A Peek into the Metaverse: Detecting 3D Model Clones in Mobile Games

Chaoshun Zuo
The Ohio State University
zuo.118@osu.edu

Chao Wang
The Ohio State University
wang.15147@osu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Abstract

3D models are indispensable assets in metaverse in general and mobile games in particular. Yet, these 3D models can be readily extracted, duplicated, or cloned, a reality that poses a considerable threat. Although instances of games duplicating 3D models from other games have been documented, the pervasiveness of this issue remains unexplored. In this paper, we undertake the first systematic investigation of 3D model cloning within mobile games. However, multiple challenges have to be addressed for clone detection, including scalability, precision, and robustness. Our solution is 3DSCAN, an open source 3D Scanning tool for Clone Assessment and Notification. We have evaluated 3DSCAN with about 12.2 million static 3D models and 2.5 million animated 3D models extracted from 176K mobile games. With these 3D models, 3DSCAN determined that 63.03% of the static models are likely cloned ones (derived from 1,046,632 distinct models), and 37.07% animated 3D models are likely cloned ones (derived from 180,174 distinctive models). With a heuristic-based clone detection algorithm, 3DSCAN finally detected 5,238 mobile games likely containing unauthorized 3D model clones.

1 Introduction

Modern computer games rely heavily on their scenes, which are distinctive visual states showcasing game-specific information. To enrich the immersive experiences for players, contemporary games such as World of Warcraft and FIFA incorporate 3D elements rather than simplistic 2D designs. In particular, static 3D objects [1], characterized by their unique color and texture, and their optional animations, constitute the fundamental elements of 3D models. These models directly impact the player's experience as they form the building blocks for game scenes. However, crafting these 3D models is a laborious and time-consuming task, typically undertaken by professional artists, an expensive human resource [2]. Consequently, 3D models have emerged as a core asset in game production and, with their growing importance in the metaverse [3, 4], their relevance is set to increase in line with the evolution of metaverse development.

However, this rise in prominence is accompanied by amplified security concerns. Specifically, 3D models are susceptible to cloning, unauthorized reuse, or resale, posing a considerable threat to the intellectual property of game developers and broader security landscapes. This vulnerability is rooted in the necessity to render 3D models on client devices to provide seamless user experiences, such as minimizing network latency. Given this need, client devices inevitably retain copies of 3D models that currently lack adequate protection. As a result, malicious entities can conveniently extract these 3D models from game binaries (for example, through `AssetStudio` [5] for games developed with the `Unity` engine, or `UE Viewer` [6] for `Unreal Engine`).

This predicament is not hypothetical but has resulted in numerous instances of 3D model theft [7–9] causing significant disruptions. For instance, the popular game Magnet Simulator, with over 145 million plays [10], was discontinued [11] due to stolen 3D models from Bubble Gum Simulator. The ramifications of such theft are extensive, impacting not only the original creators but also game developers who might inadvertently use stolen 3D models.

Therefore, it is imperative to identify the cloned 3D models and notify their creators as well as game developers. Historically, detecting copied or stolen code in software has been an important topic in software security, and numerous algorithms (e.g., [12–14]) have been developed for this purpose, since duplicating code from others could introduce security problems (e.g., propagating vulnerabilities) or violating the copyright. Similar to program code, cloning 3D models could also cause harms, as described in the above Magnet Simulator case. As such, in this paper, we seek to make a first step towards systematically detecting 3D model clones in mobile games. Intuitively, a simple approach might involve comparing hashes of 3D models, analogous to copyrighted movie detection in the P2P network [15], where each movie is first indexed by its cryptographic hash, and any matching copies are identified as illegitimate network sharing.

Unfortunately, generating an index for a 3D model poses substantial challenges. In theory, one might attempt to index the 3D model file using a hash (such as SHA-256) similarly to copyrighted movie detection. However, this method lacks robustness, as a 3D model can be easily and even unintentionally modified. A static 3D object, comprised of three-dimensional vertices, faces defined by these vertices, and materials like textures, can be subject to squashing or

stretching. These alterations can notably change the vertices representing the 3D model while maintaining the same visual impact. Furthermore, game IDE optimization often results in the same 3D model extracted from different games differing in raw data.

To address these challenges, we propose a robust and efficient 3D model indexing scheme founded on two key insights. Primarily, while squashing or stretching might alter the vertices, the relationship between vertices and faces remains consistent in cloned 3D models. Thus, our first insight is to abstract this relationship, which enables us to derive an index for the static 3D models. Regarding 3D models with animation, an additional time dimension is present that defines the motion (i.e., the movement trajectory) of the 3D objects. Our second insight, therefore, involves measuring the Euclidean distance [16] of the trajectories between two compared models using interpolation to identify clones.

We have implemented this scheme into our open source tool, 3DSCAN[1], which stands for 3D Scanning tool for Clone Assessment and Notification. We employed it to detect 3D model clones among the 176,361 `Unity`-engine based mobile games retrieved from Google Play. 3DSCAN successfully extracted 12,200,055 static 3D models and 2,451,304 animated 3D models from these games. 3DSCAN further determined that 63.03% of static 3D models are cloned from 1,046,632 distinct ones, and 37.07% of animated 3D models are clones from 180,174 distinct models, values that we elaborate on further in the results section.

Contributions. We offer the following contributions:

- **Novel Problem**. We make the first step towards the large-scale study of 3D model clones in mobile games, addressing detection challenges and the clone prevalence among games. While our focus is primarily on one aspect of the metaverse, we hope that our study will serve as a catalyst for further research in the realm of 3D model security.

- **Efficient Techniques**. We have crafted two practical techniques to detect clones by (1) indexing the static 3D models using a value-insensitive normalization, and (2) aligning animated models by measuring the Euclidean distance via interpolation of motion trajectories.

- **Empirical Results**. Our evaluation with 176,361 mobile games suggests that 63.03% of static 3D models and 37.07% of animated 3D models are clones. With a heuristic-based clone detection approach, 3DSCAN identified 5,238 games likely containing unauthorized 3D model clones.

---

[1]The source code of 3DSCAN can be found at `https://github.com/OSUSecLab/3DScan`.

2   Background

The Compositions of 3D Models. A 3D model consists of (*i*) a mesh that describes the basic object (including its color and texture) in a scene and (*ii*) the optional animation that describes how the mesh transforms its shape at different times so that the mesh can show different motions [1]. Consequently, there can be two types of 3D models in a scene:

(I) Static Mesh, which is static and will not move or transform. Examples of static meshes include the wall in a game. Note that the appearance of a mesh is defined by its shape and material.

- **Shape**. A shape consists of a set of vertices, edges, and faces (i.e., surfaces) [1]. A mesh can have thousands or more vertices. By connecting vertices, edges can be formed naturally, since an edge is a particular type of line segment that joins two vertices. A loop connected by 3 edges (or more) represents the face surrounded. And those faces together form the shape of the object represented by the mesh. For example, Figure 1 shows an example of a mesh file. The vertices are defined from line 2 which starts with "v". Each vertex is defined by its coordinates in *x*, *y*, *z* axis. The order of their definition also indicates their IDs. The faces are defined from line 7 which starts with "f". Each face is defined by 3 vertices (also 3 edges) which is specified by its ID.

- **Material**. A material describes how the surface (i.e., faces) of a shape looks like [1]. The most common way is through color or texture. For example, a red ball can be represented by a sphere mesh with a red surface. However, the surface of an object can be very complicated (e.g., with more than just one single color). To represent such objects, texture is used [1]. For example, a picture of a real-world object can be projected over a mesh surface by mapping the vertices on the mesh with the picture (i.e., UV mapping). In addition, other attributes can also be used to describe the material, such as its reflectivity and transparency [1].

(II) Animated Mesh, which can move or show with different motions, such as the avatar in a game that can run, jump, or walk. To show different motions, the mesh needs to be a skeletal mesh with skeletons. A skeleton is a set of bones that are interconnected and in a hierarchical structure (e.g., parent and child relationship) [1]. A mesh that has skeleton is also called a skeletal mesh. The vertices of a skeleton mesh are attached to the bones, and the vertices will move by following the movement of the bones on which they are attached, e.g., a child bone will move by following the movement of its parent bone. In particular, to attach vertices to a bone, there will be a map between the vertices and the bone to which it attaches. Just as skin in the real world can be moved by several bones, particularly near a joint, a vertex within a 3D model can map to multiple bones. For each
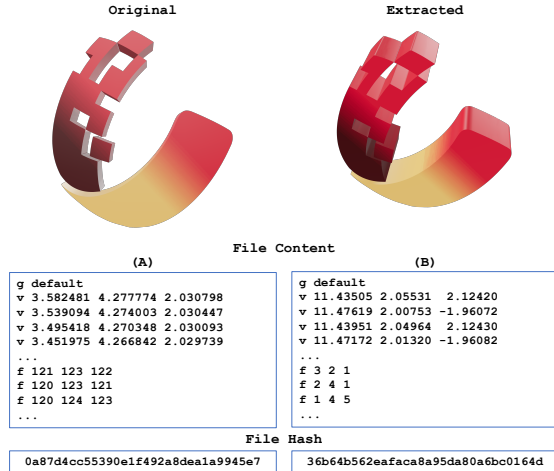
Figure 1: An example of a cloned 3D USENIX association logo model.

corresponding bone, a specific weight is assigned to indicate the degree of influence that bone exerts over the vertex.

A motion can be presented as a series of poses, similar to a film using a series of static frames [1]. By showing different poses sequentially, the mesh can make motions. Such a series of poses is called animation, and each pose is called a keyframe [1]. An animation normally consists of a series of keyframes. A keyframe records the status of each bone at the point-in-time and the time when it should be displayed. The status includes rotation, scale, and position on the x, y, and z axis. By moving the bone sequentially based on the time defined in the keyframes, the animation controls the movement of the mesh faces. In addition, to provide a smooth movement, a rendering tool will automatically use interpolation to generate the poses between two sequential keyframes.

