6.857 Project Paper

Charles Wang, Philip Tegmark

***Abstract*—Speedruns are an increasingly common method of competition where competitors attempt to complete a video game in the fastest time possible. As with any competition, cheating is an issue. Here we discuss a number of speedrun cheating methods, such as splicing, playing prerecorded inputs, and game code modification, and propose two countermeasures against some of those methods. The first countermeasure is a method of verifying that livestreamed speedruns of 3D games are not spliced. The second countermeasure is a way to make it harder to cheat using prerecorded inputs, by recording a player's hands during their speedrun. Limitations of both of these countermeasures are also discussed.**

I. Introduction

Typically, speedruns are verified by the runner—the player who is doing the speedrun—submitting a screen capture video of them speedrunning the game in question. These videos can be faked and manipulated in a number of ways, and cheating in speedruns has been documented across games for more than a decade.

Current methods for determining if a speedrun is fake are ad hoc, and require specific mistakes to be made by the forger of the speedrun. There have been proposals for stricter requirements, such as requiring inputting randomly generated button inputs during down time and submitting videos of the recordings of the player's hands. These proposals have not been adopted due to feelings that cheating in speedrunning is not a big enough issue to justify such invasive measures. However, as speedrunning grows in popularity, preventing cheating is becoming a more pressing issue as it gets harder for the community to self-police.

Achieving a fully computationally secure solution to preventing speedrunning in cheating is impossible. Without any in-person verification, all that a runner can be asked to supply is video streams. If one permits arbitrary video editing capabilities, then the runner could always shim the game controller to output computer generated controller inputs and then edit any video or audio feeds so that the recorded behavior of the runner matches the inputs sent to the game. Therefore, the countermeasures discussed in this paper will be more limited in scope, relying on assumptions limiting the capability of the runner to edit video and reverse-engineer code.

An important consideration when designing countermeasures to cheats in speedrunning is intrusiveness. A primary concern expressed by runners is that security measures will increase the barriers to entry and impact the amateur nature of the sport. Therefore, any additional setup on the part of the runner, such as requiring a live internet connection or requiring the runner to set up a camera, should be minimized.

Since we were unable to create a system with adequate security that does not have any such intrusiveness, we will present an approach that requires no filming but requires live gameplay and developer support and an approach that requires filming but does not require liveness or any special consideration on the part of the developer.

The former approach is specific to 3D games and defends against splicing attacks specifically. It works by having the game re-texture objects upon request in a way that attests that a livestreamed speedrun of a 3D game is in fact being played live, and is therefore not a recording. This is important because if a speedrun is not a recording, then that speedrun cannot be a spliced recording, and therefore this approach gives us a way to certify that a speedrun is free of splicing.

The latter approach consists of requiring the player's hands to be filmed and for the player inputs to be recorded and submitted along with the gameplay video. The recorded inputs can be compared against the gameplay footage and the video for verification.

We will then discuss under which circumstances these schemes will be effective and potential attacks on them.

II. Forgery tactics

There are a number of types of methods for forging a speedrun. Of the following, the most common is splicing, but the other methods are also common [6].

A. Splicing

Splicing consists of stitching together segments of recordings of multiple playthroughs of the game to create a gameplay video of a time that is faster than would be achievable by the runner otherwise. Splicing allows the runner to retry sections at will instead of having to have a good time on every segment of the game all in a single attempt.

B. Tool Assisted Cheating

An legitimate category of competition for deterministic games is the Tool Assisted Speedrun (TAS), where a sequence of inputs to the game is designed frame by frame to be played by a machine into the game. These runs are compared against each other, separately from the runs by human players. TASes allows for frame perfect tricks and performing reliably feats that are impossible for a human runner. A method of cheating is to take a TAS and attempt to pass it off as a normal run, which is known as 'TASbotting'. If the inputs sent to the game are constructed by stitching toghether inputs recorded from legitimate runs, this can be nearly impossible to detect when examining only a recording of the gameplay footage.

