect the secrets in software. Without relying on any other support, the best strategy so far is using various obfuscation techniques [7] including control flow flattening, signal-based control flow hiding, and code encryption and packing. Unfortunately, such protection is often quickly circumvented by individuals who subsequently upload the DRM-free executables onto the Internet for illicit download. The result is a constant arms race between developers and crackers as each side attempts to outwit the other.

Recently, Intel released Software Guard Extensions (SGX)—a security extension to the x86 instruction set [4] that has been market available starting with the Skylake CPUs. SGX provides enclaves, regions of encrypted code and data that run in isolation and cannot be viewed by an attacker. This hardware based guarantee provides substantial new opportunities for protecting the confidentiality and integrity of software. The fact that this technology is being produced for commodity hardware means that it may be widely deployed on user machines, and therefore common use of this technology will be possible within a few years of its release.

Considering the great protection potential enabled by SGX, it is not clear how game developers can use it. To bridge this gap, in this paper we make a first step of exploring how we can leverage SGX to protect the secrets inside computer games. Specifically, based on knowledge of game software development and the nature of such software, we formulate a number of design principles for protecting both game integrity and confidentiality, and in doing so design a framework with a set of APIs for game developers to use.

In short, we make the following contributions in this paper. We propose and demonstrate specific applications of SGX on games, some of which apply to software in general. More specifically, we provide a protection model for categorizing game protections and a set of design principles for developing new games with SGX protections in mind (§3). We have designed a prototype framework for integrating SGX with games (§4), and implemented proof-of-concept protections in an existing game (§5). In doing so, we demonstrate the new opportunities SGX offers and outline the path that future work in this area should take.

2. BACKGROUND

Cheat Prevention. Cheating in multiplayer games is a serious concern for game developers because a small minority of cheaters can ruin the game experience for all the players [8]. For games that

SysTEX '16, December 12-16 2016, Trento, Italy

ISBN 978-1-4503-4670-2/16/12…$15.00

DOI: http://dx.doi.org/10.1145/3007788.3007792

rely on a large community for their appeal, such as Massive Multiplayer Online (MMO) or competitive multiplayer games, cheaters can threaten the longevity of a game and cost developers money.

Computer games often use a client/server model, in which multiple player clients connect to a game server, which coordinates, validates, and disseminates game information for the connected clients. Since the server connects multiple clients but does not have to be run on a user's machine, this allows it to be run by a trusted entity and the server itself can attempt to detect cheating attempts by clients. Unfortunately, there is overhead associated with verifying each client's messages, and in games with large numbers of players or messages, the server may become a bottleneck.

DRM. The ability to exactly replicate digital information allows for the easy copying and sharing of applications at little to no cost. This often results in illegal sharing of proprietary games and applications against the wishes of the content providers that own the rights to the content. DRM, or Digital Rights Management, refers to the management and enforcement of rights to digital intellectual property [12]. This involves restricting the ability to access protected content only to those with a valid license for the content [17].

A general model of DRM involves the owner of the content, the license distributor (who may be the same as the content owner), and the end user. The content owner distributes its content in some form of secure container, usually involving some form of encryption. This secure container is delivered to the end user, who obtains a license in order to be able to access the content. With a valid license, they can decrypt and access secure container's content [12].

However, the existing DRM model has a significant weak point in that it requires a trusted component on the user's device that actually verifies their license and allows (or denies) access to the content. While the trusted component is supposed to be tamper-resistant and closed-source in order to frustrate reverse engineering [12], such protection is through obfuscation that merely slows an attacker down. Such an approach does not provide real security guarantees, and even complex DRM may be fairly quickly circumvented by determined attackers [2]. The current approaches lack a way to sufficiently protect the trusted client-side component.

