Preventing Score Forgery with Input Verification

Alec Thomson Alan Huang Casey McNamara Chris Billie

Abstract

Recently, single-player video games with online leader-boards have become popular, both on browsers and mobile devices. Unfortunately, these online leader-boards are often very lax about security and it remains relatively simple for a dishonest player to post a forged score. We present a basic defense against score forgery that makes use of an input log and a trusted verifier to verify that a player actually earned, or was entirely capable of earning, a claimed score.

1 Introduction

As games on mobile and browser platforms become more popular, social services such as leader-boards and achievements are becoming more common even among single-player games. Popular services such as the *iOS Game Center* and the *Kongregate* browser portal provide developers with simple APIs for posting scores and maintaining leader-boards. Unfortunately, the security of these services only extends to authenticating the player and provides little if any protection from score forgery. As a result, the leaderboards of popular games are often littered with forged scores. Since some developers use competitive play on the leaderboards as the primary draw of their games, these forgeries can have a negative impact on both the popularity and the commercial success of their games.

Preventing forged scores is not easy. An obvious solution would be to sign each score with a key known only to the game code, but the problem of securely storing a secret on an untrusted device makes this solution impractical. Even if the secret is protected by trusted hardware such as a TPM, the player could potentially hi-jack communication between the game and the TPM, tricking the TPM into signing a forged score. Unless the entire game is running on trusted hardware (which would be both costly and slow), the possibility of forgeries remains.

To prevent score forgery, we rely on a simple abstraction. We assume that a player can be trusted to have "earned" a score if they are capable of producing a sequence of inputs that produces the same score when run on a trusted version of the game's code. When the game posts a score to the leader-board, it is required to provide both the claimed score and a "proof" that the claim is legitimate.

To implement this abstraction, we built a system for the *Flixel* game engine that records a players inputs and produces a log that can later be verified by a trusted version of the game. The design of this system is outlined in Section 2. To test both the performance and usability of the system, we modified an existing game to make use of the system. The results of this modification are outlined in Section 3. Section 4 explores the limitations of the system and Section 6 concludes.

2 Design

The design of the score verification system is split into two main parts, a client library for recording games and communicating with a server, and the server that verifies the recordings, both outlined below. A key design decision for the system was to allow it to be as general as possible so different recordings and verifiers can be used on a per-game basis.

2.1 Client Library

We implemented a client library for recording and verifying scores for games made with the popular *Flixel* game engine. Flixel is a wrapper around common game functions and class structures written in Actionscript 3.0 for use with developing Flash games. As a debugging feature, Flixel already provides a facility for recording inputs and playing them back with high fidelity. We modified and extended this system to allow for a developer to record and verify scores with minimal changes to game code.

The recording system in Flixel works by measuring the input state of the game each frame and recording a log of the state if it changed since the previous recording. Entries in the log end up looking like the following:

```
<frame #, key state, mouse state>
```

Once recorded, Flixel can use this log to modify game-visible inputs in-between frames and effectively replay a player's play session.

For our client library, we created a new class called "FlxScoreVerifier" that performs recording, verification, and communication with the server. At the start of the game, the game calls "FlxScoreVerifier.init()" which will either start recording the current play-through (on the client's machine) or start verifying a provided set of scores (on the verification server). When a score is obtained, the game calls "FlxScoreVerifier.postScore(scoreName, scoreValue)" which will then either send a claimed score and a recorded input log to the score server (if we're recording scores on the client's machine) or will mark a score as verified (if we're verifying scores on the server). This system was chosen to be symmetric so that the same game executable could function as recorder and verifier.

Since many games record scores on a per-level basis, we also introduced functions to allow a game to begin recording or verifying at a specific state in the game. When a game wishes to record a per-level play-through, it tells the Verifier Library to not begin recording immediately on startup. Instead, it will call "FlxScoreVerifier.initAtState(stateName, stateParameter)" when it reaches the desired level state. The "stateParameter" field is used to specify a specific field name and value in the state object if the same state is used to represent multiple levels. A common implementation in Flixel is to have a single "PlayState" represent all the levels while a "levelIndex" parameter specifies the exact level. When scores are posted, the library will provide additional information in the score log so the verifier can perform verification on the correct state with the correct parameter value.

