SYMBOLIC VERIFICATION OF REMOTE CLIENT BEHAVIOR IN DISTRIBUTED SYSTEMS

Robert Anderson Cochran III

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2016

Approved by:

Michael K. Reiter

Cristian Cadar

Somesh Jha

Dinesh Manocha

Fabian Monrose

ABSTRACT

Robert Anderson Cochran III: SYMBOLIC VERIFICATION OF CLIENT BEHAVIOR IN DISTRIBUTED SYSTEMS
(Under the direction of Michael K. Reiter)

A malicious client in a distributed system can undermine the integrity of the larger distributed application in a number of different ways. For example, a server with a vulnerability may be compromised directly by a modified client. If a client is authoritative for state in the larger distributed application, a malicious client may transmit an altered version of this state throughout the distributed application. A player in a networked game might cheat by modifying the client executable or the user of a network service might craft a sequence of messages that exploit a vulnerability in a server application. We present symbolic client verification, a technique for detecting whether network traffic from a remote client could have been generated by sanctioned software. Our method is based on constraint solving and symbolic execution and uses the client source code as a model for expected behavior. By identifying possible execution paths a remote client may have followed to generate a particular sequence of network traffic, we enable a precise verification technique that has the benefits of requiring little to no modification to the client application and is server agnostic; the only required inputs to the algorithm are the observed network traffic and the client source code. We demonstrate a parallel symbolic client verification algorithm that vastly reduces verification costs for our case study applications *XPilot* and *TetriNET*.

To Mom and Dad.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Michael Reiter, for his support and guidance throughout my Ph.D. study. I am sincerely grateful for his enthusiasm, patience, and mentorship. I would also like to thank Prof. Cristian Cadar, Prof. Somesh Jha Prof. Dinesh Manocha, and Prof. Fabian Monrose for graciously serving on my dissertation committee. I am grateful for the time they devoted to reading both my proposal and this dissertation; and for their many questions, insightful comments, and helpful suggestions. I also thank my research collaborators and co-authors, Darrell Bethea, Andrew Chi, Marie Nesfield, and Prof. Cynthia Sturton as well as all other members of the UNC security group.

I thank Dr. Ben Livshits for his mentorship at Microsoft Research and for the opportunity to work on projects built upon symbolic reasoning. I also thank Prof. Brian Dean, Prof. Robert Geist, and Prof. Jim Martin at Clemson University for introducing me to computer science research and for encouraging me to pursue graduate school.

I thank all my friends in the UNC Computer Science Department, particularly my fellow bandidos: Darrell Bethea, David Millman, Brittany Fasy, and Catie Welsh. I thank Jim Kuras and Lile Stephens for their friendship and artistic inspiration. I also sincerely thank everyone in the improv comedy community of Chapel Hill.

Last but not least, I would like to thank my amazing family. I thank my parents, Robert and Janeen Cochran for their constant encouragement throughout my academic journey and their endless love and support. I also thank each of my siblings: William, John Clark, Brooks, Harry, and Rose. I love each of you very much and could not have finished this dissertation without you.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

A malicious client in a distributed system can undermine the integrity of the larger distributed application in a number of different ways. Misbehavior may be the result of a malicious user attempting to either disrupt services or to gain advantage in a system. For example, a server application with a vulnerability may be compromised directly by a modified client. Alternatively, if a client is authoritative for state data in a larger distributed application, a malicious client may transmit an illegal version of this state throughout the distributed application. Real world attacks might consist of a player in a networked game cheating by modifying the client executable or the user of a network service might craft a sequence of messages that exploit a vulnerability in a server application.

There are many techniques for identifying malicious behavior in a distributed system. Some of these methods include, for example, probabilistic models of expected traffic patterns, attack signatures for known vulnerabilities and process level monitoring to identify control-flow attacks. In this dissertation, however, we address the problem of identifying malicious clients through the verification of legitimate client behaviour.

Existing techniques for verifying the correctness of client behavior in distributed applications suffer from imprecision, increased bandwidth consumption, or significant computational expense. One approach to defend against client misbehavior is for the server to validate client messages using a model of client behavior derived from the sanctioned client software. For example, Giffin et al. [38] and Guha et al. [43] developed methods to confirm that requests are consistent with a control-flow model of the client. Unfortunately, these approaches may admit false negatives; compromised clients that make calls consistent with their control-flow models (but that may still manipulate application state) can escape detection, in a manner analogous to mimicry attacks on intrusion-detection systems [86, 77]. In other work, greater precision has been achieved, but with greater expense. For example, the *Ripley* system [82] replays each client on the server in order to

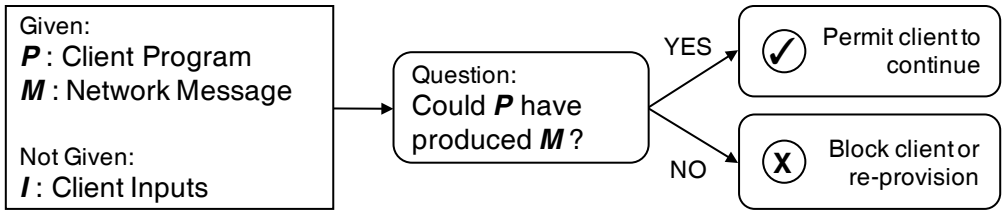


Figure 1.1. Abstracted client verification problem

validate the client's requests, but this incurs the bandwidth overhead of transmitting all client-side inputs (user inputs, timer values, etc.) to the server to permit replay and the computational overhead of replaying the client on the server side.

This dissertation presents a technique that resolves the tension between precision, bandwidth consumption and computational expense with a verification technique that validates legitimate client behavior as being consistent with the sanctioned client software. We accomplish behavior verification without encumbering the application with substantially more bandwidth use and without sacrificing accuracy. That is, any conclusion reached as to whether the sequence of client behaviors could, in fact, have been produced from the sanctioned client software is correct.

1.1 Problem

The client verification problem can be abstracted as follows. A server wishes to verify that a client is running the sanctioned software, but does so without modifying the client (e.g., by having the client send its inputs to the server). It therefore has no direct access to client state: the server knows only its own state and the network messages, M, that have been sent. The problem can then be phrased as: *Given a client program P, is it possible for P to yield output M?* This is illustrated in Figure 1.1.

Unfortunately, the general problem of determining whether an arbitrary program P can produce an output M is undecidable [79]. Fortunately, even though the general problem is undecidable, a particular instance can be tractable. The technique that we present in this dissertation is an application of symbolic execution [12, 55], which has been widely studied and applied for various purposes. Dynamic analysis techniques like symbolic execution typically face scaling challenges as code complexity and execution length grow, and our case is no exception. Nevertheless, important advances in the performance of constraint solving and symbolic execution in the recent past enable

our methods to be viable. In this dissertation we demonstrate that symbolic execution can be used as a foundation for verification remote client behavior.

1.2 Thesis Statement

Network messages from a remote client in a distributed system can be verified using a technique based on symbolic execution, in some cases at a rate that keeps pace with the application, to determine if the messages were generated by sanctioned software.

1.3 Motivation

We evaluate our framework in the context of online games. Online games provide a useful proving ground for our techniques due to the frequent manipulation of game clients for the purposes of cheating [93, 61, 90] and due to the pressure that game developers face to minimize the bandwidth consumed by their games [67]. As such, our techniques are directly useful for cheat detection in this domain. 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 [63]. 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, 33].

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 [75], and for subscription games, player loss translates directly to a reduction in revenue. Game developers and operators are not the only ones for whom the stakes are high. Hoglund and McGraw [46] 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.

The defense that we propose in this dissertation addresses a class of malicious behavior 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.* [90, Section 4.2.3]

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

1.4 Contributions

The key contributions of this dissertation are:

- A novel technique, *symbolic client verification*, that can verify the behavior of a remote client by determining whether a sequence of messages from a remote client could have been generated by sanctioned software.
- An extension of symbolic client verification to optimize identification of legitimate clients that uses training data to improve performance.
- A parallel algorithm for symbolic client verification that significantly improves performance.
- Evaluations of symbolic client verification in the framework of online games on a game of our own design, called *Cap-Man*, and two real-world games, *XPilot* and *TetriNET*.

The rest of this dissertation is outlined as follows. Chapter 2 gives an overview of symbolic execution and related work. Chapter 3 introduces symbolic client verification along with an evaluation in the context of online games. Chapter 3 is based on work that appears in the proceedings of the 17th ISOC Network and Distributed System Security Symposium [8] and in the ACM Transactions on Information and System Security [9]. Chapter 4 presents an optimistic version of symbolic client verification that prioritizes legitimate clients and also provides an evaluation. Chapter 4 is based on work published in the proceedings of the 20th ISOC Network and Distributed System

Security Symposium [26]. Chapter 5 presents a parallel algorithm for symbolic client verification and demonstrates the improvements it provides with an evaluation on verification of legitimate client in the context of online games. Portions of Chapter 5 have been published in a technical report [22]. Chapter 6 concludes the dissertation.

CHAPTER 2: BACKGROUND AND RELATED WORK

In this dissertation we use the analysis of source code to develop a model of client behavior, against which messages from the client are checked by a verifier. In this chapter we place our proposed methods in the wider context of security and verification methods by describing related work and background information. We start with an overview of cheating in online games and specific methods for detecting malicious behavior in such environments. We then give an overview of existing techniques for the detection of remote client misbehavior in the general case. While this dissertation places an emphasis on online games, it is useful to understand our contributions in the context of other verification methods; so we give an overview of verification methods in distributed systems and the emerging field of verifiable computing. Finally, we give background information on symbolic execution, which serves as the foundation for the proposals in the following chapters.

2.1 Detecting and Preventing Misbehavior in Online Games

As long as humans have played games, there have been players that have wished to cheat; and as long as there have been cheaters, players have devised methods to prevent or detect this cheating. One of the earliest examples of a cheat prevention method existed over 2000 years ago. In Ancient Rome, many games were based on dice and those who wished to keep players honest used a device to prevent the dice thrower from influencing the result [59]. The *pyxis cornea* was a table-top sized "tower" for rolling dice fairly; the dice were placed into an open funnel at the top and rolled down a internal staircase until exiting onto the table. Games of the modern era are much more complicated than those of Ancient Rome but the tension between honest players and cheating players still exists.

Modern techniques for the detection and prevention of malicious behavior in online games come in many varieties. In practice however, methods consist largely of client-side monitoring and server-side heuristics which are potentially incomplete, manually programmed and effort-intensive. Commenting on the issues developers face, game consultant Hawkins states:

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. [45]

We now give an overview of the techniques developers and researchers have investigated for the detection and prevention of malicious behavior in online games and how they relate to the proposals of this dissertation. For more information on this area, a number of authors have conducted surveys on the problem of of cheating in online games: Yan and Randell [93], Lyhyaoui et al. [61], and Webb and Soh [90].

Monitoring

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*, *World of Warcraft*'s *Warden*, *Valve*'s *VAC* [81] and [30, 31, 53, 66, 71]). Such approaches require consideration of how to defend this functionality from tampering, and some commercial examples have met with resistance from the user community. Accusations of client-side monitoring that violates standard expectations for player privacy have been levied against both *Warden* [89] and *VAC* [13]. These systems aggressively ban users that trigger cheat detection monitoring; *Valve*'s *VAC* has banned over 2 million player accounts [76]. However, the systems do have false positives; in 2010, *Valve* erroneously banned 12,000 players [64]. In contrast, our approach does not rely on monitoring functionality being added to clients and does not have false positives.

Bot Detection

Other work focuses on wholly different cheats than we consider in this dissertation. One example is game "bots" that perform certain repetitive or precise tasks in place of human gamers [19, 71, 92, 20, 65]. 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).

Network Manipulation

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., [4, 29]). 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 network delay suppression and so can be used together with them.

Peer-to-Peer Auditing

Finally, various works have examined security specifically for peer-to-peer games, e.g., using peer-based auditing [41, 49, 52]. Our technique may be applicable in some peer-to-peer auditing schemes, but we focus on the client-server setting in this dissertation.

2.2 Detecting Remote Client Misbehavior in the General Case

Detecting the misbehavior of remote clients in a client-server application is an area that has received considerable attention outside the domain of online games. We now expand our scope to examine methods for detecting remote client misbehavior in all types of networked software.

Model-based Verification

The proposals of this dissertation are a special case of model-based verification, a strategy where a model of proper client behavior is constructed and then compared against actual client behaviors. Giffin et al. [38] developed such an approach for validating remote system calls back to home server from potentially untrusted remote 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. Later work by the same authors takes a more precise approach with a model-based intrusion detection method that uses context-sensitive Dyck models [37]. Another example is work by Guha et al. [43]: through static analysis of the client portion of 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. Other works [10, 74] address parameter tampering attacks in web forms by using server-side proxies to infer a parameter constraint model when a web form is served to a client. Each client request using the web form is checked against the parameter model.

The techniques we propose in this dissertation follow a similar paradigm to the above approaches. We use analysis (in our case, of source code) to develop a model of client behavior, against which inputs (messages from the client) are compared. Unfortunately, compromised clients that make calls consistent with control-flow models may still manipulate application state [21] and can escape detection, in a manner analogous to mimicry attacks on intrusion-detection systems [86, 77]. The primary differentiator of our approach from these 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 (described below in Section 2.5) 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 many clients that have frequent communication with the server can bound the amount of uncertainty that the verifier faces in checking the client's messages.