Intel SGX. Intel SGX is, at its core, a set of x86 Instruction Set Architecture (ISA) extensions. There are two new instructions, each of which is divided into multiple leaf functions [3] (18 leaf functions in SGX revision 1 [9] and 6 more in SGX revision 2 [10]). Since using these instructions directly would be difficult, error-prone, and would require a large time investment, most developers will likely end up using a higher level API that abstracts away the low-level details. SGX provides the ability to place code and data in a secure *enclave*, which is an isolated execution environment. SGX hardware, as a part of the CPU, protects the enclave against malicious software, including the operating system, hypervisor, or even low-level firmware code (e.g., SMM).

In particular, when the processor accesses enclave data, it automatically transfers to a new CPU mode, called *enclave mode*. The enclave mode enforces additional hardware checks on each memory access, such that only code inside the enclave can access its own enclave region. The enclave data is stored in a reserved memory region called the Enclave Page Cache (EPC). To defend against known memory attacks such as memory snooping, memory content in the EPC is encrypted by the Memory Encryption Engine (MEE). The memory content in the EPC is decrypted only when entering the CPU, where the code and data are protected by the *enclave mode*, and then re-encrypted when leaving the CPU back to the EPC memory region.

3. OVERVIEW

Scope, Assumptions, and Threat Model. We focus on the protection of network connected, multi user games (or applications) since there is no particular need to prevent cheating in a purely single player game because there is no way for an attacker to negatively affect other players; the only possible negative benefit an attacker could receive is undeserved bragging rights. DRM, however, also applies to single player games. We assume that an attacker may have complete control over all software on a platform except for its attested enclaves, including the application hosting the enclaves. We also assume that an attacker is capable of directly accessing the platform's memory, but not the processor itself. We therefore treat all code and data outside the enclave as potentially modified by an attacker, including network traffic and enclave inputs. However, we do not consider potential attacks on the security of SGX itself.

3.1 Protection Model

We have developed a general protection model for defending games, and have divided protection into two specific categories: integrity (for Cheat Prevention), and confidentiality (for DRM). Cheat prevention ensures the integrity of cheat prevention mechanisms, and DRM is concerned with hiding crucial components.

Integrity. In order to prevent players from cheating, we must prevent them from performing actions or making changes that are not allowed. To this end, we must protect the integrity of certain sections of an application. Since SGX provides guarantees that the code and data in an enclave will not be modified, we can carefully choose certain components to place in enclaves to provide assurances that players cannot make unauthorized changes. We divide protecting integrity into two categories: data integrity and code integrity. These two categories do have some overlap, but this split provides a useful logical separation of the intent behind each kind of protection.

- **Data Integrity**. Certain data, such as player position or score, must be carefully monitored by either a game client or server so that players cannot modify it in invalid ways. Placing this data in an enclave immediately yields the benefit of preventing the simple attack of modifying values with a hex editor, but protections can extend further than that. Since properly protecting data also requires protecting the code that can modify it, this naturally involves code integrity protection as well. If a limited interface to the data can be designed such that only allowed operations can be performed, then we can trust the client machine to make allowed changes to that data.
- **Code Integrity**. While protecting data integrity may naturally lead to protecting the integrity of the code that interacts with it, there may also be cases where the main intention is to protect certain code instead of the data it operates on. Compared to other cases, this is more straightforward, as all that is necessary is to move the code into an enclave and attest that the enclave is unmodified.

Confidentiality. While cheat prevention with SGX does not necessarily require hiding information from the user, DRM requires that the workings of an application are sufficiently hidden and tied to the license authentication mechanism such that it would be very difficult for an attacker to remove or fool the authentication component and derive a version without copy protection. To this end, we aim to enforce DRM by encrypting components that are critical for a program's operation. Since SGX by itself easily provides a trusted component (e.g., for checking a license key), the challenge is to make it difficult for attackers to circumvent that SGX mechanism by encrypting components of the application and having them rely on the enclave to decrypt them. Since these components must be encrypted in order to hide how they work, we consider this protecting the confidentiality of an application. As with integrity, we split this protection into data confidentiality and code confidentiality.