One modification we had to make to the Flixel replay system was to allow the game to include frames in the log even when no input change had occurred. This allowed us to mark the point in the log where the score had been posted. Prior to implementing this change, the replay log would often end before the score had been posted because of a delay between the player's last input and acquisition of the score, an effect that would

confuse the verification server.

2.2 Verification Server

The verification server is a basic server structure for performing score verification, storage, and handling for scores on a per-game basis. The server listens on a certain port for connections. When a game connects to the server, it sends an xml digest with a set of claimed scores and a log of the player's inputs. This digest can be customized for different games, the only requirement being that it correctly identifies the game to post the score to. Once the server has determined that the score report is well-formed and corresponds to an existing game, it spawns a new game-specific handler to manage the report. This handler can then record the score, perform verification, or store the log for auditing at a later date. We designed the system to be fairly general so developers could opt to use custom verifiers or only perform verification under certain circumstances (such as if the score is in the top 10 or if the score is later flagged as suspicious).

Our Flixel specific verifier works by spawning a new Flash Player instance (in our case, inside a firefox process) with a trusted version of the game code and a local server to listen for verification from the trusted code (since Flash cannot easily write output to a file or standard out). The recorded log and the port number of the local server are both passed to the trusted game as arguments in the game's URI query string. The game then operates as normally, replacing the "recordScore" function with a "playReplay" function. When a score is achieved, the game will contact the local server to inform it that the score was verified. When the end of the replay is reached, the game will send a "ReplayEnd" message to the local server to inform it that verification is over.

Meanwhile, the local server will listen for score verification until either all the claimed scores are verified, the game sends a "ReplayEnd" message, or the game exceeds a certain amount of time during verification. Afterwards, the local server will either accept or reject the claimed scores and kill the spawned process. To calculate the timeout for a given score, our verifier uses the frame number of the last frame in the log as a basis for roughly how long the verification should take. The function then becomes:

```
timeout = 20 + last_frame*4/fps
```

This function is to ensure that the verifier is given enough time to run while allowing for overhead in starting the process and possible delays during a heavy server load.

3 Evaluation

In evaluating our system, we wanted to explore both the performance drawbacks of using the system and the ease of integrating the system with an existing game.

3.1 Size Analysis

As the logs are sent over the network, the size of the logs is important. Since the log only records frames where the input state changes, the rate of log growth depends on how often the player interacts with the game. For a keyboard only game, the player might only change the keyboard state twice every second on average. For a mouse game, the player might change the position of the mouse quite often, possibly as much as 30 times per second (the standard flash frame

rate). Each recorded log entry generally has 5-10 bytes. If the player is making only two input changes per second, the log will grow at a rate of approximately 72kB/hour, while if the player is making 30 input changes per second, the log will grow at a rate of approximately 1MB/hour. In both cases, we consider this size to be fairly manageable. One possible optimization exists if the game only cares about mouse position when the mouse is *clicked*. In this case, the log would record much fewer input changes and the size would grow at a considerably slower rate. In the worst case, pieces of the log could be sent to the server at regular checkpoints to avoid sending one giant log all at once at the end of the game.

3.2 Time Performance

On the server side, each verification takes roughly the same amount of time that it took to play the game. While this might be fine for a short arcade game with little computational overhead, the server might have trouble verifying every score for a large popular game. Doing so would require dedicated infrastructure similar to that used by MMORPGs which generally do most of their computation on the server-side. To avoid this problem, the server could either perform verification only for the highest scores or when a posted score has been marked as suspicious. To avoid unnecessarily transporting logs over the network, the client could refuse to send a score that doesn't exceed the top score already posted to the server for this player. In general, we believe it is possible to run verification efficiently while still avoiding pollution of the leader-board.

3.3 Integration of System with Existing Game

To test the ease of integrating our system with an existing game, we modified a Flixel-based game owned by one of the authors to use the system to post scores to the server. The game we chose was the same one that was easily exploitable during our analysis of *Kongregate*'s security.

The integration was very straightforward and required changes to about three lines of code. Since the game records scores on a per-level basis, we used the state specific version of the library calls with the additional parameter representing the index of the current level. At the start of the level, the game begins recording inputs. When the level is completed, the game posts scores to the server. If the level is reset, the game will also reset the recording.