Authoritative State

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. A system for implementing web applications to have this property is Swift [25]. 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 applications in the Ripley system [82]. 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 represents a middle ground in terms of programmer effort between automatic partitioning, which can require extensive manual annotation of the program [25], and client replication on the server, which requires less programmer effort, but more bandwidth to forward all inputs and greater server-side computation. Another consideration of moving all

authoritative state to the server is that it is known to increase the bandwidth consumed by interactive applications owing to the need for every access to authoritative state to reach the server (e.g., [67, p. 112]).

Auditing

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. [50]. 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 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 re-trace 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 client state, 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 no additional client-side computation, client-side storage or client-server bandwidth. Moreover, verification of clients in this scheme must be done *during* an active session, since clients cannot retain the needed information forever. In contrast, our approach supports auditing at any time in the future by the server operator, provided that it records the needed messages (to which it already has access).

2.3 Verifying Distributed Systems

While the verification of client behavior in the domain of online games was the initial motivation for the work in this dissertation, we argue that the proposed methods for verifying remote client misbehavior can be the first step in a framework for behavior verification in a larger class of distributed systems. Furthermore, the challenges of combating faulty or misbehaving nodes in distributed systems encompasses the challenges found in client-server systems but with increased complexity due the immense number of conditions that can arise due to asynchrony and partial failures. In this section, we place our behavior verification technique in the wider context of

methods for testing and verifying distributed system implementations and behaviours. A key distinction to note is that in this dissertation we are not checking a model of the overall client-server implementation or formally verifying the implementation. Our proposals are concerned with verifying observed network behavior received from the client against a model derived from the client source code.

Model Checking

Model checking is a state space exploration method that can be used to enumerate all states or paths in a system for the purposes of verifying properties or specifications. Surveys on recent advances in software model checking are due to Jhala et al. [51] and Bérard et al [6]. As mentioned above, our methods are related to model checking in that we use the client source code itself as a model for client behavior. Many advances in model checking have addressed the scaling challenges introduced as the space of possible configurations or states increases. In distributed systems this scaling challenge is particularly acute. Nevertheless, model checking approaches have been used to validate distributed system implementations. For example, the MODIST [94] system allows discovery of bugs or errors in unmodified binaries via an OS-level interposition layer that introduces all network or environmental actions in a distributed system. The model checking engine uses several search strategies to explore possible states which are instantiated by replaying actions. The OS-level interposition layer and heuristic search approach of MODIST is similar to our methods of behavior verification. However, unlike our techniques, MODIST has no notion of symbolic actions and thus the error conditions can only be triggered if the model checking engine introduces the appropriate sequence of actions. Another approach, MACE [54], uses a high level language with primitives that impose restrictions and structure on how a distributed system can be written, while still allowing for efficient model checking at a high level. In contrast, our methods operate on pre-existing software. The above systems integrate model checking into the testing and development stages of distributed system design, while our proposed techniques observe and verify runtime behavior, namely the outputs of the client software as received by the verifier. Pip [69] is a system that also allows for the verification of runtime behavior in a distributed system but differs from our work in that it requires the developers to define a specification for expected behavior, our work uses the client source code itself as the behavior model.

Software Verification

Recently, mechanically verified implementations of distributed systems have proven to be more robust and safe than ad-hoc implementations with accompanying hand-written proofs of correctness (e.g., for Byzantine fault tolerance algorithms [58] or for the Chord distributed hash table [99]). Systems for implementing formally verified software been built using domain-specific languages that enable formal verification, such as the Coq [7] interactive proof assistant and the TLA+ [57] proof specification system. These implementations include OS kernels [42], components of web browsers [70] and distributed protocols (e.g., Verdi [91] and IronFleet [44]). In this dissertation we do not formally verify the overall client-server implementation itself for fault tolerance and liveness, but rather use the implementation of the remote software to verify observed network behavior. Our techniques are, in theory, compatible with existing methods for verifying distributed systems, and may provide additional security properties. The guarantees above are only as good as the assumptions made about the types of faults that may occur; some work formally verifies software under general network or system errors, but leaves out consideration of targeted attacks.

2.4 Verifiable Computation

The recent growth of cloud services and mobile devices has motivated research into the development of techniques for checking the correctness of results provided by an untrusted cloud service. This domain is related to the client behavior verification problem, but in reverse; a computationally weak client wishes to offload computation to a remote server or entity. The remote server however may return an incorrect result because of a fault or error, or there may be an incentive for the server to cheat by either not performing a costly computation or returning a result that is somehow beneficial to the server. The study of *verifiable computation* is concerned with verifying that the result of some computation, performed by a cloud service or remote client, is provably correct. Before the field of verifiable computation was formalized, early solutions for ensuring the integrity of outsourced computation did so via replication. The SETI@home [3] project outsources computational work for analysis of astronomical data and uses redundant computation on different clients to identify results that may be erroneous or malicious. Ideally, however, the outsourced work would be verified at a cost that is smaller than performing the outsourced computation itself. A

scheme for verifiable computation using abstractions from the theory of computation was first introduced by Generro et. al [35]. They defined a scheme that is a combination of fully homomorphic encryption [36] and Yao's garbled circuits [96]. While impractical, this work showed that verifiable computation is achievable. Other work has improved this scheme, allowing for the outsourced task to be represented in a high level language that is transformed into an arithmetic circuit [68, 85, 5]. Others use advances in interactive proofs [40] or probabilistically checkable proofs [48] to permit a verifier to probabilistically confirm that an outsourced computation was performed correctly. Verifiable computing has been studied in under several different assumptions and client-server configurations; including, multi-client outsourcing [24], computations on encrypted data [32] and biometric computations [11]. Walfish et al. [87] provide a useful survey of several approaches to verifiable computation with performance and implementation comparisons.

There are several key differences between the solutions found in the area of verifiable computation and the proposals of this dissertation. While some improvements have been made (e.g., Pinocchio [68]), current systems for verifiable computation induce significant overhead on the client and must be implemented in specialized languages to allow transformation into an arithmetic circuit representation. Such transformations can induce an increase in the representation size that is exponential in the size of the input if the task contains a large number of loops or memory accesses. Our proposals do not induce any additional overhead on the client and can be implemented in any language that can be complied into a byte-code representation supported by the verifier (in our case, C). Secondly, while the techniques of verifiable computing could be generalized into supporting a wide variety of client-server applications, current verifiable computation systems would require existing implementations to be rewritten. Our technique supports several classes of applications without any changes to preexisting message format and frequency.

2.5 Symbolic Execution

Symbolic execution was introduced in the 1970's [12, 55], but it has only recently become a practical and viable software tool. At a high level, 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 can follow every possible code path in the target program, building a constraint for each one that must hold on execution of that path. Symbolic execution can help locate software bugs, for example, by providing constraints that enable a constraint solver to generate concrete inputs that cause errors to occur. Specifically, 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 that 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.

Applications for Security

Symbolic execution has seen significant interest in the security community for generating vulnerability signatures [14, 28], generating inputs that will induce error conditions [18, 95], automating mimicry attacks [56], and optimizing privacy-preserving computations [88], to name a few. The approach that we take to the behavior verification problem in this dissertation is an application of symbolic execution. In our case, we symbolically execute the client software with client-side inputs unknown to the server marked symbolic and then determine whether the messages received from the client violate the postconditions derived from the software.

Applications for Software Testing

Among applications of symbolic execution, software testing has received the most research attention. Symbolic execution can be an effective method of increasing the degree of code coverage in a testing tool by generating test cases that cover a high percentage of paths in a program. For example, *DART* [39] first concretely executes a program with an arbitrary input, recording the

path constraint implied by its choice at each branch point. The path constraint is then modified by negating a clause and a satisfying assignment to the constraint is found to derive a new input that will cover a different path in the program. More recent examples of this approach, which is also called *concolic* execution, include *CUTE* [72], *JPF* [84] and *Pex* [78, 2]. In contrast, our approach expands the verifier's search for paths to explain client messages as needed, starting from an initial collection of paths, but it does so without solving for inputs to exercise a path concretely and without the goal of achieving high path coverage.

Applications for Debugging

The applications of symbolic execution that are most related to our own are in debugging and diagnostics. Zamfir et al. [98] developed a debugging tool that uses symbolic execution to reconstruct the likely path a program took before it crashed, from the core dump file recorded by the operating system when the crash occurred. Their technique finds a feasible path or set of paths through a program that allow the program to reach the memory and process state that the core dump file indicates. *SherLog* [97] is another error diagnosis tool that uses a log file instead of a core dump file to indicate how a program executed. *SherLog* performs path analysis (not symbolic execution per se, but a similar technique) to determine the likely execution paths and variable values implied by a given set of log files. Similarly, symbolic execution has been used to discover the constraints for the paths through a program that reaches a vulnerability or error condition [14, 16, 18, 95]. Viewed through the lens of our work, the core dump file, log file, or error condition in these previous works is analogous to a "client message", and these tools similarly seek to find an execution that could explain it. However, the structure of our verification task — namely successively building an execution path to explain an entire sequence of messages — and the performance demands that we seek to meet give rise to a technique which we believe to be novel.

Parallel Symbolic Execution

In Chapter 5, we propose a method for behavior verification that is built upon a thread-level parallel extension to the KLEE [17] symbolic execution framework. Other efforts have demonstrated the feasibility and performance benefits of parallelized symbolic execution engines [15, 73, 62].

Bucur et al. demonstrated a parallelization approach that divides the symbolic execution work across multiple processes or worker nodes. In contrast, we propose a shared memory design, rather than a message passing approach, that has several advantages for the application of behavior verification. A shared memory approach better leverages opportunities to make symbolic execution optimizations that include: identification of duplicate states, utilization of symbolic state merging and shared constraint solving structures. Furthermore, during load-balancing in the Bucur et al. method, symbolic execution tasks are re-executed rather than transferred between processes. For testing at scale, the overhead of this redundant work is reasonable, but not for our unique application of symbolic execution. Additionally, our shared memory approach uses global knowledge of the progress of each execution path under consideration to make more informed decisions about which execution paths to explore next.

CHAPTER 3: SYMBOLIC CLIENT VERIFICATION

In this chapter we demonstrate a technique to detect any type of client modification 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 client 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 interaction with the client will generally increase the computational cost of our verification, verification need not be done on the critical path and can be performed selectively (e.g., only for randomly selected or suspicious users).

Our strategy exploits the fact that clients are often structured as an event loop that processes user inputs, server messages, or other events in the context of current client 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 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 three case studies on online games. 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. Third, we utilize *TetriNET*, a multiplayer version of *Tetris*, to demonstrate the potential bandwidth savings that symbolic client verification can enable. Together, these case studies illustrate the limits and benefits of our approach and serve as a foundation for future work.

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 operation despite message loss.

3.1 Goals, Assumptions and Limitations

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

As we present and evaluate our approach, it requires access to source code for the client, though potentially a similar approach could be developed with access to only the executable. The approach should be attractive to 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 Section 3.3, 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 client. We assume in this dissertation that the client is structured as a loop that processes inputs (user inputs, or messages from the server) and that updates the server about certain aspects of its status that are necessary (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 Section 3.3, 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 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 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 user could write their own version of the 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.

3.2 Client Verification Approach

Our detection mechanism analyzes client output (as seen by the server) and determines whether that output could in fact have been produced by a valid client. Toward that end, a key step of our approach is to profile the client's source code using symbolic execution and then use the results in our analysis of observed client outputs. We begin with an application of symbolic execution; shown in our context in Section 3.2.1–Section 3.2.4. The symbolic execution engine that we use in our work is KLEE [17], 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 client program (and does not interfere

with its behavior). Values or messages are then valid if they could have been emitted by a valid client.

At a high level, the symbolic client verification technique works by generating postconditions for all of the paths through key event loops in the client application, and then during verification, constructing chains of postconditions in combination with the message trace we are verifying. If the complete postcondition chain is satisfiable, then the message trace could have been produced by a legitimate client. The computational expense required for this method of verification lends it to be useful primarily in an offline fashion and only after modifying test applications to constrain the search spaces they present.

3.2.1 Generating Round 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*. Round constraints consist of three classes of symbolic variables: *state* variables, *input* variables and *network* variables.

