# Towards Providing Security For Mobile Games

Amir Yahyavi[†], Jeffrey Pang[‡], and Bettina Kemme[†]
[†]McGill University | [‡]AT&T Research

## ABSTRACT

Mobile security for games is often ignored when discussing architectural design of mobile systems. Substantial increase in the power and networking capabilities of smartphones has led to the emergence of mobile games, and mobile gamers now outnumber "core" gamers. These games include many genres of gaming with different and new requirements. Here we discuss existing and potential new avenues of cheating in mobile games and suggest several solutions on how different services can be provided by carriers, ecosystems, and developers to improve the security.

## 1. INTRODUCTION

Mobile games, their requirements, and effects are often treated as an afterthought in architectural design of mobile systems. Nearly 136 million people in the U.S. now own a smartphones (58% of the mobile subscribers) with numbers increasing fast. Around 34% of all mobile subscribers play mobile games [2] and this rate is much higher in the rising smartphone user group. Many of these smartphones incorporate several sensors such as a GPS chip, an accelerometer, a digital compass, a proximity sensor, an ambient light sensor, a microphone, and a camera. This variety of sensors has given rise to a huge number of innovative mobile applications including games: These sensors, along with multitouch support, turn the smartphone into an effective gaming controller. Increase in the processing power of smartphones further transforms them into full fledged gaming platforms. in fact, mobile games already provide 15% of the overall gaming revenues (10 Billion) and are fast on the rise [8]. They also provide higher profits, in terms of man years invested, than console and desktop games. As a result, architectural support for mobile game security becomes important [7].

Cheating occurs when a player gains an unfair advantage over other players or gains unauthorized access to items (e.g., paid content or unlocked achievements). This can result in loss of revenue for developers as it might discourage honest players from playing or purchasing items and turn them into cheaters as well. Effectively addressing mobile game security requires support from the operating system, carriers, and developers.

While mobile ecosystems can be fairly controlled, given the prevalence and legality of modifying the operating systems, e.g., jailbreaking and custom ROMs, cheaters can easily gain access and tamper with game codes. Furthermore, mobile devices can create new opportunities for cheating such as faking sensor readings. Currently, it is fairly easy to fabricate or lie about readings from the sensors. These fake readings can include modifying the pictures taken by the cameras or fake locations by fabricating readings from GPS sensors [10].

Given the new challenges that arise in mobile environments, it is crucial to understand the implications of mobility on cheating and the options to avoid or detect it. Therefore, this paper proposes a comprehensive analysis of how existing cheating mechanisms change in mobile environments, how new opportunities arise, and how they all impact mobile game play. From there, we outline how a framework of stakeholders, consisting of game developers, game engine, operating system ecosystems and carriers can help in securing mobile services for games.

## 2. BACKGROUND

Many of the existing cheating mechanisms can exist in mobile environments. Here we focus on cheating types that exploit vulnerabilities in the game design and do not consider other forms of abuse such as *social misconduct* and *collusion* outside the game environment (e.g., exchanging information through chatting).

### 2.1 Existing Cheating Mechanisms

We provide a summary of commonly used cheats, shown in Table 1, as well as a short definition [15]. We further highlight the threats that are different in mobile environments. Some cheating mechanisms get worse as a result of *characteristics* of mobile environments, while some become harder to *detect*. One of the advantages of mobile environments, however, is that due to the limited capabilities of the smartphones the cheating softwares can, in some cases, be less complicated. On the other hand, if validations are to be run on the smartphones, they too will have limited resources available and therefore, tests need to be less complicated than their desktop counterparts.

The main causes for worse security are:

- *Exacerbated vulnerabilities:* Cellular networks make it easier to perform some types of cheating. For example, sniffing network packets can be easier, specially if

players are relatively close to each other.

- *Harder Detection & prevention:* Higher latencies, loss, and jitter makes detection of cheaters harder. For example, higher latency makes it harder to detect cheaters who withhold their updates and/or delay them.