- **Data Confidentiality**. Since an enclave becomes opaque after it is started, any data decrypted inside it remains hidden. If code written to handle the data is also contained within the

enclave, then it is feasible to keep the data self-contained. If the data eventually must be exposed to the user in some form, then an attacker may be able to extract the information from memory unless there is an encrypted path all the way to the output device. However, even if data is eventually exposed in memory, since the key is never exposed, an attacker must first extract the data from memory and then write or extract code that loads that decrypted data into the correct locations. This raises the bar over simply extracting decryption keys.

- **Code Confidentiality**. Code confidentiality poses an interesting challenge, as it requires an enclave be able to dynamically decrypt and execute code that was not originally present in the enclave, and therefore any enclave intending to run encrypted code on an untrusted machine must be self-modifying. The benefits of decrypting code in the enclave are significant. An attacker never has access to even compiled code, and therefore any algorithms hidden in this manner provide true black-box behavior.

3.2 Desired Properties for Protected Content

There are several properties in games we wish to protect. Since some of them may not hold in existing applications, the application needs to be modified in order to allow the content to be moved into an enclave. Specifically, we believe new applications that will support SGX need to take such considerations into account.

- **Isolated**. SGX enclaves have many restrictions on both the data that can be sent to and from them and the actions that can be taken inside them. System calls cannot be made directly inside them, and C++ objects cannot easily be passed across the enclave boundary. In addition, a complicated interface to an enclave opens more potential attacks due to increased complexity. For example, the more function calls that can be made into the enclave, the more opportunities there are for an attacker to call functions in an order unanticipated by a developer. As such, it is best to have a restricted interface between enclave code and the rest of the application. In order to do this, potential enclave code should not rely on external libraries or passed object references. If the code must send or receive objects, the objects must support serialization/deserialization or their contents must be sent across the boundary, which may open new attack vectors. However, this requirement is very challenging to obtain, as many components that we may wish to move to the enclave may be tightly coupled with the rest of the application, and therefore the source may require substantial refactoring.
- **Crucial**. Enclaves have a restricted amount of memory (a limit of 128MB globally provided by current SGX hardware), and therefore it may not be feasible to move all or most of an application into an enclave. While it is possible to move enclave pages to unprotected memory [13], enclaves do not appear to be intended for large amounts of code and data, and too-large enclaves may suffer performance penalties. It is important to find the critical components that the application uses and to limit the size and number of enclaves at once. For the purposes of cheat prevention, it is important to find a minimal set of data and functions that prevent undesired behaviors. For DRM, we must find functionality that is both absolutely necessary for program execution and difficult to reverse-engineer from black-box input and output behavior. If an attacker is able to deduce an algorithm from its behavior, they may be able to reconstruct the hidden component and not have to rely on the enclave.

3.3 Types of Content

We have divided the types of game content we seek to protect into four rough categories. These can be further divided, with some overlaps, into content best used for protecting integrity and content best used for protecting confidentiality. The first two, game state and integrity checks, are useful for protecting integrity, while the last two, initialization data and game logic, are useful for protecting confidentiality. Examples of desired content are shown in Table 1.

	Integrity	**Confidentiality**
Data	Player Position Score	Media Content Configuration Data
Code	Velocity Checks Collision Detection	Algorithms Scripts

Table 1: Our protection model with examples of the kind of content to protect for each category.