3D Model Creation. Creating a high quality 3D model often requires not only creativity from art, but also knowledge from science and engineering such as the knowledge of biological structure, skin, and architectural finishes. Although 3D model creation is a complicated task, at a high level it can be divided into the following four subtasks:

(1) **Creating shapes**. The shape is the frame of a 3D model and consists of vertices and faces. Typically, 3D design software offers some primitive shapes such as cube and ball. But for a customized object, 3D artists need to build it from scratch, or they can apply other devices, such as a LiDAR scanner, to scan a real object to get a corresponding mesh. The shape of a 3D object can be complicated. In our experiment, we have seen meshes with up to 2,741,574 vertices and 3,473,208 faces.

(2) **Designing appearances**. One factor that determines the appearance of a 3D model is its texture. Multiple ways can be used to create the texture. For instance, a 3D artist

can take a picture of a real object, and then project the picture to the mesh by connecting the vertices to the UV axis of the picture. Furthermore, to make the object look more "real", other properties like reflectivity and transparency also need to be defined properly. There are some existing bundles of such properties, e.g., "wood material" that will make the mesh surface looks like wood.

(3) **Building skeletons**. A skeleton is needed for a 3D model if it involves movements. To build a skeleton, a 3D artist first needs to define the number of bones and their hierarchy structure, such as the bone in an arm should attach to the bone of a body. Then the artist needs to select the vertices and attach them to different bones, a tedious process that requires focus and knowledge such as human body structure.

(4) **Designing animations**. For a movable 3D object, it requires the design of animations. Simple motions like the rotation of a wheel can be easily represented by equations. For complicated motions, a 3D artist can either manually design some particular poses to generate part of the keyframes and then use interpolation to generate other frames between them, or use a digital video recorder to capture a moving object in the real world, from which to further generate keyframes by mapping the captured object with the bones in the 3D model.

Today, 3D model creation has become a profession. There are several 3D model markets such as the popular CGTrader [17]. In the markets, some of them are free, but most of them need to be paid. The price can range from a few dollars (e.g., a piece of log costs $2) to a few thousand (e.g., a human anatomy costs $4,299 [18]). If game producers do not have dedicated 3D teams, they can purchase the needed 3D models from the market to develop their games. When the game app is released, the 3D models will be encoded and embedded in the game.

3D Model Extraction. A 3D model can be directly extracted from game binaries. Tools such as `AssetStudio` [5] and `UtinyRipper` [19] are able to access and export the 3D models embedded within `Unity` binaries. It is important to note that the extraction of 3D models using `AssetStudio` or similar tools could raise intellectual property and copyright concerns, since this could violate the game developer's terms and conditions. Therefore, it is crucial for developers and users to respect the intellectual property rights of game creators and obtain proper permissions when working with extracted assets.

3D Model Clone. In this paper, we define a model "clone" is a model that is an exact-copy of another model with minimal changes, including stretching or squeezing, noise introduced by the extracting tool, and noise introduced by the game IDE. However, we do not consider models to be cloned if they have structural changes, such as the manual removal of certain parts of the model. It's important to note that two iden-

tical models will be considered "clones" even if they have been independently created. This definition applies mostly to simple models like a square plane with 4 vertices. For more complex models with numerous points or surfaces, the likelihood of intentionally creating identical "clone" models reduces significantly due to their intricate properties and dimensions. In addition, it is also considered "clones" if two developers use the same purchased 3D model in their games.

3 Overview

3.1 Motivation and Goal

The security community has a deep tradition in developing methods to identify and mitigate harms, and especially the harms created by adversaries. For instance code similarity detection has been developed to identify software theft [12] as described in §8.1. In this work, since a 3D model can be easily stolen, cloned, and reused from a game binary and there are hundreds of thousands of mobile games today, our goal is to understand the landscape of the 3D model reuse (or clones) among mobile games. However, to the best of our knowledge, there are no such tools or algorithms that are designed to detect 3D model clones at scale. Therefore, we have to design an algorithm that is:

- **Scalable**. Given the vast array of mobile games with tens or hundreds of 3D models each, encompassing numerous vertices, edges, faces, keyframes, and motions, our algorithm must be designed with scalability in mind. That is, it should be capable of effectively managing and processing this massive quantity of 3D models.

- **Precise**. The identified cloned models should accurately represent cloned instances, given the reputational implications for the games involved. In other words, our approach should prioritize eliminating false positives for the sake of accuracy.

- **Robust**. During the extraction of a 3D model from a game binary, various forms of noise may be introduced. Additionally, the IDE tool could alter the raw data of the mesh during integration into the game. Thus, our algorithm should be robust against such noise. Specifically, it should be equipped to handle variations such as stretching or squeezing, noise generated by the extraction tool, and noise introduced by the game IDEs, among others.

We hope that the development of our algorithm can advance future research at the intersection of computer security and 3D game (or metaverse) design. In particular, we believe that it can serve two essential purposes: (1) It illuminates the landscape of the 3D model ecosystem by providing answers to key fundamental questions. This includes gauging the quantity and complexity of existing 3D models, estimating the effort required for unauthorized model cloning, and determining the prevalence of 3D model reuse. With

this ability, it can be used to identify the cloned 3D models in games. And the results can be used to further notify their creators as well as game developers. (2) It provides a foundational benchmark and becomes an integral part of the tool chain for future studies on 3D model clone detection. Utilizing this tool as a base, future research could build upon this groundwork to propose different detection techniques, such as those based on similarity or signature recognition.

3.2 Scope and Assumptions

This study primarily focuses on Android mobile games for our exploration of 3D model clones. The principal reason for this focus is the relative ease and accessibility of data collection from Android platforms, as compared to iOS. Notably, a substantial number of Google Play apps have already been amassed by resources like AndroZoo [20]. Conversely, gathering data from iOS applications would necessitate a jailbroken device to download and extract app content, a process which is labor-intensive and difficult to scale. Consequently, our decision to concentrate on Android games allows for more efficient and large-scale research.

In addition, while there are multiple game engines, we focus on the Android mobile games created by the `Unity` engine, and this decision is underpinned by two primary considerations. First, to extract 3D models from games, we leverage existing GUI-based, engine-specific 3D model extraction tools (e.g., `AssetStudio` [5] for `Unity` games, or `UE Viewer` [6] for `Unreal Engine` [21]) and adapt them for automated 3D model extraction. Second, `Unity` holds the lion's share of the game engine market, boasting a 48% market share [22], with Unreal Engine trailing behind at 13%.

Finally, we focus on detecting the 3D model clones that are minimally altered copies of original models, accounting for slight transformations such as stretching or squeezing, as well as noise introduced by extraction tools or game IDEs. We consciously avoid considering models as clones if they demonstrate significant structural changes, such as manually removed model parts. While this approach might lead to some false negatives, we believe it is a reasonable compromise for the proof-of-concept stage, presenting a conservative estimate (lower bound) of the prevalence of cloned 3D models in mobile games. We note that detecting clones that have undergone significant alterations is a daunting task, primarily due to the challenges inherent in quantifying such changes. Truly accurate detection of unauthorized and altered clones may necessitate techniques like 3D watermarking [23], which fall outside the scope of this work.

3.3 Challenges and Insights

Again, an intuitive approach to detecting a clone of a 3D model might involve comparing the cryptographic hashes

of the 3D files. However, this methodology does not hold up well in practice. Upon conducting preliminary experiments, we observed that even identical static 3D models could present different raw data for their meshes, a variation that occurred when the model was extracted from one game and then integrated into another. This issue becomes significantly more complex with animated 3D models, where changes are often more drastic due to the potential for keyframes to be dropped. Consequently, a robust strategy is required to address these two primary changes. In the following, we delve deeper into these challenges and offer insights into our methods for overcoming them.

Handling the Noise in Static 3D models. The raw data of a mesh might alter during extraction or upon reuse in another game. Specifically, we have identified four potential alterations to a mesh. For illustrative purposes, let's denote M_o as the original 3D model and M_e as the model extracted from an APK. The four identified changes are as follows:

(1) **Vertex Coordinates**: The x, y, and z coordinates of the vertices can vary. This is primarily due to two reasons: (*a*) an attacker might squash or stretch the model, altering the vertex coordinates, and (*b*) the 3D model extracting tools, such as `AssetStudio`, might adjust the model, given that it has a model extraction precision parameter with a default value of 25%. Figure 1 presents a concrete example, which clearly demonstrates the changes in vertex coordinates, even though the overall shape remains identical.
(2) **Vertex Count**: Alongside alterations in vertex coordinates, we observed that some duplicated vertices might disappear during model extraction or integration.
(3) **Vertex Order**: Our experimentation has also revealed changes in the ordering of vertices, which does not visually affect the shape. However, the cause of this change remains unconfirmed due to the complex logic of the IDE tools, whose source code we do not possess.
(4) **Face Order**: Since there is no particular order required among faces, it has been observed that the order of some faces might alter.

As such, the hash values of 3D models cannot be straightforwardly employed for indexing, as illustrated in Figure 1. However, it's observed that despite potential changes in vertex coordinates, the relationship between vertices and the number of faces is preserved. Specifically, through extensive experimentation, we found that sorting the vertices ensures consistent order between M_o and M_e. Furthermore, removing duplicated vertices even aligns the order perfectly. Nonetheless, the order information alone is insufficient to accurately represent the model. Consequently, we encode the faces using the ordered vertices' IDs.

Moreover, to mitigate the effects of face order alteration, we sort faces based on the IDs of the faces' vertices. This

results in a sequence of faces, encoded by vertex IDs, which represents the relationship, rather than any absolute coordinate values. To enhance performance efficiency, we generate a cryptographic hash (such as SHA-256 or BLAKE2 [24]) of the sequence and use this hash for model indexing. Any models yielding the same hash are identified as clones. This indexing-based similarity detection algorithm proves scalable and efficient, as it circumvents the need for pairwise comparison.