We divide cheating types into the following three categories [15] and then discuss how they are affected as a result of architectural differences in mobile systems: (1) Interrupting information dissemination (2) Illegal game actions (3) Unauthorized information access.

**Interrupting information dissemination:** Includes dropping, corrupting, delaying, changing the rate of the updates, or sending wrong or inaccurate information. This helps a player blind or confuse others about his current state. As a result, the chances of his avatar being targeted is decreased or he gains an unfair advantage in his attacks.

- *Disconnections & Loss:* Given higher disconnection and loss rates in mobile environments, *detection* of cheating types such as *escaping* where the player terminates the connection in order to escape imminent loss, or *blinding opponents* where updates are not sent to some players, is harder.
- *Latency:* Higher latency in mobile environments makes it harder for different *time cheatings* mechanisms to be detected. For example, in *look ahead cheating*, cheater deliberately delays his updates to base his actions on those he receives from others or sends messages with fake old time stamps. Similarly, *fast rate cheat* where higher rate of actions is submitted, is harder to detect.
- *Retransmissions:* Given easier sniffing in wireless environments (see below), *replay attacks* are easier to perform in which the cheater replays the messages sent by other, e.g., a health reduction, to damage others.

**Illegal game actions:** A cheater can circumvent the game physical laws (e.g., limited velocity) and unduly change his state (e.g., increase his health or ammunitions) by tampering with the game code. Players may also use third party software to perform game tasks such as aiming.

In closed mobile environments, it's harder to perform modifications to the game, e.g., *app code tampering*. However, modified apps can be used through jailbreaking. In addition, open source operating systems such as Android can be more vulnerable through use of custom ROMs. Similarly, *aimbots* can be used to facilitate aiming. Given limited memory of these devices, memory injection tools can be used more effectively to modify critical game values.

**Unauthorized information access:** Available information in the game that is not supposed to be disclosed to the players (e.g., position of players behind walls) can be exploited to increase the chances to kill other players or to foresee danger and evade. Due to the nature of wireless medium, it's easier to *sniff* the packets sent. This information can be used for cheating. In addition, a simple *rate analysis* can help cheaters in gaining unauthorized information in the game (e.g., in multi-resolution games where updates are sent at different rates). Furthermore, players' communications can be more easily *jammed* due to the nature of wireless communications, especially by players in close physical proximity. This can be used simply as a DoS attack or to gain unfair advantage in the game. These advantages include preventing other players from receiving necessary game information or performing their actions.

## 2.2 Existing Security Mechanisms

Many cheat *prevention* (proactive) and *detection* (reactive) techniques have been proposed to address cheating in networked games [13]. Most of these techniques can be used in mobile environments: Cryptographic measures, signatures and checksums, are effective in eliminating message sniffing and illegal modifications.

Dealing with delayed and dropped messages is harder in mobile environments due to higher latency and loss. The state of the player in the game (i.e., winning, losing) along with statistical analysis (frequency of disconnects when player is losing) can be used in improving the detection of escaping cheats. One approach to detect these cheaters is by delaying the updates and determining who waits for the updates (e.g., AC/DC [4]).

Use of CAPTCHA texts has been proposed[5, 11] to deal with aiming bots. CAPTCHA texts are an example of a Turing test. Distinguishing aiming bots from competent players based solely on the quality of their game is difficult. Learning approaches have been proposed that partially address this problem [1]. In addition, several tools exist to detect tampering with the game code or to detect cheating processes running in the memory which can be applied to mobile environments as well. PunkBuster [9] and Valve Anti-Cheat (VAC) [12] are examples of such systems. In addition, hardware-based solutions [3, 10] can be used to perform stealth measurements for verification.

Penalization typically happens in the form of expelling players from their current games and/or banning them from accessing future sessions.