C. Game code modification

Another approach to cheating in a speedrun is to modify the code of the game. One example would be to modify the settings of a car so that it drove slightly faster. If the game is non-deterministic, another approach would be to alter the random number generation code of the game so that better events are more likely to happen.

When considering adding code to games for the purpose of verifying a speedrun, another game mod consideration is the potential for reverse engineering and tampering with the verification code. This will be presumed to be difficult and countering this possibility will not be considered here.

III. Current Detection Methods

Currently, there are no computational security measures preventing speedrun forgery. Fakes have been detected through inspection of videos for mistakes by the forger, such as jumps in the video or audio and analyzing the video for consistency with known properties of the game. A splice was detected by counting frames of animation lengths and comparing with the known number frames that the animation lasts. A game code modification was detected by noting that a specific car was emitting smoke effects that should not have happened without game code modification. There was also a recorded instance of detection of TASbotting by noting that a player's hands in a livestream did not match up with the sent video [6]. Another case of game code modification was detected by performing statistical analysis of the recorded gameplay to show that the probability of the quality of the drops matching that in the recorded gameplay was less than 10^{-26} and thus could have only a negligible probability of having been achieved legitimately [4].

These detection methods are all community based and there is no common system for finding cheats. There are no security measures imposed on the runners other than submitting a video of gameplay.

IV. Modeling

Here we will be considering the case where there is a runner seeking to achieve a verified run with the desired time and the verifier attempting to distinguish between real and fake runs. When a game is played, the player is generating a sequence of inputs. The game inputs could either be the sequence of button presses for a console game controlled by a game controller or a sequence of key presses and mouse movements for a PC game controlled by keyboard and mouse. In each of these cases, the inputs can be represented by a sequence, with each element of the sequence corresponding to the state of the controller at a single frame.

The output of the game is a sequence of video frames. In the case of a non-livestreamed game, the input sequence and output sequence can be lumped into single objects, giving the game as a (possibly deterministic) function taking the inputs k to the output video $G(v)$.

Another case is that of a livestreamed game. A runner livestreams a speedrun of a game to an audience via a streaming service (e.g. Twitch.tv). The livestream has some known and fixed stream delay D. That is, anyone in the audience will see the events of the Stream unfolding an amount of time D after they occur on the runner's computer (stream delays are a common practice in livestreaming done to prevent a the livestream from buffering). Requests can be sent to the game by the verifier.

V. Difficulty Assumptions

For the case of a splicing attack, we will be considering the case where the player has access to an large number of videos of legitimate gameplay and decides to attempt to forge a speedrun with a faster time using the aforementioned videos. The 3D live verification system is designed for a slightly more powerful attacker that is able to access information in the game's code but unable to re-implement sections of the game's code, such as the rendering engine in particular.

We will additionally be making assumptions about the difficulty of editing videos. Due to the fact that game outputs are an especially regular type of video, editing such videos are not necessarily always difficult. For example, consider a 2D game with a low pixel count, sprites, and no anti-aliasing. Under these conditions, even without any machine learning, such things as the

foreground and background can be perfectly segmented using simple pattern matching algorithms. Due to this reason, we will be focusing on three dimensional video editing problems, such as editing videos of 3D games and videos of real objects, such as the player's hands.

In the case of the 3D games, we will be assuming that it is impossible for the player to segment between foreground and background and determine the pose of a random object with sufficient precision to be able to forge textures within the the delay time D.

In the case of handcams, the assumption is that creating a forged video of the player's hands, h, corresponding to a sequence of inputs k other than the one that was played passing inspection. This is not quite realistic, as it could be possible to state that a key press occurred a frame or two off of the original video and successfully pass it off. However it is reasonable to assume that attempting to play to match a given sequence of keystrokes to within the fudge factor and small changes to the keystrokes will not be able to change the time by enough due to the fact that the later part of the video is still unchanged.