Toy Example

For example, consider the toy game client in Figure 3.1(a). This client reads a keystroke from the user and either increments or decrements the value of the location variable loc based on this key. 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 3.1(b). First, we initialize the variable key not with a concrete input value read from the user (line 4) but instead as an unconstrained symbolic variable (line 4). We then replace the instruction to send output to the server (line 11) with a breakpoint in the symbolic execution (line 12). Finally, we create a new symbolic state variable, $prev\_loc$ (line 1), 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 2).

```
1: loc ← 0;
2:
3: while true do
4:     key ← readkey();
5:     if key = ESC then
6:         endgame();
7:     else if key = '↑' then
8:         loc ← loc + 1;
9:     else if key = '↓' then
10:        loc ← loc − 1;
11:    sendlocation(loc);
```

(a) Toy game client

```
1: prev_loc ← symbolic;
2: loc ← prev_loc;
3: while true do
4:     key ← symbolic;
5:     if key = ESC then
6:         endgame();
7:     else if key = '↑' then
8:         loc ← loc + 1;
9:     else if key = '↓' then
10:        loc ← loc − 1;
11:    msg ← loc
12:    breakpoint;
```

(b) Symbolically instrumented toy game client

Figure 3.1. Example game client

Symbolically executing one loop iteration of this modified program, we see that there are four possible paths 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:

$$(loc = prev\_loc + 1) \vee (loc = prev\_loc - 1) \vee (loc = prev\_loc) \tag{3.1}$$

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:

$$(prev\_loc = 8) \wedge (loc = 9) \wedge$$
$$[(loc = prev\_loc + 1) \vee (loc = prev\_loc - 1) \vee (loc = prev\_loc)]$$

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

$$(prev\_loc = 9) \wedge (loc = 12) \wedge$$
$$[(loc = prev\_loc + 1) \vee (loc = prev\_loc - 1) \vee (loc = prev\_loc)]$$

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 disseration 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.

3.2.2 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 Section 3.2.1 with the round number j:

$$\mathcal{G}(j) = \{\ loc_j = loc_{j-1} + 1\ ,\ loc_j = loc_{j-1} - 1\ ,\ loc_j = loc_{j-1}\ \}$$

Note that each member of $\mathcal{G}(j)$ corresponds to a disjunct in (3.1). If in round $i = 2$ the server receives the message $msg_2 = 9$ from the client, then it generates the constraint $\mathsf{M} =$ "$loc_2 = 9$", because the value "9" in the message represents information corresponding to the variable loc in the client code. Then, combining M with each $\mathsf{G} \in \mathcal{G}(2)$ gives the three constraints:

$$loc_2 = 9 \wedge loc_2 = loc_1 + 1$$
$$loc_2 = 9 \wedge loc_2 = loc_1 - 1$$
$$loc_2 = 9 \wedge loc_2 = loc_1$$

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., $loc_2 = 9$ above).

Algorithm Symbolic Client Verification

```
1: procedure VERIFY(msg_i, C_{i-1})
2:     C_i ← ∅
3:     M ← msgToConstraint(msg_i)
4:     for G ∈ G(i) do
5:         for C ∈ C_{i-1} do
6:             C' ← C ∧ G ∧ M
7:             if isSatisfiable(C') then
8:                 C_i ← C_i ∪ {C'}
```

Figure 3.2. 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 $\mathcal{C}_1 = \{loc_1 = 8\}$, then the verifier can use each round constraint to generate the following possible accumulated constraints:

$$loc_1 = 8 \wedge [loc_2 = 9 \wedge loc_2 = loc_1 + 1]$$
$$loc_1 = 8 \wedge [loc_2 = 9 \wedge loc_2 = loc_1 - 1]$$
$$loc_1 = 8 \wedge [loc_2 = 9 \wedge loc_2 = loc_1]$$

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

$$\begin{aligned} \mathcal{C}_2 &= \{loc_1 = 8 \wedge [loc_2 = 9 \wedge loc_2 = loc_1 + 1]\} \\ &= \{loc_1 = 8 \wedge loc_2 = 9\} \end{aligned}$$

The basic algorithm for constructing $\mathcal{C}_i$ from $\mathcal{C}_{i-1}$ and msg_i is thus as shown in Figure 3.2. 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 $\mathcal{C}_i$.

3.2.3 Constraint Pruning

Every accumulated constraint in $\mathcal{C}_i$ is a conjunction $\mathsf{C} = c_1 \wedge \ldots \wedge c_n$ (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 6) 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 7), 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 $\mathsf{G} \in \mathcal{G}(j)$ refers only to variables for round j and $j-1$ — the formulas G and M in line 6 will refer only to variables in rounds i and $i-1$. Therefore, if there are blocks of conjuncts for C' in line 6 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 7), it can safely remove the conjuncts in that block from C'.

3.2.4 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 Section 3.3, 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 Section 3.3.4). 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 a server-to-client 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 3 of Figure 3.2.

3.3 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*).

3.3.1 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 those particular misuses of the protocol were 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. While this example is illustrative, we emphasize that our goal is not to identify new cheating vulnerabilities on the *XPilot* server, but rather to illustrate how a pre-existing game client can be adapted for verification in our framework and how the verifier performs under different configurations.

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 Section 3.3.3 and Section 3.3.4, 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.

3.3.2 Client 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 Section 3.7 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 3 of Figure 3.2) with $\mathsf{M} = \mathsf{true}$.

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 Section 3.3.4, in particular.

Bounding loops

The number of round constraints can grow rapidly as new branch points are encountered during path traversal. Loops in the code can be especially problematic; a loop with up to n iterations induces

$\Omega(n^2)$ round constraints. During symbolic execution of *XPilot*, most loops have a concrete number of iterations, but there are some loops that iterate over a symbolic variable. If this symbolic variable is unbounded, then the number of round constraints can become impractical to manage. While some loops are not explicitly bounded in the code, they are implicitly bounded by the environment during normal execution. For example, the user input loop in *XPilot* is a while loop that continues until there are no longer any queued user input events to process. During normal execution, the number of iterations of this loop is limited by how fast a player can press a key. As such, during symbolic execution, we limited the input processing loop to a maximum of three iterations because we observed that during gameplay, this queue never contained more than three events. The user input loop and the packet processing loop in *XPilot* required this type of modification, but all other loops were exhaustively searched.

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.

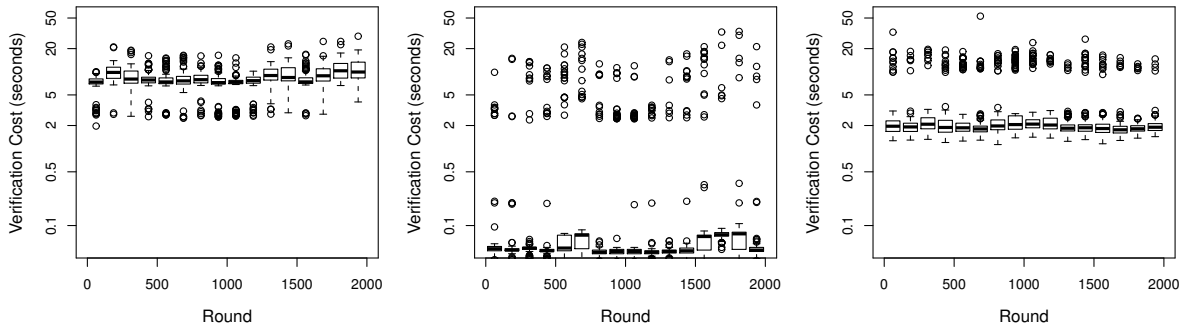
Third, KLEE is built upon LLVM and requires the input executable to be compiled into the LLVM intermediate representation (IR). Like all software, *XPilot* does not execute in isolation and makes use of external libraries; not all of these were compiled into LLVM IR. Specifically, the graphics library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

3.3.3 Verification with Lazy Round Constraints

In this section we measure the performance of verification using lazy round constraints. As discussed in Section 3.2, 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 Section 3.2, 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 3.2, 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 Section 3.2.

We ran our lazy client verifier on a 2,000-round *XPilot* game log (about a minute of game-play time) on a single core of a 3GHz processor. Figure 3.3(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. In these experiments, the verifier's memory usage remained below 256MB. 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.



(a) Cost per round (lazy)

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

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

Figure 3.3. Verification cost per round while checking a 2,000-round *XPilot* game log

By employing an *XPilot*-specific optimization, we were able to significantly improve verification performance. After the trimming described in Section 3.3.2, 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 3.3(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.

3.3.4 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.

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 3.4 shows an *XPilot* frame, containing a packet of type `PKT_FUEL` and potentially others (indicated by "..."). Packets are encoded as a single header byte followed by a packet data section that can carry anything from a single byte to an arbitrary-length string, depending on the packet type. Frames may contain multiple packet types and multiple instances of the same packet type.

Consider the client's frame-processing algorithm. Given a frame, it first 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.

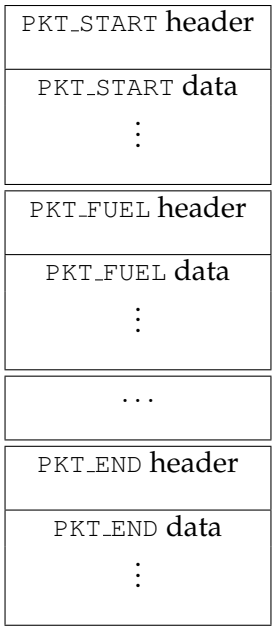


Figure 3.4. *XPilot* frame layout

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. 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

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

where p is a variable for the packet type and $\text{constraints\_for}(\texttt{PKT\_START})$ 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* protocol to avoid using the 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 this type of packet, generates a constraint defining the

value of the destination array directly, as the packet handler would have. Then, when symbolically executing the packet handlers, 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. This information allows the client to draw the item on the screen, but it does 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.

[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.

Eager Verification Performance

We ran our eager client verifier on the same 2,000-round *XPilot* game log and on the same computer used in Section 3.3.3. Figure 3.3(c) describes the per-round validation cost (in seconds) using a box-and-whiskers plot. As in Figure 3.3(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 Section 3.3.4) equal to the client sends in that round. Similar to Figure 3.3(b), Figure 3.3(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 recall that they were by far the least frequent. The verifier's memory usage remained below 100MB throughout these verification runs.

Comparing Figures 3.3(b) and 3.3(c), the times for the eager approach are much 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 are much more complex than in the lazy approach, where 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 7 of Figure 3.2) 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 Section 3.3.4, achieving feasible generation of eager round constraints required substantial additional manual tuning, and consequently greater opportunity for programmer error. As such, it appears that the eager approach is inferior to the lazy approach for *XPilot*. Another comparison between the two approaches, with differing results, will be given in Section 3.4.

3.4 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.

3.4.1 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 navigating 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] 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 new bomb until their previous bomb has detonated.

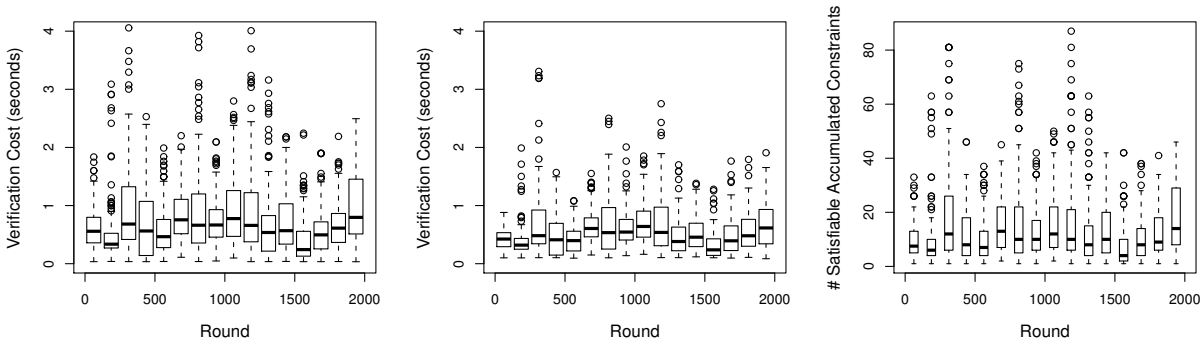
Cap-Man uses a client-server architecture, which we designed specifically to go against current game-development best practices: i.e., it is the *server*, not the client, which has a minimum of authoritative state. The client tracks his own map position, power-up-mode time remaining, and bomb-placement 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*.

---

[2] In our preliminary work [8], 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 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 3.5. Verifying a 2,000-round *Cap-Man* game log

3.4.2 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 3.5 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 3.5(a)) and a mean and standard deviation of 310ms and 193ms for verification using eager round constraints (Figure 3.5(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. The verifier's memory usage remained below 100MB throughout verification of both types. While in the *XPilot* case study, eager verification required significantly greater development effort (see Section 3.3.4), this additional effort was unnecessary with *Cap-Man* due to its relative simplicity.

Figure 3.5(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* 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.

3.5 Case Study: *TetriNET*

As discussed earlier, our verification technique presents opportunities to reduce the bandwidth consumed by an online game, since it allows the client's management of state to be verified with less-than-complete information. In our third case study, we used a pre-existing game called *TetriNET* in order to explore simple bandwidth-savings measures and their impact on client verification performance.

3.5.1 The Game

TetriNET is a multiplayer clone of the classic puzzle game *Tetris*. It was originally developed in 1997 but remains popular and has been reimplemented on many different platforms. In the game, random *tetrominoes*, which are geometric shapes consisting of four connected blocks, automatically advance down each player's playing field, one at a time. As each does, the player can rotate it into four possible orientations or slide it horizontally left or right. The objective of the game is to arrange tetrominoes so that, as they land, they create gapless horizontal rows of blocks on the playing field. When such a gapless row is created, it is cleared, each row of blocks above it falls down one row, and — in the primary multiplayer departure from *Tetris* — a line with gaps is added to the bottom

of the other players' fields. A player loses when there is no room for an additional tetromino to enter her playing field. *TetriNET* is implemented in C and has a client-server architecture similar to *Cap-Man* and *XPilot*. The server does virtually no checking on client messages and so is vulnerable to cheats; e.g., a client could indicate it cleared a row that has gaps or placed a new tetromino in a spot that the client should not have allowed it to reach.