## 2.3 New Gaming & Cheating Mechanisms

In addition to existing cheating opportunities, mobile platforms are vulnerable to new cheating types as well. This can be a result of new vulnerabilities introduced by the mobile environment or due to new gaming environments such as *augmented reality* (AR) [1] games. These games feature live view of a physical real-world environment whose elements are augmented by computer-generated sensory input such as sound, video, graphics

[1] g.co/projectglass

Table 1: Popular cheating mechanisms in distributed games

| Type | Name | Description | Mobile Games |
|---|---|---|---|
| **Disruption Of Information Flow** | Escaping | Terminating the connection in order to escape imminent loss | Worse |
| | Time Cheating (look ahead) | Delaying the updates to base one's actions on those received from others | Worse |
| | Network Flooding | Overflowing the game server to create lags and disrupt game play | Better |
| | Fast Rate Cheat | Mimicking a rate of game event generation that is faster than the real one | Worse |
| | Suppress-Correct Cheat | Dropping consecutive updates, then sending an invalid update afterwards | Worse |
| | Replay Cheat | Resend signed & encrypted updates of a different player | Worse |
| | Blind Opponent | Dropping updates to opponents, blinding them about the cheater's actions | Worse |
| **Invalid Updates** | Client-side Code Tampering | Modifying the client-side code to get an unfair advantage | Same |
| | Aimbots | Using an intelligent program to provide it with automatic weapon aiming | Same |
| | Spoofing | Sending messages, pretending to be a different player | Worse |
| | Consistency Cheat | Sending different updates to different players | Same |
| **Unauthorized Access** | Sniffing | Logging & accessing different information sent across the network | Worse |
| | Maphack | Hacking to see through walls and obstacles | Same |
| | Rate Analysis | Analyzing the updates rates to detect players attention and escape | Worse |

that constitute the game items. As a result new cheating opportunities emerge:

- *Faking sensor readings:* it is relatively easy for malicious applications to fabricate or lie about readings from these sensors as most systems currently lack software or hardware checks [10]. By lying about the gyroscope, accelerometer, and other sensor readings, players can gain unauthorized access to information or gain an unfair advantage. In addition, these sensor readings are used as controller input for the game. Therefore, by faking sensor readings the cheater can use cheating programs such as auto aiming tools.
- *Faking Location:* One main difference between smartphones and current game consoles or PCs is in providing real world location services. This provides the potential for location based games (e.g., geo-caching) as well as augmented reality games. In such cases, faking the location can result in access to information such as virtual items in the game that are only available to players in close proximity to the item.
- *Exploiting Consistency Techniques:* Many techniques have been proposed or optimized to deal with higher loss and latency and lower bandwidth in mobile networks. These techniques can be exploited by the cheaters. For example, dead reckoning used to deal with high latency and loss can be abused in suppress-correct cheats.
- *Micro-Transactions:* one of the most popular forms of gaming is *freemium* models where the game is offered for free but in game items and enhancements are offered for a purchase. Cheating in collecting or purchasing these items is one the most common forms of cheating.

# 3. TRUST MODEL & DESIGN PRINCIPLES

## 3.1 Trust Model

We identify five main components involved in securing mobile games: (1) Game app, (2) Game server, (3) Carrier, (4) OS, (5) Phone hardware. Since we are focusing on the *mobility* aspects of providing security and do not discuss back-end security, we assume that the game server and carriers can be trusted. Given jailbreaking and custom ROMs in which security features can be turned off, we assume the operating system cannot be trusted. Tampering with game code is commonplace in cheating in desktop and mobile games, therefore, apps cannot be trusted. Given the ease of faking phone hardware readings [10], unless trusted platform modules (TPM) exist, we assume the phone hardware cannot be trusted.

## 3.2 Design Principles

Several design principles have to be taken into account in designing an efficient anti-cheating mechanism.