There are machine learning techniques that may be able to solve this problem with an accuracy sufficient to defeat our scheme. More specifically, there are existing methods for determining the orientation of an object [1] [5], segmenting foreground from background [3], and inferring the lighting on a surface [2]. However, it is reasonable to assume that potential speedrun forgers will be unable to access the resources and expertise required to generate a fake in this manner.

VI. 3D Live Verification System

A. System Overview

In this section we will deal with how to verify that a livestreamed speedrun of a 3D game is not a spliced recording. We define a 3D (3-dimensional) game as a game in which the in-game world is 3-dimensional, and most of the objects in the world of the game are able to rotate and translate in three dimensions relative to the player's in-game view. For example, a game like *Portal* (see figure 1) is a 3D game because the in-game world is 3-dimensional; and because as the player character moves around in the game, or as visible objects move around in the game, the player will see these visible objects move and rotate relative to the screen.

Some well-known examples of 3D games are *Portal*, *Mario Kart*, and *Dark Souls*.

We propose a 3D Live Verification System, which (provided that our hardness assumption (see below) is not broken) can verify that a livestreamed speedrun of a 3D game is in fact being played live, and is therefore not a recording. This is important because if a speedrun is not a recording, then that speedrun cannot be a spliced recording.

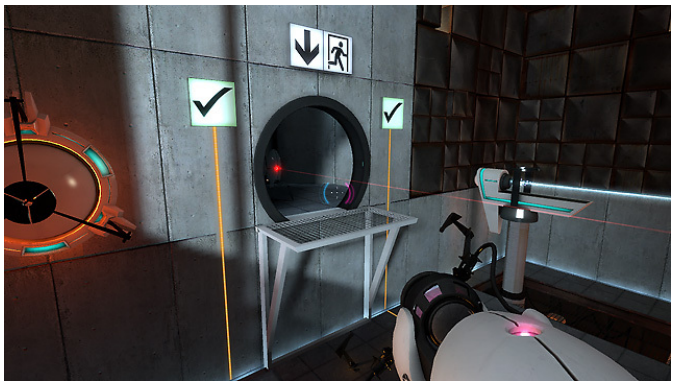

Fig. 1: A screenshot from *Portal*, an example of a 3D game.

In order to implement this system, we need to introduce into our model a trusted party called the verifier, who watches the speedrun as it is livestreamed. Over the course of the speedrun, the verifier sends a number of "retexturing requests" to the runner's copy of the game, in order to attest the speedrun's liveness. We also require that the game being played is built by its developers to include a "speedrunning mode" that, when turned on, allows the game to properly respond to these re-texturing requests.

In order to verify the speedrun's liveness, the verifier sends retexturing requests to the runner's copy of the game at random semi-regular intervals (e.g. 5-20 sec apart). For each retexturing request, the verifier randomly chooses a time T in the near future, and randomly chooses a point P on the runner's game screen. T and P are then sent to the runner's copy of the game as a retexturing request. At time T (as measured by the runner's computer's clock), the runner's copy of the game must do the following:

1) Undo the effects of the previous retexturing request, if there was one. (This is done primarily for cosmetic reasons—if too many objects are retextured at once, the game will likely start to look ugly and/or disorientingly different from how it usually looks.)
2) Find whatever opaque in-game object O is visible at point P on the runner's game screen.
3) Change object O's texture (the way that the object's surface looks) from its original texture t_1 to a noticeably different texture t_2.

For an example of what this might look like in practice, see Figure 2.

