Is Runtime Verification Applicable to Cheat Detection?

Margaret DeLap, Björn Knutsson, Honghui Lu, Oleg Sokolsky,
Usa Sammapun, Insup Lee and Christos Tsarouchis
Department of Computer and Information Science, University of Pennsylvania
{delap,bjornk,hhl,sokolsky,usa,lee,cstsarou}@cis.upenn.edu

ABSTRACT

We investigate the prospect of applying runtime verification to cheat detection. Game implementation bugs are extensively exploited by cheaters, especially in massively multiplayer games. As games are implemented on larger scales and game object interactions become more complex, it becomes increasingly difficult to guarantee that high-level game rules are enforced correctly in the implementation. We observe that although implementing high-level rules in code is complex because of interference between rules, checking for rule compliance at runtime is simple because only a single rule is involved in each check. We demonstrate our idea by applying the Java-MaC runtime verification system to a simple game to detect a transaction bug that is common in massively multiplayer games.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Validation

General Terms

Design, Reliability, Security, Verification

Keywords

Multiplayer game, Cheat detection, Runtime verification.

1. INTRODUCTION

Cheating is perpetrated by game participants to gain undue advantage for themselves, harm other players, or cause general mayhem and disruption. Although there is no universal agreement on what constitutes a "cheat", cheat prevention is regarded as crucial to the quality of service of online multiplayer games.

Our focus is on Massively Multi-player Games (MMGs), since they tend to exist in persistent worlds and run on subscription models. This means that *cheating* has lasting effects, as well as threatening the revenue model of the companies running the games, since players tend to abandon games where cheating isn't curbed. Additionally, since there exists, through the unsanctioned sale of in-game goods and currency, both an unofficial market and exchange rate for the proceeds of cheating, the incentive for cheating is extremely well defined in real-world terms.

MMGs are commonly supported by a client-server architecture. They are vulnerable to cheating due to their complex rule sets and implementations. As games become more complex and game objects — and interactions between them — multiply, it becomes increasingly hard to guarantee that high level game rules are enforced correctly in the implementation.

Our approach detects cheating at runtime based on the observation that cheating can be expressed as deviant program behavior. Implementing certain anti-cheating rules in the game code itself may require complex programming, incur high performance cost, or result in unpopular game policies, while dynamically verifying those same game rules is relatively simple. In particular, although different events may interfere with one another, we need to verify only whether a particular rule is enforced or not.

Runtime verification has been used, with success, to verify dynamic properties of real-time control systems consisting of sequential events [6, 7, 10]. It is potentially applicable to games because they are essentially control systems, albeit with simultaneous events and more complex object behaviors.

This paper first surveys cheating, in particular those forms exploiting implementation bugs. We then demonstrate that the correctness of transactions, a common interaction in games, can be verified using an existing run-time verification framework with a simple rule checking logic.

2. GAME ARCHITECTURE

Online games come in two general flavors: those with large persistent worlds, often called Massively Multiplayer Games (MMGs), and those with smaller transient worlds, often just called (classical) Multi-Player Games (MPGs). MMGs are exemplified by Massively Multiplayer Online Role-Playing Games (MMORPG) like EverQuest, while MPGs include First Person Shooters (FPS) like Quake III and Real-Time Strategy games (RTS) such as WarCraft III. MMGs like Lineage, developed by NCsoft, have recorded two million registered players, and 180K concurrent players in one night. Conversely, a large and long MPG would have fewer than a hundred players playing in a world lasting up to a few hours.

The core of online games is the use of computing engines to *simulate* entities that interact with each other, where the entities are modeled as objects. An object consists of a collection of fields (state variables and attributes) and a set of methods that model the behavior of the component. The relationship of objects to one another is specified through three approaches:

- Attributes that indicate those state variables and parameters of an object that are accessible to other objects.
- Association between objects (*e.g.*, one object is part of another).
- Interactions between objects that indicate the influence of one object's state on the state of another object.

A typical multiplayer game *world* is made up of immutable landscape information (*terrain*); characters controlled by players (*PCs* or *avatars*); *mutable objects* such as food, tools, and weapons; mutable landscape information (*e.g.*, breakable windows); and non-player characters (*NPCs*) controlled by automated algorithms. NPCs can be either allies, bystanders or enemies, and are not always immediately distinguishable from PCs, except by their interaction.