Handling the Noise in Dynamic (i.e., Animated) 3D Models. Just as with static 3D models, the raw data of animated 3D models can undergo modifications during game development. These modifications pertain not only to shape data but also to animation data. Specifically, the alterations in data can be categorized into two types:

(1) **Alteration in the number of keyframes**. Our observations indicate that IDE tools may eliminate some keyframes between M_o and M_e, possibly due to animation optimization.
(2) **Modulation of keyframe values (i.e., rotation and scale)**. We note that the timings of the keyframes may be adjusted, leading to consequential alterations in bone status data.

The key strategy to mitigate this challenge resides in our observation that, should M_o and M_e be similar animated 3D models, their shapes would exhibit similarity. Furthermore, we discern that the skeleton structure between M_o and M_e remains consistent for cloned models. Hence, we first apply shape information to group animated 3D models. Then, using the skeleton information, we further segregate these groups into sub-groups. Ultimately, within each sub-group, we conduct a direct comparison between pairs of potential clone models to identify the genuine clones. Of course, if two clones are allocated to different groups/sub-groups, their identification would be missed.

3.4 3DSCAN **Overview**

We have developed 3DSCAN, a tool designed to scan and detect 3D model clones in mobile games, drawing on the insights discussed earlier. Specifically, when provided with a Unity-engine based mobile game, 3DSCAN employs the following three-step process to ascertain the presence of any clones:

(1) **3D Model Extraction (§4.1)**. The first step involves extracting the 3D models from the game APK. This is accomplished by modifying the open-source tool `AssetStudio` [5].
(2) **3D Model Indexing (§4.2)**. To identify similar 3D models robustly and efficiently, 3DSCAN applies a value-insensitive normalization algorithm to the extracted 3D models, then indexes them using BLAKE2 hashes.

(3) **Cloned 3D Model Detection (§4.3)**. By comparing the indexes of 3D models, 3DSCAN identifies potential static 3D model clones. If the models are animated meshes, 3DSCAN then leverages skeleton and motion track information to further narrow down and meanwhile perform a pair-wise comparison to detect the cloned ones.

4 Detailed Design

4.1 Extracting 3D Models

While 3D models can be remotely rendered by cloud platforms such as Google Stadia [25] and locally displayed on mobile devices such as smartphones [26, 27], these operations demand high network bandwidth and significant cloud computation power. Consequently, most mobile games we have examined opt to directly package the 3D models into the APK file during game compilation. As 3D model files tend to be large, they are invariably encoded within the APK for more compact file size. This necessitates unpacking the APK and decoding the files to retrieve the 3D models. Please note that these 3D models are proprietary to the game developers; we utilize them solely for academic research without distribution, in compliance with the *fair use* doctrine of U.S. copyright law.

For `Unity` games, the 3D files are encoded using proprietary methods, thus requiring decoding. While it would be feasible to reverse engineer these file formats, a number of open-source tools such as `AssetStudio`[5] and `UtinyRipper`[19] are readily available for extracting models from `Unity` games, thanks to the engine's popularity. Of these, `AssetStudio` stands as the most favored, boasting over 7,000 stars on GitHub.

Specifically, `AssetStudio` is a GUI tool that decodes resource files and subsequently exports the 3D models in a standard format. For instance, a static 3D model will be exported to the `OBJ` file (as shown in Figure 1), while animated models will be exported to the `FBX` format. Given our need for raw data for subsequent steps and the large number of `Unity` games in our possession, it was necessary to modify this GUI program into a console tool that directly produces the raw data when supplied with an APK. We have thus successfully modified `AssetStudio` to parse models directly from a command line input and generate the following raw data of a 3D model:

- **The Shape**. The coordinates of the *x*, *y*, and *z* axes of each vertex, as well as their respective IDs. Each face is defined by the corresponding vertices IDs.

- **The Skeleton**. The number of bones and their hierarchical structure, represented as $(bone_p, bone_c)$ to denote that $bone_c$ is attached to $bone_p$.
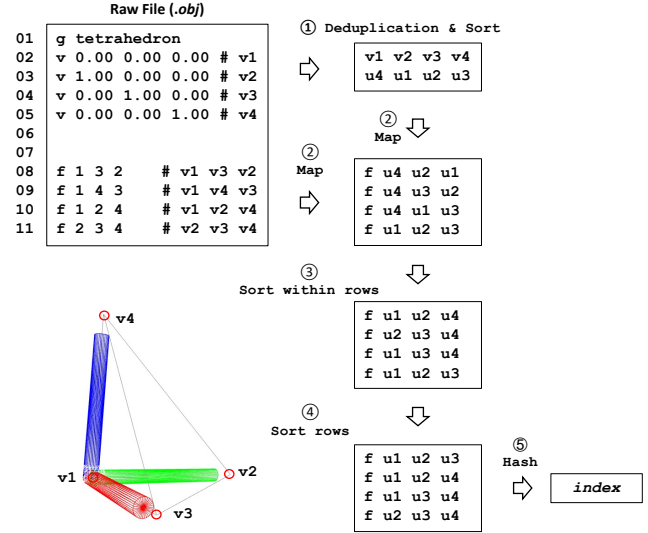

Figure 2: Illustration of each step of the indexing algorithm with our running example.

- **The Animation**. The list of keyframes, each containing time, bone, rotation, translation, and scale details.

4.2 Indexing 3D Models

With the aid of `AssetStudio`, we can extract large-scale 3D models from Android APKs. However, directly hashing these models is ineffective (as demonstrated in Figure 1), and we need to seek alternative approaches. To this end, we propose an indexing-based algorithm, which calculates a singular value based on the raw data of the 3D model. This singular value is then used to index the model, enabling models with identical indexes to be identified as clones. This type of index algorithm is efficient, boasting a time complexity of $O(1)$ with *n* models. Nevertheless, crafting such an algorithm presents a challenge because the calculation must be (1) sufficiently general to allow cloned models with disparate raw data to yield identical indexes, and (2) adequately specific to differentiate unique models. Furthermore, given the potential changes to the raw data during model import and export by IDE tools, normalization is required.

Hence, we introduce a normalization-based indexing algorithm for detecting model clones. Recall that the shape (vertices, edges, faces) of a 3D model conveys a wealth of information, making it an ideal candidate for normalization. However, as discussed in §3.3, four types of shape data can be altered between M_o and M_e: vertices values, vertices number, vertices order, and face order, due to the impact of the `Unity` IDE and model extraction tools. Fortunately, we have identified three types of data that remain preserved and can be harnessed to generate the index:

Algorithm 1 Static 3D Model Indexing

```
1:  Input: vs_o: the vertices in original order; fs_o: the faces in original order
2:  Output: hash: the hash that can represent this model
3:  procedure MODELINDEXING(vs_o, fs_o)
4:      vs_ou ← SET(vs_o)
5:      vs_n ← SORT(vs_ou)
6:      vs2ids ← DICT(∅)
7:      for i in (0, LENGTH(vs_n)) do
8:          v_i ← vs_n[i]
9:          vs2ids[v_i] ← i
10:     end for
11:     fs_n ← ∅
12:     for f_oi in fs_o do
13:         f_ni ← ∅
14:         for vid_oj in f_oi do
15:             v_i ← vs_o[vid_oj]
16:             vid_ni ← vs2ids[v_i]
17:             f_ni ← f_ni ∪ vid_ni
18:         end for
19:         fs_n ← fs_n ∪ f_ni
20:     end for
21:     for f_ni in fs_n do
22:         f_ni ← SORT(f_ni)
23:     end for
24:     fs_ns ← SORT(fs_n)
25:     hash ← BLAKE2(fs_ns)
26:     return hash
27: end procedure
```

- **Vertices Relationship**. Although the x, y, and z coordinates of the vertices may vary, the relationship between two vertices in the 3 dimensions is consistently maintained in M_o and M_e. For instance, if the coordinates of $X(v_a)$ are less than $X(v_b)$ on the X-axis of M_o, this relationship will persist in M_e.

- **Face Numbers**. To ensure the same visual effects, the number of faces remains consistent in M_o and M_e.

- **Face-Vertices Relationship**. While the vertices or face IDs may be modified, their relationship is preserved. Specifically, if a face f_a exists in M_o, a corresponding face f_b will exist in M_e to deliver the same visual effect.

Based on those consistent data, we have designed an algorithm as presented in Algorithm 1. To clearly illustrate the algorithm, we use a tetrahedron model in Figure 2 as a running example to show detailed steps in generating its hash index. At a high level, the algorithm takes the shape information as input and outputs a hash for the model. Specifically, it first removes duplicated vertices in line 4 by comparing the coordinates x, y, z. In the example, since there are no duplicate vertices, all vertices will be kept. Second, the remaining vertices will be sorted according to their coordinates in line 5. Each vertex will be assigned with a new ID (on lines 6-10) based on its position in the sorted list. The corresponding step in the example is illustrated in step ①. The four vertices will be sorted in descending order. As a result, the new order would be $[v2, v3, v4, v1]$, based on which the new IDs will be assigned. For instance, the coordinates of $v1$ is 'smaller' than other vertices, so it is the fourth item in the sorted list. As such, its new ID is $u4$. Third, each face will be represented by the new vertex IDs (at lines 12-20), which is illustrated in step ② in the example. For instance,

the first face (i.e., $\{v1, v3, v2\}$) will be represented as $\{u4, u2, u1\}$. Fourth, the IDs will be sorted within the face at line 21-23, which is illustrated in step ③ in the example. For instance, the first face (i.e., $\{u4, u2, u1\}$) will be sorted (i.e., descending) to $\{u1, u2, u4\}$. Next, the faces will be sorted according to their vertex IDs at line 24, which is illustrated in step ④ in the example. For instance, the IDs of fourth face (i.e., $\{u1, u2, u3\}$) is 'larger' than any other face, so it will be sorted to the first place. Eventually, the BLAKE2 hash value will be generated for the model by the sorted faces at line 25, which is illustrated at step ⑤ in the example. We can clearly see that our hash is not generated by any concrete coordinate values in the 3D models but rather by the order of faces along with the order of its sorted vertices.