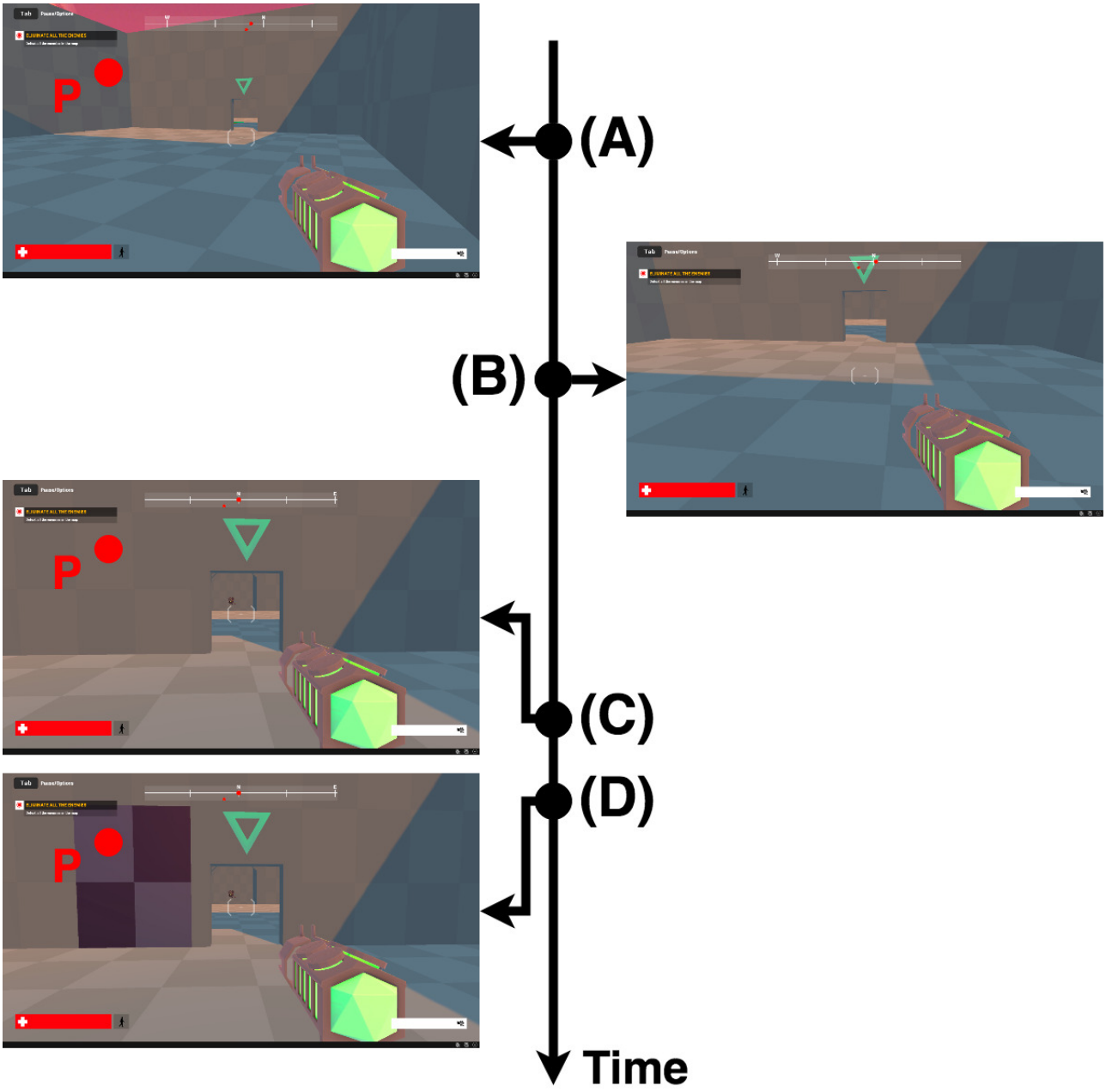

Fig. 2: The game screen of a runner playing a 3D game using our system, as the game responds to a retexturing request sent by the verifier. Point P, although not actually highlighted on the runner's screen, is still shown here for clarity. **A)** The verifier chooses P and T , and sends them to the runner's game. **B)** The runner's game receives P and T. **C)** Time T arrives. The game determines onto which object O point P falls (in this case O is a piece of the wall), and **D)** The game retextures object O.

It is important that the clocks used by the runner's computer, the verifier's computer, and the streaming service must all be in sync, and no two of these clocks should differ by more than some pre-determined length of time δ_{clock} (e.g. 1 second). Note that this kind of synchronization is already implemented in the popular *LiveSplit* program. Also, note that retexturing must be guaranteed to take no longer than some amount of time $\delta_{retexture}$ to occur.