4 Limitations and Drawbacks

4.1 Imperfect Replay Fidelity

One potential drawback of the system as it exists is that while Flixel's system provides fairly high fidelity playbacks, it does not guarantee perfect fidelity, primarily because of non-deterministic frame delays. While this will generally not be a problem for most games, some games might require perfect fidelity replays to avoid false positives and negatives. For these games, we offer some potential solutions.

One solution is to modify the recording system to also record the elapsed time in between frames (often referred to as the "deltaTime") for each frame. The verifier can then replace the actual verification deltaTime with the time recorded in the log (keeping it within reasonable bounds to avoid abuse) and effectively sim-

ulate the non-determinism of the original play-through. This solution is not completely ideal as the game recording will grow proportionally to the number of frames rather than the number of input state changes, which can be significantly larger and slower.

Another solution is to "quantize" the deltaTime, i.e. force the deltaTime to a certain exact value each frame which the verifier can then reproduce. This solution is also not ideal because it can effect the usability and playability of a game. A forcibly fixed deltaTime will often make games feel choppy and less smooth. A developer will have to analyze the costs and benefits of perfect replay fidelity versus playability if considering this option.

4.2 Automated Play-throughs

One aspect of cheating that our system does not address is the use of cheating software to allow an AI to play the game rather than the player. For games that focus heavily on reflexes, automated play-throughs can be a common form of cheating. Since our system assumes that a score was "earned" if the player can recite inputs that will produce the same score, automated AI players will be able to produce scores that appear to be legitimate. Preventing automated play-throughs is actually difficult since even with preventative software installed on the player's machine, there is nothing to stop the player from having a camera observe the screen while a robotic arm provides inputs.

In online "speedrun" leagues, players will avoid this problem by recording both a view of the game on their screen and a view of their hands on the input device, relying on observers to verify that nothing fishy is going on. For the purposes of this paper, we chose to ignore this problem. Our primary concern was with online leader-boards becoming polluted by scores that are often impossible to achieve given the constraints of the game. By a certain logic, even scores produced by an automated play-through were legitimately "earned", albeit by a non-human player.

4.3 Abuse of Random Seeds

A possibility with the current design is that an attacker could choose to play a game with the same random seed multiple times. By effectively removing randomness from a game, the attacker could more easily obtain significant scores. As long as the random seed is chosen by the client's game, this vulnerability remains. One possible solution is to have the server provide random seeds at the start of a game. The server could either store seeds for a given game instance or sign seeds with its private key. When the score is returned, the server would have to verify that it produced the random seed in the log. To avoid replay attacks, the server will also need to record a timestamp or require that a given seed be used by a game only once.

5 Related Work

With *TrInc*, Levin et al. explored using trusted hardware to prevent cheating in peer-to-peer competitive games [3]. While TrInc could possibly be applied to prevent score forgery in certain situations, it would be very difficult to adapt to existing games and leader-board services because of the trusted hardware requirement and the difficulty in designing attestations adapted for individual games.

Bursztein et al. [1] looked at other forms of cheating in multiplayer competitive games and

built a system that hides information from untrusted clients. For our project, we chose to focus on single-player browser and mobile games where there is a single client performing game computations, so the work is not directly applicable.

Finally, the work done by Haeberlen et al. using accountable virtual machines and tamper evident logs to present evidence of cheating software[2] was one of the primary inspirations for this project. We adapted the idea of a trusted version of the game's code from the idea of an accountable virtual machine and chose to trust any client that can produce a legitimate log, avoiding the need for tamper-evident logs. Since AVMs were primarily designed to detect when cheating software has been installed, they can be used to provide further security against cheating in conjunction with our system.

6 Conclusions

We explored the problem of score forgery among single-player video games that communicate with leader-board services. We found that existing leader-board services provide little to no protection from forgeries and that it is difficult to adapt existing security measures (cryptography, trusted hardware) to prevent forgeries. As a solution to the problem, we implemented an engine-level system that records player inputs and uses a trusted version of the game's code to verify that those inputs produce the claimed scores. This system appears to perform well and is easy to adapt to existing games. For most games, it provides a high level of security against forgeries while for some games (turn-based puzzle games), it provides perfect security against forgeries. We believe this system can already be applied to existing games and hope it is useful in practice.

References

[1] Elie Bursztein, Jocelyn Lagarenne, Mike Hamburg, and Dan Boneh. Openconflict preventing real time map hacks in online games. In *Security and Privacy(S&P)*, May 2011.

[2] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, October 2010.

[3] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, April 2009.