4.3 Identifying Cloned 3D Models

The purpose of 3D model indexing is to identify the cloned 3D models. In particular, our indexing is based on the shape, which is the basic element of both static 3D models and also animated 3D models. When the indexes of the 3D models, the cloned static 3D models can be directly identified. More importantly, this index can also significantly help identify animated 3D models, since we can first use the same shape to narrow down the "suspicious" cloned 3D models. That is, 3DSCAN directly detects static 3D models, whereas to detect animated 3D models, it uses static 3D model detection first to narrow down the scope (for efficient reasons) and then performs a pair-wise comparison.

(I) Detecting Cloned Static 3D Models. If a model does not contain any animation, it is a static 3D model. The index value computed at §4.2 can now be used directly to detect static model clones. That is, if two models have the same index value, then 3DSCAN decides that one of them is a clone of the other. Also, since one model can be cloned by several games, 3DSCAN automatically groups the 3D models based on their indexes. Note that 3DSCAN cannot determine the origin among the cloned one, which requires additional techniques such as 3D watermarking [23] and again this is beyond our current scope.

(II) Detecting Cloned Animated 3D Models. An animated 3D model is composed of the static 3D model, the skeleton, and the animation. Ideally, similar to our static 3D model clone detection, we wish to have an indexing-based approach. Unfortunately, detecting animated 3D model clone is far more complicated than that of the static 3D model, as we cannot simply use the shape for the detection, because an animated 3D model contains animations and meanwhile the animation is applied to the skeleton (but different animated 3D models can have the same skeleton). Particularly, although the skeleton does not get changed, the animation information can be changed in several ways (more details will be provided below). As such, we cannot

Algorithm 2 Skeleton Hashing

```
 1: Input: bone: the root bone of the skeleton
 2: Output: hash: the hash that can represent this skeleton
 3: procedure SKELETONHASHING(bone)
 4:     bones ← ∅
 5:     for child in CHILDRENBONES(bone) do
 6:         childs ← SKELETONHASHING(child)
 7:         bones ← CONCAT(bones, childs)
 8:     end for
 9:     hash ← BLAKE2(bones)
10:     return hash
11: end procedure
```
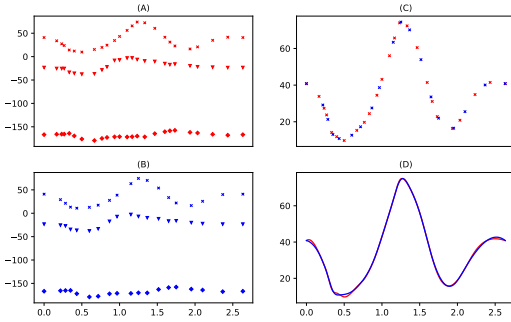
Figure 3: Animation Interpolate

use an index-based approach and instead we propose a divide-and-conquer algorithm, by which we first divide the animated 3D models into groups, and then conquer the clone detection within each group with a pair-wise comparison.

Dividing animated 3D models by its static 3D model and skeleton. In addition to the static 3D model, the skeleton can also be used to divide animated 3D models since a cloned copy must have the same skeleton. As discussed in §2, a skeleton is a set of bones constructed in a hierarchy structure, which follows the same structure of the target object such as human bones, for instance, the left foot bone is attached to the left leg bone. The head of the hierarchy structure is the root bone. Since a skeleton is essentially represented in a tree structure, we have to design an algorithm to index the skeleton as well.

Inspired by the Merkle tree [28], we propose a recursive algorithm as illustrated in Algorithm 2 for our purpose. Specifically, a hash value will be used to represent each bone, and the hash value for the root bone will be used to index the skeleton. The hash value of a bone is based on the hash values of its children's bones (that is, line 6). In particular, the hash values of the children's bones will be concatenated as shown in line 7, and the BLAKE2 hash function will be applied on the concatenated results to generate the hash for the parent bone (that is, line 9). As shown in line 4, if the bone does not have children, its value will be the BLAKE2 hash of the empty string. As such, we are able to index skeletons and further narrow down the animated 3D models into groups.

Conquering the clone detection via pair-wise comparison. Among each identified group of the "suspicious" an-

imated 3D models, a pairwise comparison is still needed to finally decide the clones. However, this is a challenging task since every type of raw data in an animation could have been changed. In particular, an animation is a list of keyframes that show at different timing points, and each keyframe contains its timing point and a list of bone statuses which contain the rotation and scale information for the bone that needs to perform in this particular keyframe. The rotation and scale information contain the angle of rotation, translation, and scale value in x, y, and z axis. With bone status information for different keyframes, the rendering platform will apply interpolation [1] to predict bone status between keyframes, so it can present a smooth motion.

When focusing on a specific bone during the animation, it can be shown in a scatter plot over time (the horizontal-axis) and rotation (the vertical-axis), and the status information is scattered data points in this plot. Figure 3 (A) presents the rotation information of the `leftUpLeg` bone in a `walk` animation for a human-like model M_o. The horizontal axis is the time and the vertical axis is the rotation angles of the bone in the 3D (represented by x, y, and z coordinates). For example, at time point 0 (i.e., initial state), the bone should rotate 40.83 on x-axis in the 3D, -23.27 on y-axis, and -166.77 on z-axis. The Figure 3 (B) is the data of the same bone in M_e. We can see that the data have changed dramatically. Specifically, (1) the timing point has been changed. For example, the second time point in M_o is 0.17, however, the second timing point in M_e is 0.22; (2) the values of the rotation angle have been changed. Similarly to shape data, the absolute values of the rotation angle have been changed; (3) the number of keyframes has been changed. There are 25 keyframes in M_o, but only 19 keyframes in M_e.

Therefore, to compare these two animations, it is impractical to simply compare their keyframes one by one, since the timing point of each keyframe has been changed, and they could not be aligned. As such, we have to find another way. Luckily, since the animation is a motion applied on a set of bones, by following the principle of divide-and-conquer again, we can first compare the movement similarity of the same bone in two animations, then calculate the average similarity based on the similarity of all the bones' movements. With such an average similarity, we cluster similar animations into groups.

Next, to compare the movements of the same bone in two animations, since the movements contain the rotation, translation, and scale data in x, y, and z-axis (9 types in total), we need to divide them further. In particular, we compare them one by one (i.e., x to x, y to y) and then merge them by calculating the average value. More specifically, we first compare the rotation data on the x-axis of the bone in two animations, then compare the other 8 types of data. And eventually, we calculate the average similarity based on them. When focusing on a specific comparison, such as the rotation angles in the x-axis as presented in Figure 3 (C) in which we have in-

cluded the `leftUpLeg` bone's x rotation angles of both M_o (i.e., in red) and M_e (i.e., in blue), we can see that such data are two sets of scattered data points. Although we have divided the comparison to such a low level, it is still challenging to compare the scattered data points.

Fortunately, inspired by the interpolation algorithm [1] on the rendering platform, we have designed an interpolation-based comparison algorithm. In particular, for each of the scattered data point sets, we compute a curve that crosses the points. Then we get two curves for the two data point sets, for example, Figure 3 (D) shows the two interpolation curves of the data in Figure 3 (C). As such, the similar comparison of the two sets of scattered data points has been transformed into similar comparison of two curves, which is a well-studied problem [29] in computer graphics.

There are several algorithms for calculating curve similarity, such as Euclidean distance (ED), Dynamic Time Warping (DTW) [30], and Longest Common Subsequences (LCSS) [31]. We use Euclidean distance for this purpose. The formula is:

$$ED = \frac{1}{n} \sum_{k=1}^{n} |curve_o(x_k) - curve_e(x_k)|$$

The high-level idea is that, with n samples on the x coordinates, for each of the samples, we calculate the distance of the corresponding values on the two curves. The average distance is the distance of the curves. In particular, assume that x_k is the k-th sample, $curve_o(x_k)$ returns the corresponding angle value of x_k on the curve of M_o. With this, it will produce the Euclidean distance of the two curves. Specifically, we equally sample 100 x coordinates for the calculation. For each sample, the Euclidean distance will then be calculated, for example, when time is at 0.5 (i.e., $x = 0.5$), the rotation angle of M_o is 11.11 and M_e is 9.76. As such, the Euclidean distance is 1.35 (at $x = 0.5$). The final Euclidean distance will be calculated by averaging the Euclidean distance of the 100 samples. Currently, two models are detected similar if their average ED is less than 0.95, and this threshold is obtained through experimenting with 100 cloned animations.

5 Evaluation

We have implemented a prototype of 3DSCAN by modifying `AssetStudio` for 3D model extraction from game APKs and developing our algorithm in C# and Python. In the interest of encouraging further research, we have made the source code of 3DSCAN available on Github. In this section, we report our evaluation results. We begin by outlining our experimental setup in §5.1, followed by a detailed presentation of the results for each component in §5.2.

5.1 Experiment Setup

Dataset Collection. To explore the prevalence of 3D model clones in Android games developed with `Unity`, our first task was to assemble a collection of relevant games. To this end, we engineered a crawler based on the `Scrapy` framework [32] to retrieve all the app package names from Google Play. As apps featuring 3D models are typically games, our focus was on the free games category, yielding 315,321 free games in total. We then used `gplaycli` [33] to download the APK files from the Google Play server by interacting with its Web APIs. To identify games developed with `Unity`, we checked the magic number of the files located in the `assets` folder in each APK. Ultimately, we identified 176,361 `Unity`-based games, requiring 5TB of hard drive space for storage.

Environment Setup. Our experiment was conducted on a Dell server equipped with dual Intel Xeon 8268s CPUs (48 cores @ 2.9GHz per CPU), 192GB memory, and a 96TB hard drive. The server runs on Red Hat Enterprise Linux (RHEL version 7.9). Given that `AssetStudio` is developed with C#, which necessitates a `.NET` environment, we used the `Wine` framework to run it on RHEL.