The state of a player includes his position in the world and the state of his game avatar, such as its abilities, health and possessions. Avatar states are often persistent and can be carried along from one game session to another. Similar states exist for NPCs and game objects. In general, a player is allowed three kinds of actions: *position change*, *player-object interaction*, and *player-player interaction*. A player interacting with objects (including NPCs) or other players may, subject to game rules, change objects' state as well as the state of his avatar, *i.e.*, drinking from a bottle would empty the bottle and decrease the thirst of the avatar.

3. TYPES OF CHEATS

Cheaters can exploit many different aspects of a game; we have attempted to classify the vulnerabilities into three categories:

- *Rule design* – loopholes in game rules are similar to those in real world laws. Cheaters can exploit them to gain advantage at the cost of disturbing the game economy.
- *Architecture* – putting function and data on the client can give the client opportunities to cheat, usually by changing the client side code or observing supposedly hidden states.
- *Implementation* – a specific implementation may fail to enforce rules, regardless of rule design and game architecture.

The three aspects are orthogonal from the cheaters' point of view, and some cheats may exploit combinations of them. Which type of exploit is most common often depends on the game genre — role-playing games tend to have complicated rules and bugs in rule design, while FPS games have more cheats due to client side data and functionality designed for latency hiding; both genres are susceptible to implementation bugs.

Defining what constitutes cheating becomes easier as we move from high level game rules to low level implementations. Clever exploits of game rules can be tolerated unless they become a nuisance. At the implementation level, on the other hand, exploiting an implementation bug is relatively clear cut. We will now give concrete examples and detailed discussions of exploits of these three types.

3.1 Rule design

Game rules govern objects and their attributes as well as the variety of possible game events. A rule either specifies the bounds of an attribute value, or specifies the precondition and postcondition of an event. For example, the preconditions for a "purchase" event can be the location of two participants and their respective possessions. The rule specifies the "price" of each merchant, and how each participant's possessions are updated after the purchase. Good rule design will make the game challenging, encourage interaction, and balance the overall game economy. It takes experience and extensive testing to make good rules. Even the genre and goals of the game have effects on what types of cheating are easy or useful. Two common cheats that exploit rule design flaws are automatic play and collusion. Other cheats that make use of the interaction between rule design and implementation bugs are described in Section 3.3.1.

Automatic play. Mechanical and repetitive game events can be automated by the player. There are various forms of automatic play and some may be seen as more annoying than others. The usefulness of automatic play depends on the overall type of game rather than on specific rules.

One example of automatic play is reflex augmentation. By automating tasks that would otherwise involve some human reaction time delay, such as aiming, it produces superior results that otherwise would be impossible to achieve. Similarly, a script that automatically perform a series of simple task defeats the common use of time/reward efficiency to deter players from repeatedly doing simple, low-yield actions to advance.

In both cases, these cheats ruin the game experience for other players as well as upsetting the game balance. If many of the players are in reality computer controlled or computer augmented, the player may as well play a single-player game. Also, if anyone can become a high-level player with no real skills, this erodes the sense of achievement a player gets from playing the "real" game.

Collusion. Collusion occurs when players have an agreement to cooperate that is made outside of the normal play of the game. In any given game, some forms of cooperation, such as teaming up, may be entirely normal and legitimate, while others are more questionable. For example, a cheater might persuade other people to create new characters in a certain game for the sole purpose of having them lose to him. In general, since collusion involves out-of-game cooperation and communication, in-game rules or policies are unlikely to have any effect, but game designers can at least attempt to avoid rules that cannot be enforced within the game.

3.2 Architectural problems

This is a somewhat nebulous category since all online games require some code to be run locally (the user must have a computer). Moving parts of the game code onto players' own machines can decrease latency significantly, but unfortunately may also allow players to observe or alter parts of the game to their advantage. This problem is commonly exploited in FPS games, because their tight bound on response time forces more of the game to be run locally on the player's machine. A well known cheat is information exposure, in which a player can modify local code or eavesdrop the local network to reveal information that would otherwise be hidden from him. Recently, we have proposed a peer-to-peer architecture that distributes most MMG game state to player machines [9], which may expose MMGs to similar cheating problems.

3.3 Implementation bugs

In general, game rules are *local* – each is concerned with only a single attribute or event, but objects and their interactions are hierarchal, such that each object can interact with a variety of objects and different levels of the hierarchy. Enforcing local rules across multiple levels of dynamic interactions is both complex and expensive.