***Simplicity:*** One of the main advantages of mobile games is their rate of return on investments, i.e., mobile games are generally cheaper to make and return a higher profit. Complex cheat detection and prevention technique make the cost of development and maintenance much higher.

***Low Overhead:*** Excessive network use can cost the customer money in addition to draining the battery. Similarly, high computational costs degrade the game performance and drain the battery.

***Privacy Concerns:*** Sending the necessary information for verification purposes can raise privacy concerns for the players as players may need to submit additonal location information, pictures, list of nearby wireless networks, cell information, etc. Privacy and security present a trade off. Several protocols suggested that provide some level of privacy while collecting data that rely on aggregating the data to provide privacy or using negative surveys [6].

***API Abuse:*** Security techniques can often themselves be abused by cheaters. In addition, they may introduce new vulnerabilities for the environment. For example, an API providing location verification using publicly available information about the player can be abused by other players to locate players they are not supposed to know about.

To devise a comprehensive security solution we will first explain how different components involved in mobile gaming can help in prevention and detection of cheating. Here, we mostly focus on the new or exacerbated threats in mobile environments, however, in some cases solutions may apply to desktop versions of the games as well. These components each can play a role in improving the security:

Table 2: Suggested API for securing game services

| Category | Method | Description |
|---|---|---|
| **Validate Inf. Flow** | VerifyNetStats(Netstats *nets, Loc *loc) | Verifies the network connectivity stats such as loss and latency |
| **Validate Location** | ValidateLocWiFi(LocInfo *Loc, WifiInfo *SSIDs) | Validates location against WiFi info e.g., list of nearby SSIDs |
| | ValidateLocCell(LocInfo *Loc, CellInfo cell) | Validates location against the cell tower information |
| | VisualLocValidation(GeoTaggedPic pic) | Validates the player's location through visual verification |
| | ValidatePath(Path *path) | Validates accessibility of player's path and location |
| **Validate Readings** | PhysicalVerification(ActionList *alist, SensorReadings *sr ) | Validates last $n$ actions using sensor readings for duration $t$ |
| | VerifyLocQuestion(QuestionHash qh, Answer ans) | Verification by questions about local information |
| | PhysicsCheck(SensorReading *sr) | Validate state updates or predictions according to physical rules |
| | TuringTest(SensorReading *sr, Credentials *cr) | Novel Turing tests to ensure human player |
| | TPMSign(TPMSign s, Message m) | TPM verifies running specific hardware and/or software |
| **Validate Access** | FaceRecognition(UserPic pic) | Authentication and Bot prevention |
| | ValidateTransaction(Transaction *Tr) | Validate micro-transactions |

- ***Game Apps:*** Can best perform verifications to ensure that the platform (e.g., OS, hardware) is not compromised. Good authentication, authorization, and cross-checking actions all rely on a well designed game app.
- ***Game Servers:*** In addition to performing authentication and maintaining players' records, servers can perform verifications by demanding additional information when players request certain data. More complicated detections techniques can also be delegated to servers.
- ***Operating Systems:*** Other than the general security mechanisms such as memory isolation, the OS can ensure the correctness of sensor readings as well as higher security for financial transactions.
- ***Carriers:*** Are in the best place to verify, independently from smartphone readings, the physical location of the phone and perform authentication for the user. They can also provide statistics on network quality.

# 4. SECURITY ARCHITECTURE

We propose a security architecture in which different stakeholders provide support to prevention and detection of location cheating, fake sensor readings, and disruption of information flow. The security architecture is exposed as a set of APIs where each method tackles a specific issue.

## 4.1 New Security Mechanisms

A summary of the proposed APIs is presented in table 2. We provide cheating scenarios that explain how cheating is performed and discuss how these APIs that can address the issue. In addition to discussing these new methods, we further elaborate on how they can be implemented using the currently available APIs as well as suggesting new necessary architectural support from OS and carriers to help further improve their accuracy. As an example we focus on the APIs currently provided by two large ecosystems: AT&T[2] and Android operating system[3], however, other popular operating systems and carriers provide similar functionalities.

