Server-side Verification of Client Behavior in Online Games[*]

Darrell Bethea Robert A. Cochran Michael K. Reiter

University of North Carolina at Chapel Hill
{djb,rac,reiter}@cs.unc.edu

July 11, 2010

Abstract

Online gaming is a lucrative and growing industry but one that is slowed by cheating that compromises the gaming experience and hence drives away players (and revenue). In this paper we develop a technique by which game developers can enable game operators to validate the behavior of game clients as being consistent with valid execution of the sanctioned client software. Our technique employs symbolic execution of the client software to extract constraints on client-side state implied by each client-to-server message, and then uses constraint solving to determine whether the sequence of client-to-server messages can be "explained" by any possible user inputs, in light of the server-to-client messages already received. The requisite constraints and solving components can be developed either simultaneously with the game or retroactively for existing games. We demonstrate our approach in two case studies: one of the open-source game *XPilot*, and one of a game similar to *Pac-Man* of our own design.

1 Introduction

Multi-player online games are very popular and profitable and are growing more so. Since 1996 the computer game industry has quadrupled — in 2008 alone, worldwide video-game software sales grew 20 percent to $32 billion [29]. Estimates place revenue from online games at $11 billion, with games such as *World of Warcraft*, which has more than 10 million subscribers worldwide, bringing in around $1 billion in revenue for parent company Blizzard Entertainment [1, 14].

Since its inception, the online game industry has been plagued by cheating of numerous types, in some cases with financial repercussions to the game operator. *Age of Empires* and *America's Army* are examples of online games that suffered substantial player loss due to cheating [36], and for subscription games, player loss translates directly to a reduction in revenue. And game developers and operators are not the only ones for whom the stakes are high. Hoglund and McGraw [20] argue that "games are a harbinger of software security issues to come," suggesting that defenses against game cheats and game-related security problems will be important techniques for securing future massive distributed systems of other types.

In this paper, we develop an approach to detect a significant class of cheats in which a player changes a game client to allow behaviors that a sanctioned game client would not allow. To

[*] A preliminary version of this paper appeared in the *Proceedings of the 17th Internet Society Network and Distributed System Security Symposium*, February 2010.

accomplish this, the player might modify the client executable or in-memory data structures of a running client, for example. Today, the most robust defense against such client modification is to maintain authoritative state at the server, beyond the reach of direct manipulation by cheaters. This defense, however, exacts a heavy price from game operators, owing to the increased bandwidth use that results due to sending low-level client events (in the limit, every player input) to the server for accessing such state and conveying the effects back to clients. As bandwidth is one of the largest costs for large-scale game operators [32] and also a recurring one, this tension between bandwidth use and cheat prevention is problematic:

> *In the US and European markets, a good goal to shoot for is 4-6 kilobits per second (kps)/player or less. ... If you can get the bit rate down to 2kps, you're "golden." It's hard to see how that can happen, however, without putting dangerous amounts of data directly into the client, which is just asking for trouble from talented cheaters and hackers.* [32, p. 112]

The movement of games to all manners of devices using wireless, volume-priced communication only reinforces the importance of keeping bandwidth utilization to a minimum. Moreover, even with the amount of detailed client information collected at the server, server-side checking today is heuristic (and thus potentially incomplete) and manually programmed (and thus effort-intensive):

> *Players love to cheat — especially in online games ... be ready to add server-side support to prevent user cheating with methods that you were not able to predict.* [19]

In this paper we demonstrate a technique to detect any type of cheating that causes the client to exhibit behavior, as seen by the server, that is inconsistent with the sanctioned client software and the game state known at the server. That is, our approach discerns whether there was *any possible sequence* of user inputs to the sanctioned client software that could have given rise to each message received at the server, given what the server knew about the game client based on previous messages from the client and the messages the server sent to the client. In doing so, our approach remedies the previously heuristic and manual construction of server-side checks. Moreover, our approach potentially enables new game designs that reduce bandwidth use by placing more authoritative state at the client, since our approach verifies that the client's behavior is consistent with legal management of that state. While reducing the interaction with the client will generally increase the computational cost of our verification, the verification need not be done on the critical path of game play and can be performed selectively (e.g., only for suspected or winning players). Moreover, it can benefit from the dramatic growth of inexpensive computing power (larger numbers of cores) in game-operator server farms.

Our strategy exploits the fact that game clients are often structured as an event loop that processes user inputs, server messages, or other events in the context of current game state and then sends an update to the server on the basis of its processing. We symbolically execute the loop to derive a predicate that characterizes the effects of the loop, and specifically the update sent to the server, as a function of its inputs and game state. By partially instantiating these predicates on the basis of the actual messages the server receives from a client and what the server previously sent to the client, a *verifier* can then use a constraint solver to determine whether the resulting predicate is satisfiable. If so, then this indicates that the messages are consistent with proper client execution — i.e., there were *some* user inputs that could have yielded these messages.

We demonstrate our approach with two case studies. In the first, we apply our technique to the open-source game *XPilot*. Because *XPilot* was developed as is commonplace today, i.e., with low-level client events being sent to the server, this case study does not fully illustrate the strengths of our approach. However, it does demonstrate the (few) ways in which we found it necessary to adapt *XPilot* to use our technique efficiently. For the second case study, we use a game of our own design that is similar to *Pac-Man* but that has features to better exercise our technique. Together, these two case studies illustrate the limits and benefits of our approach and serve as guidance for game developers who are considering using this technique for detecting cheating in their games.

Following our initial investigation of these case studies, we investigate the impact of message loss on our verification technique. We extend our technique to improve verification performance in the face of message loss on the network. We then evaluate this extension using *XPilot*, since it is an example of a game built to use an unreliable transport protocol for performance reasons and consequently to continue gameplay despite message loss.

2 Related Work

Detecting the misbehavior of remote clients in a client-server application is an area that has received considerable attention. One strategy, of which ours is a special case, is to construct a model of proper client behavior against which actual client behaviors are compared. Giffin et al. [16] developed such an approach for validating remote system calls back to the home server from computations outsourced to (potentially untrusted) worker machines. In that work, remote system calls are compared to a control flow model generated from the binary code of the outsourced computation, specifically either a non-deterministic finite-state automaton or a push-down automaton that mirrors the flow of control in the executable. A more recent example is work by Guha et al. [18]: through static analysis of the client portion of Ajax web applications (HTML and JavaScript), their system constructs a control-flow graph for the client that describes the sequences of URLs that the client-side program can invoke. Any request that does not conform to this graph is then flagged as potentially malicious.

The technique we develop here follows this paradigm. We similarly use analysis (in our case, of source code) to develop a model of client behavior, against which inputs (messages from the client) are compared. The primary differentiator of our approach from previous works is soundness: only sequences of client messages that could have actually been produced through valid client execution, on the inputs sent by the server, will be accepted. This precision is accomplished though our use of symbolic execution to derive the complete implications of each message value to the client-side state. While this would hardly be tractable for any arbitrary client-server application, the control-loop structure of game clients and the frequent communication that is typically necessary for game play bounds the amount of uncertainty that the verifier faces in checking the client's messages.

A different approach to protecting against client misbehavior in client-server settings is to ensure that clients manage no authoritative state that could affect the server or the larger application; as discussed in the introduction, this is commonplace today for games. A recent system for implementing web applications to have this property is Swift [10], for example. The extreme of this approach is for the client to simply forward all unseen inputs (e.g., user inputs) to the server, where a trusted copy of the client-side computation acts on these inputs directly; e.g., this is implemented for Web 2.0 applications in the Ripley system [37]. In contrast, our approach detects any client behavior that is inconsistent with legal client execution, without requiring that all low-level events be sent to the server. Our approach also represents a middle ground in terms of programmer effort between automatic partitioning, which can require extensive manual annotation of the program [10], and client replication on the server, which requires none. In our case studies, we found that our approach was largely automatic but did require manual tuning in some cases to be efficient.

If the preceding approach can be viewed as a "pessimistic" way of eliminating trust in the client to manage authoritative state, one might say an "optimistic" version was proposed by Jha et al. [24]. Instead of moving authoritative state to a trusted server, a trusted *audit server* probabilistically audits the management of authoritative state at the client. In this approach, each game client periodically commits to its complete state by sending a cryptographic hash of it to the audit server. If later challenged by the audit server, the client turns over the requested committed state and all information (client-to-server and server-to-client updates, user inputs) needed to retrace and validate the client's behavior between this state and the next committed state. This approach, however, introduces additional costs to the client in the form of increased computation (to cryptographically hash game state, which can be sizable), storage (to retain the information needed to respond to an audit), and bandwidth (to transmit that information in the event of an audit). Our approach introduces none of these and can even enable bandwidth savings. Moreover, verification of clients in this scheme must be done *during* game play, since clients cannot retain the needed information forever. In contrast, our approach enables the game operator to audit at any time in the future, provided that it records the needed messages (to which it already has access).