Bugs commonly occur because of implementation oversight, especially in large game worlds. For example, a "castle" object would naturally employ standard "door" and "window" objects. While the door is handled with custom code that prevents it from being opened except by a special fancy spell, the window objects are standard. Later extensions of the common window class, also in use in other parts of the world, may provide a way by which they can be opened or broken. Or, they may have been placed in such a way as to render them unreachable, but the later addition of a movable "ladder" object may allow players access to them. Thus, as games age and extensions are added to keep them interesting, the original implicit rules may be overlooked.

3.3.1 Meta-game events

Many cheats exploit *meta-game events*, events that are not part of the game world and the rules in it, but rather belong to the environment in which the game operates, *e.g.*, network disconnects. In chess, accidentally knocking a piece off the board would be a meta-game event. Meta-game events cause particular difficulties in implementing games. First of all, client failures may leave the game in an inconsistent state. Furthermore, rules that are designed to make dying and network outage less painful for unlucky players can also be exploited by cheaters who die and disconnect intentionally.

One class of cheats can occur when a relatively complex or expensive synchronization procedure would be required to carry out an action correctly, but instead it is implemented as an interruptible sequence of events. This may be a bug, or it may be the result of a design choice. The common exploit during disconnect/timeout is to take advantage of the fact that these transactions are not atomic, by disconnecting before a transaction completes. This exploit manifests itself in cheats such as the "dup" cheat.

Cloning or "duping" is a notorious cheat since it allows a player's wealth to grow exponentially. Suppose player A, who has a units of money, is colluding with player B, who has b money units. A gives M units of money to B, which requires two updates to add money to b and remove it from a. A disconnects temporarily after $b = b + M$ is committed, but before $a = a - M$ occurs. A can later reconnect to reclaim the money from his cohort, and the cheat can proceed through many iterations. This inconsistency can also be seen as resulting from multi-tiered design, where the meta-game event regarding A's disconnection is handled separately from regular in-game transaction events.

3.4 Anti-cheating approaches

Both MMGs and MPGs suffer from problems stemming from implementation bugs. Detection, usually through human observation, is the key to preventing this kind of cheating. Just as crime stands out more in a small village than in a city, the smaller scale of MPGs means that discrepancies become more obvious. Even with a hundred players, one that mysteriously advances in score/money/level, or performs incredible feats, will stand out. Among hundreds of thousands of players in a game spanning months, it is much harder for fellow players to keep track of each other and extrapolate others' advancement. MMGs usually have dedicated administrators to monitor players and investigate reported cheaters.

Commercial anti-cheating products include Punkbuster [4] and Cheating Death [2]. They involve installing more code on the client side to verify the client's game code or detect a certain pattern of cheating. This makes cheating harder, but fundamentally does not prevent it. Research approaches include Terra [5], a virtual machine-based trusted computing platform. They too rely on static code verification and do not fundamentally prevent cheating. Finally, [1] presents an algorithm for preventing a certain type of cheat (lookahead cheat) in turn-based games, and [3] extends this idea to games that use dead-reckoning. This approach can be useful against an important class of cheats, although to be used in existing games it would require some rewriting of the code.

4. OUR APPROACH

Because games are complex control systems whose implementation may require complex programming, incur high performance cost, or result in unpopular game policies, we choose to detect exploits at runtime. Dynamically verifying game rules is relatively simple, even though implementing game rules is complex because of large object hierarchies and meta-game events. In particular, although different events may interfere with each other, we need to verify only whether a particular rule is enforced or not.

Runtime verification mainly targets cheats that exploit implementation bugs, but can also be used to monitor system resources such as player capability and wealth. Since gaining advantage is a major goal of cheating, the effect of many cheats is reflected in player state changes, such as a quicker increase of capabilities. For example, we can monitor the rate of wealth increase by specifying a time interval and defining the rate as the difference of wealth between the beginning and end of the interval. We could then compute the average and max among different intervals for each player, and raise an alarm if either of them is larger than some threshold.

4.1 Atomicity of transactions

Transactions are a common form of interaction among players. As previously explained, transaction atomicity is often not guaranteed in games. As a result, many cheats exploit the lack of atomicity, particularly in the context of meta-game events.

Figure 1 shows the state machines of a simple transaction between players A and B, in which A initiates the transaction to transfer one unit of money to B. It represents the server states of each player. The two state machines communicate with each other as well as with their corresponding client objects. We include only one meta-game event, client disconnection, in the state machines. This meta-game event may be caused by a network outage between the client machine and the server or by failure of the client machine. The transaction should be aborted when disconnection occurs.