- **Game State**. Game state can be thought of as any data that comprises the current status of a game in progress. Examples of this include player position, score, lives, orientation or velocity, as well as the status of elements of the game world, such as map data or inventory items. It is often desirable to prevent changes to this data.
- **Integrity Checks**. Integrity checks comprise the code that performs checks on the constraints on game state data. While game state focused on the integrity of the data, this focuses on the integrity of the code that verifies this data.
- **Initialization Data**. In order to prevent an unauthorized application from starting, it is useful to protect the confidentiality of assets required before a game can begin. This may include media such as sounds, textures, or 3D models, or it may include configuration files.
- **Game Logic**. Another way to prevent an unauthorized application from running is to hide critical logic that cannot be decrypted and run without authorization. Game logic might include game scripts (e.g., it might be possible to place a scripting language interpreter such as Lua's completely inside an enclave) or small but necessary semantically (but not necessarily computationally) complex calculations.

4. DETAILED DESIGN

Cheat prevention and DRM systems are relatively complex in efforts to obfuscate their purpose and prevent analysis. However, we aim to make a clear solution using SGX and therefore not depend on a convoluted design. Establishing security guarantees with a clearly explained straightforward design avoids "security through obscurity" and follows cryptographic best practices.

In order to provide the correct security guarantees, our design must carefully establish a chain of trust that is never broken. Regardless of whether we wish to ensure integrity or confidentiality, the client's machine will need to contact a server controlled by a trusted entity at least once in order to attest to its correctness. After that point, the server can be the authoritative voice to confirm whether the client machine contains trusted enclaves.

We have also created a set of APIs that follow our design. The API is designed to be general enough to work with any implementation on top of the low-level SGX assembly instructions. While our final implementation used the official SGX SDK, other interfaces are possible. In fact, many of the API calls in our abstract framework are in fact aliases of existing SDK functions, and we directly use these functions in our implementation. However, by abstracting away the specific details of an underlying framework, our general design and API should be applicable and adaptable for any interface built upon SGX. In the following, we describe the detailed design of our framework.

4.1 Integrity

Different games have different kinds of information that must be protected in order to prevent cheating. For this reason, the actual code or data that must be protected for the integrity depends on

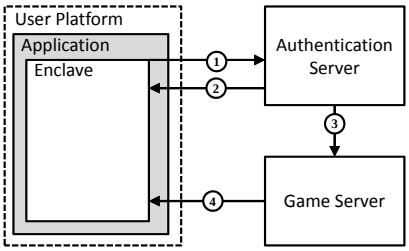

Figure 1: Our design for protecting code and data integrity.

the specific game that we are protecting and therefore is an implementation issue. Once found, this code or data must be moved to a secure enclave. When the user starts the application, the enclave contacts the trusted server and performs remote attestation. After this point, we can trust that the code and data inside it has not been modified. This allows a server to avoid verifying anything more than the integrity and freshness of client messages; any checks on message semantics can be done in the client code since it is trusted and therefore can simplify the server. The specific steps of this process are shown in Figure 1. Next, we go over each step in detail.

- **Step ① : Start Remote Attestation**. First, we establish trust by performing remote attestation on one or more enclaves on the client machine. The enclave contacts an authentication server with the cooperation of the untrusted application. The untrusted component cannot make any changes to any of the messages or attestation will fail, and therefore an attacker can only provide denial of service when the enclave tries to exchange messages with the server. This step corresponds with our `sgx_ra_start` and `sgx_ra_ respond` API calls. RA in this case stands for remote attestation. `sgx_ra_start` initializes the attestation state in the enclave and returns the first message to be sent to the server, while `sgx_ra_respond` handles the first message from the server and returns the enclave's response.
- **Step ② : Verify Enclave**. After the exchange of attestation messages is complete, the server will have the platform's quote for the enclave, allowing it to verify that its contents have not been modified. If the final result meets the server's requirements, the server can send its final verification result back to the enclave. This step corresponds with our `sgx_ra_verify` call. It allows the enclave to know whether the server has verified it.
- **Step ③ : Share Credentials**. If the verification server is not the same as a game server, it may transfer information for the secure channel to one or more other servers that actually handle game data. By using keys derived from the secure session (or fresh keys delivered from the authentication server for each server), each game server can also be assured of the enclave's integrity.
- **Step ④ : Enclave Communicates with Game Server**. When the game communicates with a server, both for initializing a game session and during a game, messages are routed through the enclave for encryption and decryption, and any outgoing messages involving the client's game state either can originate in the enclave or the enclave can perform integrity checks on data coming from untrusted components. The messages sent to the server must either be signed or encrypted, depending on use case, but data verification can all be performed on the client side, substantially lightening the load on game servers by avoiding the need to verify data from all clients connected to the server. This step corresponds with our `sgx_encrypt_message`, `sgx_decrypt_message`, `sgx_sign_message`, and `sgx_verify_message` calls.