If these two assumptions about time hold true, then the verifier can expect to see the result of their retexturing request sometime in the time interval $[(T + D) - \delta_{clock}, (T + D) + \delta_{clock} + \delta_{retexture}]$, as measured from their computer's clock. If the verifier does in fact see the correct retexturing occur during this interval (and the retexturing looks like it's supposed to), then they can be presume that the speedrun was in fact happening live at time T. But if the verifier does not see the correct retexturing occur during this interval, then they can not affirm the liveness of the speedrun at time T.

Retexturing requests should occur frequently enough that at least a sizeable number N (e.g. 20) of them occur over the course of the speedrun. If for all retexturings, the verifier sees the correct retexturing occur during the correct time interval, then they verify that the speedrun was in fact live and not spliced. But if at least one retexturing does not happen or does not look correct, then the verifier will reject the run.

There are also some additional caveats to mention regarding textures t_1 and t_2. t_1 and t_2 should be different enough that an experienced player (i.e. someone who has played the game before, and who knows it well enough to understand more-or-less what is happening in the speedrun) can always or almost always definitively tell which of these two textures object O is using. However, t_1 and t_2 should also not be too different, otherwise transitioning between them could be distracting to the runner.

It is also important that an experienced player can correctly predict what t_2 should be. That is, given a retexturing request with some on-screen point P and time T, an experienced player should be able to pause the speedrun livestream right before the retexturing occurs, figure out what object O is, and then correctly determine what texture t_2 it will be retextured to.

B. Security Analysis

If the runner was livestreaming a recorded speedrun, however, they would most likely be unable to include the requested retexturing in the stream, at which point the verifier would know that the speedrun was recorded. This is because, to pass off a recording as a live speedrun to this system, the runner must include the correct retexturings in their recording at the correct times. This would either need to be done in advance, which we argue is extremely unlikely to succeed, or would need to be done in real time, which breaks our hardness assumption. Therefore, provided that our hardness assumption holds, a runner livestreaming a video recording would be unable to comply with retexturing requests, and would fail to pass our system's test of liveness.

If the runner wants to try to include the retexturings in their recording in advance, then for every retexturing request that the verifier will send, the runner must correctly guess the object O that gets retextured, and guess the time T when the retexturing takes place. They need to guess T closely enough that the verifier still sees the retexturing occurring in the interval $[(T+D)-\delta_{clock}, (T+D)+\delta_{clock}+\delta_{retexture}]$. Given that there will be at least a sizeable number N retexturings, and that the probability of correctly guessing O and T is at least moderately unlikely, the probability therefore that the runner will be able to correctly include all retexturings in the entire speedrun is negligible. Note that N can and should be made large enough to guarantee that the probability of a runner correctly guessing all re-texturings is negligible.

As for the runner editing their recording in response to retexturing requests during the livestream, we make a hardness assumption that this cannot be done. This hardness assumption is guaranteed if: for at least one frame in the streamed speedrun video, the runner is not able to convincingly edit that frame to reflect the most recent retexturing request within an amount of time $T + D + 2 * \delta_{clock} + \delta_{retexture} - T_r$, where T_r is the time when the runner's computer received said retexturing request (as measured by its clock). If this second assumption is guaranteed, then at least one frame in the speedrun recording will not be edited to look correct in time, and the speedrun will fail to be verified as taking place in real time.

Anyway, assuming that the hardness assumption holds, the runner will be unable to fool our system by editing a recording in real time in response to retexturing requests.

Given that guessing the retexturings in advance and editing the recording in real time don't work, the runner could also try doing some combination of both, but we fail to see how doing so would significantly increase the runner's odds of successfully fooling our system.

VII. Handcams

Another method for ensuring speedrun veracity for deterministic games is through filming the runner's hands.