The design depicted in Figure 1 has a loophole in that the transaction is not guaranteed to be atomic if the client fails. For example, if client A fails at the $Wait$ state, B can still commit the transaction. As a result, $B++$ is executed without the corresponding $A--$. This loophole has been extensively exploited in MMORPGs, by intentionally disconnecting A to duplicate A's asset. In the reverse case, if client B is disconnected at the $Update$ state after it has sent a positive reply to A, the item will be lost in the transaction. Players will not, however, intentionally try to reach this state.

To detect non-atomic transactions, we annotate each transaction with a unique ID tid and the IDs of the two parties, $\langle tid, id_A, id_B \rangle$. Given the state machines of the transaction, once both parties agree to commit, the commit events should happen within a short interval unless client failures occur. We then monitor commit events, and detect a non-atomic event if one party commits but the other party does not commit within a certain time interval afterward. This time interval can be determined by the interval at which the game system writes state to stable storage.

- Transaction between A and B: $\langle tid, id_A, id_B \rangle$
- Raise alarm if
  - $(A.commits \wedge (\neg B.commit - within - T))$
  - OR $(B.commits \wedge (\neg A.commit - within - T))$

Note that all events relevant to the runtime verification are local game events within the transaction, while the rule implementation must consider external meta-game events that may interrupt the transaction. Since considering the effect of meta-game events on all possible in-game events is very difficult in complex games, verifying game rules at runtime is simpler than code verification.

5. RUNTIME VERIFICATION OF SYSTEM PROPERTIES

Continuous monitoring of the runtime behavior of a system can improve our confidence in the system by ensuring that the current execution is consistent with its requirements at runtime. We have developed a Monitoring and Checking (MaC) framework for runtime monitoring of software systems [8].

Figure 2 shows the overall structure of the MaC framework. The framework includes two main phases: static and dynamic. During the static phase, *i.e.*, before a target program is executed, runtime components such as a filter, an event recognizer, and a runtime checker are generated from a target program and a formal requirements specification. During the dynamic phase, the instrumented target program is executed while being monitored and checked with respect to the requirements specification. A filter is a collection of probes inserted into the target program. The essential functionality of a filter is to keep track of changes of monitored objects and send

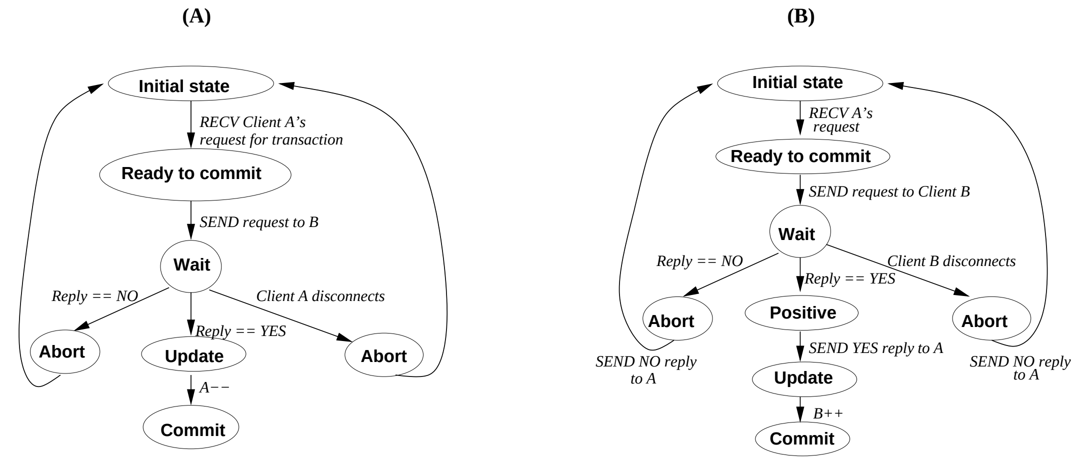

Figure 1: State machines for a transaction initiated by client A. These are server states.

pertinent state information to the event recognizer. An event recognizer detects an event from the state information received from the filter. Events are recognized according to a low-level specification. Recognized events are sent to the runtime checker. Although it is conceivable to combine the event recognizer with the filter, we chose to separate them to provide flexibility in an implementation of the architecture. A runtime checker determines whether or not the current execution history satisfies a requirement specification. The execution history is captured from a sequence of events sent by the event recognizer.