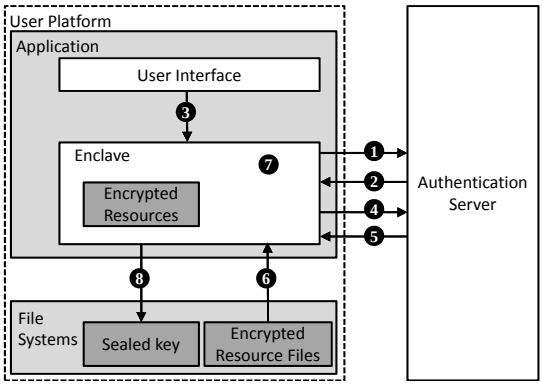

Figure 2: Our design for protecting code and data confidentiality.

4.2 Confidentiality

For hiding secrets in an application, the decision must first be made as to which secrets to hide and therefore, similar to integrity, it is application-specific. However, once the secrets have been identified, they can be encrypted and stored somewhere with the application or removed to be downloaded later. Note that encrypted code or data can be stored in multiple ways. The two that we focus on are either storing the encrypted data in the compiled binary or enclave shared object, or storing the encrypted data in separate files that are loaded in later. Encrypting assets in files allows the data to be allocated dynamically at runtime. Certain kinds of data, such as images or audio, more naturally fit into the external file model, while other kinds, such as code, may make more sense to be decrypted in place. Such decisions therefore depend on the data. In the following, we describe the steps (outlined in Figure 2) for achieving confidentiality. Since the first two steps of establishing the remote attestation are similar to our integrity model, we omit those details here.

- **Step ❸ : Retrieve License Key**. Once a trusted connection has been established, we can verify that the user is authorized to use the software. For games, this likely will involve the user entering a license key when the user opens the game for the first time. License keys can be distributed with physical or digital copies of a game. For example, an installer may display a window asking for the key, or the game may prompt the user for the license key when it first starts.
- **Step ❹ : Send License Key**. The client enclave can send the license key along the encrypted channel to the server. This prevents attackers from extracting license keys from network traffic, although it does not defend from dishonest users distributing license keys. Meanwhile, since license key checking is performed on a trusted server, illicitly redistributed keys can be detected by watching for specific key activations (e.g., to protect against license key reuse, a publisher can simply restrict the number of times a specific key can be reused). This step corresponds to our `sgx_encrypt_license` API, which accepts the plain text of the license key and returns the encrypted message to send to the server. Once the license key is encrypted for the secure channel, the application can send it to the server for validation.
- **Step ❺ : Receive Decryption Key**. Since a vendor can control the number of activations per key, it therefore can determine whether to grant access to the application. Once the server has verified that a user has a valid license, it then sends a decryption key for the encrypted assets across the trusted channel to an authentication server. This step corresponds to our `sgx_receive_decrypt_key` API, which accepts the decryption key encrypted by the secure channel's key.
- **Step ❻ : Retrieve Encrypted Assets**. Once an enclave receives a decryption key for the content, it is then able to de-

crypt all encrypted assets. The enclave may accept encrypted data passed in from external files or use encrypted data contained within it. When receiving external files, the enclave relies on untrusted components of the application, but since all data being read in is encrypted, any attempted corruption of the encrypted contents can be easily detected. The assets are encrypted before the application is distributed in an offline process. This corresponds to our `sgx_cryptstore_encrypt_to_file` and `sgx_cryptstore_encrypt_to_bytes` APIs, which are called by code that may not be present and can be disallowed in the released version of a game if it has no need to re-encrypt the assets. Before the data is encrypted, it is inserted into a `cryptstore` instance with the `sgx_cryptstore_add` call.