5.2 Experiment Results

We have tested 3DSCAN with 176,361 `Unity`-based games. To accelerate our analysis, we run 80 instances in parallel for extracting and indexing the 3D models. The experiment took our server 125 hours, and 3DSCAN identified 89,519 of the games containing 3D models. Further, it found 22,517,361 static 3D models and 7,696,801 animated 3D models, which indicates that each game has 251.54 static 3D models and 85.98 animated 3D models on average. Meanwhile, within a game, the same model can be used several times, which can be identified by the resource file used by the model. By removing the repeated models within each game, we eventually found 12,200,055 static 3D models and 2,451,304 animated 3D models.

With our indexing algorithm, 3DSCAN identified 5,557,146 distinctive static 3D models. The models can have up to 2,741,574 vertices (which is a model of courtyard in game Restoration VR) and 3,473,208 faces (which is a model of corona virus in game Lockdown Area). By identifying similar 3D models among static 3D models with their indexes, 3DSCAN found that 4,510,514 only exist in one game and 1,046,632 (i.e., 18.83%) have been found in different games. Also, it found more than 94.75% games have used cloned models. The most popular 3D model is the model of a floor which only has 4 vertices and 2 faces. It has been found in 29,176 games. Also, there are 41,322 distinctive skeletons in all animated 3D models. By combining the model index and the skeleton, it identified 188,015 groups of animated 3D models. By comparing the animations

Game Category	# Games	# Static	Avg.	# Animated	Avg.
Action	13,622	3,628,174	266.35	2,050,971	150.56
Adventure	6,737	1,931,868	286.75	835,033	123.95
Arcade	14,829	2,480,407	167.27	634,569	42.79
Board	1,278	85,320	66.76	33,072	25.88
Card	670	40,021	59.73	38,394	57.30
Casino	793	42,234	53.26	56,662	71.45
Casual	12,176	2,331,971	191.52	705,614	57.95
Educational	2,447	541,045	221.11	171,626	70.14
Music	726	55,419	76.33	27,862	38.38
Puzzle	6,749	1,312,079	194.41	466,566	69.13
Racing	6,745	2,554,523	378.73	243,820	36.15
Role Playing	3,258	1,069,659	328.32	539,948	165.73
Simulation	13,135	5,270,549	401.26	1,272,250	96.86
Sports	3,249	553,301	170.30	264,845	81.52
Strategy	2,304	546,875	237.36	307,952	133.66
Trivia	412	51,594	125.23	25,274	61.34
Word	389	22,322	57.38	22,343	57.44

Table 1: The 3D models distribution across game categories. Note that Google mandates that a game can only belong to one category.

within each group, 1,722,795 distinctive animated models have been identified. Among them, the most popular animated 3D model is the model of a driver with animation `turnSteeringWheel` that has been found in 1,544 games.

Detailed Results. To provide a clear presentation of the experimental performance of 3DSCAN with these games, we detail the performance of each of its three components in the following.

(I) Extracting 3D models. Again, 3DSCAN extracted 12,200,055 static 3D models and 2,451,304 animated 3D models in total. However, we also found that, not all of the `Unity` based games contain 3D models, because there are many 2D games. In total, we found 89,519 of them contain 3D models. The 3D model numbers are varied among different game categories. As such, we present the model information based on the category of the game in Table 1. We can see that the `Simulation` game category is the top category that contains more 3D models. Meanwhile, on average, the games in the `Simulation` category also contain more 3D models. One possible reason is that this type of game is commonly simulating real-word activities (such as vehicle simulation) which require more models than other game categories. For animated 3D models, the `Action` category contains more than that of other categories. However, on average, the `Role Playing` category contains more animated 3D models than others. It indicates that the games in `Action` category would use the same animated 3D models several times and the game in `Role Playing` category needs more different kinds of animated 3D models. Furthermore, we also present the model information by grouping the games according to the installed times in the first 6 columns in Table 2. We can see that popular games tend to use more models (i.e., columns 4 and 6 are increasing) for rich content.

(II) Indexing 3D models. To understand the complexity of the 12,200,055 static 3D models for the indexing algorithm, we present the distribution of the shape information
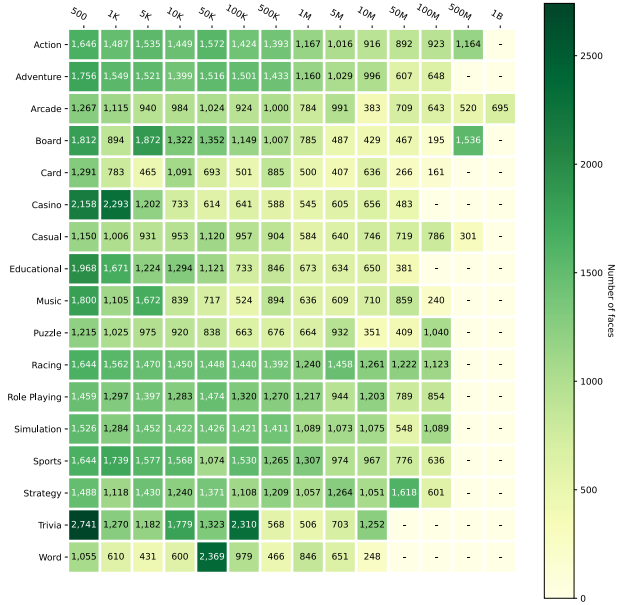

Figure 4: Model average faces distribution across games.

in Figure 5a. In particular, the horizontal-axis is the number of vertices (for red curve) and faces (for blue curve); the vertical-axis is the percentage of 3D models that have fewer vertices or faces than the corresponding value in the horizontal-axis. We can see that more than 20% of models have more than 1,000 vertices and there are even some models (about 2.52%) have more than 10,000 vertices. Similarly, 80% of models have more than 820 faces, and there are even 2.30% of models have more than 10,000 faces. Meanwhile, we also present the average vertices and faces data for games with different install numbers in columns 7 and 8 of Table 2. Interestingly, while the popular games tend to have more models, the model complexity is actually decreasing (i.e., columns 7, 8 are decreasing). To clearly show this finding, we present a heat map of the distribution of average faces of games based on their categories and install numbers in Figure 4. We can see that the models in some categories (e.g., `Action`) are more complex than others. Meanwhile, in all categories, the trend of model complexity is decreasing, while the install number is increasing.

We also present the distribution of the extracted skeletons, which can be found in Figure 5b. The *x*-axis is the number of bones the skeleton has, and the *y*-axis is the number of such skeletons. We can see that most skeletons have less than 100 bones. But the skeleton can also be complex; the most complex skeleton in the experiment has 979 bones, which is a pant that has motions in game `Misguided`.

(III) Identifying Cloned 3D Models. After our indexing, 3DSCAN is ready to detect the clones. In total, 5,557,146 distinct static 3D models have been identified and 1,046,632

# Installs	# Games	(I)				(II)		(III)				
		Static 3D Models		Animated 3D Models		Model Average		Distinct Model		Model Clones		
		Total	Average	Total	Average	Vertices	Faces	Static	Animated	Static	Animated	Average
500	33,182	4,969,430	149.76	1,357,767	40.92	1,766.91	1,470.54	1,404,839	338,742	409,252	72,485	14.52
1K	13,177	2,948,531	223.76	919,989	69.82	1,612.37	1,325.13	918,647	198,108	398,695	62,881	35.03
5K	5,475	1,392,774	254.39	429,659	78.48	1,646.80	1,341.39	515,873	114,060	241,060	35,022	50.43
10K	11,659	3,389,492	290.72	1,249,804	107.20	1,630.57	1,320.78	1,033,665	247,982	452,565	68,981	44.73
50K	4,528	1,468,631	324.34	540,708	119.41	1,729.43	1,378.50	486,907	137,904	264,041	33,113	65.63
100K	9,460	2,998,522	316.97	1,161,467	122.78	1,613.92	1,318.86	949,642	226,499	428,137	67,664	52.41
500K	3,457	1,265,555	366.08	446,742	129.23	1,592.51	1,309.02	476,733	133,743	242,269	37,844	81.03
1M	5,521	2,323,020	420.76	796,189	144.21	1,277.82	1,040.69	961,134	371,945	350,072	43,033	71.20
5M	1,313	663,554	505.37	234,583	178.66	1,291.44	1,038.19	285,578	88,566	122,666	16,824	106.24
10M	1,448	890,168	614.76	473,049	326.69	1,000.09	825.86	483,649	131,387	174,879	25,898	138.66
50M	202	152,370	754.31	60,573	299.87	853.12	756.63	77,433	12,977	29,382	1,798	154.36
100M	89	51,016	573.21	23,896	268.49	923.47	838.93	37,040	5,664	10,367	364	120.57
500M	7	2,934	419.14	1,989	284.14	760.56	702.61	2,271	123	243	0	34.71
1B	1	1,364	1,364.00	386	386.00	730.06	695.31	1,048	376	316	10	327.00

Table 2: The 3D models distribution across games that grouped by their install times.

(a) Models distribution

(b) Skeleton distribution

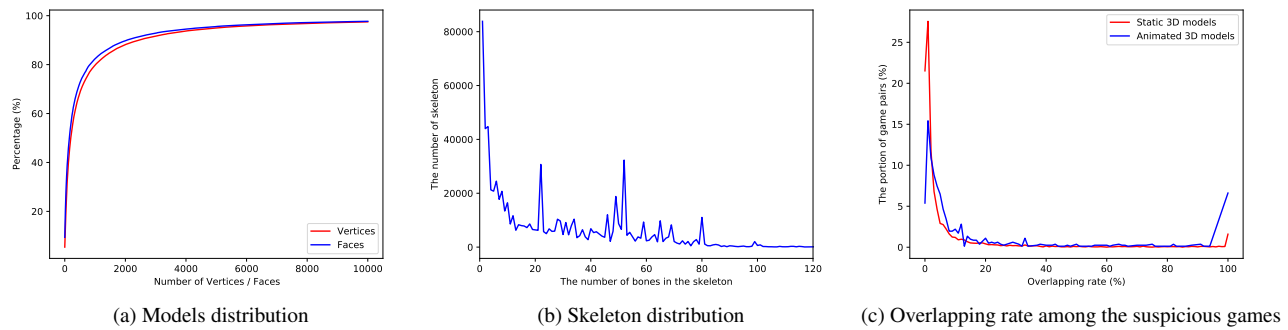(c) Overlapping rate among the suspicious games

Figure 5: Some characteristics of the tested static 3D models

of them have been found in different games. The columns 9-13 in Table 2 present the experiment data for this component. We can see that, popular games tend to contain more cloned models (i.e., the last column is increasing). To understand the static model clones, we further group the models based on the number of games that contain them and present the data in Figure 6a. We can see that 4.5 million of them only exist in one game and 0.5 million of them exist in two games. And more than 530,000 of them exist in more than 2 games. There are even 458 models that have occurred in more than 500 games. As such, more than 80% distinct models are exclusive models (i.e., only exist in one game). Furthermore, to understand the model clone among games, we define a term `sharing rate` (i.e., *sr*) for each game which can measure its clones of the models. In particular, the rate of sharing is the average occurrence of all its models, which can be calculated using the following equation:

$$sr_G = \frac{1}{n} \sum_{k=1}^{n} occurrence(M_k)$$

The `occurrence(`M_k`)` represents the occurrence of model M_k. By grouping the games based on their share rates, we create Figure 6c. We can see that only around 6,700 games'

sharing rate is 1.0. Recall that 80% models are exclusive. As such, there are some common models used by many games.