Enabling the use of our verification tool with *TetriNET* required some consideration of how the user input operates. A tetromino in play automatically moves down one row every 0.5 seconds, and when the piece can move no further, the position is fixed and a new random piece starts falling from the top of the game screen. The placement of the tetromino in a permanent resting point defines the end of single round of gameplay, at which time the x and y coordinates and the rotation z are sent to the server. Until the end of the round, though, the player can move the piece horizontally or rotate it as many times as she wishes. So, even though there is a finite number of final fixed positions for a game piece in a given round, there are theoretically an infinite number of possible inputs sequences that could lead to each valid final position. To minimize the number of input sequences that must be explored symbolically, we considered a restricted version of gameplay where the gameboard must be empty above each tetrimino at the time of its placement, and so three rotations and six horizontal moves sufficed to reach any such placement on the 12-column gameboard.

3.5.2 Evaluation

To demonstrate bandwidth reduction in *TetriNET* using our verification technique, we simply reduced the information in the client-to-server messages. *TetriNET* was modified so that only every k rounds was the complete tuple (x, y, z) sent to the server (an "unabridged message"). In other rounds, the client sent a partial tuple, omitting x, y, or z or both y and z. Figure 3.6 shows the tradeoff between the bandwidth reductions accomplished and the costs of verifying client behavior using the lazy client verifier, where the bandwidth reductions were calculated assuming x, y and z are sent in four, five, and two bits, respectively. (The actual *TetriNET* implementation is not engineered for bandwidth reduction and so uses payload space more wastefully.) These graphs each represent five random play sessions of 100 rounds each and the same five play sessions were used for each of the four experiments. Note that $k = 1$ is equivalent to verification of an unmodified game client. In the experiment where y is omitted, the verification cost does not increase because

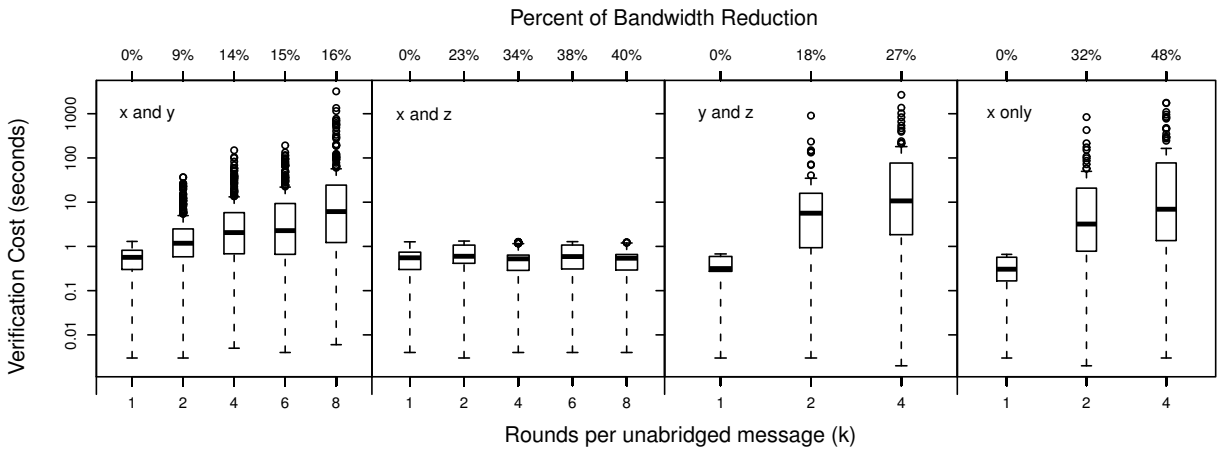


Figure 3.6. Verification of 100-round *TetriNET* logs under different configurations of client-to-server message content.

there is no ambiguity as to y's value when x and z are provided. During these verification runs, the verifier's memory usage remained below 512MB.

3.6 Verification with 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 Section 3.7). 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 3.2. 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. Because it is possible for a valid client on a poor network connection to generate indistinguishable behavior, this cheat is not in the class that our verifier detects. Nevertheless, as discussed in Chapter 2, our techniques are compatible with existing methods that address this type of cheat.

Another consequence of message loss is that the performance of verification can be severely impacted by it. The performance results in Section 3.3–3.4 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 a function of (only) 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.

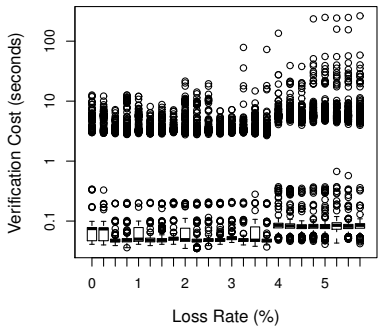


Figure 3.7. Verification (lazy) of 2000-round *XPilot* log with loss of client-to-server messages at the rate indicated on horizontal axis.

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* and *TetriNET* use TCP and so do 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.

To show the effectiveness of this approach, we repeated the lazy verification of 2000-round *XPilot* game logs using *XPilot*-specific optimizations (c.f., Figure 3.3(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 3.7 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 3.7 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 3.8 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 too large to be practical.

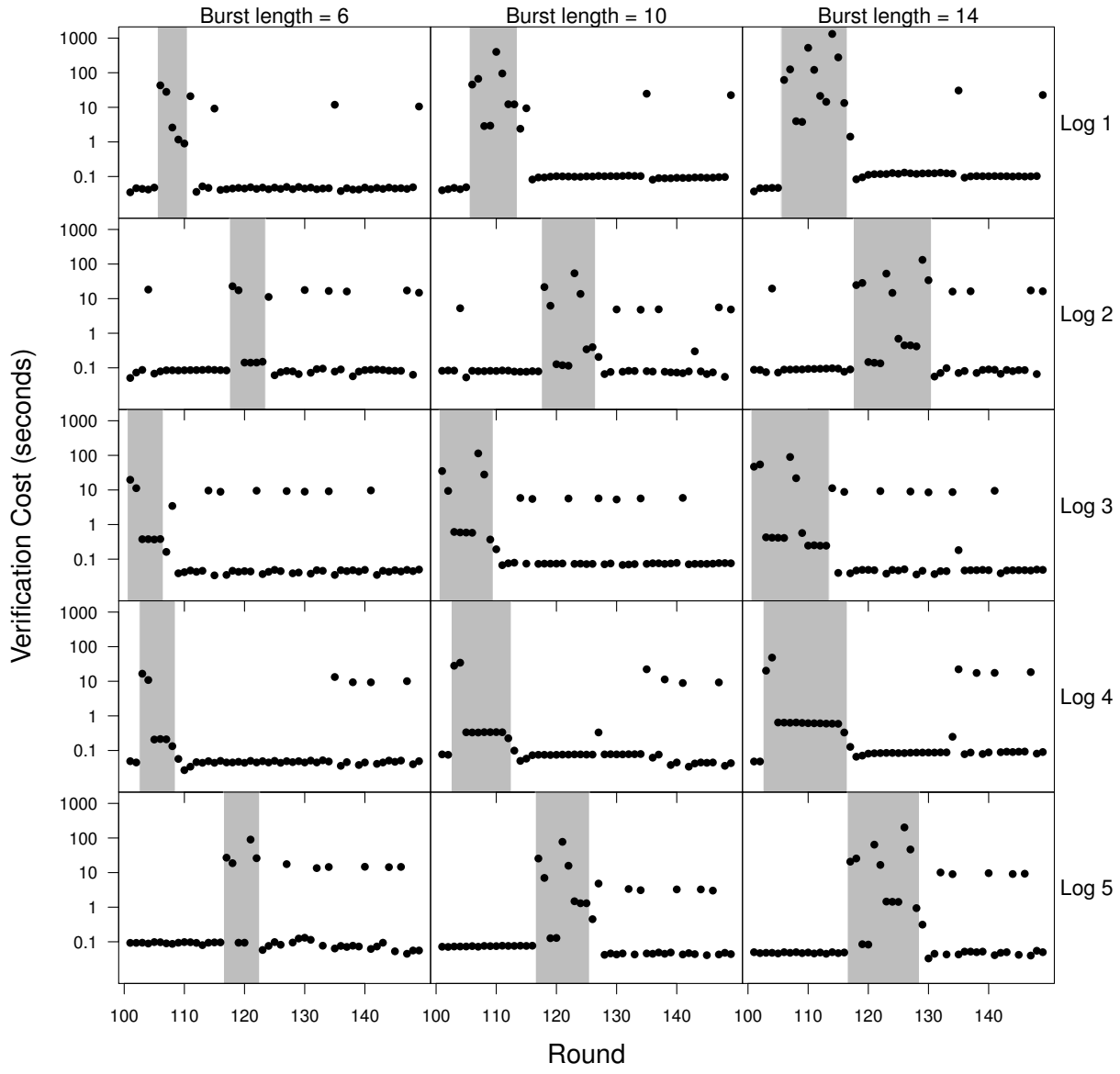


Figure 3.8. Verification (lazy) of *XPilot* logs with randomly induced bursts of client-to-server message losses. Shaded areas designate rounds in which losses occurred.

3.7 Acknowledgement Scheme for *XPilot*

As discussed in Section 3.3.2, 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 [47], 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.

3.8 Summary

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

We demonstrated our technique in the context of online games with three case studies. In the first, we applied our validation approach to *XPilot*, an existing open-source 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 applying 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 then showed simple ways to leverage our technique to reduce bandwidth consumption in a game called *TetriNET*, and returned to *XPilot* to demonstrate a strategy for dealing with message loss.

CHAPTER 4: GUIDED CLIENT VERIFICATION

In this chapter we develop a client-checking algorithm that retains precision while permitting better tradeoffs between bandwidth costs and computational expense in the common case of a legitimate client. Our algorithm builds from the approach of Chapter 3 but makes use of a training phase to guide a search for a path through the client program that could have produced a message observed at the server. One configuration of our algorithm incurs no additional bandwidth costs, like the approach in Chapter 3, but completes verification much more efficiently in the common case of a legitimate client. Another configuration of our algorithm consumes minimal additional bandwidth — in our tests, at most two bytes per client-to-server message — and completes verification even faster in the common case of a legitimate client. Moreover, we reiterate that our algorithm is precise in the sense of having no false negatives and no false positives. That is, any sequence of client messages that our technique declares legitimate actually is, in the sense that there exist inputs that would have driven the sanctioned client software to send that sequence of messages,[1] and any sequence of client messages that our technique declares impossible is actually inconsistent with the client software.

To definitively conclude that a sequence of client messages is impossible (the uncommon case), our algorithm incurs a cost similar to the technique in Chapter 3. As such, we expect our algorithm to be useful primarily as an data reduction technique that prunes the client messages that must be logged for offline analysis by that (or another) technique. In addition, clients whose messages are not verified quickly by our technique can be serviced only provisionally (e.g., with fewer privileges and/or logging to enable undoing their effects) while their verification is completed offline.

We evaluate our algorithm in the context of online games. Online games provide a useful proving ground for our techniques due to the frequent manipulation of game clients for the purposes of cheating [93, 61, 90] and due to the pressure that game developers face to minimize the bandwidth

[1] More precisely, the only source of false negatives is the fidelity of modeling values returned by components with which the client software interacts (e.g., the client OS). This will be discussed further in Section 4.7.

consumed by their games [67]. Our evaluations show, for example, that verifying the behavior of a valid client in the *TetriNET* game can often keep up with the pace of gameplay. Moreover, our algorithm succeeds in verifying legitimate messages traces of the highly interactive *XPilot* game without game restrictions required in Chapter 3.

The technique that we develop here is an extension of the technique presented in Chapter 3 and as such, is also an application of symbolic execution [12]. Dynamic analysis techniques like symbolic execution typically face scaling challenges as code complexity and execution length grow, and our case is no exception. We believe that the technique we develop here to prioritize path analysis on the basis of historical usage may be more broadly useful, i.e., outside of behavior verification in distributed systems, to contain the expense of dynamic analysis.

The rest of this chapter is structured as follows. We discuss necessary background in Section 4.1, and present our algorithm in Section 4.2 and Section 4.3. Evaluation results for this algorithm are presented in Section 4.7, and we conclude in Section 4.8.

4.1 Goals, Assumptions and Limitations

The optimistic verification technique [26] is optimized for the common case of verifying a message trace from a legitimate client. The method is designed to retain precision while permitting better a tradeoff between bandwidth costs and computational expense. We make use of a training stage to inform the verifier of past execution paths and an optional instrumented version of the client client software that can convey "hints" while running that can aid verification.

As in Chapter 3, the goal that motivates this chapter is the construction of a verifier to detect a client in that exhibits behavior, as seen by the server, that is inconsistent with the sanctioned client software and the application state known at the server. That is, the verifier discerns whether there was *any possible sequence* of 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 client based on previous messages from the client and the messages the server sent to the client. In doing so, our approach should enable an automated, server-side validation procedure for client messages.