A deterministic game is a game where entering the same inputs into the game twice will give the same outputs. This will defend against TASbotting, game code modification, and splicing. And unlike the 3D Live Verification System, filming the runner's hands can provides verification of a speedrun that is not happening live. It does however require more intrusive setup on the part of the runner.

The hand-filming protocol is as follows. When the runner plays a speedrun of a game, they record not just a screen-cap video of the speedrun, but also a video of their hands and a recording of all inputs that they make into the game (typically these inputs come from the keyboard or game controller that the runner is using), producing a triplet of recordings (h, k, v). This triplet is then submitted to a verifier V. To verify the run, V plays the recorded key presses through the game G, generating $G(k)$, which is then compared with v. If $G(k)$ does not match v, then the run is rejected. The verifier then compares the recorded video of the player's hands, h with the recorded key presses k, looking for splices in the video and inconsistency between the video and the player's key presses. If the result of this process on h and v, $M(h, v)$, is a success, and $G(k)$ matches v, then the run is accepted.

Consider an adversary attempting to generate a spurious submission to be accepted by the verifier. Since replacing v with $G(k)$ will never cause an rejection, the problem reduces to that of generating an h and k such that $G(v)$ is a completion of the game in the desired time. However, by assumption, it is impossible to generate a valid h, k pair that passes $M(h, k)$ unless k was generated by the legitimate player oracle. Therefore, the player can only generate a valid speedrun with the desired time if it was run legitimately.

VIII. Limits

A major constraint on the adoption of security measures is intrusiveness. Requiring additional work from the runner's part beyond simply pressing record on the computer and playing the game would meet with resistance and the additional labour costs and barriers to entry incurred and subsequent harm to the community could easily outweigh the benefits of reducing cheating. This is especially important in the case of the handcam approach as it requires the runner to buy a camera and be able to set it up in such a way to make a high quality video recording of the hands. Additionally, runners may not like the privacy loss of requiring filming of real life settings.

One problem with the game engine based approaches is that many of the most popular games to speedrun are decades old, and will not have any such speedrunning countermeasures in place. Another issues is the possibility that the speedrun heavily involves game states not anticipated by the developer, potentially rendering countermeasures useless.

All of these countermeasures hinge crucially on hardness assumptions on editing video. Video game outputs are an especially regular video type, and it is possible that machine learning systems will be capable of isolating any watermarks and working around them. Splicing video of real world objects, such as the player's hands is more difficult, but could potentially be done.

IX. Conclusion

As speedrunning becomes an increasingly popular and mainstream sport, preventing cheating is becoming an increasingly urgent concern. Here we have presented two candidate solutions to prevent cheating. One is an method implemented by game developers to give a verification mode to their game to allow for guarding 3D games from splicing attacks. This method requires a live internet connection but does not any additional setup on the part of the runner. The other method guards against a wider range of potential attacks and only requires that the game be deterministic, but requires the runner to setup additional equipment to be verified. None of these solutions are perfect and no perfect solution is possible. Despite this, we hope that these proposals could help prevent forged speedruns and help speedrunning mature as it grows larger and it becomes harder for community monitoring to be able to keep all issues in check.

References

[1] Garrick Brazil, Gerard Pons-Moll, Xiaoming Liu, and Bernt Schiele. Kinematic 3d object detection in monocular video. *CoRR*, abs/2007.09548, 2020.

[2] Graham Finlayson, Clément Fredembach, and Mark S. Drew. Detecting illumination in images. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007.

[3] Xiaolong Liu, Zhidong Deng, and Yuhan Yang. Recent progress in semantic image segmentation. *CoRR*, abs/1809.10198, 2018.

[4] Matt Parker. How lucky is too lucky?: The minecraft speedrunning dream controversy explained.

[5] Ashutosh Saxena, Justin Driemeyer, and Andrew Y. Ng. Learning 3-d object orientation from images. In *2009 IEEE International Conference on Robotics and Automation*, pages 794–800, 2009.

[6] Steven T Wright. How the scourge of cheating is changing speedrunning, Dec 2019.