Other work on defeating cheating specifically in online games comes in many varieties. Useful surveys of the problem are due to Yan and Randell [42], Lyhyaoui et al. [28], and Webb and Soh [40]. One common approach to defeat a variety of cheats involves augmenting the client-side computer with monitoring functionality to perform cheat detection (e.g., *PunkBuster* and [12, 13, 26, 31, 34]). Such approaches require consideration of how to defend this functionality from tampering, and some commercial examples have met with resistance from the user community (e.g., *World of Warcraft*'s *Warden*, see [39]). In contrast, our approach does not rely on monitoring functionality being added to clients. Other work focuses on wholly different cheats than we consider here. One example is game "bots" that perform certain repetitive or precise tasks in place of human gamers [8, 34, 41, 9, 30]. Bots that utilize the sanctioned game client to do so (as many do) will go undetected by our scheme, since the client behavior as seen by the server could have been performed by the sanctioned game client on inputs from a real human user (albeit an especially skilled or patient one). Another cheat that has received significant attention occurs when clients delay or suppress reporting (and choosing) their own actions for a game step until after learning what others have chosen in that step (e.g., [2, 11]). Such attacks can also go unnoticed by our techniques, if such delay or suppression could be explained by factors (e.g., network congestion) other than client modification. Our techniques are compatible with all proposed defenses of which we are aware for both game bots and delay/suppression and so can be used together with them. Finally, various works have examined security specifically for peer-to-peer games, e.g., using peer-based auditing [17, 22, 25]. Our technique may be applicable in some peer-to-peer auditing schemes, but we focus on the client-server setting here.

Our approach to validating client-side execution utilizes symbolic execution, a technique that has seen significant interest in the security community for generating vulnerability signatures [4], generating inputs that will induce error conditions [7, 43], automating mimicry attacks [27], and optimizing privacy-preserving computations [38], to name a few. A recent approach to generating weakest preconditions has shown promise as a more efficient alternative to symbolic execution in some applications [5, 23], and we plan to investigate the application of this technique to our problem to make client checking even more efficient.

3 Goals, Assumptions and Limitations

The defense that we develop in this paper addresses a class of game cheats that Webb and Soh term *Invalid commands*:

> *Usually implemented by modifying the game client, the invalid command cheat results in the cheater sending commands that are not possible with an unmodified game client. Examples include giving the cheater's avatar great strength or speed. This may also be implemented by modifying the game executable or data files. Many games suffer this form of cheating, including console games such as* Gears of War. [40, §4.2.3]

Importantly, our technique will even detect commands that are invalid in light of the history of the client's previous behaviors witnessed by the game server, even if those commands could have been valid in some other execution. Simply put, our approach will detect any client game play that is impossible to observe from the sanctioned client software.

We designed our cheat detection technique primarily for use by game developers. As we present and evaluate our approach, it requires access to source code for the game, though potentially a similar approach could be developed with access to only the game executable. The approach should be attractive to game developers because it can save them significant effort in implementing customized server-side verification of client behaviors. Our approach is comprehensive and largely automatic; in our case study described in §5, we needed only modest adaptations to an existing open-source game.

In order for detection to be efficient, our technique depends on certain assumptions about the structure of the game client. We assume in this paper that the game client is structured as a loop that processes inputs (user inputs, or messages from the game server) and that updates the game server about certain aspects of its status that are necessary for multiplayer game play (e.g., the client's current location on a game map, so that the server can update other players in the game with that location). Updates from the client to the server need not be in exact one-to-one correspondence to loop iterations. However, as the number of loop iterations that execute without sending updates increases, the uncertainty in the verifier's "model" of the client state also generally increases. This increase will induce greater server-side computation in verifying that future updates from the client are consistent with past ones. As we will see in §5, it is useful for these updates from the client to indicate which server-to-client messages the client has received, but importantly, the information sent by the client need not include the user inputs or a full account of its relevant state. Indeed, it is this information that a game client would typically send today and that we permit the client to omit in our approach.

Due to the scope of what it tries to detect, however, our technique has some limitations that are immediately evident. First, our technique will not detect cheats that are permitted by the sanctioned client software due to bugs. Second, modifications to the game client that do not change its behavior as seen at the server will go unnoticed by our technique. For example, any action that is possible to perform will be accepted, and so cheating by modifying the client program to make difficult (but possible) actions easy will go undetected. Put in a more positive light, however, this means that our technique has no false alarms, assuming that symbolic execution successfully explores all paths through the client. As another example, a client modification that discloses information to the player that should be hidden, e.g., such as a common cheat that uncovers parts of the game map that should be obscured, will go unnoticed by our technique. In the limit, a player could write his own version of the game client from scratch and still go undetected, provided that the behaviors it emits, as witnessed by the server, are a subset of those that the sanctioned client software could emit.

4 Our Approach

Our detection mechanism analyzes client output (as seen by the game server) and determines whether that output could in fact have been produced by a valid game client. Toward that end, a key step of our approach is to profile the game client's source code using symbolic execution and then use the results in our analysis of observed client outputs. We begin with a summary of symbolic execution in §4.1, and then discuss its application in our context in §4.2–§4.6. The symbolic execution engine that we use in our work is KLEE [6], with some modifications to make it more suitable for our task.

Before we continue, we clarify our use of certain terminology. Below, when we refer to a *valid* client, we mean a client that faithfully executes a sanctioned game-client program (and does not interfere with its behavior). Values or messages are then valid if they could have been emitted by a valid game client.

4.1 Symbolic Execution

Symbolic execution is a way of "executing" a program while exploring all execution paths, for example to find bugs in the program. Symbolic execution works by executing the software with its initial inputs specially marked so they are allowed to be "anything" — the memory regions of the input are marked as symbolic and are not given any initial value. The program is executed step-by-step, building constraints on the symbolic variables based on the program's operations on those variables. For example, if the program sets $a \leftarrow b + c$, where a, b, and c are all marked as symbolic, then after the operation, there will be a new logical constraint on the value of a that states that it must equal the sum of b and c. When the program conditionally branches on a symbolic value, execution forks and both program branches are followed, with the true branch forming a constraint that the symbolic value evaluates to true and the false branch forming the opposite constraint. Using this strategy, symbolic execution attempts to follow each possible code path in the target program, building a constraint that must hold on execution of that path.

Symbolic execution can help locate software bugs by providing constraints that enable a constraint solver (KLEE uses STP [15]) to generate concrete inputs that cause errors to occur. For example, if execution reaches an error condition (or a state thought to be "impossible"), then a constraint solver can use the constraints associated with that path to solve for a concrete input value which triggers the error condition. Having a concrete input that reliably reproduces an error is a great help when trying to correct the bug in the source code.

4.2 Generating Constraints

The first step of our technique is identifying the main event loop of the game client and all of its associated client state, which should include any global memory, memory that is a function of the client input, and memory that holds data received from the network. These state variables are then provided to the symbolic execution tool, which is used to generate a constraint for each path through the loop in a single round. These constraints are thus referred to as *round constraints*.

```
100: $loc \leftarrow 0$;
101:
102: while true do
103:     $key \leftarrow$ readkey();
104:     if $key =$ ESC then
105:         endgame();
106:     else if $key =$ '↑' then
107:         $loc \leftarrow loc + 1$;
108:     else if $key =$ '↓' then
109:         $loc \leftarrow loc - 1$;
110:     end if
111:     sendlocation($loc$);
112: end while
```

(a) A toy game client . . .

```
200: $prev\_loc \leftarrow$ symbolic;
201: $loc \leftarrow prev\_loc$;
202: while true do
203:     $key \leftarrow$ symbolic;
204:     if $key =$ ESC then
205:         endgame();
206:     else if $key =$ '↑' then
207:         $loc \leftarrow loc + 1$;
208:     else if $key =$ '↓' then
209:         $loc \leftarrow loc - 1$;
210:     end if
211:     breakpoint;
212: end while
```

(b) . . . instrumented to run symbolically

Figure 1: Example game client

For example, consider the toy game client in Figure 1(a). This client reads a keystroke from the user and either increments or decrements the value of the location variable *loc* based on the key that was read. The new location value is then sent to the server, and the client loops to read a new key from the user. Although this example is a toy, one can imagine it forming the basis for a *Pong* client.

To prepare for symbolic execution, we modify the program slightly, as shown in Figure 1(b). First, we initialize the variable *key* not with a concrete input value read from the user (line 103) but instead as an unconstrained symbolic variable (line 203). We then replace the instruction to send output to the server (line 111) with a breakpoint in the symbolic execution (line 211). Finally, we create a new symbolic state variable, *prev_loc* (line 200), which will represent the game state up to this point in the execution. The state variable *loc* will be initialized to this previous state (line 201).

Symbolically executing this modified program, we see that there are four possible paths through the main loop that the client could take in any given round. In the first possible path, *key* is ESC, and the game ends. Note that this branch never reaches the breakpoint. The second and third possible paths are taken when *key* is equal to '↑' and '↓', respectively. The final path is taken when *key* is none of the aforementioned keys. These last three paths all terminate at the breakpoint.

Via symbolic execution, the verifier can obtain the constraints for all symbolic variables at the time each path reached the breakpoint. Because we artificially created *prev_loc* during the instrumentation phase, it remains an unconstrained symbolic variable in all three cases. The state variable *loc*, however, is constrained differently on each of the three paths. In the case when *key* is equal to '↑', symbolic execution reports $loc = prev\_loc + 1$ as the only constraint on *loc*. When *key* is equal to '↓', the constraint is that $loc = prev\_loc - 1$. And when *key* is not '↑', '↓', or ESC, the constraint is that $loc = prev\_loc$.