More specifically, consider a sequence of messages $msg_0, msg_1, \ldots$ that were sent or received by the client, listed in the order in which the client sent or received them; we call such a sequence a

message trace. Because the server received or sent, respectively, each of these messages, the server knows their contents. In Chapter 3 we described an efficient method for the client to inform the server of the order in which the client processed these messages. As such, the message trace $msg_0, msg_1, \ldots$ is known to the server and, so, the verifier. We do not consider the loss of client-to-server messages here in this chapter, though we could employ equally well the method to recover from such losses that we can from Chapter 3.

The verifier's goal is to find a sequence of client instructions, called an *execution prefix* and denoted Π, that begins at the client entry point and is *consistent* with the message trace $msg_0, msg_1, \ldots$. In this thesis we consider only single-threaded clients, and so Π must represent single-threaded execution. More specifically, Π_n is *consistent* with $msg_0, msg_1, \ldots, msg_n$ if the network I/O instructions (SEND and RECV) in Π_n number $n+1$ and match $msg_0, msg_1, \ldots, msg_n$ by type — i.e., if msg_i is a client-to-server message (respectively, server-to-client message), then the i-th network I/O instruction is a SEND (respectively, RECV) — and if the branches taken in Π_n were possible given the contents of $msg_0, msg_1, \ldots, msg_n$. There may be many prefixes Π consistent with $msg_0, msg_1, \ldots$ (e.g., depending on inputs to the client, such as user inputs or system-call return values), but if there are none, then the trace $msg_0, msg_1, \ldots$ is impossible given the sanctioned client software.

The goal of the verifier is simply to determine if there exists an execution prefix that is consistent with $msg_0, msg_1, \ldots$; if not, then the verifier detects the client as compromised. Assuming that client compromise is rare, our goal is to optimize locating such a prefix so that legitimate clients (the common case) can be verified as quickly as possible. While ideally both validation of legitimate clients and detection of compromised clients would be achieved online (i.e., at the pace of message receipt), the number of execution prefixes to explore through the client will generally make it infeasible to definitively detect a compromised client, since doing so requires checking that there is no prefix Π that is consistent with the message trace $msg_0, msg_1, \ldots$. However, we seek to show that through judicious design of the verifier, it can validate most legitimate clients quickly. Requests from clients that the server cannot validate quickly can then be subjected to stricter (though presumably more expensive) sandboxing and/or logging for further analysis offline.

4.2 Training

The algorithm we present in this chapter to meet the goals described in Section 4.1 incorporates a training phase that is used to configure the verifier.

4.2.1 Requirements

The training phase uses message traces of client behavior that should reflect to the greatest degree possible the actual client behavior that will be subjected to verification. For example, in the case of a client-server game, the training phase should make use of message traces of valid gameplay. We stress that the training phase requires only valid message traces (i.e., for which there exists an execution prefix consistent with each), and any invalid message traces will be detected as such during the training process (albeit at substantial computational expense). As such, there is no risk of "poisoning" the training process with invalid message traces, and gathering valid message traces for training purposes can be done by executing the sanctioned client software artificially or by recording message traces from actual client-server sessions.

4.2.2 Algorithm

As we will discuss in Section 4.3, during verification the verifier will attempt to find an execution prefix Π_n that is consistent with the message trace $msg_0, \ldots, msg_n$ incrementally, i.e., by appending to an execution prefix Π_{n-1} that is consistent with $msg_0, \ldots, msg_{n-1}$. To do so, it searches through *execution fragments* in an effort to find one that it can append to create Π_n. The goal of the training phase, then, is to determine the order in which to search possible execution fragments.

More specifically, let an *execution fragment* be any nonempty path (i) beginning at the client entry point, a SELECT, or a SEND in the client software, (ii) ending at a SEND or RECV, and (iii) having no intervening SEND or RECV instructions. Training produces a set Φ of execution fragments. As we will discuss in Section 4.3, the verifier will examine execution fragments in an order guided by Φ to extend an execution prefix Π_{n-1} to reach an execution prefix Π_n that is consistent with a message trace $msg_0, \ldots, msg_n$. Ideally, Φ would include the execution fragments that are commonly exercised during execution or reasonable approximations thereof.

The algorithm for constructing Φ starts from at least one message trace $msg_0, msg_1, \ldots$ and execution prefix Π that is consistent with it. We do not necessarily require that Π is the *actual* execution prefix that was executed to produce the trace, though if that execution prefix could be recorded for the purposes of training, then it will certainly suffice. Alternatively, Π could be produced from the trace (in an offline fashion) using techniques from Chapter 3.

Given the execution prefix Π, the algorithm symbolically executes the sanctioned client software on the path Π, maintaining the resulting symbolic state throughout this execution. This symbolic state consists of memory regions populated by symbolic values with constraints. The constraints on symbolic values are those implied by execution of the path Π; e.g., every branch condition involving a symbolic value will generally add another constraint on that value, perhaps in relation to other symbolic values. A memory region is *concrete* if it is constrained to be a single value.

From this symbolic execution, the training algorithm generates a "postcondition" for each distinct execution fragment contained in Π. Specifically, after each execution of a fragment in Π, the constraints on the symbolic state form a "postcondition term" for that fragment. The disjunction of all postcondition terms collected after execution of the same fragment then forms the postcondition for that fragment. Moreover, since the same fragment may appear in other execution prefixes $\hat{\Pi}$, the postcondition terms from all such executions can contribute to the postcondition of the fragment.

We use this postcondition to then determine the messages in each trace with which the fragment is consistent, where "consistent" has a meaning analogous to, but somewhat more generous than, that for execution prefixes with respect to message traces. Specifically, an execution fragment is *consistent with* a message msg if the fragment ends at an appropriate network I/O instruction — SEND if msg is a client-to-server message, RECV otherwise — and in the case of a SEND, if the fragment postcondition does not contradict the possibility that msg was sent in that SEND or, in other words, if the postcondition and the asserted message contents do not imply false.

Once the set of execution fragments consistent with each message is found, the next step of the algorithm divisively clusters the execution fragments. The fragments are first clustered by the type of their last instructions (SEND or RECV) and then by their starting instructions; i.e., at the second level, all fragments in the same cluster start at the same instruction and end at the same type of network I/O instruction. Finally, each of these level-two clusters is clustered so that fragments that are only small deviations from each other (in terms of the instructions executed) are in the

same cluster. Specifically, each level-two cluster is clustered by (Levenshtein) edit distance using k-medoid clustering to a fixed number of clusters k (or fewer if there are fewer than k fragments in a level-two cluster). Once the execution fragments are clustered by edit distance, the medoid of each cluster is added to Φ. In addition, all training messages consistent with any fragment are retained as *indicators* for the fragment's cluster (and the cluster's medoid).

4.3 Verification

In this section, we discuss how the verifier, for the next message msg_n in a message trace, utilizes the clustering described in Section 4.2 to guide its search for an execution fragment of the client to "explain" the client's progress through it sending or receiving msg_n. More specifically, the verifier does so by finding an execution fragment to append to an execution prefix Π_{n-1} that is consistent with $msg_0, \ldots, msg_{n-1}$, in order to produce an execution prefix Π_n that is consistent with $msg_0, \ldots, msg_n$.

Before describing the verifier algorithm, there are two important caveats to note. First, even if there is an execution fragment that, appended to Π_{n-1}, yields a Π_n that is consistent with $msg_0, \ldots, msg_n$, it may be that this fragment is not contained in Φ. Recall that Φ is only a *partial* list of all execution fragments; it includes only the medoid fragments after clustering the execution fragments from training. As such, it will not suffice for us to limit our attention only to the execution fragments in Φ, and indeed a central innovation in our work is how we use Φ to *guide* the search for execution fragments without being limited to it.

Second, even if the client is behaving legitimately, there may be no execution fragment that can be appended to Π_{n-1} to produce an execution prefix Π_n that is consistent with $msg_0, \ldots, msg_n$. In this case, Π_{n-1} could not have been the path executed by the client through $msg_0, \ldots, msg_{n-1}$. So, the verifier will need to *backtrack* to search for another $\hat{\Pi}_{n-1}$ that is consistent with $msg_0, \ldots, msg_{n-1}$, which the verifier will then try to extend to find a Π_n consistent with $msg_0, \ldots, msg_n$. Of course, backtracking can re-enter previous message verifications, as well, and in the limit, can devolve into an exhaustive search for a path from the client entry point that is consistent with $msg_0, \ldots, msg_n$. If and only if this exhaustive search concludes with no consistent path, the client is detected as behaving inconsistently with the sanctioned client software and this exhaustive search will generally

be costly. However, for the applications we consider in Section 4.7, legitimate clients rarely require backtracking. Combined with optimizations to backtracking that we describe in Section 4.5, our algorithm is a step toward making it possible to quickly verify legitimate clients for such applications and triage those it cannot for further checking later (and sandboxing in the interim).

4.3.1 Guided Verification Algorithm

The verification algorithm takes as input an execution prefix Π_{n-1} consistent with $msg_0, \ldots, msg_{n-1}$ and that ends with the SEND or RECV at which msg_{n-1} was sent or received. The verifier can symbolically execute the sanctioned client software on the path Π_{n-1}, using the concrete messages $msg_0, \ldots, msg_{n-1}$ as those sent or received at the corresponding network I/O instructions in Π_{n-1}, to yield the symbolic state σ_{n-1} of the client.

Preprocessing for a server-to-client message

If msg_{n-1} is a server-to-client message, then presumably msg_{n-1} most directly influenced client execution immediately after it was received. So, our algorithm to produce a Π_n consistent with $msg_0, \ldots, msg_n$ first performs a preprocessing step by symbolically executing σ_{n-1} forward using the server-to-client message msg_{n-1} as the message received in its last instruction (which is a RECV). σ_{n-1} is a symbolic state and so may branch on symbolic variables as it is executed forward (even though msg_{n-1} is concrete); preprocessing explores paths in increasing order of the number of symbolic variables they include so far. This search continues until a path encounters an instruction that suggests that the processing of msg_{n-1} by the client is complete — specifically, upon encountering a SELECT or a SEND. The path starting from σ_{n-1} until this instruction are used to extend Π_{n-1} (and σ_{n-1}) to produce Π^+_{n-1} (and σ^+_{n-1}).

If msg_{n-1} is a client-to-server message, then no such preprocessing step is necessary. In this case, let Π^+_{n-1} and σ^+_{n-1} be Π_{n-1} and σ_{n-1}, respectively.

Overview of basic verification algorithm

The core of the verification algorithm starts from the symbolic state σ^+_{n-1} and uses a subset $\Phi_n \subseteq \Phi$ to guide a search for an execution fragment that can be appended to Π^+_{n-1} to yield Π_n that is consistent with $msg_0, \ldots, msg_n$. Intuitively, Φ_n includes the execution fragments from Φ that

are deemed likely to be similar to the fragment executed by the client leading up to it sending or receiving msg_n. We defer discussing the selection of Φ_n to Section 4.6; here we simply stipulate that each fragment in Φ_n begins at the instruction pointed to by the program counter of σ^+_{n-1} and ends at a SEND or RECV if msg_n is a client-to-server message or a server-to-client message, respectively. We stress that despite these constraints, appending a $\phi \in \Phi_n$ to Π^+_{n-1} will not necessarily yield a Π_n consistent with $msg_0, \ldots, msg_n$.

Our verification algorithm executes roughly as follows. The algorithm builds a strictly binary tree of paths, each starting from the next instruction to be executed in σ^+_{n-1}. (Here, by "strictly" we mean that every non-leaf node has exactly two children, not one.) The root of the tree is the empty path, and the two children of a node in the tree extend the path represented by that node through the next symbolic branch (i.e., branch instruction involving a symbolic variable). One child represents that branch evaluating to false, and the other represents that branch evaluating to true. The algorithm succeeds in finding a fragment with which to extend Π^+_{n-1} to yield Π_n if, upon extending a path, it encounters a network I/O instruction that can "explain" msg_n, i.e., that yields a state with constraints that do not contradict msg_n being the network I/O instruction's message.

Perhaps the central idea in our algorithm, though, is the manner in which it selects the next node of the tree to extend. For this purpose it uses the training fragments Φ_n. There are any number of approaches, but the one we evaluate here selects the path to extend to be the one that minimizes the edit distance to some prefix of a fragment in Φ_n (and that has not already been extended or found to be inconsistent). This strategy naturally leads to first examining the fragments in Φ_n, then other fragments that are small modifications to those in Φ_n, and finally other fragments that are further from the fragments in Φ_n. This algorithm will be detailed more specifically below.