The MaC framework includes two languages: MEDL and PEDL. The Meta-Event Definition Language (MEDL) is used to express requirements. It is based on an extension of a linear-time temporal logic. It can be used to express a large subset of safety properties of systems, including real-time properties. We use events and conditions to capture and reason about temporal behavior and data behavior of the target program execution; events are abstract representations of time progress and conditions are abstract representations of data. For formal semantics of events and conditions, see [8].

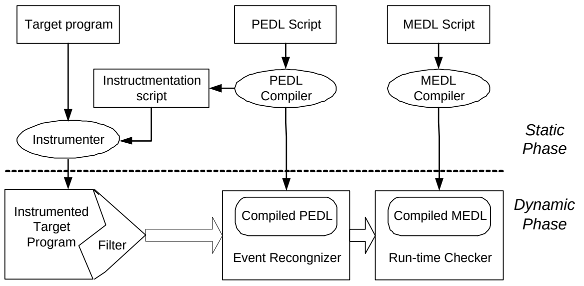

Figure 2: The Mac framework

In addition to the requirements written in MEDL, a monitoring script relates these events and conditions with low-level data manipulated by the system at runtime. Monitoring scripts are expressed in the Primitive Event Definition Language (PEDL). PEDL describes primitive high-level events and conditions in terms of system objects. PEDL is used to define what information is sent from the filter to the event recognizer, and how it is transformed into events used in high-level specification by the event recognizer. Based on the monitoring script, the system is automatically instrumented to deliver the monitored data to the event recognizer at runtime. The event recognizer, also generated from the monitoring script, transforms this low-level data into abstract events and delivers them to the runtime checker. The runtime checker verifies the sequence of abstract events with respect to the requirements specification and detects violations of the requirements.

The reason for keeping event recognition distinct from the property checking *per se* is to maintain a clean separation between the system itself, implemented in a certain way, and high-level system requirements, independent of a particular implementation. PEDL, therefore, is tied to the implementation language of the monitored system in the use of object names and types. MEDL is independent of the monitored system. The separation between PEDL and MEDL ensures that the architecture is portable to different implementation languages and specification formalisms. Note that implementation-dependent event recognition insulates the requirement checker from the low-level details of the system implementation. This separation also allows us to perform monitoring of heterogeneous distributed systems. A separate event recognizer may be supplied for each module in such a system. Each event recognizer may process the low-level data in a different way, but all deliver high-level events to the checker in a uniform fashion.

To demonstrate the effectiveness of the MaC framework, we have implemented Java-MaC [7], a MaC prototype for Java programs. Java-MaC targets Java executable code, *i.e.*, bytecode. It is easy to deploy Java-MaC, because it automatically instruments the target program and generates the runtime components of Java-MaC based on requirements specifications written in two scripting languages, MEDL and PEDL-for-Java. The system is available at `www.cis.upenn.edu/˜rtg/mac`.

6. EXPERIMENTS

Our game SimMUD [9] is modeled after the well known role playing game Multi-User Dungeon. We use a client-server version of SimMUD, implemented in Java. SimMUD has four kinds of elements: players, food, money, and the map. The map is created offline and does not change during game play. A player is described by its life or health level, the amount of money it owns, and its position in the map. Players can eat food and fight with other players, as well as transfer money to each other. Food items are described by their units of nutrition. A player can take only a limited amount of food at a time; eating food increases the player's health level depending on the number of nutrition units in the food. Once the food in one location is gone, it can be regenerated, but is likely to appear in a different location. For our transaction experiment, we are concerned with changes to players' amounts of money.

The transaction we simulate and check in SimMUD is the one described in Section 4. A description of how to use MaC to check for faulty transactions follows. The MEDL script defines a more general faulty transaction, while the PEDL script specifies detection of money-related transactions for this particular game.

```
import event transactionStart, transactionEnd,
       increment, decrement;

event incrementFirst = increment
    when [transactionStart,decrement);
event decrementFirst = decrement
    when [transactionStart,increment);
event incrementSecond = increment
    when [decrementFirst, transactionEnd);
event decrementSecond = decrement
    when [incrementFirst, transactionEnd);
event firstCommit =
    incrementFirst || decrementFirst;
event secondCommit =
    incrementSecond || decrementSecond;

alarm noCommit =
    start( currentTime - time(firstCommit) > T )
    when [firstCommit, secondCommit);
```

Figure 3: MEDL script for checking transaction atomicity

```
export event transactionStart, transactionEnd,
      increment, decrement;
event transactionStart = startM(
      Server.handleTransactionMessage() );
event transactionEnd = endM(
      Server.handleTransactionMessage() );
event increment = endM( Player.incMoney() );
event decrement = endM( Player.decMoney() );
```