Therefore, there are three possible paths that can lead to a message being sent to the server. If the server receives a message from a client — and the client is a valid client — then the client must have taken one of these three paths. Since each path introduces a constraint on the value of *loc* as a function of its previous value, the verifier can take the disjunction of these constraints, along with the current and previous values of *loc* (which the server already knows) and see if they are all logically consistent. That is, the verifier can check to see if the change in values for *loc* match up to a possible path that a valid game client might have taken. If so, then this client is behaving according to the rules of a valid game client. The disjunction of round constraints in this case is:

$$(loc = prev\_loc + 1) \vee (loc = prev\_loc - 1) \vee (loc = prev\_loc) \tag{1}$$

For example, suppose the verifier knows that the client reported on its previous turn that its *loc* was 8. If the client were to then report its new location as $loc = 9$, the verifier could simply check to see if the following is satisfiable:

$$\begin{aligned}&(prev\_loc = 8) \wedge (loc = 9) \wedge \\ &[(loc = prev\_loc + 1) \vee (loc = prev\_loc - 1) \vee (loc = prev\_loc)]\end{aligned}$$

Of course, it is satisfiable, meaning that the new value $loc = 9$ could in fact have been generated by a valid game client. Suppose, though, that in the next turn, the client reports his new position at $loc = 12$. Following the same algorithm, the verifier would check the satisfiability of

$$\begin{aligned}&(prev\_loc = 9) \wedge (loc = 12) \wedge \\ &[(loc = prev\_loc + 1) \vee (loc = prev\_loc - 1) \vee (loc = prev\_loc)]\end{aligned}$$

Because these round constraints are *not* satisfiable, no valid game client could have produced the message $loc = 12$ (in this context). Therefore, the verifier can safely conclude that the sender of that message is running an incompatible game client — is cheating.

There are also constraints associated with the variable *key*. We have omitted these here for clarity, showing only the constraints on *loc*. We have also omitted the constraints generated by the preamble of the loop, which in this case are trivial ("$loc = 0$") but in general would be obtained by applying symbolic execution to the preamble separately. Had there been any random coin flips or reading of the current time, the variables storing the results would also have been declared symbolic, and constraints generated accordingly. While file input (e.g., configuration files) could also be declared symbolic, in this paper we generally assume that such input files are known to the verifier (e.g., if necessary, sent to the server at the beginning of game play) and so treat these as concrete.

4.3 Accumulating Constraints

While the branches taken by a client in each round may not be visible to the verifier, the verifier can keep a set of constraints that represent possible client executions so far. Specifically, the verifier forms a conjunction of round constraints that represents a sequence of possible paths through the client's loop taken over multiple rounds; we call this conjunction an *accumulated constraint* and denote the set of satisfiable accumulated constraints at the end of round i by $\mathcal{C}_i$. This set corresponds to the possible paths taken by a client through round i.

The verifier updates a given set $\mathcal{C}_{i-1}$ of accumulated constraints upon receiving a new client message msg_i in round i. To do so, the verifier first combines the values given in msg_i with each round constraint for round i, where each symbolic variable in the round constraint represents client state for round i, and the round constraint characterizes those variables as a function of the variables for round $i - 1$. The verifier then combines each result with each accumulated constraint in $\mathcal{C}_{i-1}$ and checks for satisfiability.

For example, let us parameterize the round constraints for the toy example in §4.2 with the round number j:

$$\mathcal{G}(j) = \{\ \mathit{loc}_j = \mathit{loc}_{j-1} + 1\ ,\ \mathit{loc}_j = \mathit{loc}_{j-1} - 1\ ,\ \mathit{loc}_j = \mathit{loc}_{j-1}\ \}$$

Note that each member of $\mathcal{G}(j)$ corresponds to a disjunct in (1). If in round $i = 2$ the server receives the message $\mathit{msg}_2 = 9$ from the client, then it generates the constraint $M =$ "$\mathit{loc}_2 = 9$", because the value "9" in the message represents information corresponding to the variable loc in the client code. Then, combining M with each $G \in \mathcal{G}(2)$ gives the three constraints:

$$\begin{array}{l} \mathit{loc}_2 = 9 \wedge \mathit{loc}_2 = \mathit{loc}_1 + 1 \\ \mathit{loc}_2 = 9 \wedge \mathit{loc}_2 = \mathit{loc}_1 - 1 \\ \mathit{loc}_2 = 9 \wedge \mathit{loc}_2 = \mathit{loc}_1 \end{array}$$

Note that the combination of the client message with each round constraint involves both instantiation (e.g., using $j = 2$ above) as well as including the specific values given in the client message at that round (i.e., $\mathit{loc}_2 = 9$ above).

```
$\mathcal{C}_i \leftarrow \emptyset$
$M \leftarrow \mathsf{msgToConstraint}(msg_i)$
for $G \in \mathcal{G}(i)$ do
    for $C \in \mathcal{C}_{i-1}$ do
        $C' \leftarrow C \wedge G \wedge M$
        if $\mathsf{isSatisfiable}(C')$ then
            $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup \{C'\}$
        end if
    end for
end for
```

Figure 2: Construction of $\mathcal{C}_i$ from $\mathcal{C}_{i-1}$ and msg_i

These three round constraints each represent a possible path the client might have taken in the second round. The verifier must therefore consider each of them in turn as if it were the correct path. For example, if $\mathcal{C}_1 = \{\mathit{loc}_1 = 8\}$, then the verifier can use each round constraint to generate the following possible accumulated constraints:

$$\begin{array}{l} \mathit{loc}_1 = 8 \wedge [\mathit{loc}_2 = 9 \wedge \mathit{loc}_2 = \mathit{loc}_1 + 1] \\ \mathit{loc}_1 = 8 \wedge [\mathit{loc}_2 = 9 \wedge \mathit{loc}_2 = \mathit{loc}_1 - 1] \\ \mathit{loc}_1 = 8 \wedge [\mathit{loc}_2 = 9 \wedge \mathit{loc}_2 = \mathit{loc}_1] \end{array}$$

Since the second and third constraints are not satisfiable, however, this reduces to

$$\begin{array}{rcl} \mathcal{C}_2 & = & \{\mathit{loc}_1 = 8 \wedge [\mathit{loc}_2 = 9 \wedge \mathit{loc}_2 = \mathit{loc}_1 + 1]\} \\ & = & \{\mathit{loc}_1 = 8 \wedge \mathit{loc}_2 = 9\} \end{array}$$

The basic algorithm for constructing $\mathcal{C}_i$ from $\mathcal{C}_{i-1}$ and msg_i is thus as shown in Figure 2. In this figure, msgToConstraint simply translates a message to the constraint representing what values were sent in the message. It is important to note that while $|\mathcal{C}_i| = 1$ for each i in our toy example, this will not generally be the case for a more complex game. In another game, there might be many accumulated constraints represented in $\mathcal{C}_{i-1}$, each of which would have to be extended with the possible new round constraints to produce $\mathcal{C}_i$.

4.4 Constraint Pruning

Every accumulated constraint in $\mathcal{C}_i$ is a conjunction $C = c_1 \wedge \ldots \wedge c_n$ (or can be written as one, in conjunctive normal form). In practice, constraints can grow very quickly. Even in the toy example of the previous section, the accumulated constraint in $\mathcal{C}_2$ has one more conjunct than the accumulated constraint in $\mathcal{C}_1$. As such, the verifier must take measures to avoid duplicate constraint checking and to reduce the size of accumulated constraints.

First, the verifier partitions the conjuncts of each new accumulated constraint C' (line 304) based on variables (e.g., loc_2) referenced by its conjuncts. Specifically, consider the undirected graph in which each conjunct c_k in C' is represented as a node and the edge $(c_k, c_{k'})$ exists if and only if there is a variable that appears in both c_k and $c_{k'}$. Then, each connected component of this graph defines a block in the partition of C'. Because no two blocks for C' share variable references, the verifier can check each block for satisfiability independently (line 305), and each block is smaller, making each such check more efficient. And, since some accumulated constraints C' will share conjuncts, caching proofs of satisfiability for previously-checked blocks will allow shared blocks to be confirmed as satisfiable more efficiently.

Second, because round constraints refer only to variables in two consecutive rounds — i.e., any $G \in \mathcal{G}(j)$ refers only to variables for round j and $j-1$ — the formulas G and M in line 304 will refer only to variables in rounds i and $i-1$. Therefore, if there are blocks of conjuncts for C' in line 304 that contain no references to variables for round i, then these conjuncts cannot be rendered unsatisfiable in future rounds. Once the verifier determines that this block of conjuncts is satisfiable (line 305), it can safely remove the conjuncts in that block from C'.

4.5 Server Messages

Round constraints are not a function of only user inputs (and potentially random coin flips and time readings) but also of messages from the server that the client processes in that round. We have explored two implementation strategies for accounting for server messages when generating round constraints:

- *Eager*: In this approach, *eager round constraints* are generated with the server-to-client messages marked symbolic in the client software, just like user inputs. Each member of $\mathcal{G}(i)$ is then built by conjoining an eager round constraint with one or more conjuncts of the form "$svrmsg = m$", where *svrmsg* is the symbolic variable for a server message in the client software, and m is the concrete server message that this variable took on in round i. We refer to this approach as "eager" since it enables precomputation of round constraints prior to verification but, in doing so, also computes them for paths that may never be traversed in actual game play.
- *Lazy*: In this approach, *lazy round constraints* are generated from the client software after it has been instantiated with the concrete server-to-client messages that the client processed in that round; these round constraints for round i then constitute $\mathcal{G}(i)$ directly. Since the server messages are themselves a function of game play, the lazy round constraints cannot be precomputed (as opposed to eager round constraints) but rather must be computed as part of verification. As such, the expense of symbolic execution is incurred during verification, but only those paths consistent with server messages observed during game play need be explored.