Algorithm Client Verification

100: **procedure** verify(σ^+_{n-1}, msg_n, Φ_n)
101: nd $\leftarrow$ makeNode() ▷ Initialize root node
102: nd.path $\leftarrow \langle\rangle$
103: nd.state $\leftarrow \sigma^+_{n-1}$
104: Live $\leftarrow \{\mathsf{nd}\}$
105: **while** |Live| > 0 **do**
106: nd $\leftarrow \arg \min_{\mathsf{nd}' \in \mathsf{Live}} \min_{\phi \in \Phi_n} \min_{\phi' \sqsubseteq \phi} \mathsf{editDist}(\mathsf{nd}'.\mathsf{path}, \phi')$ ▷ Select node
107: Live $\leftarrow$ Live $\setminus \{\mathsf{nd}\}$
108: $\pi \leftarrow$ nd.path
109: $\sigma \leftarrow$ nd.state
110: **while** σ.next $\neq \bot$ **and** isNetInstr(σ.next) $=$ **false and** isSymbolicBranch(σ.next) $=$ **false do**
111: $\pi \leftarrow \pi \parallel \langle \sigma.\mathsf{next} \rangle$
112: $\sigma \leftarrow \mathsf{execStep}(\sigma)$
113: **if** isNetInstr(σ.next) $=$ **true then**
114: **if** $((\sigma.\mathsf{constraints} \wedge \sigma.\mathsf{next.msg} = msg_n) \not\Rightarrow$ **false**$)$ **then**
115: **return** $\pi \parallel \langle \sigma.\mathsf{next} \rangle$ ▷ Success!
116: **else if** isSymbolicBranch(σ.next) $=$ **true then**
117: nd.child$_0 \leftarrow$ makeNode()
118: nd.child$_0$.path $\leftarrow \pi \parallel \langle \sigma.\mathsf{next} \rangle$
119: nd.child$_0$.state $\leftarrow [\ \mathsf{execStep}(\sigma)\ \mid \sigma.\mathsf{next.cond} \mapsto$ **false** $]$
120: **if** nd.child$_0$.state.constraints $\not\Rightarrow$ **false then**
121: Live $\leftarrow$ Live $\cup \{\mathsf{nd.child}_0\}$
122: nd.child$_1 \leftarrow$ makeNode()
123: nd.child$_1$.path $\leftarrow \pi \parallel \langle \sigma.\mathsf{next} \rangle$
124: nd.child$_1$.state $\leftarrow [\ \mathsf{execStep}(\sigma)\ \mid \sigma.\mathsf{next.cond} \mapsto$ **true** $]$
125: **if** nd.child$_1$.state.constraints $\not\Rightarrow$ **false then**
126: Live $\leftarrow$ Live $\cup \{\mathsf{nd.child}_1\}$
127: **return** $\bot$ ▷ Failure

Figure 4.1. Basic verification algorithm, described in Section 4.3.1

Details of basic verification algorithm

The algorithm for verifying a client-to-server message is summarized more specifically in Figure 4.1. This algorithm, denoted verify, takes as input the symbolic state σ^+_{n-1} resulting from execution of Π_{n-1} from the client entry point on message trace $msg_0, \ldots, msg_{n-1}$ and then the preprocessing step described above if msg_{n-1} is a server-to-client message; the next message msg_n; and the execution fragments Φ_n described above (and detailed in Section 4.6). Its output is either an execution fragment that can be appended to Π^+_{n-1} to make Π_n that is consistent with msg_0, ..., msg_n, or undefined ($\perp$). The latter case indicates failure and, more specifically, that there is no execution prefix that can extend Π^+_{n-1} to make Π_n that is consistent with msg_0, ..., msg_{n-1}. This will induce the backtracking described above to search for another $\hat{\Pi}_{n-1}$ that is consistent with msg_0, ..., msg_{n-1}, which the verifier will then try to extend to find a Π_n consistent with msg_0, ..., msg_n.

The aforementioned binary tree is assembled as a collection of nodes created in lines 101, 117, and 122 in Figure 4.1. Each node has fields path, state, and children child_0 and child_1. The root node nd is initialized with $\mathsf{nd.path} = \langle\rangle$ and $\mathsf{nd.state} = \sigma^+_{n-1}$ (103). Initially only the root is created (101–103) and added to a set Live (104), which includes the nodes that are candidates for extending. The algorithm executes a **while** loop (105–126) while Live includes nodes (105) and the algorithm has not already returned (115). If the **while** loop exits because Live becomes empty, then the algorithm has failed to find a suitable execution fragment and $\perp$ is returned (127).

This **while** loop begins by selecting a node nd from Live that minimizes the edit distance to some prefix of a fragment in Φ_n; see line 106, where $\phi' \sqsubseteq \phi$ denotes that ϕ' is a prefix of ϕ. The selected node is then removed from Live (107) since any node will be extended only once. The state σ of this node (109) is then executed forward one step at a time ($\sigma \leftarrow \mathsf{execStep}(\sigma)$, line 112) and the execution path recorded ($\pi \leftarrow \pi \parallel \langle \sigma.\mathsf{next} \rangle$, where $\parallel$ denotes concatenation) until this stepwise execution encounters the client exit ($\sigma.\mathsf{next} = \perp$, line 110), a network I/O instruction ($\mathsf{isNetInstr}(\sigma.\mathsf{next}) =$ **true**), or a symbolic branch ($\mathsf{isSymbolicBranch}(\sigma.\mathsf{next}) =$ **true**). In the first case ($\sigma.\mathsf{next} = \perp$), execution of the main **while** loop (105) continues to the next iteration. In the second case ($\mathsf{isNetInstr}(\sigma.\mathsf{next}) =$ **true**) and if the constraints $\sigma.\mathsf{constraints}$ accumulated so far with the symbolic state σ do not contradict the possibility that the network I/O message $\sigma.\mathsf{next.msg}$ in the

next instruction σ.next is msg_n (i.e., (σ.constraints $\wedge$ σ.next.msg $= msg_n$) $\not\Rightarrow$ **false**, line 113), then the algorithm returns successfully since $\pi \parallel \langle\sigma.\mathsf{next}\rangle$ is an execution fragment that meets the verifier's goals (115).

Finally, in the third case (isSymbolicBranch(σ.next) $=$ **true**), the algorithm explores the two possible ways of extending π, namely by executing σ.next conditioned on the branch condition evaluating to **false** (denoted [execStep(σ) | σ.next.cond $\mapsto$ **false**] in line 119) and conditioned on the branch condition evaluating to **true** (124). In each case, the constraints of the resulting state are checked for consistency (120, 125) and the consistent states are added to Live (121, 126).

4.4 Edit-distance calculations

As discussed previously, one insight employed in our verify algorithm is to explore paths close to the training fragments Φ_n first, in terms of edit distance (line 106). Edit distance between strings s_1 and s_2 can be computed by textbook dynamic programming in time $O(|s_1| \cdot |s_2|)$ and space $O(\min(|s_1|, |s_2|))$ where $|s_1|$ denotes the character length of s_1 and similarly for s_2. While reasonably efficient, this cost can become significant for large s_1 or s_2.

For this reason, our implementation optimizes the edit distance computations. To do so, we leverage an algorithm due to Ukkonen [80] that tests whether editDist$(s_1, s_2) \leq t$ and, if so, computes editDist(s_1, s_2) in time $O(t \cdot \min(|s_1|, |s_2|))$ and space $O(\min(t, |s_1|, |s_2|))$ for a parameter t. By starting with a small value for t, we can quickly find nodes nd$' \in$ Live such that for some $\phi \in \Phi_n$ and $\phi' \sqsubseteq \phi$, editDist(nd$'$.path, ϕ') $\leq t$. Only after such nodes are exhausted, do we then increase t and re-evaluate the nodes still in Live. By proceeding in this fashion, verify incurs cost per edit-distance calculation of $O(t \cdot \min(|s_1|, |s_2|))$ for the distance threshold t when the algorithm returns, versus $O(|s_1| \cdot |s_2|)$.

Second, when calculating editDist(nd$'$.path, ϕ), it is possible to reuse intermediate results from a previous calculation of editDist(nd$'$.path, ϕ') in proportion to the length of the longest common prefix of ϕ and ϕ'. (Since Φ_n contains only fragments beginning with the instruction to which the program counter points in σ_{n-1}, their common prefix is guaranteed to be of positive length.) To take maximum advantage of this opportunity to reuse previous calculations, we organize the elements of Φ_n in a prefix tree (trie), in which each internal node stores the intermediate results that can be reused when calculating editDist(nd$'$.path, ϕ) for the execution fragments Φ_n that share the prefix

represented by the interior node. In a similar way, the calculation of editDist(nd'.path, ϕ) can reuse intermediate results from the editDist(nd.path, ϕ) calculation, where nd'.path extends nd.path. In this way, the vast majority of edit distance calculations are built by reusing intermediate results from others.

Third, though the verification algorithm as presented in Figure 4.1 assembles each path π instruction-by-instruction (lines 110–112), the paths nd'.path and fragments Φ_n are not represented as strings of instructions for the purposes of the edit distance calculation in line 106. If they were, it would not be atypical for these strings to be of lengths in the tens of thousands for some of the applications we consider in Section 4.7, yielding expensive edit-distance calculations. Instead, nd'.path and Φ_n are represented as strings of basic block identifiers for the purposes of computing their edit distance. In our evaluation, this representation resulted in strings that were roughly an order of magnitude shorter than if they had been represented as strings of instructions.

4.4.1 Judicious use of edit distance

Despite the optimizations just described, calculating edit distances incurs a degree of overhead. As such, we have found that for highly interactive applications, it is important to employ edit distance only when Φ_n is likely to provide a useful guide in finding a π with which to extend Π_{n-1} to obtain Π_n.

For the applications with which we have experimented, the primary case where using edit distance is counterproductive is when $\min_{\phi \in \Phi_n} \min_{\phi' \sqsubseteq \phi} \mathsf{editDist}(\mathsf{nd}'.\mathsf{path}, \phi')$ is large for every $\mathsf{nd}' \in \mathsf{Live}$. Because nodes are explored in increasing order of their edit distances from their nearest prefixes of training fragments, this condition is an indication that the training fragments Φ_n are not a good predictor of what happened in the client application leading up to the send or receipt of msg_n. This condition implies that verify now has little useful information to guide its search and so no search strategy is likely to be a clear winner, and thus in this case we abandon the use of edit distance to avoid calculating it. That is, we amend verify so that when

$$\min_{\mathsf{nd}' \in \mathsf{Live}} \min_{\phi \in \Phi_n} \min_{\phi' \sqsubseteq \phi} \mathsf{editDist}(\mathsf{nd}'.\mathsf{path}, \phi') > d_{\max}$$

for a fixed parameter $d_{\max}$ ($d_{\max} = 64$ in our experiments in Section 4.7), verify transitions to selecting nodes $\mathsf{nd}' \in \mathsf{Live}$ in increasing order of the number of symbolic variables introduced on $\mathsf{nd}'.\mathsf{path}$. The rationale for this choice is that it tends to prioritize those states that reflect fewer user inputs and is very inexpensive to track.

4.4.2 Selecting nd

In each iteration of the main **while** loop 105–126 of verify, the next node nd to extend is selected as that in Live with a minimum "weight," where its weight is defined by its edit distance to a prefix of an element of Φ_n. Since the only operations on Live are inserting new nodes into it (lines 121, 126) and extracting a node of minimum weight (line 106), Live is represented as a binary min-heap. This enables both an insertion of a new element and the removal of its min-weight element to complete in $O(\log |\mathsf{Live}|)$ time where $|\mathsf{Live}|$ denotes the number of elements it contains when the operation is performed. This (only) logarithmic cost is critical since Live can grow to be quite large; e.g., in our tests described in Section 4.7, it was not uncommon for Live to grow to tens of thousands of elements.

4.4.3 Memory management

The verification algorithm, upon traversing a symbolic branch, creates new symbolic states to represent the two possible outcomes of the branch (lines 119 and 124). Each state representation includes the program counter, stack and address space contents. While KLEE [17] (on which we build) provides copy-on-write semantics for the address-space component, it does not provide for garbage collection of allocated memory or a method to compactly represent these states in memory. To manage the considerable growth in memory usage during a long running verification task, we utilize a caching system that selectively frees in-memory representations of a state if necessary. If at a later time a freed state representation is needed (due to backtracking, for example), our system reconstructs the state from a previously checkpointed state. This method adds to the overall verification time but reduces the extent to which memory is a limiting factor.

4.5 Backtracking and Equivalent State Detection

As discussed at the start of Section 4.3, if verify(σ^+_{n-1}, msg_n, Φ_n) returns $\perp$ (line 127), then it is not possible that the client legitimately executed Π^+_{n-1}, producing state σ^+_{n-1}, and then sent/received msg_n. If msg_{n-1} is a client-to-server message (and so $\Pi^+_{n-1} = \Pi_{n-1}$), verification must then *backtrack* into the computation verify(σ^+_{n-2}, msg_{n-1}, Φ_{n-1}) to find a different fragment to append to Π^+_{n-2} to yield a new execution prefix $\hat{\Pi}_{n-1}$ consistent with msg_0, ..., msg_{n-1} and resulting in state $\hat{\sigma}_{n-1}$. Once it does so, it invokes verify($\hat{\sigma}_{n-1}, msg_n, \hat{\Phi}_n$) to try again. To support this backtracking, upon a successful return from verify(σ^+_{n-2}, msg_{n-1}, Φ_{n-1}) in line 115, it is necessary to save the existing algorithm state (i.e., its Live set and the states of the nodes it contains) to enable it to be restarted from where it left off. If msg_{n-1} is a server-to-client message (and so $\Pi^+_{n-1} \neq \Pi_{n-1}$), then backtracking is performed similarly, except the computation of verify(σ^+_{n-2}, msg_{n-1}, Φ_{n-1}) is resumed only after all possible extensions $\hat{\Pi}^+_{n-1}$ of Π_{n-1} have been exhausted, i.e., each corresponding verify($\hat{\sigma}^+_{n-1}$, msg_n, $\hat{\Phi}_n$) has failed.