## 4.2 Location cheating

[2] `http://developer.att.com/developer`

[3] `http://developer.android.com/reference/`

Consider a player that has to be in a certain location to find an item, e.g., a geo-cache, unlock an achievement or to see a virtual item in an AR game [7]. In this case players have incentive to fake their location and tools to *fake* location are readily available in app stores or trough third party channels. To deal with this the game server can require additional location information about the player to be submitted along with the request for local information. In such a case, when the game server suspects a player is a cheater or it is simply performing a random audit, this additional information is used for verification. Currently there are three methods available for determining the physical location of a player: (1) *Cell ID (cell tower)*, (2) *WiFi*, (3) *GPS*, GLONASS, or other future platforms. Any combination of these methods can be used to verify the location reported. In order to verify the location these sensors can be cross checked or third party map information can be used:

- *WiFi verification:* available wireless network information can help identify the location of the player (Wi-Fi-based positioning system (WPS)). Sending a list of nearby available wireless SSIDs and their strength is currently extensively used in helping increase the accuracy and speed of the location detection (e.g., Google's WPS database). This can be used to also verify the location of the user where player are required to occasionally or upon request send in a list of nearby SSIDs that will be matched against existing databases. Android currently provides WiFi location services through its `LocationManager` API which provides a 200 meter or better accuracy, as does AT&T, and any WiFi location service can be usable. Proposed `ValidateLocWiFi` provides such an interface.
- *Cell verification:* connection to a cell tower determines that the player is located within the range of the tower. In addition, its signal strength can help approximate the player's location and cell triangulation can be performed by carriers. While this information is useful, given long range of some cell towers it might not be accurate enough. Furthermore, this information may not be directly accessible by game companies. Android `LocationManager` provides such an interface. Similarly AT&T provides Network-based location services in its API. The accuracy depends on the cell density in an

area and can be up to several thousand meters. Proposed `ValidateLocCell` provides an interface for validating reported location information against cell tower information available which is provided by the operating system, carrier, or other openly available databases. Listing 1 provides a sample code on how these two APIs can be used.

- *Question answering:* the player can be asked a question about his current location in which case he should be able to answer if he's in the right spot. Many map services provide "local places information" APIs which can be used. An example would be: *"name the pizzeria that you're in front of?"*. Google Places API (`PlacesService`) provides nearby location information as does AT&T. Other similar ecosystems exist. `VerifyLocQuestion` provides such an interface.
- *Accessibility:* information about accessibility of the location sent by the player can help determine cheaters. For example, unreachable positions (e.g., middle of a lake) reported as well as the path between movements (e.g., crossing a river with no bridge) can be verified using accessibility information. `ValidatePath` provides an interface for such validation. Android provides Google maps API, as do many other ecosystems. Third party open solutions, e.g., OpenStreet Maps, also exist.
- *Visual Verification:* the smartphones' camera can be used in visual verification of players' locations. This will become more usable with use of AR devices. Visual verifications can also be done against available databases of location pictures (e.g., Google Street View and goggle or Microsoft Synth). `VisualLocValidation` provides such an interface. Google goggles and project glass currently does not yet provide an API, however, future API is expected. Google also maintains a large street view database that can be used.

Listing 1: Location Request Verification

```
1  % Game
2  move()
3  server.getLocalItems(gps, cellid, ssidlist)
4
5  % Game Server
6  def getLocalItems(gps, cellid, ssidlist):
7      if randomAudit == True or trustRating < threshold:
8          validLoc = validateLocation(gps, cellid, ssidlist)
9          if validLoc:
10             return items
11         else:
12             //player.lowerTrust() or player.penalize()
13
14 def validateLocation(gps, cellid, ssidlist):
15     cellValid = carrierAPI.ValidateLocation(gps, cellID)
16     ssidValid = WiFiLocAPI.ValidateLocation(gps, ssidlist)
17     if cellValid == Valid and ssidValid == Valid:
18         return Valid
19     return Invalid
20
21 % Carrier
22 def ValidateLocation(gps, cellID):
23     cell = carrierCells.get(cellID)
24     if distance(cell.loc, gps) < cell.range:
25         return Valid
26     return Invalid
```