In either case, it is necessary that the server log the messages it sent and that the verifier know which of these messages the client actually processed (versus, say, were lost). In our case study in §5, we will discuss how we convey this information to the server, which it records for the verifier.

As discussed above, the eager approach permits symbolic execution to be decoupled from verification, in that eager round constraints can be computed in advance of game play and then augmented with additional conjuncts that represent server messages processed by the client in that round. As such, the generation of round constraints in the eager approach is a conceptually direct application of a tool like KLEE (albeit one fraught with game-specific challenges, such as those we discuss in §5.4.1). The lazy approach, however, tightly couples the generation of round constraints and verification; below we briefly elaborate on its implementation.

To support the lazy approach, we extend KLEE by building a model of the network that permits it access to the log of messages the client processed (from the server) in the current round i and any message the client sent in that round. Below, we use the term *active path* to refer to an individual, symbolically executing path through the client code. Each active path has its own index into the message log, so that each can interact with the log independently.

To handle server-to-client messages from the log, we intercept the `recv()` system call and instead call our own replacement function. This function first checks to see that the next message in the network log is indeed a server-to-client message. If it is, we return the message and advance this active path's pointer in the log by one message. Otherwise, this active path has attempted more network reads in round i than actually occurred in the network log prior to reaching the breakpoint corresponding to a client-message send. In this case, we return zero bytes to the `recv()` call, indicating that no message is available to be read. Upon an active path reaching the breakpoint (which corresponds to a client send), if the next message in the log is not a client-to-server message, then this active path has attempted fewer network reads than the log indicates, and it is terminated as invalid. Otherwise, the round constraint built so far is added to $\mathcal{G}(i)$, and the logged client message is used to instantiate the new conjunct M in line 301 of Figure 2.

4.6 Scaling to Many Clients

Implementing our technique on a real-world online game with a large user base might require its own special implementation considerations. As we will see in §5, our eager and lazy implementations are not yet fast enough to perform validation on the critical path of game play. So, the game operator must log all the messages to and from clients that are needed to validate game play offline. That said, the need for logging will not be news to game operators, and they already do so extensively:

> *LOG EVERYTHING, and offer a robust system for reviewing the logs. When hunting down bugs and/or reviewing player cries of foul, nothing makes the job of the GM easier than knowing that he/she has perfect information and can state with 100% accuracy when a player isn't telling the whole truth.* [35]

As such, our approach introduces potentially little additional logging to what game operators already perform. Nevertheless, to minimize this overhead, game operators might use a log-structured file system [33]. Such file systems write data sequentially in a log-like structure and are optimized for small writes (as would be the case when logging client and server messages). Log-structured file systems have been implemented for NetBSD and Linux, for example.

Once the messages are logged, they can be searched later to extract a specific game trace to be checked (e.g., for a winning player). The checking itself can be parallelized extensively, in that the trace of a player can be checked independently of others', and even blocks within accumulated constraints C' (see §4.4) can be checked in parallel. Traces can also be partially checked, by starting in the middle of a trace, say at round i with client-to-server message msg_i, and checking from that point forward (i.e., with $\mathcal{C}_{i-1} = \{\text{true}\}$). Of course, while such a partial check can validate the internal consistency of the part of the trace that is checked, it will not detect inconsistencies between the validated part and other parts.

5 Case Study: *XPilot*

In our first case study, we apply our technique to *XPilot*, an open-source multiplayer game written in about 150,000 lines of C code. *XPilot* uses a client-server architecture that has influenced other popular open source games. For example, the authors of *Freeciv* used *XPilot*'s client-server architecture as a basis for the networking in that game. *XPilot* was first released over 15 years ago, but it continues to enjoy an active user base. In fact, in July 2009, 7b5 Labs released an *XPilot* client for the Apple iPhone and Apple iPod Touch (see `http://7b5labs.com/xpilotiphone`), which is one of several forks and ports of the *XPilot* code base over the years. We focus on one in particular called *XPilot NG* (*XPilot Next Generation*).

5.1 The Game

The game's style resembles that of *Asteroids*, in which the player controls an avatar in the form of a spaceship, which she navigates through space, avoiding obstacles and battling other ships. But *XPilot* adds many new dimensions to game play, including computer-controlled players, several multiplayer modes (capture the flag, death match, racing, etc.), networking (needed for multiplayer), better physics simulation (e.g., accounting for fuel weight in acceleration), and updated graphics. In addition, *XPilot* is a highly configurable game, both at the client and the server. For example, clients can set key mappings, and servers can configure nearly every aspect of the game (e.g., ship mass, initial player inventory, probability of each type of power-up appearing on the map, etc.).

As we have discussed, developers of today's networked games design clients with little authoritative state in order to help address cheating. In keeping with that paradigm, *XPilot* was written with very little such state in the client itself. Despite this provision, there are still ways a malicious user can send invalid messages in an attempt to cheat. In *XPilot*, there are some sets of keys that the client should *never* report pressing simultaneously. For example, a player cannot press the key to fire (`KEY_FIRE_SHOT`) while at the same time pressing the key to activate his shield (`KEY_SHIELD`). A valid game client will filter out any attempts to do so, deactivating the shield whenever a player is firing and bringing it back online afterward. However, an invalid game client might attempt to gain an advantage by sending a keyboard update that includes both keys. As it happens, the server does its own (manually configured) checking and so the cheat fails in this case, but the fact that the client behavior is verifiably invalid remains. There are numerous examples of similar cheats in online games that servers fail to catch, either because of programming errors or because that particular misuse of the protocol was unforeseen by the game developers. In our evaluations, we confirmed that our technique detects this attempt to cheat in *XPilot*, as expected. This detection was a direct result of the logic inherent in the game client, in contrast to the manually programmed rule in the *XPilot* server.

At the core of the architecture of the *XPilot* client is a main loop that reads input from the user, sends messages to the server, and processes new messages from the server. In §5.3 and §5.4, we describe the verification of *XPilot* client behavior by generating lazy round constraints and eager round constraints for this loop, respectively. However, we first describe modifications we made to *XPilot*, in order to perform verification.

5.2 Game Modifications

Message acknowledgments Client-server communication in *XPilot* uses UDP traffic for its timeliness and decreased overhead — the majority of in-game packets are relevant only within a short time after they are sent (e.g., information about the current game round). For any traffic that must be delivered reliably (e.g., chat messages between players), *XPilot* uses a custom layer built atop UDP. Due to *XPilot*'s use of UDP and the fact that it can process arbitrary numbers of messages in a single client loop, we added to *XPilot* an acknowledgement scheme to inform the server of which inbound messages the client processed in each loop iteration and between sending its own messages to the server. The server logs this information for use by the verifier. There are many possible efficient acknowledgement schemes to convey this information; the one we describe in Appendix A assumes that out-of-order arrival of server messages is rare.

These acknowledgments enable the server to record a log of relevant client events in the order they happened (as reported by the client). For each client-to-server message that the server never received, the verifier simply replaces the constraint M implied by the missing message (see line 301 of Figure 2) with $M = \mathsf{true}$.

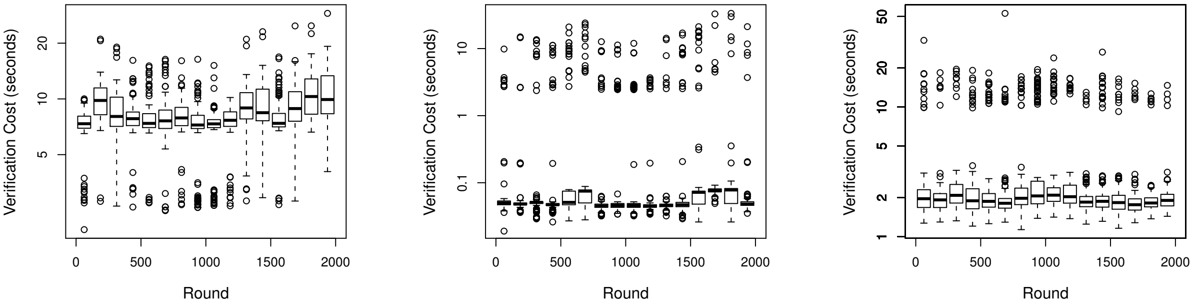
Floating-point operations *XPilot*, like most games of even moderate size, includes an abundance of floating-point variables and math. However, it is not currently possible to generate constraints on floating-point numbers with KLEE or to check them using STP. Therefore, we implement *XPilot*'s floating-point operations using a simple fixed-point library of our own creation. As a result, symbolic execution on the *XPilot* client produces constraints from this library for every mathematical operation in the client code involving a symbolic floating-point number. These constraints, in turn, inflate the verification speeds reported in §5.4, in particular.

Client trimming The *XPilot* client, like presumably any game client, contains much code that is focused on enhancing the user gaming experience but that has no effect on the messages that the client could send to the server. To avoid analyzing this code, we trimmed much of it from the game client that we subjected to analysis. Below we summarize the three classes of such code that we trimmed. Aside from these three types of code, we also trimmed mouse input-handling code, since all game activities can be performed equivalently using the keyboard.

First, several types of user inputs impact only the graphical display of the game but have no effect on the game's permissible behaviors as seen by the server. For example, one type of key press adjusts the display of game-play statistics on the user's console. As such, we excised these inputs from the client software for the purposes of our analysis.