- **Step ❼ : Decrypt Assets**. There are two options for how to handle game assets. The first option is to keep the decrypted code or data in the enclave and therefore keep it completely hidden. This is useful for small amounts of critical code or data. However, if the data are, for example, encrypted images, it may not be feasible to keep the entirety of them inside the enclave at once. Therefore, the second option is to immediately pass the decrypted data back out to the main application in main memory. In this case, the assets such as the images themselves would not be the primary secret that would be hidden, but instead the decryption key and loading mechanism. This step corresponds to our `sgx_cryptstore_decrypt_from_file` and `sgx_cryptstore_decrypt_from_bytes` APIs. Using the newly retrieved decryption keys, the assets are decrypted. Whether the data can be retrieved from the enclave is up to the developer. If it can be retrieved, the data can be accessed with our `sgx_cryptstore_get` API.
- **Step ❽ : Seal Decryption Key**. After a game has been authenticated and decryption keys have been delivered, it would be inefficient and inconvenient if the authentication process had to be repeated every time the game was restarted. This implies that the enclave must somehow save its data in a secure manner, and fortunately SGX's sealing mechanism allows for this. For large assets that are normally stored on disk, the enclave can simply seal the decryption key. This can allow for a single-player game that normally does not require any external connection to only require a single connection to authenticate with a server and retrieve the decryption key. Afterwards it can operate autonomously without ever checking in with the server again. Sealing and unsealing the key corresponds to our `sgx_seal_decrypt_key` and `sgx_unseal_decrypt_key` calls. The seal call saves the key to an encrypted file. The call to unseal the key will only succeed if the key has already been sealed in a previous session. However, if it does succeed, all authentication steps (from **Step ❶** to **Step ❺**) can be skipped because the key has already been obtained.

5. CASE STUDY

We have applied our framework to protect the real world game Biniax2. In this section, we describe what we have modified and our experience of protecting this game. We chose this game for our case study because it is a mid-sized (3,540 LOC) open source game written in C, which simplifies the efforts of modifying it to work with SGX.

Objective. While the game supports multiple users playing on a single machine, it does not contain any networking features. Our focus is therefore on facilitating copy-protection mechanisms—preventing the game from being reverse engineered. For this specific example, we decided to encrypt the assets that were used in the game, including images, sounds, and text. Not having access to these assets would make the game completely unplayable, as it comprises most of the player's experience. Each asset has been saved to a cryptstore, each of which is encrypted and written to disk. When the game launches, the contents are read, decrypted inside the enclave, and loaded into their correct data structures.

In a straightforward copy protection scheme, a user would enter their license key when the game is installing, and the server would send back a key to decrypt the assets. In this case, we do not actually provide a real authentication server, and instead have the game behave as if the user entered the correct license key. Therefore, we give the enclave access to the decryption key. During initialization, the enclave uses this key to decrypt each asset as it is loaded. We protect the *confidentiality* of these assets until they are loaded into memory. Since the graphics and sound cannot fully reside in SGX and there is no secure I/O, we are forced to leak such data in memory so that it can be output to the player. However, this simple change raises the bar for an attacker attempting to bypass the enclave, as the bytes would have to be extracted from memory and the asset loading code would have to be modified to directly use the extracted data. This requires less effort from a developer than an attacker.