## 4.3 Fake Sensor Readings

Smartphones can act as a game controller, therefore, faking different sensor readings can give the players an advantage. This can be achieved using an emulator (e.g., Android emulator) that hosts the operating system and generates readings. It can also be done using a compromised build of the operating system that allows for such readings to be generated by a bot. In such a scenario a cheater uses aiming bots and similar tools that can generate the necessary sensor readings to control a player's avatar. The game has to verify the hardware and OS it is being run on in order to ensure they are valid and not compromised.

- *Turing Tests:* are used to verify that a human is indeed playing and not a bot. Given the range of sensors available in smartphones, new forms of Turing tests can be developed. For example, a novel CAPTCHA test can be designed that asks a player to lift and move the phone in a certain pattern or touch a certain point on the screen. Accurate sensor readings (e.g., Android `SensorManager`) can be easily used to verify the movement pattern. `TuringTest` provides a general interface for this verification which can be provided by the operating system.
- *Physics verification:* during gameplay, where the smartphone is used as a controller, detection of *micro movements* as the result of interaction with the phone, and matching them against the actions inside the game can be used to detect aimbots and similar cheating software. These movements have already been shown to provide detailed interaction information such as detecting typing[14]. Furthermore, different sensors on the smartphone can be used to verify each other's readings, e.g., location changes should be reflected in the accelerometer and gyroscope readings that can be used to verify direction and acceleration of the player. Most operating systems provide detailed sensor readings (e.g., using the `Hardware & SensorManager` APIs in Android). However, to verify the reported readings to the game (e.g., by an aimbot), the OS needs to maintain a short time history of readings. This has to be securely managed as otherwise it can be exploited itself. `PhysicsCheck` provides such checks which can be provided by the OS.
- *Hardware based security:* mechanisms to secure sensor readings through hardware and software checks have been proposed [10]. Trusted Platform Modules (TPM) can verify that a specific software has been booted on a specific hardware. Similar techniques to the ones suggested for hardware based security measures in desktop games can be used: [3] proposes use of hardware-based,

stealth measurements such as Intel's Active Management Technology (AMT)[4] that has been used in rootkit detection in operating systems. In this approach, a tamper-resistant processor resident on a client and isolated from the system's primary processor is used to perform randomly-timed measurements *underneath* the host's software stack to detect cheats. `TPMSign` provides such an interface requiring hardware support.

- *Visual Identifications:* ubiquitous presence of cameras in smartphones makes it possible to use them to visually confirm the fact that a human is present as well as his identity. Face unlocking technologies, that already exist in the smartphones, can be provided to 3rd party apps via APIs. `FaceRecognition` provides an interface for authentication and authorization of the players. This can be provided by the operating system, or third party providers. Google currently provides face unlocking services but it is not yet presented as an API to the developers. Third party solutions also exist.

### 4.4 Disruption of Information Flow

Cheaters can disable they communications (e.g., by simply turning of WiFi or mobile data) enabling them to perform *escaping* or *suppress-correct cheats* cheats. Determining whether the network statistics are acceptable, heavily depends on the network conditions that can be verified by the carriers. Only carriers have enough information about signal strength, network speed, average and burst loss rates, and other network statistics in certain locations. In such a scenario the game server or other players can use the carrier API to verify their recorded network statistics of the player (e.g., update loss they have experienced) and decide whether it falls within accepted threshold (extracted using honest players) or not. `VerifyNetStats` provides such an interface using available network statistics which can be best provided by carriers. AT&T currently provides basic APIs that can provide information about *device capabilities*, however, it doesn't provide network information to apps. Android provides considerable network information available to apps. For example, `TelephonyManager` and `NetworkInfo` provides detailed information about network and its type, speed, state, etc. and `ConnectivityManager` provides information about connectivity type (e.g., wireless, cellular, etc.). `TrafficStats` provides network statistics such as number of sent and received packets and similar detailed statistics. These information can be required by a game server when performing verifications.