Second, there are certain "reliable" messages the server sends the client (using the custom reliable-delivery protocol built over UDP). Reliable traffic is vital to the set-up and tear-down of games and game connections, but once play has begun, reliable messages are irrelevant for game play. Types of messages the server sends reliably are in-game chat messages (both among players and from the server itself), information about new players that have joined, and score updates, all of which are relatively infrequent and purely informational, in the sense that their delivery does not alter the permissible client behaviors. As such, we ignored them for the purpose of our analysis.

Third, KLEE is built upon LLVM and requires the input executable to be compiled into the LLVM intermediate representation (IR). Like all software, *XPilot* does not execute in isolation and makes use of external libraries; not all of these were compiled into LLVM IR. Specifically, the graphics library was not symbolically executed by KLEE, and instead any return values from graphics calls that *XPilot* later needed were simply declared symbolic.



(a) Cost per round (lazy)

(b) Cost per round (lazy) with *XPilot*-specific optimizations

(c) Cost per round (eager) with *XPilot*-specific optimizations

Figure 3: Verification cost per round while checking a 2,000-round *XPilot* game log

5.3 Verification with Lazy Round Constraints

In this section we measure the performance of verification using lazy round constraints. As discussed in §4, lazy round constraints are generated once the client-to-server and server-to-client messages are known. Thus, the only unknown inputs to the game client when generating lazy round constraints are the user inputs and time readings (and random coin flips, but these do not affect server-visible behavior in *XPilot*).

In generating lazy round constraints, we departed slightly from the description of our approach in §4, in that we inserted multiple breakpoints in the client event loop, rather than only a single breakpoint. Each breakpoint provides an opportunity to prune accumulated constraints and, in particular, to delete multiple copies of the same accumulated constraint. This is accomplished using a variant of the algorithm in Figure 2, using constraints derived from prefixes of the loop leading to the breakpoint, in place of full round constraints. Some of these extra breakpoints correspond to the (multiple) send locations in *XPilot*'s loop. Aside from this modification, we implemented our approach as described in §4.

We ran our lazy client verifier on a 2,000-round *XPilot* game log (about a minute of game-play time) using a machine with a 3GHz processor. Figure 3(a) describes the per-round validation cost (in seconds) using a box-and-whiskers plot per 125 rounds: the box illustrates the 25th, 50th, and 75th percentiles; the whiskers cover points within 1.5 times the interquartile range; and circles denote outliers. The per-round verification times averaged 8.6s with a standard deviation of 2.74s. As an aside, in every round, there was exactly one remaining satisfiable accumulated constraint, indicating that, without client state, there is little ambiguity at the verifier about exactly what is happening inside the client program, even from across the network.

By employing an *XPilot*-specific optimization, we were able to significantly improve verification performance. After the trimming described in §5.2, the user input paths that we included within our symbolic execution of the client each caused another client-to-server message to be sent, and so the number of such sends in a round indicates to the verifier an upper bound on the number of user inputs in that round. As such, we could tune the verifier's symbolic execution to explore only paths through the client where the number of invocations of the input-handling function equals the number of client messages for this round in the log. This optimization yields the graph in Figure 3(b). Notice that there are three distinct bands in the graph, corresponding to how many times the input-handling function within the game client was called. The first band contains rounds which called the input handler zero times and represents the majority (90.1%) of the total rounds. These rounds were the quickest to process, with a mean cost of 53.8ms and a standard deviation of 21.1ms. The next-largest band (5.1%) contains rounds which called the input handler only once. These rounds took longer to process, with a mean of 3.26s and a standard deviation of 1.05s. The final band represents rounds with more than one call to the input-handling function. This band took the longest to process (12.9s, on average), but it was also the smallest, representing only 4.1% of all rounds.

5.4 Verification with Eager Round Constraints

In this section we discuss verification of *XPilot* using eager constraint generation. Recall that eager round constraints are precomputed from the sanctioned client software without knowledge of the messages the client will process in any given loop iteration. However, we found this approach to require moderate manual tuning to be practical, as we describe below.

5.4.1 Manual Tuning

A direct application of our method for generating eager round constraints for the *XPilot* client loop would replace the user key press with symbolic input and any incoming server message with a symbolic buffer and then use KLEE to symbolically execute the resulting client program. Such a direct application, however, encountered several difficulties. In this section we describe the main difficulties we encountered in this direct approach and the primary adaptations that we made in order to apply it to the *XPilot* client. These adaptations highlight an important lesson: the eager technique, while largely automatic, can require some manual tuning to be practical. Because our technique is targeted toward game developers, we believe that allowing for such manual tuning is appropriate.

Frame processing In *XPilot*, messages from the server to the client describing the current game state are called *frames*. Each frame is formed of a chain of game *packets* (not to be confused with network packets). The first and last packets in a frame are always special start-of-frame and end-of-frame packets, called `PKT_START` and `PKT_END`. Figure 4 shows an *XPilot* frame, containing a packet of type `PKT_FUEL` and potentially others (indicated by "...”). Packet headers are a single byte, followed by packet data that can carry anything from a single byte to an arbitrary-length, NULL-terminated string, depending on the packet type. Frames may contain multiple packet types and multiple instances of the same packet type.

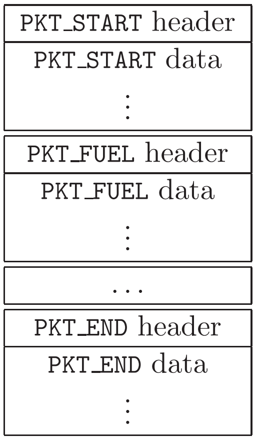


Figure 4: *XPilot* frame layout

Consider the client's frame-processing algorithm. Given a frame, it reads the packet header (i.e., the first byte), then calls the handler for that packet, which processes the packet and advances the frame pointer so that the new "first byte" is the packet header of the next packet in the frame. This continues until the packet handler for `PKT_END` is called, the return of which signifies the end of the frame handling. Therefore, given a completely symbolic buffer representing the frame, our symbolic execution would need to walk the client code for *each possible sequence* of packets in a frame, up to the maximum frame size. But *XPilot* has dozens of packet types, some of which include a very small amount data. As evidence of the infeasibility of such an approach, consider the following (very conservative) lower bound on the number of packet sequences: There are at least 10 types of packets that we considered whose total size is at most 5 bytes. The maximum size for a server-to-client frame in *XPilot* is 4,096 bytes, which means there is room for over 800 of these packets. That gives *at least* 10^{800} possible packet sequences that symbolic execution would traverse to generate constraints, which is obviously infeasible.

To make eager constraint generation feasible, then, we adapt our approach to generate round constraints by starting and stopping symbolic execution at multiple points within the loop, as opposed to just the beginning and end of the loop. In particular, we apply symbolic execution to the frame processing and user input processing portions of the loop separately, to obtain *user-input constraints* and *frame-processing constraints*, which in turn the verifier pieces together *during verification* to construct the round constraints. Moreover, the verifier can construct the frame-processing constraints on the basis of the particular frame the server sent to the client. It does so dynamically from packet-processing constraints that characterize how the client should process each packet in the particular frame. For example, if the only packet types were `PKT_START`, `PKT_FUEL`, `PKT_TIME_LEFT`, and `PKT_END`, the packet-processing constraints representing the processing of a single packet would be

$$\begin{array}{l}
(p = \texttt{PKT\_START}) \wedge (\text{constraints\_for}(\texttt{PKT\_START})) \\
(p = \texttt{PKT\_FUEL}) \wedge (\text{constraints\_for}(\texttt{PKT\_FUEL})) \\
(p = \texttt{PKT\_TIME\_LEFT}) \wedge (\text{constraints\_for}(\texttt{PKT\_TIME\_LEFT})) \\
(p = \texttt{PKT\_END}) \wedge (\text{constraints\_for}(\texttt{PKT\_END}))
\end{array}$$

where p is a variable for the packet type and constraints_for(`PKT_START`) represents the additional constraints that would result from symbolic execution of the packet handler for `PKT_START`. With this new model of packet processing, the verifier can build a frame-processing constraint to represent any given frame from the logs. In this way, when the verifier checks the behavior of a given client, it does so armed with the frames the server sent to the client, the messages the server received from the client, and the frame-processing constraints that characterize the client's processing of each frame, which the verifier constructs from the packet-processing constraints.

Packet processing Certain individual packet types present their own tractability challenges as well. For example, the payload for a certain packet begins with a 32-bit mask followed by one byte for each bit in the mask that is equal to 1. The client then stores these remaining bytes in a 32-byte array at the offsets determined by the mask (setting any bytes not included in the message to 0). In the packet handler, the *XPilot* client code must sample the value of each bit in the mask in turn. Since the payload (and thus the mask) is symbolic, each of these conditionals results in a fork of two separate paths (for the two possible values of the bit in question). Our symbolic execution of this packet handler, then, would produce over 4 billion round constraints, which is again infeasible. We could have changed the *XPilot* network protocol to avoid the using mask, sending 32 bytes each time, but doing so would increase network bandwidth needlessly. Instead, we note that the result of this packet handler is that the destination array is set according to the mask and the rules of the protocol. We thus added a simple rule to the verifier that, when processing a packet of this type, generates a constraint defining the value of the destination array directly, exactly as the packet handler would have. Then, when symbolically executing the packet handlers, prior to verification, we can simply skip this packet.