The most significant performance optimization that we have implemented for backtracking is a method to detect the equivalence of some symbolic states, i.e., for which execution from these states (on the same inputs) will behave identically. If the states σ_{n-1} and $\hat{\sigma}_{n-1}$ are equivalent and if a valid client could not send msg_n after reaching σ_{n-1}, then equivalently it could not send msg_n after reaching $\hat{\sigma}_{n-1}$. So, for example, if $\hat{\sigma}_{n-1}$ was reached due to backtracking after verify(σ_{n-1}, msg_n, Φ_n) failed, then the new execution prefix $\hat{\Pi}_{n-1}$ that produces $\hat{\sigma}_{n-1}$ should be abandoned immediately and backtracking should resume again.

The difficulty in establishing the equivalence of σ_{n-1} and $\hat{\sigma}_{n-1}$, if they are in fact equivalent, is that they may not be syntactically equal. This lack of equality arises from at least two factors. The first is that in our present implementation, the address spaces of the states σ_{n-1} and $\hat{\sigma}_{n-1}$ are not the same, but rather are disjoint ranges of the virtual address space of the verifier. Maintaining disjoint address spaces for symbolic states is useful to enable their addresses to be passed to external calls (e.g., system calls) during symbolic execution. It also requires us to assume that the client program execution is invariant to the range from which its addresses are drawn, but we believe this property is true of the vast majority of well-behaved client applications (including the ones we use in our evaluation).

A second factor that may cause σ_{n-1} and $\hat{\sigma}_{n-1}$ to be syntactically distinct while still being equivalent is that the different execution prefixes Π_{n-1} and $\hat{\Pi}_{n-1}$ leading to these states may induce differences in their pointer values. Consider, for example, the trivial C function in Figure 4.2, which reads an input character and then branches based on its value; in one branch, it allocates `*buf1` and then `*buf2`, and in the other branch, it allocates `*buf2` and then `*buf1`. Even if the address spaces of different states occupied the same ranges, and even if the memory allocator assigned memory deterministically (as a function of the order and size of the allocations), the addresses of `buf1` and `buf2` would be different in states that differ only because they explored different directions of the symbolic branch `if (c == 'x')`. These states would nevertheless be equivalent, assuming that the client application behavior is invariant to its state's pointer values (again, a reasonable assumption for well-behaved applications).

```
void foo(char **buf1, char **buf2) {
    char c;
    c = getchar();
    if (c == 'x') {
        *buf1 = (char *) malloc(10);
        *buf2 = (char *) malloc(10);
    } else {
        *buf2 = (char *) malloc(10);
        *buf1 = (char *) malloc(10);
    }
}
```

Figure 4.2. Example code that may induce different pointer values for variables in otherwise equivalent states.

To detect equivalent states σ_{n-1} and $\hat{\sigma}_{n-1}$ that are syntactically unequal due to the above causes, we built a procedure to walk the memory of two states in tandem. The memory of each is traversed in lock-step and in a canonical order, starting from each concrete pointer in its global variables (including the stack pointer) and following each concrete pointer to the memory to which it points. (Pointers are recognized by their usage.) Concrete, non-pointer values traversed simultaneously are compared for equality; unequal values cause the traversal to terminate with an indication that

the states are not equivalent.[2] Similarly, structural differences in simultaneously traversed memory regions (e.g., regions of different sizes, or a concrete value in one where a symbolic value is in the other) terminate the traversal. Symbolic memory locations encountered at the same point in the traversal of each state are given a common name, and this common name is propagated to any constraints that involve that location. Finally, equivalence of these constraints is determined by using a constraint solver to determine if each implies the other. If so, the states are declared equivalent.

4.6 Configurations

Thus far, we have not specified how Φ_n is populated from the set Φ of medoids resulting from clustering the execution fragments witnessed during training (Section 4.2). We consider two possibilities for populating Φ_n in this chapter.

4.6.1 Default configuration

The default algorithm configuration constructs Φ_n from the contents of msg_n. If the closest training message is at distance m from msg_n, for a measure of distance described below, then the algorithm computes the set M_n^α of training messages less than distance αm from msg_n, for a fixed parameter $\alpha \geq 1$. (In Section 4.7, we use $\alpha = 1.25$.) An execution fragment ϕ is *eligible* to be included in Φ_n if (i) ϕ is the medoid of some cluster for which there is an indicator message $msg \in M_n^\alpha$, and (ii) ϕ begins at the instruction to which the program counter in σ_{n-1}^+ points, where σ_{n-1}^+ is the symbolic state that will be passed to verify along with msg_n and Φ_n. Then, Φ_n is set to include all eligible fragments up to a limit β; if there are more than β eligible fragments, then Φ_n consists of an arbitrary subset of size β. (In Section 4.7, we use $\beta = 8$.)

The distance measure between messages that we use in our evaluation in Section 4.7 is simply byte edit distance between messages of the same directionality (i.e., between server-to-client messages or between client-to-server messages). If msg and msg_n do not have the same directionality,

[2]The state could still be equivalent if the differing concrete values do not influence execution, but our method does not detect the states as equivalent in this case.

then we define their distance to be ∞, so that only training messages of the same directionality as msg_n are included in M_n^α.

4.6.2 Hint configuration

The "hint" configuration requires that the client software has been adapted to work with the verifier to facilitate its verification. In this configuration, the client piggybacks a hint on msg_n that is a direct indication of the execution fragment it executed prior to sending msg_n. This extra hint, however, increases the bandwidth utilized by client-to-server messages, and so it is important that we minimize this cost.

Specifically, in this configuration, the client software has knowledge of the clustering used by the verifier, as described in Section 4.2.2. (For example, the server sends it this information when the client connects.) The client records its own execution path and, when sending a client-to-server message msg_n, maps its immediately preceding execution fragment to its closest cluster in the verifier's clustering (using edit distance on execution fragments). The client then includes the index of this cluster within msg_n as a "hint" to the verifier. The server extracts the cluster index from msg_n and provides this to the verifier.

Intuitively, the medoid ϕ of the indicated cluster should be used as the sole element of the set Φ_n, but only if ϕ begins at the instruction pointed to by the program counter of the symbolic state σ_{n-1}^+ to be provided to verify as input.[3] For the applications we evaluate in Section 4.7, however, we adapt this idea slightly and interpret this cluster index within the set of all clusters whose fragments begin at that instruction and end at a SEND. Then, Φ_n is set to contain only the medoid of this cluster. (If the cluster index exceeds the number of clusters whose fragments begin at that instruction, or if msg_n is a server-to-client message, then the default approach above is used to create Φ_n, instead.) In this way, the cluster hint can be conveyed in exactly $\lceil \log_2 k \rceil$ extra bits on each client-to-server message, where k is the number of clusters allowed by the verifier in its third level of clustering (see Section 4.2.2). While sending a hint does increase bandwidth cost, it does so minimally; e.g., in

[3] An alternative is for the verifier to backtrack immediately if ϕ begins at a different instruction, since in that case, σ_{n-1}^+ is apparently not representative of the client's state when it executed the fragment leading up to it sending msg_n. For the applications we evaluate in Section 4.7, however, backtracking usually incurred more verification cost even in this case.

Section 4.7, we consider $k = 256$ (1 byte per client-to-server message) and $k \leq 65536$ (2 bytes per client-to-server message).

Despite the fact that the client sends the hint to the server, the client remains completely untrusted in this configuration. The hint it provides is simply to accelerate verification of a legitimate client, and providing an incorrect hint does not substantially impact the verifier's cost for declaring the client compromised.

4.7 Evaluation

To evaluate our technique, we built a prototype that uses the KLEE [17] symbolic execution engine as a foundation. Our implementation includes approximately 1000 modified source lines of code (SLOC) in KLEE and additional 10,000 SLOC in C++. That said, at present we have not completed the client-side implementation of the hint configuration described in Section 4.6, and so we instead simulate the client-side hint in our evaluation here. We stress, therefore, that while we accurately measure the verifier's performance in both the default and hint configurations, the additional client overheads implied by the hint configuration are not reported here. The experiments described in this section were performed on a system with 24GB of RAM and a 2.93GHz processor (Intel X5670).

We limit our evaluation to the verifier's *performance*. By design, our verification algorithm has no false positives — i.e., if a message trace is declared to be inconsistent with the sanctioned client software, then it really is (though this is subject to an assumption discussed in Section 4.5). Similarly, the only source of false negatives arises from the limited fidelity of the constraints used to model values returned by components with which the client software interacts (e.g., the OS). We could improve that fidelity by subjecting these components to symbolic execution, as well, but here we limit symbolic execution to the client software proper.

To evaluate performance, we apply our algorithm to verify behavior of legitimate clients of two open-source games, namely *XPilot* and *TetriNET* (described in Section 4.7). We limit our attention to *legitimate* clients since this is the case in which we make a contribution; i.e., our approach is designed to validate legal behavior more quickly than previous work, but confirms illegal behavior in time comparable to what the previous technique in Chapter 3 would achieve. We employ these games for our evaluation for numerous reasons: they are complex and so pose challenging test

cases for our technique; they are open-source (and our tools require access to source code); and games is a domain that warrants behavior verification due to the invalid-command cheats to which they are often subjected [90].

In our evaluation we observe two data points for each msg_n in a message trace, verification *cost* and verification *delay*. The verification cost for message msg_n, denoted $cost(n)$, accounts for all time spent in verify(σ_{n-1}, msg_n, Φ_n). This is measured as the wall-clock time from the algorithm takes to determine that msg_n is valid.[4] Our second data point for evaluation is the verification *delay*, which is the delay in time between the arrival of message msg_n at the server (where a server-to-client message "arrives" when it is sent) and the discovery of an execution prefix Π_n that is consistent with $msg_0, \ldots, msg_n$. Delay differs from verification time by representing the fact that verification for msg_n cannot begin until after verification for msg_{n-1} completes. To formally define delay, we first define the verification *completion time*, $\mathsf{T}_{comp}()$, for msg_n inductively as follows:

$$\mathsf{T}_{comp}(0) = cost(0)$$
$$\mathsf{T}_{comp}(n) = \max\{\mathsf{T}_{arr}(n), \mathsf{T}_{comp}(n-1)\} + cost(n)$$

where $\mathsf{T}_{arr}(n)$ is the wall-clock time when msg_n arrived at the verifier. Since the verification of msg_n cannot begin until after both (i) it is received at the verifier, at time $\mathsf{T}_{arr}(n)$ and (ii) the previous messages $msg_0, \ldots, msg_{n-1}$ have completed verification, at time $\mathsf{T}_{comp}(n-1)$. $\mathsf{T}_{comp}(n)$ is calculated as the cost $cost(n)$ incurred after both conditions (i) and (ii) are met. Then, the *delay* of msg_n is $delay(n) = \mathsf{T}_{comp}(n) - \mathsf{T}_{arr}(n)$.

XPilot

XPilot is an open-source, multi-player, client-server game that has been developed in many revisions over more than 15 years including, e.g., a version for the iPhone that was released in July 2009. The version we used in our evaluation is *XPilot NG* (*XPilot Next Generation*) version 4.7.2. Its client consists of roughly 100,000 SLOC. Beyond this, the scope of symbolic execution included all

[4] If backtracking occurs, that time is included in the verification time of the associated round, i.e., the verification time for msg_n may include the total wall clock time of more than one instance of verify.

needed libraries except `Xlib`, whose functions were replaced with minimal stubs, so that the game could be run without display output. Moreover, `uClibc` was used in lieu of the GNU C library.

In the game, the user causes her spaceship to navigate a two-dimensional world occupied by obstacles, objects such as power-ups that the user can collect by navigating her spaceship over them, and both fixed and mobile hostiles that can fire on her ship (some of which are ships controlled by other players). Each player's goal is to earn the highest score. Despite its "2D" graphics, the game incorporates sophisticated physics simulation; e.g., ships with more fuel have greater mass and thus greater inertia.

Upon startup, the *XPilot* client reads local files that, e.g., define the world map. (Our evaluation assumes that these initialization files are available to the verifier, as they must be to the server, as well.) The *XPilot* client then enters an event loop that receives user input and server messages, processes them (including rendering suitable changes on the client's display), and sends an update to the server. These updates can include information about the current status of the user's ship's shields (whether they are up or down), weapons (whether any are firing), position, orientation, acceleration, etc. Various limitations imposed by the client, such as that a client cannot both have its shields up and be firing at the same time, are obvious targets for a user to override by modifying the game client in order to cheat. Our behavior verification technique will detect such game cheats automatically.