**Transaction security:** Invalid in-app transactions are one of the most popular forms of cheating. However, most transaction security problems can be solved by correct implementation of billing and authentication services. `ValidateTransaction` provides an interface for further validation of in-game transactions. This can be provided by the app-store ecosystem (usually by the operating system provider, e.g., Google Play), or third party eco-systems (e.g., Amazon). Carriers, the game ecosystem itself or other third party services (e.g., PayPal) can be used for checking the credentials as well. Google Play store provides `IInAppBillingService` that is used for in app billing. AT&T provides `Notary Management` API that can further be used for in-app payments. It also enables signatures and encryption for secure communication. *Visual identification* (e.g., Face Unlocking in Android) can further help in authentication of users.

[4]Intel vPro Technology `http://www.intel.com/content/www/us/en/architecture-and-technology/vpro/vpro-technology-general.html`

## 5. CONCLUSION

Mobile games are a fast growing industry. Addressing security concerns in these environment requires more attention from all stakeholders, i.e., carriers, platforms, game developers, and hard designers, and requires support from all them.

## 6. REFERENCES


[1] K. Chen, H. K. Pao, and H. C. Chang. Game bot identification based on manifold learning. In *NETGAMES*. ACM, 2008.

[2] Comscore smartphone report. `http://www.comscore.com/Insights/Press_Releases/2013/5/comScore_Reports_March_2013_U.S._Smartphone_Subscriber_Market_Share`, 2013.

[3] W. Feng, E. Kaiser, and T. Schluessler. Stealth measurements for cheat detection in on-line games. In *NETGAMES*. ACM, 2008.

[4] S. Ferretti and M. Roccetti. AC/DC: an algorithm for cheating detection by cheating. In *NOSSDAV*. ACM, 2006.

[5] P. Golle and N. Ducheneaut. Preventing bots from playing online games. *Computers in Entertainment*, 3(3):3, 2005.

[6] W. He, X. Liu, H. Nguyen, K. Nahrstedt, and T. Abdelzaher. Pda: Privacy-preserving data aggregation in wireless sensor networks. In *INFOCOM*, pages 2045 –2053, may 2007.

[7] W. He, X. Liu, and M. Ren. Location cheating: A security challenge to location-based social network services. In *ICDCS*, pages 740–749, 2011.

[8] NPD, Gfk, IDG, MSFT Estimates. `http://news.xbox.com/2013/05/x360-aaron-greenberg-industry-growth`, 2013.

[9] PunkBuster: the original anti-cheat system for online multiplayer games. `http://www.evenbalance.com/`, 2013.

[10] S. Saroiu and A. Wolman. I am a sensor, and i approve this message. In *HOTMOBILE*, pages 37–42. ACM, 2010.

[11] T. Schluessler, S. Goglin, and E. Johnson. Is a bot at the controls?: Detecting input data attacks. In *NETGAMES*, pages 1–6. ACM, 2007.

[12] VAC: valve anti cheat. `https://support.steampowered.com/kb_article.php?p_faqid=370`, 2013.

[13] S. D. Webb and S. Soh. Cheating in networked computer games: a review. In *DIMEA*, pages 105–112. ACM, 2007.

[14] Z. Xu, K. Bai, and S. Zhu. Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. In *WISEC*, pages 113–124. ACM, 2012.

[15] A. Yahyavi, K. Huguenin, J. Gascon-Samson, J. Kienzle, and B. Kemme. Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In *ICDCS*, 2013.