To avoid similar modifications to the extent possible, we pruned the packets the verifier considers during verification to only those that are necessary. That is, there are several packet types that will not alter the permissible behaviors of the client as could be witnessed by the server, and so we ignored them when applying our technique. Most of these packet types represent purely graphical information. For example, a packet of type `PKT_ITEM` simply reports to the client that a game item of a given type (e.g., a power-up or a new weapon) is floating nearby at the given coordinates. These instructions allow the client to draw the item on the screen, but they do not affect the valid client behaviors as observable by the verifier.[1]

User input The first part of the client input loop checks for and handles input from the player. Gathering user-input constraints is fairly straightforward, with the exception that *XPilot* allows players to do an extensive amount of keyboard mapping, including configurations in which multiple keys are bound to the same function, for example. We simplified the generation of constraints by focusing on the user actions themselves rather than the physical key presses that caused them. That is, while generating constraints within the user-input portion of *XPilot*, we begin symbolic execution *after* the client code looks up the in-game action bound to the specific physical key pressed, but *before* the client code processes that action. For example, if a user has bound the action `KEY_FIRE_SHOT` to the key '`a`', our analysis would focus on the effects of the action `KEY_FIRE_SHOT`, ignoring the actual key to which it is bound. However, as with other client configuration options, the keyboard mapping could easily be sent to the server as a requirement of joining the game, invoking a small, one-time bandwidth cost that would allow the verifier to check the physical key configuration.

5.4.2 Eager Verification Performance

We ran our eager client verifier on the same 2,000-round *XPilot* game log and on the same computer as the test in §5.3. Figure 3(c) describes the per-round validation cost (in seconds) using a box-and-whiskers plot. As in Figure 3(b), we employed here an *XPilot*-specific optimization by observing that the number of client messages in a round bounds the number of user inputs in that round. As such, in piecing together round constraints, the verifier includes a number of copies of user-input constraints (see §5.4.1) equal to the client sends in that round. Similar to Figure 3(b), Figure 3(c) exhibits three bands (the third comprising a few large values), corresponding to different numbers of copies. The large percentage of rounds contained no user inputs and were the quickest to process, with a mean cost of 1.64s and a standard deviation of 0.232s. The second band of rounds — those with a single user input — took longer to process, with a mean of 11.3s and a standard deviation of 1.68s. Remaining rounds contained multiple user inputs and took the longest to process (34.2s, on average), but they were by far the least frequent.

Comparing Figures 3(b) and 3(c), the times for the eager approach are substantially slower than those for the lazy approach, when applied to *XPilot*. This performance loss is due to the fact that a large portion of the *XPilot* client code is dedicated to handling server messages. And while the verifier in the eager case has preprocessed this portion of the code, the resulting round constraints

[1] In particular, whether the client processes this packet is irrelevant to determining whether the client can pick up the game item described in the packet. Whether the client obtains the item is unilaterally determined *by the server* based on it computing the client's location using the low-level client events it receives — an example of how nearly all control is stripped from clients in today's games, owing to how they cannot be trusted.

are much more complex than in the lazy approach, where the the verifier knows the exact values of the server messages when generating round constraints. This complexity results in constraint solving in the eager case (line 305 of Figure 2) being more expensive.

It is also important to recall that lazy and eager are not interchangeable, at least in terms of game developer effort. As discussed in §5.4.1, achieving feasible generation of eager round constraints required substantial additional manual tuning, and consequently greater opportunity for programmer error. As such, it appears that eager is an inferior approach to lazy for *XPilot*. Another comparison between lazy and eager, with differing results, will be given in §6.

6 Case Study: *Cap-Man*

Our client verification technique challenges the current game-design philosophy by allowing servers to relinquish authoritative state to clients while retaining the ability to validate client behavior and thus detect cheating. As a way of demonstrating this notion, we have written a game called *Cap-Man* that is based on the game *Pac-Man*. In some ways *Cap-Man* is easier to validate than *XPilot* was — it represents a considerably smaller code base (roughly 1,000 lines of C code) and state size.

That said, *Cap-Man* is interesting as a case study for three reasons. First, whereas *XPilot* was written with virtually no authoritative client state, we will see that *Cap-Man* is intentionally rife with it, providing a more interesting challenge for our technique because it is so much more vulnerable to invalid messages. Second, the size of its code base allows us to conduct a more direct comparison between lazy and eager verification. Third, *Cap-Man* differs from *XPilot* in that the set of possible user inputs per round is substantially larger than the set of paths through the client's event loop. That is, in *XPilot*, there is nearly a one-to-one correspondence between user inputs and paths through the client event loop, which dampens the improvement that our technique offers over, e.g., the verifier simply running the client on all possible inputs in each round. *Cap-Man* demonstrates the scalability of our technique to many possible user inputs when this is not the case, thereby separating our technique from other such possible approaches.

6.1 The Game

Cap-Man is a *Pac-Man*-like game in which a player controls an avatar that is allowed to move through a discrete, two-dimensional map with the aim of consuming all remaining "food" items before being caught by the various enemies (who are also wandering the map). Each map location is either an impenetrable wall or an open space, and the open spaces can contain an avatar, an enemy, pieces of food, a power-up, or nothing at all. When a player reaches a map location that contains food or a power-up, he automatically consumes it. Upon consuming a power-up, the player enters a temporary "power-up mode," during which his pursuers reverse course — trying to escape rather than pursue him — and he is able to consume (and temporarily displace) them if he can catch them. In addition to these features (which were present in *Pac-Man* as well), we have added a new feature to *Cap-Man* to invite further abuse and create more uncertainty at the server: A player may set a bomb (at his current location), which will then detonate a number rounds in the future selected by the user from a predefined range (in our implementation, between 3 and 15 rounds).[2]

[2] In our preliminary work [3], a bomb detonated in a fixed number of rounds. We changed the game to accommodate a user-selected number of rounds to bomb detonation in order to demonstrate the ability of our technique to scale to

When it detonates, it kills any enemies (or the player himself) within a certain radius on the map. Players are not allowed to set a second bomb until the first bomb has detonated.

Cap-Man uses a client-server architecture, which we designed specifically to go against current game-development best practices: i.e., it is the *server*, not the client, which has a minimum of authoritative state. The client tracks his own map position, power-up-mode time remaining, and bomb-placement details. Specifically, at every round, the client sends a message to the server indicating its current map position and remaining time in power-up mode. It also sends the position of a bomb explosion, if there was one during that round. Note that the client never informs the server when it decides to *set* a bomb. It merely announces when and where detonation has occurred. The server, in contrast, sends the client the updated positions of his enemies — this being the only game state for which the server has the authoritative copy.

The design of *Cap-Man* leaves it intentionally vulnerable to a host of invalid-message attacks. For example, although valid game clients allow only contiguous paths through the map, a cheating player can arbitrarily adjust his coordinates, ignoring the rules of the game — a cheat known in game-security parlance as "telehacking." He might also put himself into power-up mode at will, without bothering to actually consume a power-up. Finally, there is no check at the server to see whether or not a player is lying about a bomb placement by, for example, announcing an explosion at coordinates that he had not actually occupied within the past 15 rounds. In fact, the *Cap-Man* server contains no information about (or manual checks regarding) the internal logic of the game client.

In order to detect cheating in *Cap-Man*, we apply our technique in both its lazy and eager variations. Due to *Cap-Man*'s smaller size and simpler code structure, we can generate round constraints over an entire iteration of the main loop in each case, without the need to compartmentalize the code and adopt significant trimming measures as we did for *XPilot*.

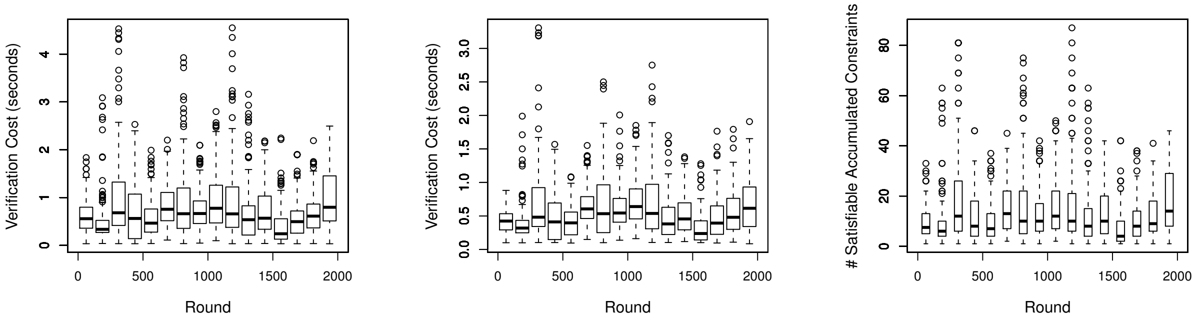
6.2 Evaluation

Using our technique, we are able to detect invalid-command cheats of all the types listed above. Below we present the results of client-validity checks on a game log consisting of 2,000 rounds (about 6-7 minutes of game-play time), during which the player moved around the map randomly, performing (legal) bomb placements at random intervals.

Figure 5 shows that the verification costs for *Cap-Man* were consistently small, with a mean and standard deviation of 752ms and 645ms for verification via lazy round constraints (Figure 5(a)) and a mean and standard deviation of 310ms and 193ms for verification using eager round constraints (Figure 5(b)). The lazy method was (on average) roughly 2.5 times slower than the eager method, owing to the overhead of symbolic execution to compute round constraints for each round individually during verification. While in the *XPilot* case study, eager verification required significantly greater development effort (see §5.4.1), this additional effort was unnecessary with *Cap-Man* due to its relative simplicity.