Finally, by grouping the animated 3D models by their static model index and skeleton, 3DSCAN identified 188,015 groups of animated 3D models. With further comparison, 1,722,795 distinct animated 3D models have been identified. They are using 179,541 static 3D models. On average, each model can have around 10 motions. Similarly to the static model clone, we present the data of the animated 3D model clone in Figure 6b. We can see that 1.5 million (i.e., more than 88%) animated 3D models exist only in one game. Most of them (i.e., 94%) exist in one or two games. Therefore, we can conclude that static 3D model clone is more common than animated model clone. However, similar to static 3D model clone, there are some popular animated models, for instance, 46 animated 3D models occurred in more than 500 games.

Accuracy Analysis. While 3DSCAN delivers promising results, it's not without its potential inaccuracies. Theoretically, the system may yield both false positives (FP) and false negatives (FN), as some information may be lost during model indexing. To measure the FP rate (i.e., dissimilar models identified as similar), we randomly selected 1,000 pairs of non-duplicate static 3D model indexes and 1,000

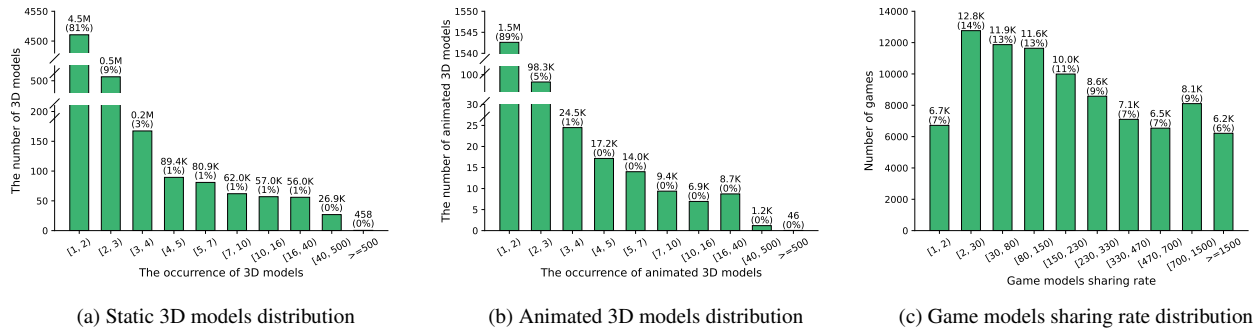| (a) Static 3D models distribution | (b) Animated 3D models distribution | (c) Game models sharing rate distribution |

Figure 6: Some characteristics of the tested animated 3D models.

pairs of non-duplicate animated 3D model groups. In each pair, the two models have distinct indexes. Upon manual review of the models' appearances, we did not encounter any cases where different models shared the same index.

While FPs can be quantified, measuring FNs (i.e., models that appear identical but bear different indexes) is more challenging. To tackle this, we leveraged the `name` field of the model, defined by the developers. We selected 1,500 models for FN analysis, gathering a list of 50 common object words, such as 'book' and 'dog'. For each word, we collected 30 models with names containing the corresponding word. Following a manual review of models for each word, we found no instances of similar models possessing different indexes, suggesting no FNs in the models we validated.

Note, however, that manual review of these models can take months. To expedite this process, we developed a helper program that renders static models into `jpg` files and animated 3D models into `gif` files. This streamlined presentation allows for review using keyboard controls alone, reducing the verification process to about a week for two authors.

6   Detecting Unauthorized Clones

Despite having identified 1,046,632 static 3D model clones and 180,174 animated 3D model clones, determining the legality of these clones (i.e., discerning authorized from unauthorized clones) remains challenging. This is due to the potential scenario where developers have legitimately purchased these models from markets. The key difference between legal and illegal clones hinges on whether the creator has authorized the clone. Given that it is unrealistic to detect illegal clones based solely on the model data, we conducted a manual review of thousands of model clones. From this, we made three key conjectures: (1) Popular game developers often create their own models rather than purchasing them; (2) Models in popular games are more likely to be appropriated without authorization due to the games' visibility; (3) Complex models are more likely to be illicitly copied

than simpler ones. Based on these insights, we propose a heuristic-based approach for detecting unauthorized clones.

Specifically, if complex models are only found in a popular game and a less-popular game from a different developer, it is plausible that the less-popular game has appropriated the models from the popular one. Our detection process is as follows: (1) We identify complex static and animated 3D models shared across games; (2) We exclude pairs of games developed by the same team; (3) We retain only those pairs that comprise a popular game (defined as a game with over 1 million installs) and a less-popular game (defined as a game with installs constituting 1% of the popular game's installs). Ultimately, we identified 6,850 pairs of games with static 3D model clones and 817 pairs of games with animated 3D model clones that may potentially be unauthorized clones.

However, recognizing potential unauthorized clones might also result in false positives. For example, the developers of both the popular and less-popular games could be the only purchasers of a particular model from a 3D market, leading to misidentification of the less-popular game as having unauthorized clones. To confirm unauthorized clones, one approach could be to contact the developers of the games with stolen models, but this is not feasible for thousands of games. Instead, we reported these potential instances of resource abuse to Google, who we believe are best equipped and incentivized to regulate the app market.

Collecting evidence for Google, however, is a complex task. We need to ensure that the model is not publicly available, necessitating a search for each model on popular 3D markets. Given the daunting prospect of searching for each model, we opted to randomly select 1,000 models from the dataset, excluding other models from the same game in subsequent selections. Through this process, we discovered 26 free models available on 3D markets. After removing these, we prepared the game and 3D model metadata for Google. In total, we submitted 974 suspicious games to Google.

Confirming resource abuse (i.e., 3D asset clone) is also time-consuming for Google. As of this writing, Google is

	Victim Game	# Install	Category	Developer	Game w/ Cloned Models	# Install	Category	Developer	Removed?
Game Clone	Flip Diving	50M	Sports	MotionVolt Games Ltd	Cliff Diving	5K	Simulation	BBLoveG	✓
	Zombie 3D Gun Shooter	10M	Arcade	Fun Shooting Games .	Unkilled Hunter	1K	Action	Bsbh Studio	✓
	Agent Action	10M	Action	Saygames Ltd	Agent Legend	100	Adventure	Keep200019	✓
	Bike Racer	10M	Racing	Games Saga	Stunt Bike Racer 3D	100	Simulation	Liudingjun	✓
	Shoot Goal	10M	Sports	Bambo Studio	Dream Football Strike	10K	Sports	Crazy Sports Co.	✓
Idea Clone	Temple Run	500M	Arcade	Imangi Studios	Eternal Parkour	500	Action	Mangoqueen	✓
	Join Clash 3D	100M	Arcade	Supersonic Studios LTD	Join Clash 3D: Crowd Rush	5K	Arcade	Gamezo	✓
	Talking Tom Hero Dash	100M	Action	Outfit7 Limited	Jungle Cat Runner	100	Role Playing	Vigour3D	✓
	Hungry Shark Evolution	100M	Arcade	Ubisoft Entertainment	Scuba Hunter	100	Adventure	Arcadian Lab	✗
	Robot Helicopter	5M	Action	Naxeex Robots	Us Army Robot Helicopter	5K	Strategy	Dreams Games Inc,	✓
Model Clone	Subway Surfers	1,000M	Arcade	Sybo Games	Bob The Robber Runner	100	Adventure	Appspros	✗
	Cover Strike	100M	Action	Free Actions	Military Commando Mission	1M	Adventure	New Games 2021 .	✓
	Color Bump 3D	100M	Arcade	Good Job Games	Color Bumper Ball	1K	Arcade	Coz Rdev	✗
	Flight Simulator	100M	Simulation	I6 Games	Airplane Free Fly Simulator	10K	Simulation	Bf Games Studio	✓
	Bridge Race	100M	Casual	Supersonic Studios Ltd	Tower Stack	1K	Casual	Jacob Van Haag	✓

Table 3: The unauthorized model clones among the inspected games.

still verifying the games, and we are prepared to provide additional information if necessary. Google has already begun removing some of them from the market. By checking the package names, we found that 360 (36.96%) reported games have disappeared from Google Play. We would like to emphasize that our intention is not to act as "cyber police," but rather to provide objective data on model similarities. Reporting suspicious unauthorized clones to Google is merely a step towards further confirmation. We also plan to explore other routes, like contacting developers directly and allowing them to make the final decision.