Modifications. The first challenge in using SGX with an existing application is the fact that it must be ported to SGX by partitioning the application into trusted and untrusted sections, if there is no library OS support. We therefore had to decide what components to place in the enclave. In order to avoid significant alterations and provide an example of an approach requiring minimal developer effort, we used our cryptstore library to encrypt the assets of the application. This lead to a total of 29 encrypted assets, including 923KB of images (e.g., `background0.png` or `font.png`), 160KB of sound effects (e.g., `sfx1.wav`), and 14KB of text (e.g., `help.txt`), comprising a total of over 1MB of protected data.

A comparison between the original game and our SGX port is shown in Table 2, and some additional statistics about SGX-Biniax2 are given in Table 3. Our ported version has 22% more lines of code than the original; this is likely mostly due to the additional code for the enclave functions, as the code added to protect the assets could mostly reuse our framework functions. The binary size increase is likely for the same reason. In addition, due to the fact that we store the assets with our cryptstore API, there is an insignificant size increase of around 1% when they are encrypted. This is caused by the small amount of metadata stored along with the asset.

Performance. In order to evaluate the effects of our modifications on performance, we tested the time it took to initialize the game, measuring the time before and after all assets were loaded. Meanwhile, we measured how much time it took to initialize only the enclave. We started the game 10 times and took the average and standard deviation. Our evaluation machine runs Ubuntu 14.04.4 LTS with 64GB of memory and an Intel i7-6700 Skylake CPU running at 3.40Ghz. The results are shown in Table 2 and Table 3.

Our results may at first appear rather startling, as a 72% overhead is rather significant. However, the start up time in either case is still far faster than a human can detect, and there is no human-perceptible difference between the two. While this overhead may increase for larger games, a slightly longer loading time may be tolerable for the benefit the encryption provides. In addition, note that all the assets are loaded only during initialization, and all code after that point is identical to that from the original game, meaning there is no overhead after the game starts.

One interesting discovery is how much of the initialization comes solely from the enclave initialization itself; it comprises roughly half of the additional overhead over the baseline, with asset decryption likely making up the other half. However, enclave initialization should be a fixed overhead and will likely not be performed often, so this is also not a concern.

Metric	Biniax2	SGX-Biniax2	Percent Increase
Lines of Code	3540	4326	22.20%
Initialization Time (ms)	141.58±4.23	243.59±4.11	72.05%
Binary Size (bytes)	35038	38353	9.46%
Asset Size (bytes)	1084486	1097259	1.18%

Table 2: Comparison of several metrics between the original Biniax2 game and our modified version that we ported to SGX.

Metric	Value
Lines of Code in Enclave	580
Enclave Size (bytes)	100425
Enclave Initialization (ms)	53.22±4.21
Assets Encrypted	29

Table 3: Statistics for our modified SGX-Biniax2.

Future Work. This case study is a simple yet interesting look at copy protection for an application, but it does not provide all the guarantees that are possible when using SGX. We only encrypted assets that are eventually output to the player and therefore must leave the enclave memory; there is no secure I/O channel, meaning that all images and sounds must be leaked to attackers, and in the case of cheat prevention attackers can fake user inputs. It would benefit SGX greatly to be somehow combined with secure I/O.

However, certain assets may never need to leave the enclave (e.g., internal data structures). Such assets would have to be completely reconstructed (solely using black-box inputs/outputs from the enclave) by an attacker intending to circumvent copy protection. For future studies, we aim to provide these stronger guarantees by encrypting data that never needs to leave the enclave. In addition, code for a built-in scripting language could also be encrypted, potentially making all in-game scripts completely hidden from an attacker.

We also did not demonstrate all steps in our framework. In order to more thoroughly demonstrate our framework, we could set up a mock authentication server and require a license key to be entered when the game first starts. Then, we could also demonstrate sealing and unsealing by having the game unseal the decryption key for all subsequent launches. In addition, we did not provide a case study for cheat prevention, as many games with network play are complicated, and partitioning the application into secure and non-secure sections is nontrivial. However, we also intend to look more into cheat prevention in the future.