Figure 5(c) shows the number of satisfiable accumulated constraints during eager verification, which did not trend upward during the run. In lazy verification, the number of satisfiable accumulated constraints was virtually identical. (Variations in our pruning implementations caused less than 1% of the rounds to differ, and then by at most 12 accumulated constraints.) In the case of *XPilot*, the number of satisfiable accumulated constraints was always 1, but in *Cap-Man*

a larger number of possible user inputs.



(a) Cost per round (lazy)

(b) Cost per round (eager)

(c) Satisfiable accumulated constraints per round (eager)

Figure 5: Verifying a 2,000-round *Cap-Man* game log

there were often multiple accumulated constraints that remained satisfiable at any given round. This increase resulted primarily from state the *Cap-Man* client maintains but does not immediately report to the server (e.g., whether a bomb has been set, and with what detonation timer). The relationship between this hidden state and the number of satisfiable accumulated constraints is an important one. Consider the verification of a *Cap-Man* game that is currently in round i, with no bomb placements in the last 15 rounds (unbeknownst to the verifier). The verifier must maintain accumulated constraints that reflect possible bomb placements at each of rounds $i - 14$ through i. Upon encountering msg_{i+1} with an announcement of a bomb explosion, the verifier can discard not only all current accumulated constraints which do *not* include a bomb placement in any of rounds $i - 14$ through $i - 2$, but also those accumulated constraints which *do* include bomb placements in rounds $i - 1$ through $i + 1$, because players can only have one pending bomb at a time. This rule was not manually configured into the verifier— it was inferred automatically from the client code itself.

7 Message Loss

Games today must be built to tolerate a range of networking conditions, including occasional message loss. While there are standard approaches to recovering lost messages, such as message retransmission at the transport level (i.e., using TCP) or at the application level, retransmission is avoided in some games for two reasons. First, the importance of some messages diminishes quickly, and so by the time the message would be retransmitted, the utility of doing so is lost. Second, retransmission can introduce overheads that high-performance games cannot tolerate.

Lost server-to-client messages pose little difficulty to our client verification technique; all the verifier requires is to know what server-to-client messages the client processed and when, which can be communicated from the client efficiently (e.g., see Appendix A). Lost client-to-server messages pose more difficulty, however. Intuitively, our technique can handle client message loss by instantiating the constraint M for a missing round-i message msg_i to simply $M =$ “true” in Figure 2. However, this has two negative consequences.

First, from the server's (and verifier's) perspective, it is impossible to distinguish a lost message from one the client only pretended to send. This can be used by a cheating client to gain latitude in terms of the behaviors that the verifier will consider legitimate. For example, whenever a power-up appears on the game map, an altered game client could collect it by reporting its player's position at the power-up's location. So as to not be caught by the verifier, the client could alter its state to reflect having sent messages that would have been induced by the player actually moving to that location, even though these messages were never sent and so, from the server's perspective, were lost. We emphasize, however, that this attack will go undetected only if the client having sent these messages is possible with the sanctioned client software. As such, a straightforward way to mitigate this attack is to build the client software so that the number of messages it sends beyond the last one acknowledged by the server is restricted.

Another consequence of message loss is that the performance of verification can be severely impacted by it. The performance results in §5–6 did not reflect the loss of any client messages; instead, the game logs that we validated included all messages that the client sent. However, in practice message loss causes the accumulated constraints $\mathcal{C}_i$ to grow dramatically, since *any* path through the client that causes a message to be sent is deemed possible in round i. As a result, in experimenting with message loss in *XPilot*, we found that in the face of lost messages, the performance of our technique decays very substantially.

As such, we propose a lightweight scheme to enable our technique to retain its performance in the face of (limited) message loss. Rather than retransmitting messages, our technique communicates a small amount of additional information per client-to-server message to enable the verifier to prune accumulated constraints effectively in the face of message loss. Intuitively, the client remembers the path through its event loop that it traverses in round i and then conveys evidence of this to the server over the next several messages. The server records this evidence for the verifier, which uses it to prune round constraints considered for round i.

There are several ways to instantiate this intuition within our framework. Here we describe one implementation that works well in *XPilot*. In this implementation, the "evidence" that the client conveys to the server for the path it traversed in round i is a hash of the fields of the message it sent in round i that are constant as a function of the path traversed. Rather than send the entire hash in a subsequent message, however, the client "trickles" this hash value to the server, e.g., one bit per message, so that subsequent message losses still enable the server to collect a number of hash bits for each round. After the client's messages are recorded at the server, the verifier collects these bits and uses them to prune the round constraints considered at each step of verification where it is missing a message.

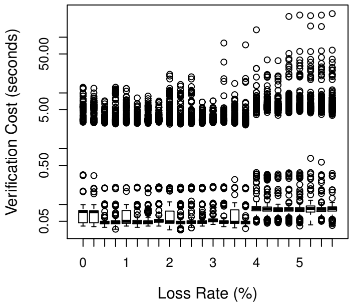


Figure 6: Verification (lazy) of 2000-round *XPilot* log with loss of client-to-server messages at the rate indicated on horizontal axis.

We have prototyped this approach in the context of lazy verification, in order to validate the ability of the *XPilot* verifier to retain its performance in the face of message losses. (*Cap-Man* uses TCP and so does not face message-loss issues.) The hash we use is a 16-bit BSD `sum`, and the k-th bit of the round-i message hash is carried on the round $i + k$ message ($1 \leq k \leq 16$). As such, each message carries an extra 16 bits composed of bits from the previous 16 client-to-server messages.

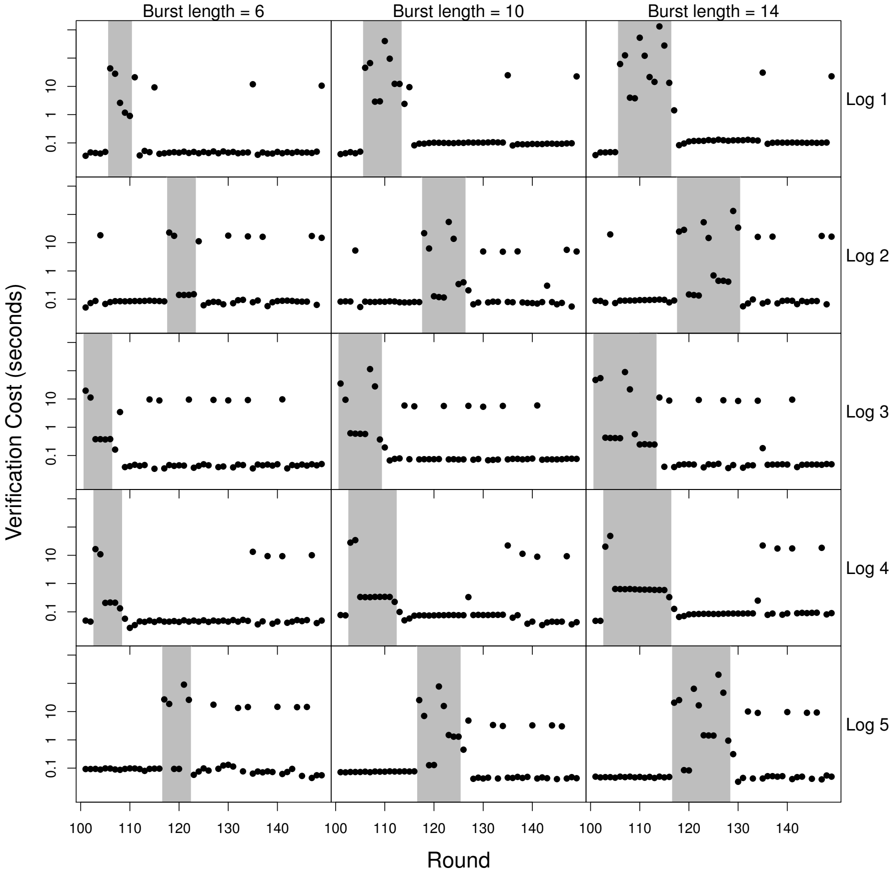


Figure 7: Verification (lazy) of *XPilot* logs with randomly induced bursts of client-to-server message losses. Shaded areas designate rounds in which losses occurred.

To demonstrate the effectiveness of this approach, we repeated the lazy verification of 2000-round *XPilot* game logs using *XPilot*-specific optimizations (c.f., Figure 3(b)) but introduced client-to-server message losses to show that our approach tolerates them seamlessly. We experimented with two types of message loss. In the first, each client-to-server message is lost with a fixed probability. Figure 6 shows box-and-whiskers plots that illustrate the per-round verification costs that resulted, as a function of this loss rate. Note that a message loss rate of 4% earns a "critical" designation at a real-time monitoring site like `www.internetpulse.net`. As Figure 6 shows our technique can easily handle such a high loss rate.

A second type of loss with which we experimented is a burst loss, i.e., the loss of a contiguous sequence of client-to-server messages. Figure 7 shows the verification costs per round in five different message logs in which a burst loss of length 6, 10, or 14 client-to-server messages is introduced at a random point between the 100th and 150th round. As these graphs show, the verification costs do tend to spike in the region where the burst loss occurs, but the verification costs remain feasible and recover after the burst to their original durations. Only when the burst length exceeds 16 (not shown) do the verification costs become and remain sufficiently large to be considered impractical.

8 Conclusion

The need to detect cheats in online games has heavily influenced game design for well more than a decade. Cheating has driven game developers to minimize or eliminate the management of authoritative state at game clients. These measures have direct impact on the game operator's bottom line, in particular due to the inflated bandwidth costs that result and to the manual and heuristic (and hence ongoing) effort of programming server-side checks on client behaviors.