Result analysis. To delve deeper into cloning behavior within each pair of games, we introduce the concept of an overlapping rate between two games (*Ga* and *Gb*) as:

$$overlapping(Ga, Gb) = \frac{M_{Ga} \bigcap M_{Gb}}{M_{Ga} \bigcup M_{Gb}}$$

where M_{Ga} represents the models in game *Ga*. The distribution of the overlapping rate of game pairs is displayed in Figure 5c. The graph reveals that the majority of pairs have an overlapping rate of less than 20%, suggesting that only a portion of the models in a game have been illicitly appropriated. However, we also observe spikes around 100% overlap rates. To gain a better understanding of this anomaly and the general cloning behaviors, we manually reviewed the overlapping models in hundreds of pairs. We pinpointed three main reasons for these clones:

• **Game Clone**. An attacker may repackage an existing game for various motivations, such as distributing malware or monetizing by substituting the advertisement revenue account in the game. The original game and the repackaged version would exhibit a high overlapping rate (typically around 100%). The spikes around 100% overlapping rates in Figure 5c can be attributed to such game clones. Table 3 lists the top five game clones identified in our study. Interestingly, some repackaged games have been uploaded under a different game category, making it challenging for the original developers to identify them.

• **Idea Clone**. If an original developer creates a popular game with an innovative concept, an attacker might try

to replicate this idea by creating a similar game, possibly appropriating some models from the original game in the process. Table 3 presents five such cases. For instance, Join Clash 3D is a popular game known for its unique logic and design. However, we discovered the game Join Clash 3D: Crowd Rush (with 5K installs), which offers remarkably similar content, albeit with lower quality. This game shares 8.84% of static models and 11.59% of animated models with the original game.

• **Model Clone**. At times, an attacker may be interested in a specific model. Unlike idea cloning, where the attacker creates a similar game, attackers typically steal the model and use it in a game belonging to a different category. We highlight five such instances in the last five rows of Table 3. For example, we observed that the game Subway Surfers (a highly popular game) and Bob the robber Runner (a less-popular game with 1K installs) share a unique dog model that we could not locate anywhere in the market. It is highly plausible that this model from Subway Surfers has been appropriated.

Case Studies. Next, we present two case studies to emphasize the potential security implications arising from 3D model cloning. As depicted in Figure 7, we illustrate two instances of cloning, with each instance comprising an original game (on the left) and its suspected unauthorized clone (on the right).

• Join Clash 3D is a high-caliber arcade game where players navigate a team in combat against rival factions. Its unique gameplay has contributed to its immense popularity on Google Play, boasting over 100 million installs. The screenshot on the right depicts another game that bears striking resemblance to Join Clash 3D, with several identical models, including the main characters. The gameplay also mimics that of Join Clash 3D, albeit at a notably lower quality.

• Subway Surfers is an immensely popular endless running mobile game developed by SYBO Games. With its addictive gameplay and visually appealing graph-

	Victim	Suspicion	Victim	Suspicion
Model				
Screenshot				
	Join Clash 3D	Join Clash 3D: Crowd Rush	Subway Surfers	Bob The Robber Runner

Figure 7: Cases for likely unauthorized 3D model clones.

ics, Subway Surfers has captivated millions of players worldwide. The game on the right features not only the inclusion of a dog character but also utilizes multiple identical 3D models found in the renowned game Subway Surfers. Furthermore, the gameplay of this game closely imitates that of Subway Surfers with noticeably lower quality.

The implications of unauthorized cloning can be severe. First, it infringes on the intellectual property rights of the original creators. Game developers dedicate substantial resources towards crafting these 3D models, and their unauthorized usage in other games constitutes a flagrant violation of their rights. Second, unauthorized cloning raises ethical concerns within the gaming community. The practice of cloning 3D models instead of developing original ones hampers innovation and creativity in the industry. In light of this, we assert the pressing need for robust protections against 3D model cloning. This includes the advancement of secure rendering techniques and tools capable of detecting cloned 3D models.

7 Limitations and Future Work

Expanding Detection Scope Beyond Mobile Games. Currently, 3DSCAN is designed primarily to focus on mobile games, specifically those created with the Unity engine. An immediate next step involves broadening 3DSCAN's scope to include other platforms, such as iOS, console, and PC

games. It's worth noting that 3D models extend beyond games, permeating fields such as film, television, medical imaging, industrial design, and even 3D printing [34]. Identifying 3D model clones within these sectors presents an exciting avenue for future research.

An emerging field of interest is the metaverse (e.g., [3, 4]), where users interact with virtual 3D worlds via a range of devices such as smartphones, tablets, and headsets. Given the integral role of 3D models in this domain, the development of methods to protect these models from theft, and to detect their unauthorized clones, will undoubtedly be a crucial research area.

Exploring Alternative Algorithms (e.g., Machine Learning). 3DSCAN utilizes an indexing-based approach to identify static 3D model clones using hash values. This assumes a lack of significant modification to the 3D models. Alternative strategies, such as transforming 3D models into vectors rather than a singular hash value, or implementing pattern recognition algorithms with artificial neural networks (e.g., [35, 36]), could be used to detect 3D model clones. The features 3DSCAN derives are fundamental (e.g., vertex order, face sequences), but there could be more representative 3D geometry features, as employed in 3D shape retrieval [37]. Techniques that use machine learning to automatically derive representative features could also be adopted (e.g., [38, 39]). We defer exploration of these techniques to future work.

Enhancing 3D Model Security. 3D model clone detection is but one facet of 3D model security. Additional problems, such as protection and detection, remain to be addressed. Protective measures could explore techniques for preventing 3D model extraction, such as 3D model encryption, obfuscation, and binding them with specific game code. Detection measures could propose solutions for thorough 3D model extraction from games and explore techniques (e.g., watermarking models) to detect unauthorized clones.

Anticipating Adversarial Clone Detection Evasion. Our study of 3D model clone detection signals the beginning of subsequent research into adversarial clone detection and evasion. Future researchers will likely develop methods to circumvent detection systems, testing the robustness of current tools. Malicious attackers may alter models to evade detection, prompting continued refinement of our detection methods in response to more advanced evasion techniques.

Integrating Our Tool in App Review. Major app platforms like Apple and Google could utilize our tool and approach as part of their app review processes, to proactively detect and reject games that utilize unauthorized or cloned 3D models. Future research could focus on integrating clone detection methods into these platforms, thereby bolstering their ability to prevent the submission of games with cloned 3D models. As we plan to open-source our tool and database, game developers can also incorporate our clone detection capabilities into their model theft detection pipelines.

8   Related Work

8.1   Software Security

Mobile Game Security. Computer games have evolved into one of the most substantial segments of the entertainment industry. A central theme in game security revolves around the dynamic between cheating and anti-cheating measures [40]. Game cheaters exploit vulnerabilities in game design (e.g.,[41]), distribution, and execution. In response, game defenders employ various patterns (e.g., inconsistency [42, 43]) derived from cheating behavior and recently, secure hardware (e.g., using SGX for game security [44, 45]) to counteract cheats. Specifically in the domain of mobile game security, Tian et al. [46] analyzed a range of cheats, such as memory modification and network traffic manipulation, and proposed a reference framework for mobile game defense. Most recently, PaymentScope [47] was proposed to automatically detect vulnerable in-app purchasing implementations in mobile games by performing static analysis. In contrast to these studies, our focus is on the security of game assets, particularly 3D model clones. There are also numerous studies addressing mobile app security, particularly in the area of app clone detection [48, 49]. However, it is not practical to apply them for 3D model clone detection due to the large number of games that get involved.

Code Similarity Detection. Numerous works have demonstrated the importance of code similarity detection. For instance, it has been used for: (1) Vulnerability detection [50] since buggy code may have been copied to other places. Identifying them based on code similarity detection can help to patch bugs; (2) Software theft detection [12] as code similarity detection can help to identify the code that has been stolen from open source project or commercial software; (3) Malware analysis [14] as analysts can use code similarity detection to identify the known code pieces which can minimize redundant efforts during the reverse engineering of malware. Similarly, cloning 3D models, especially without authorization, can cause harms, and in this paper we make a first step towards systematically understanding the prevalence of clones and its potential harms. Additionally, the detected clones could be used to mitigate or detect other attacks, such as masquerading attack [51] where one attacking AR app may generate the same model as other apps to mislead the users.

8.2   3D Model Security and Analysis

3D Model Watermarking. To trace the provenance and identify illegal copies, Ohbuchi et al. proposed 3D model watermarking [23], a technique that stealthily embeds hidden information within original 3D models. Over time, several enhancements have been proposed [52], including authentication watermark [53], forensics tracing watermark [54], and biometric watermarks [55]. Utilizing stereographic projection [56] for 3D object classification and retrieval offers a compelling method for clone detection, especially when tracing origins. 3DSCAN complements these works by identifying clones without the need for watermarking.

3D Model Retrieval. There is a substantial body of work focusing on 3D model retrieval [37, 57]. This involves the exploration of more intricate 3D geometric features to search for 3D objects within large corpora. Recent advancements in this field include the use of convolutional neural networks [38] and deep neural networks to learn and derive features [39]. While we could have utilized these methods to construct the index, we deemed them too resource-intensive and have reserved them for future exploration.

3D Model Diversification. The issue of 3D model cloning has been well-studied in the context of crowd simulation [58]. In such scenarios, the generation of a large quantity of individually diversified 3D objects is required. The presence of any clones among these 3D models [59] or motions [60] could diminish the user's experience. Therefore, visual variety [61] and context-aware motion diversification [62] have been proposed as solutions. Furthermore, ML-based approaches [63, 64] have shown promising results

in addressing these issues. However, their efficacy has not been validated on large-scale datasets. In contrast to these studies, 3DSCAN investigates different methods and focuses on different challenges.

9 Conclusion

We have presented 3DSCAN, a robust tool designed to identify 3D model clones within the mobile gaming sector. Our comprehensive experiments conducted on a sample size of 176,361 mobile games suggested that 63.03% of static 3D models and 37.07% of animated 3D models are subject to cloning. The strategic combination of value-insensitive normalization and heuristic-based methodologies validates our approach's efficacy in detecting not only ubiquitous 3D models, but unauthorized duplicates as well. While the prevalence of common 3D models accounts for a substantial number of clones, our evaluation results unequivocally highlight the presence of unauthorized copies. This is apparent through instances of game cloning, idea duplication, and model replication, as discerned by our heuristic-based algorithm.