It will also be important to examine the security of an enclave implementation itself. E.g., if there are many enclave functions, they could be sensitive to being called out of order and may put the enclave at risk of compromising integrity or confidentiality. This adversary model gives great strength to an attacker, and we must consider the ways protections might be compromised.

6. RELATED WORK

Game Protection. Hoglund and McGraw [8] described common game cheating techniques such as memory editing [18], code injection, and network traffic forgery. CheatEngine [1] is an open source engine for cheating single player games that uses value scanning to identify the data of interest while the game runs. However, as acknowledged by the author, such an approach does not work with highly dynamic online games. Kartograph [6] is a state-of-the-art tool for hacking games and protecting games against maps hacks. More broadly, there are also various obfuscation strategies [7] to protect software assets, but they cannot provide security guarantees. In contrast, the use of SGX gives us strong guarantees, as shown in this paper.

SGX and Its Applications. Since SGX holds great potential to solve many challenging security problems [4], a number of efforts have started to explore SGX in various applications and also build platforms for SGX research (e.g.,the OpenSGX [11] project). In particular, with a shielded execution on Intel SGX, Haven [5] protects the code and data in the cloud. VC3 [16] demonstrated privacy-aware data analytics in the cloud. OpenSGX [11] provides an open source platform with fully functional and instruction compatible emulator to hasten the development of TEE based applications. Most recently, Ohrimenko et al. [14] presented a number of privacy preserving multi-party machine learning algorithms running in SGX machines for cloud users.

7. CONCLUSION

The challenges of preventing cheating in games or illegal copying of games are well-known. However, it has proved challenging to provided a trusted component to enable protections against cheating and illegal copying due to the fact that content creators do not have control over an end-user's platform. With the introduction of SGX, this becomes feasible. In this paper, we presented a general framework for protecting game integrity and confidentiality. We showed the security guarantees that SGX provides and how they can be applied to protecting games, and we demonstrated that our prototype framework can be integrated into existing games.

Acknowledgement

We thank the anonymous reviewers for their valuable comments. This research was supported in part by NSF awards CNS-1564112 and CNS-1629951. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the NSF.

8. REFERENCES

[1] Cheat engine. http://www.cheatengine.org/.
[2] Ubisoft's controversial 'always on' pc drm hacked. http://www.cnet.com/news/ubisofts-controversial-always-on-pc-drm-hacked/#!,
[3] Intel software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, Oct. 2014.
[4] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, 2013.
[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, 2014.
[6] E. Bursztein, M. Hamburg, J. Lagarenn, and D. Boneh. Openconflict: Preventing real time map hacks in online games. In *IEEE SP*, 2011.
[7] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE TSE*, 2002.
[8] G. Hoglund and G. McGraw. *Exploiting online games: cheating massively distributed systems*. Addison-Wesley Professional, first edition, 2007.
[9] Intel. Intel Software Guard Extensions Programming Reference (rev1), Sept. 2013. 329298-001US.
[10] Intel. Intel Software Guard Extensions Programming Reference (rev2), Oct. 2014. 329298-002US.
[11] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. Opensgx: An open platform for sgx research. In NDSS, 2016.
[12] W. Ku and C.-H. Chi. Survey on the technological aspects of digital rights management. In *Information Security*, pages 391–403. Springer, 2004.
[13] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*. ACM, 2013.
[14] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.
[15] W. Rosenblatt, S. Mooney, and W. Trippe. *Digital rights management: business and technology*. John Wiley & Sons, Inc., 2001.
[16] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *IEEE SP*, 2015.
[17] S. Subramanya and B. K. Yi. Digital rights management. *Potentials, IEEE*, 25(2):31–34, 2006.
[18] D. Urbina, Y. Gu, J. Caballero, and Z. Lin. SigPath: A Memory Graph Based Approach for Program Data Introspection and Modification. *ESORICS*, 2014