Figure 4: Definition of the primitive events

6.1 Detection of faulty transactions

In order to check that transactions in our game happen atomically, we have expressed the property presented in Section 4 in MEDL. The main part of the MEDL script is shown in Figure 3. To keep the script simple, we assume non-overlapping transactions. The script assumes that four primitive events in the game are observable: start and end of a transaction, and increase or decrease of a player's money, which are seen as the commit events for the players. Each transaction should have one increment and one decrement event, but they can appear in arbitrary order. The script, then, detects the first commit event (`firstCommit`), and the second commit event (`secondCommit`). To see how this is done, consider the definition of the event `incrementFirst`. The condition $[e_1, e_2)$ holds from the time event e_1 occurs in the trace until the first occurrence of e_2 after that. Therefore, `incrementFirst` will occur if the `increment` event occurs after the event `transactionStart` but before the `decrement` event occurs. Now that we defined the two commit events of the transaction, we can raise the alarm as soon as enough time elapses after `firstCommit` without the occurrence of a `secondCommit`.

We then specify how to extract the events from the particular implementation of the game that we are working with. Figure 4 shows the definition of the primitive events. In our implementation, a transaction occurs within the method `handleTransactionMessage` in class `Server`. Thus the call to this method represents the start of a transaction and return from the method represents the end of it. Commit events correspond to the completion of methods `incMoney` and `decMoney` in the class `Player`. When method `incMoney` is called but `decMoney` is not subsequently called within time limit T, suggesting that A has disconnected to interrupt the transaction, a `noCommit` alarm is raised by the MaC runtime checker.

Running the SimMUD game instrumented with these primitive events, we were able to observe the transaction in which a cheating player avoids having his money decremented by interrupting the transaction. The cost for monitoring a transaction is small: the primitive event recognizer generates four events per transaction, each of which contains 10 bytes. The event records are passed to the model checker for in-memory, real-time checking.

7. CONCLUSIONS AND FUTURE WORK

Runtime verification is a promising tool for assuring the correctness of game implementations. Although traditionally used for real-time control systems, runtime verification is applicable to games because games are essentially complex control systems. Games are also much more complex than many traditional applications of runtime verification. Game objects model the physical world, and they interact by complex and sometimes contradictory rules.

Our key idea is that although game rules may interfere with each other, it is simpler to dynamically verify whether a rule has been correctly enforced, than to try to prevent all opportunities for cheating. We demonstrate our idea by applying the Java-Mac runtime verification system to SimMUD and detecting a transaction bug that is common in massively multiplayer online games.

Although preliminary results are encouraging, several issues must be addressed before this approach can be deployed in practice. The first is functionality. Since rules in runtime verification are based on temporal logic, are they general enough to express game rules, which more closely resemble the real physical and social world? The second issue is performance. Although the verification is done in memory, it is not free. Scalability to a large number of rules, objects, and players is a challenging question. Last but not least, programmability is an issue. We hand-crafted the verification rules for this experiment, but our goal is to build verification into game engines and make it transparent to high-level game designers.

Acknowledgment

We would like to thank John Fiore and Baohua Wu for constructing the SimMUD code. This research is supported partly by the Ashton Fellowship Foundation, ONR N00014-01-1-0795, NSF CCR-0086147, NSF CCR-0209024 and ARO DAAD19-01-1-0473.

8. REFERENCES

[1] Nathaniel E. Baughman and Brian Neil Levine. Cheat-proof playout for centralized and distributed online games. In *INFOCOM '01*, pages 104–113, 2001.

[2] 2002-2003 by UnitedAdmins.com. Cheating Death. `http://www.unitedadmins.com/cdeath.php`.

[3] Eric Cronin, Burton Filstrup, and Sugih Jamin. Cheat-proofing dead reckoned multiplayer games. In *Proc. ADCOG '03*, January 2003.

[4] 2000-2004 Even Balance, Inc. Punkbuster. `http://www.punkbuster.com`.

[5] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proc. SOSP '03*, October 2003.

[6] Patrice Godefroid. Model checking for programming languages using Verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[7] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance approach for Java programs. *Formal Methods in Systems Design*, 24(2):129–155, 2004.

[8] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *Proc. ECRTS '99*, pages 114–121, June 1999.

[9] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM '04*, Hong Kong, China, March 2004.

[10] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David Dill. CMC: A pragmatic approach to model checking real code. In *Proc. OSDI '02*, pages 75 – 88, December 2002.