In this paper we have developed a new approach to validate the server-visible behavior of game clients. Our approach validates that game-client behavior is a subset of the behaviors that would be witnessed from the sanctioned game-client software, in light of both the previous behaviors from the client and the game state sent to that client. Our technique exploits a common structure in game clients, namely a loop that accepts server and user inputs, manages client state, and updates the server with information necessary for multiplayer game play. Our technique applies symbolic execution to this loop to produce constraints that describe its effects. The game operator can then check the consistency of client updates with these constraints offline, in an automated fashion. We explored both lazy and eager approaches to constraint generation and investigated the programmer effort each entails, as well as their performance in two case studies.

In our first case study, we applied our validation approach to *XPilot*, an existing open-source multiplayer game. We detailed the ways we adapted our technique, in both the lazy and eager variants, to allow for efficient constraint generation and server-side checking. While this effort demonstrated the application of our approach to a real game, it was less satisfying as a test for our technique, in that *XPilot* was developed in the mold of modern games — with virtually no authoritative state at the client. We thus also applied our technique to a simple game of our own design that illustrated the strengths of our technique more clearly.

We believe that the technique developed in this paper can change how game developers address an important class of game cheats today, and in doing so opens up new avenues of game design that permit lower bandwidth utilization and better performance. We plan to examine the application of this technique to other types of distributed applications in future work.

Acknowledgements

We are deeply grateful to Cristian Cadar, Daniel Dunbar, and Dawson Engler for helpful discussions and for permitting us access to an early release of KLEE. Srinivas Krishnan, Alana Libonati, Andy White and anonymous reviewers provided helpful comments on drafts of this paper. This work was supported in part by NSF awards CT-0756998 and TC-0910483.

References

[1] L. Alexander. World of warcraft hits 10 million subscribers, January 2008. `http://www.gamasutra.com/php-bin/news_index.php?story=17062`.

[2] N. E. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *Proceedings of IEEE INFOCOM*, April 2001.

[3] D. Bethea, R. A. Cochran, and M. K. Reiter. Server-side verification of client behavior in online games. In *Proceedings of the 17th ISOC Network and Distributed System Security Symposium*, pages 21–36, February 2010.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[5] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest pre-conditions. In *Proceedings of the 2007 Computer Security Foundations Symposium*, July 2007.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, December 2008.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, November 2006.

[8] K.-T. Chen, J.-W. Jiang, P. Huang, H.-H. Chu, C.-L. Lei, and W.-C. Chen. Identifying MMORPG bots: A traffic analysis approach. In *Proceedings of the 2006 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, June 2006.

[9] K.-T. Chen, H.-K. K. Pao, and H.-C. Chang. Game bot identification based on manifold learning. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 21–26, October 2008.

[10] S. Chong, J. Liu, A. C. Myers, X. Qi, N. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 31–44, October 2007.

[11] E. Cronin, B. Filstrup, and S. Jamin. Cheat-proofing dead reckoned multiplayer games. In *Proceedings of the 2nd International Conference on Application and Development of Computer Games*, January 2003.

[12] M. DeLap, B. Knutsson, H. Lu, O. Sokolsky, U. Sammapun, I. Lee, and C. Tsarouchis. Is runtime verification applicable to cheat detection? In *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, August 2004.

[13] W. Feng, E. Kaiser, and T. Schluessler. Stealth measurements for cheat detection in on-line games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 15–20, October 2008.

[14] Gamasutra Staff. Analyst: Online games now $11b of $44b worldwide game market, June 2009. `http://www.gamasutra.com/php-bin/news_index.php?story=23954`.

[15] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007*, pages 519–531, July 2007.

[16] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[17] J. Goodman and C. Verbrugge. A peer auditing scheme for cheat elimination in MMOGs. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 9–14, October 2008.

[18] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proceedings of the 18th International World Wide Web Conference*, pages 561–570, April 2009.

[19] S. Hawkins, consultant for Sega of America. Quoted [32, p. 182], 2003.

[20] G. Hoglund and G. McGraw. *Exploiting Online Games: Cheating Massively Distributed Systems*. Addison-Wesley Professional, 2007.

[21] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[22] T. Izaiku, S. Yamamoto, Y. Murata, N. Shibata, K. Yasumoto, and M. Ito. Cheat detection for MMORPG on P2P environments. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, October 2006.

[23] I. Jager and D. Brumley. Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Cylab, Carnegie Mellon University, February 2010.

[24] S. Jha, S. Katzenbeisser, C. Schallhart, H. Veith, and S. Chenney. Enforcing semantic integrity on untrusted clients in networked virtual environments (extended abstract). In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 179–186, May 2007.

[25] P. Kabus, W. W. Terpstra, M. Cilia, and A. P. Buchmann. Addressing cheating in distributed MMOGs. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, October 2005.

[26] E. Kaiser, W. Feng, and T. Schluessler. Fides: Remote anomaly-based cheat detection using client emulation. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[27] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*, pages 161–176, July 2005.

[28] Y. Lyhyaoui, A. Lyhyaoui, and S. Natkin. Online games: Categorization of attacks. In *Proceedings of the 2005 International Conference on Computer as a Tool (EUROCON)*, November 2005.

[29] M. Magiera. Videogames sales bigger than DVD-Blu-ray for first time, January 2009. `http://www.videobusiness.com/article/CA6631456.html`.

[30] S. Mitterhofer, C. Platzer, C. Kruegel, and E. Kirda. Server-side bot detection in massive multiplayer online games. *IEEE Security and Privacy*, 7(3):18–25, May/June 2009.

[31] C. Mönch, G. Grimen, and R. Midtstraum. Protecting online games against cheating. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, October 2006.

[32] J. Mulligan and B. Patrovsky. *Developing Online Games: An Insider's Guide*. New Riders Publishing, 2003.

[33] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[34] T. Schluessler, S. Goglin, and E. Johnson. Is a bot at the controls? Detecting input data attacks. In *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 1–6, September 2007.

[35] D. Schubert, former lead designer for *Meridian 59*. Quoted [32, p. 221], 2003.

[36] D. Spohn. Cheating in online games. `http://internetgames.about.com/od/gamingnews/a/cheating.htm`.

[37] K. Vikram, A. Prateek, and B. Livshits. Ripley: Automatically securing Web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[38] R. Wang, X. Wang, Z. Li, H. Tang, M. K. Reiter, and Z. Dong. Privacy-preserving genomic computation through program specialization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[39] M. Ward. Warcraft game maker in spying row, October 2005. `http://news.bbc.co.uk/2/hi/technology/4385050.stm`.

[40] S. Webb and S. Soh. A survey on network game cheats and P2P solutions. *Australian Journal of Intelligent Information Processing Systems*, 9(4):34–43, 2008.

[41] R. V. Yampolskly and V. Govindaraju. Embedded noninteractive continuous bot detection. *Computers in Entertainment*, 5(4):1–11, October 2007.

[42] J. Yan and B. Randell. A systematic classification of cheating in online games. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, October 2005.

[43] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

A An *XPilot* Acknowledgement Scheme

As discussed in §5.2, an efficient acknowledgement scheme allows the server (and hence verifier) knowledge of the order (and loop iterations) in which the client processed server messages and sent its own messages. Below we describe one such scheme that is optimized for messages that arrive at the client mostly in order.

In this scheme, the *XPilot* client includes a sequence number *c2sNbr* on each message it sends to the server, and similarly the server includes a sequence number *s2cNbr* on each message it sends to the client. Each message from the server to a client also includes the largest value of *c2sNbr* received from that client. In each client message, the client includes *c2sAckd*, the largest value of *c2sNbr* received in a server message so far; a sequence *lateMsgs*[] of server message sequence numbers; and a sequence *eventSeq*[] of symbols that encode events in the order they happened at the client. The symbols in *eventSeq*[] can be any of the following. Below, *s2cAckd* is the largest sequence number *s2cNbr* received by the client before sending message *c2sAckd*, and similarly *loopAckd* is the largest client loop iteration completed at the client prior to it sending *c2sAckd*.

- `Loop` denotes a completed loop iteration. The j-th occurrence of `Loop` in *eventSeq*[] denotes the completion of loop iteration $loopAckd + j$.
- `Send` denotes the sending of a message to the server. The j-th occurrence of `Send` in *eventSeq*[] denotes the sending of client message $c2sAckd + j$.
- `Recv` and `Skip` denote receiving or skipping the next server message in sequence. The j-th occurrence of `Recv` or `Skip` in *eventSeq*[] denotes receiving or skipping, respectively, server message $s2cAckd + j$. Here, a message a skipped if it has not arrived by the time a server message with a larger sequence number arrives, and so a series of one or more `Skip` symbols is followed only by `Recv` in *eventSeq*[].
- `Late` denotes the late arrival of a message, i.e., the arrival of a message that was previously skipped. The j-th occurrence of `Late` in *eventSeq*[] denotes the arrival of server message $lateMsgs[j]$.

As such, *lateMsgs*[] contains a sequence number for each server message that arrives after another with a larger sequence number, and so *lateMsgs*[] should be small. *eventSeq*[] may contain more elements, but the symbols can be encoded efficiently, e.g., using Huffman coding [21] or another coding scheme, and in at most three bits per symbol in the worst case. Note that the server can determine *s2cAckd* and *loopAckd* based on the previous messages received from the client.