Acknowledgment

References

[1] D. J. Eck, *Introduction to Computer Graphics*. Hobart and William Smith College, 2021.

[2] "This is how long it takes to learn 3d modeling," https://www.3dbiology.com/how-long-it-takes-to-learn-3d-modeling/.

[3] "Facebook goes meta: What is the metaverse and why is big tech obsessed?" https://www.cnet.com/tech/computing/facebook-goes-meta-what-is-the-metaverse-and-why-is-big-tech-obsessed/, (Accessed on 1/14/2023).

[4] "Metaverse market share | metaverse industry trend by 2028," https://www.emergenresearch.com/industry-report/metaverse-market, (Accessed on 1/14/2023).

[5] "Assetstudio," https://github.com/Perfare/AssetStudio.

[6] "Ue viewer," https://www.gildor.org/en/projects/umodel.

[7] "This game has stolen models, stolen maps and stolen music. how can i report it," https://www.reddit.com/r/Steam/comments/4ih3bc/this_game_has_stolen_models_stolen_maps_and/, (Accessed on 1/14/2023).

[8] "List of ripped models (stolen models) on the asset store - unity forum," https://forum.unity.com/threads/list-of-ripped-models-stolen-models-on-the-asset-store.1163021/, (Accessed on 1/15/2023).

[9] "I have found a model thief | cgtrader," https://www.cgtrader.com/forum/selling-buying-3d-models/model-thief, (Accessed on 1/15/2023).

[10] "X2! magnet simulator - roblox," https://www.roblox.com/games/3486025575/X2-Magnet-Simulator.

[11] "Magnet simulator," https://roblox.fandom.com/wiki/Community:LuaClifford/Magnet_Simulator.

[12] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 280–290.

[13] X. Wang, Y. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks," in *2009 Annual Computer Security Applications Conference*. IEEE, 2009, pp. 149–158.

[14] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in *2014 Eighth International Conference on Software Security and Reliability (SERE)*. IEEE, 2014, pp. 78–87.

[15] X. Lou and K. Hwang, "Collusive piracy prevention in p2p content delivery networks," *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 970–983, 2009.

[16] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and image processing*, vol. 14, no. 3, pp. 227–248, 1980.

[17] "Cgtrader - 3d models for vr / ar and cg projects," https://www.cgtrader.com/, (Accessed on 1/15/2023).

[18] "Sketchfab - the best 3d viewer on the web," https://sketchfab.com/3d-models/human-anatomy-male-explosive-view-7fff13988a094eaca9cb5057fb2fd1f2, (Accessed on 1/15/2023).

[19] "Utinyripper," https://github.com/mafaca/UtinyRipper.

[20] "Androzoo home," https://androzoo.uni.lu/, (Accessed on 1/14/2023).

[21] "Unreal engine: The most powerful real-time 3d creation tool," https://www.unrealengine.com/en-US/.

[22] "Unreal engine vs unity 3d games development: What to choose?" https://www.valuecoders.com/blog/techn ology-and-apps/unreal-engine-vs-unity-3d-games-d evelopment/, (Accessed on 1/14/2023).

[23] R. Ohbuchi, H. Masuda, and M. Aono, "Watermarking three-dimensional polygonal models through geometric and topological modifications," *IEEE Journal on selected areas in communications*, vol. 16, no. 4, pp. 551–560, 1998.

[24] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "Blake2: simpler, smaller, fast as md5," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 119–135.

[25] "Stadia - one place for all the ways we play - google," https://stadia.google.com/.

[26] D. Koller and M. Levoy, "Protecting 3d graphics content," *Communications of the ACM*, vol. 48, no. 6, pp. 74–80, 2005.

[27] D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, and R. Scopigno, "Protected interactive 3d graphics via remote rendering," *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 695–703, 2004.

[28] R. C. Merkle, "Protocols for public key cryptosystems," in *1980 IEEE Symposium on Security and Privacy*. IEEE, 1980, pp. 122–122.

[29] L. Chen, M. T. Özsu, and V. Oria, "Robust and fast similarity search for moving object trajectories," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 491–502.

[30] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series." in *KDD workshop*, vol. 10, no. 16. Seattle, WA, USA:, 1994, pp. 359–370.

[31] M. Vlachos, G. Kollios, and D. Gunopulos, "Discovering similar multidimensional trajectories," in *Proceedings 18th international conference on data engineering*. IEEE, 2002, pp. 673–684.

[32] "Scrapy | a fast and powerful scraping and web crawling framework," https://scrapy.org/.

[33] "Google play downloader via command line," https://github.com/matlink/gplaycli.

[34] J.-U. Hou, D. Kim, W.-H. Ahn, and H.-K. Lee, "Copyright protections of digital content in the age of 3d printer: Emerging issues and survey," *IEEE Access*, vol. 6, pp. 44 082–44 093, 2018.

[35] J. K. Basu, D. Bhattacharyya, and T.-h. Kim, "Use of artificial neural network in pattern recognition," *International journal of software engineering and its applications*, vol. 4, no. 2, 2010.

[36] K. Fukushima, "A neural network for visual pattern recognition," *Computer*, vol. 21, no. 3, pp. 65–75, 1988.

[37] J. W. Tangelder and R. C. Veltkamp, "A survey of content based 3d shape retrieval methods," *Multimedia tools and applications*, vol. 39, no. 3, pp. 441–471, 2008.

[38] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, "Multi-view convolutional neural networks for 3d shape recognition," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 945–953.

[39] Z. Zhu, X. Wang, S. Bai, C. Yao, and X. Bai, "Deep learning representation using autoencoder for 3d shape retrieval," *Neurocomputing*, vol. 204, pp. 41–50, 2016.

[40] G. McGraw, *Exploiting online games: cheating massively distributed systems*. Addison-Wesley, 2008.

[41] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh, "Openconflict: Preventing real time map hacks in online games," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 506–520.

[42] D. Bethea, R. A. Cochran, and M. K. Reiter, "Server-side verification of client behavior in online games," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 4, pp. 1–27, 2008.

[43] D. Liu, X. Gao, M. Zhang, H. Wang, and A. Stavrou, "Detecting passive cheats in online games via performance-skillfulness inconsistency," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 615–626.

[44] E. Bauman and Z. Lin, "A case for protecting computer games with sgx," in *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16)*, Trento, Italy, December 2016.

[45] S. Park, A. Ahmad, and B. Lee, "Blackmirror: Preventing wallhacks in 3d online fps games," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 987–1000.

[46] Y. Tian, E. Chen, X. Ma, S. Chen, X. Wang, and P. Tague, "Swords and shields: a study of mobile game hacks and existing defenses," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 386–397.

[47] C. Zuo and Z. Lin, "Playing without paying: Detecting vulnerable payment verification in native binaries of unity mobile games," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3093–3110.

[48] L. Li, T. F. Bissyandé, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 676–693, 2019.

[49] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 56–65.

[50] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 48–62.

[51] K. Lebeck, T. Kohno, and F. Roesner, "Enabling multiple applications to simultaneously augment reality: Challenges and directions," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 81–86. [Online]. Available: https://doi.org/10.1145/3301293.3302362

[52] M. Narendra, M. Valarmathi, and L. J. Anbarasi, "Watermarking techniques for three-dimensional (3d) mesh models: a survey," *Multimedia Systems*, pp. 1–19, 2021.

[53] J. Fridrich, M. Goljan, and A. C. Baldoza, "New fragile authentication watermark for images," in *Proceedings 2000 International Conference on Image Processing (Cat. No. 00CH37101)*, vol. 1. IEEE, 2000, pp. 446–449.

[54] K. R. Liu, *Multimedia fingerprinting forensics for traitor tracing*. Hindawi Publishing Corporation, 2005, vol. 4.

[55] R. C. Motwani, F. C. Harris Jr, and K. E. Bekris, "A proposed digital rights management system for 3d graphics using biometric watermarks," in *2010 7th IEEE Consumer Communications and Networking Conference*. IEEE, 2010, pp. 1–6.

[56] M. Yavartanoo, E. Y. Kim, and K. M. Lee, "Spnet: Deep 3d object classification and retrieval using stereographic projection," in *Asian conference on computer vision*. Springer, 2018, pp. 691–706.

[57] D. V. Vranic, D. Saupe, and J. Richter, "Tools for 3d-object retrieval: Karhunen-loeve transform and spherical harmonics," in *2001 IEEE Fourth Workshop on Multimedia Signal Processing (Cat. No. 01TH8564)*. IEEE, 2001, pp. 293–298.

[58] D. Thalmann, H. Grillon, J. Maim, and B. Yersin, "Challenges in crowd simulation," in *2009 International Conference on CyberWorlds*. IEEE, 2009, pp. 1–12.

[59] R. McDonnell, M. Larkin, S. Dobbyn, S. Collins, and C. O'Sullivan, "Clone attack! perception of crowd variety," in *ACM SIGGRAPH 2008 papers*, 2008, pp. 1–8.

[60] Y. Li, M. Christie, O. Siret, R. Kulpa, and J. Pettré, "Cloning crowd motions," in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Citeseer, 2012, pp. 201–210.

[61] S. Oxspring, B. Kirman, and O. Szymanezyk, "Attack on the clones: managing player perceptions of visual variety and believability in video game crowds," in *International Conference on Advances in Computer Entertainment Technology*. Springer, 2013, pp. 356–367.

[62] Q. Gu and Z. Deng, "Context-aware motion diversification for crowd simulation," *IEEE Computer Graphics and Applications*, vol. 31, no. 5, pp. 54–65, 2010.

[63] K.-H. Liu, P.-Y. Chiang, and C.-C. J. Kuo, "A machine learning approach to 3d model retrieval," in *Proceedings of Asia-Pacific Signal and Information Processing Association Annual Submit and Conference*, 2011, pp. 13–17.

[64] H. Chen and B. Bhanu, "Efficient recognition of highly similar 3d objects in range images," *IEEE Transactions on Pattern analysis and machine intelligence*, vol. 31, no. 1, pp. 172–179, 2008.