In Chapter 3 it was necessary to modify the *XPilot* client in various small ways to make its analysis tractable. We used this modified version in our evaluations in this chapter, as well, though to illustrate certain improvements enabled by our technique, we reverted an important modification. Recall that in Chapter 3, we inserted bounds to limit the number of user inputs that would be processed in any given event-loop round, since otherwise the event loop could theoretically process an unbounded number of such inputs. This unboundedness, in turn, would cause symbolic execution to explore an arbitrary numbers of corresponding input-processing loop iterations. By inserting bounds, we rectified this problem but introduced a potential source of false positives, if the deployed client software were execute more than the bounded number of input events in a given event-loop round. Here we removed these inserted limits so as to eliminate this risk of false positives and also to highlight the power of training our verifier on previous executions. After removing these limits, these input-processing loops could theoretically iterate an arbitrary number of times,

but nevertheless our verifier does not explore paths including increasingly large numbers of such iterations until it is done exploring paths with numbers of iterations similar to those encountered in the training runs.

TetriNET

TetriNET is a multi-player version of the popular single-player *Tetris* game. In the *Tetris* game, one player controls a rectangular gameboard of squares, at the top of which a *tetromino* appears and starts to "fall" toward the bottom at a constant rate. Each tetromino is of a size to occupy four connected grid squares orthogonally and has one of seven shapes. The tetromino retains its shape and size as it falls, but the user can reorient the tetromino as it falls by pressing keys to rotate it. The user can also move the tetromino to the left or right by pressing other keys. Once the tetromino lands on top of another tetromino or the bottom of the grid, it can no longer be moved or rotated. At that point, another tetromino appears at the top of the grid and begins to fall. Whenever a full row of the gameboard is occupied by tetrominos, the row disappears (potentially fracturing any tetrominos occupying a portion of it) and all rows above the removed row are shifted downward. *TetriNET* differs from *Tetris* by *adding* an empty row to all other players' grids when this occurs. The goal of the game is for a player to place as many tetrominos as possible before no more can enter her gameboard, and a player wins the multiplayer version by playing longer than other players.

The *TetriNET* client is structured as an event loop that processes user inputs and advances each tetromino in its fall down the gameboard. Only once a tetromino has landed in its resting place does the game client inform the server of the location of the tetromino and whether its placement caused any rows to be deleted (and if so, which ones). The server does not validate the client's claim that the condition for deleting the row was met (i.e., that the row was full), and so the game is very vulnerable to invalid-command cheats. Again, our technique will automatically detect such cheats.

The *TetriNET* client version (0.11) that we used in our evaluation is 5000 SLOC. As in *XPilot*, the scope of symbolic execution also included all needed libraries, though again the display output library (`ncurses`) was disabled using minimal stub functions and `uClibc` was used in place of the GNU C library. Despite its small size, a single event-loop iteration in the *TetriNET* client permits an unbounded number of user inputs to rotate or horizontally shift the tetromino, which presents problems for symbolic execution analogous to those that led us to cap the number of inputs in a

single *XPilot* event-loop iteration for exhausitive symbolic client verification in Chapter 3. As such, in the experimentation with *TetriNET*, we also limited gameplay so that a tetromino could be placed only at a location for which only empty squares were above it, so as to limit the number of user inputs needed for a tetromino placement to half the width of the gameboard plus the number of possible tetromino rotations — nine user inputs in total. We emphasize that none of these restrictions are employed in this chapter, and again the ability of our algorithm to verify the behavior of a *TetriNET* client in its unconstrained form illustrates its strengths.

Evaluation of our verification algorithm requires traces of gameplay for both training and testing. For *TetriNET*, we generated 20 traces of manual gameplay, each of 240 messages in length (which corresponds to roughly 6.5 minutes of gameplay). For *XPilot*, we generated 40 traces of manual gameplay, each consisting of 2100 messages (roughly 70 seconds of gameplay).

4.7.1 Case Study: *TetriNET*

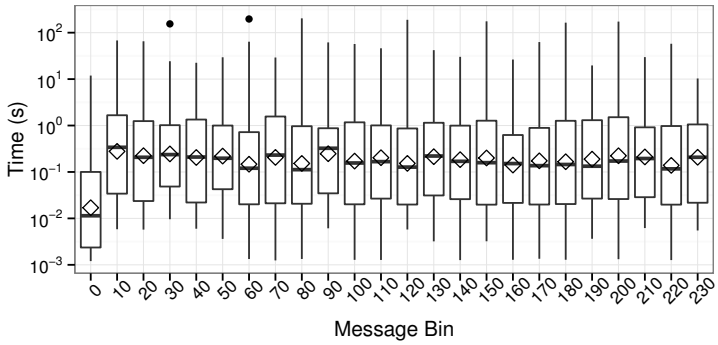
Figure 4.3 shows *TetriNET* verification costs. Figure 4.3 includes plots for both the default and hint configurations, as well as for clustering parameter values $k = 256$ and $k = 3790$; the latter case ensured a single execution fragment per cluster.

The numbers represented in Figure 4.3 were obtained by a 20-fold cross validation of the *TetriNET* traces; i.e., in each test, one of the traces was selected for testing, and the remainder were used for training. Specifically, Figure 4.3 shows the distribution of verification costs per message, binned into ten-message bins, across all 20 traces. So, for example, the boxplot labeled "0" shows the distribution of verification costs for messages $msg_0, \ldots, msg_9$ in the 20 traces. The data point for message msg_n accounts for all time spent in verify$(\sigma^+_{n-1}, msg_n, \Phi_n)$ and any immediately preceding preprocessing step (see Section 4.3.1), including any backtracking into those functions that occur. (That said, backtracking in *TetriNET* is very rare.) In each boxplot, the "box" shows the first, second (median) and third quartiles, and its whiskers extend to ± 1.5 times the interquartile range. Additional outlier points are shown as bullets. Overlaid on each boxplot is a diamond ($\Diamond$) that shows the average of the data points.

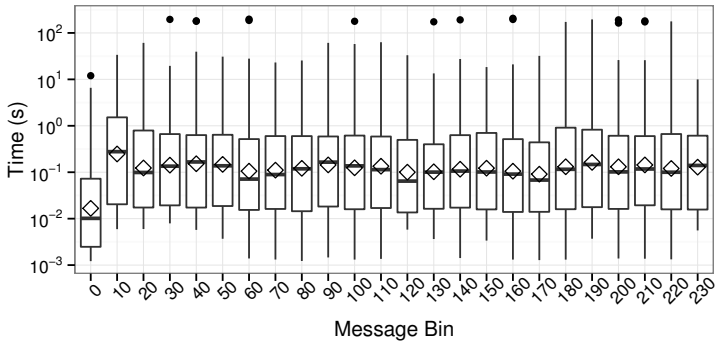
Several things are worth noting about Figure 4.3. In all cases, the distribution of verification costs is largely unaffected by the message index, i.e., where in the trace the message appears. This confirms that our implementation is mostly free from sources of increasing verification expense as

traces grow longer. This figure also confirms that more fine-grained clustering ($k = 3790$) leads to faster verification times than coarse grained ($k = 256$). Fine-grain clustering, however, results in greater bandwidth use in the hint configuration; $k = 3790$ implies an overhead of 12 bits or, if sent as two bytes, an average of 17% bandwidth increase per client-to-server message, in contrast to only 9% per client-to-server message for $k = 256$. Not surprisingly, the hint configuration generally outperforms the default.

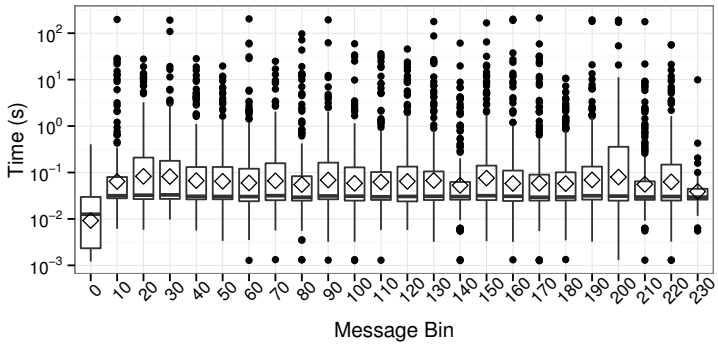
Figure 4.3 also suggests that our algorithm is, for the large majority of messages, fast enough to verify valid *TetriNET* gameplay at a pace faster than the game itself: the average verification cost per message, regardless of configuration or clustering granularity, is easily beneath the inter-message times of roughly 1.6s. That said, there are two issues that require further exploration. First, there are messages that induce verification cost in excess of 10s or even 100s, which unfortunately makes it impossible to reliably keep pace with gameplay. Nevertheless, as an optimization over previous work for verifying message traces, and as a data reduction technique to eliminate some traces (or portions thereof) from the need to log and analyze offline, our technique still holds considerable promise. Second, and working in favor of this promise, is the slack time between the arrival of messages that gives verification the opportunity to catch up to the pace of gameplay after a particularly difficult message verification.



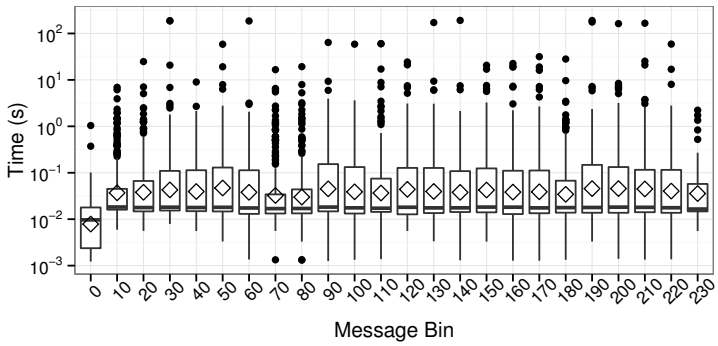
(a) Default, $k = 256$



(b) Hint, $k = 256$

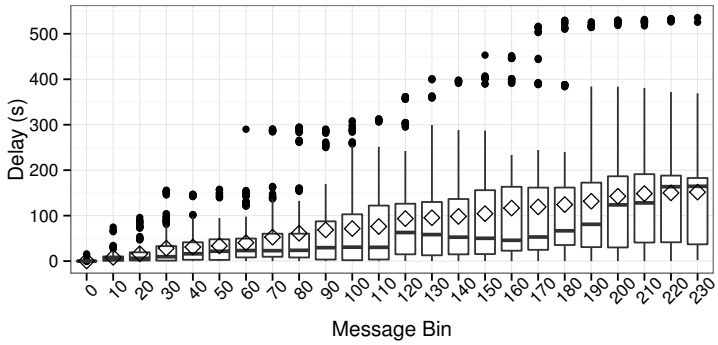


(c) Default, $k = 3790$

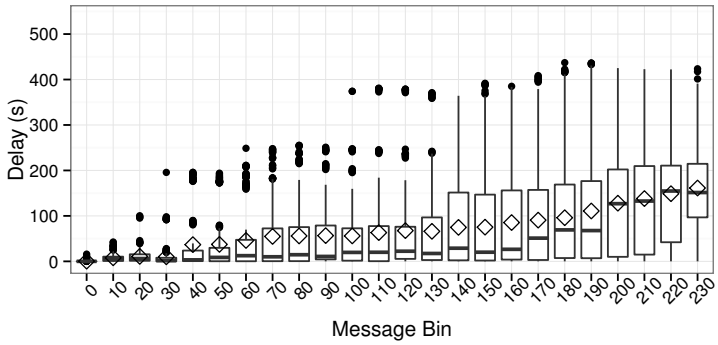


(d) Hint, $k = 3790$

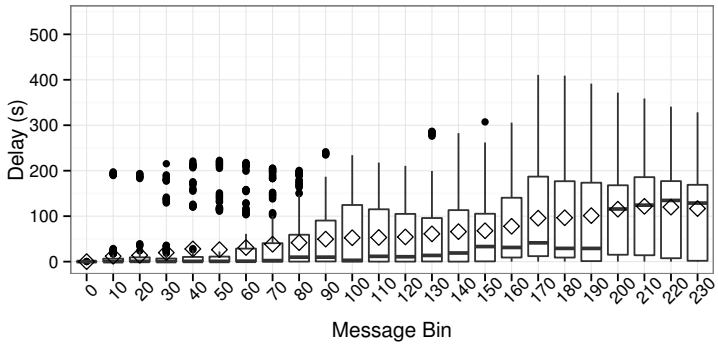
Figure 4.3. *TetriNET* verification costs. Cross-validation over 20 traces. Boxplot at x shows verification costs for messages msg_x, ..., msg_{x+9} in each trace (after training on the other traces). "$\Diamond$" shows the average.



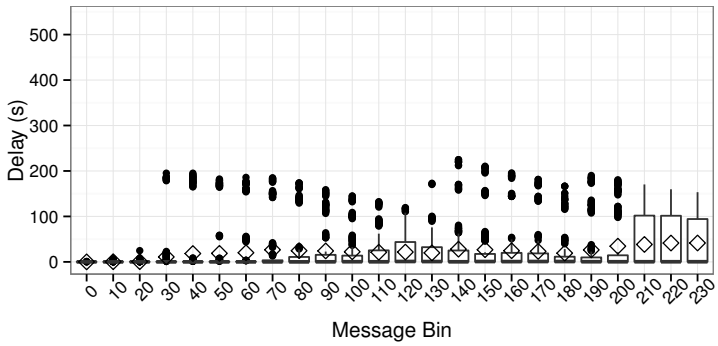
(a) Default, $k = 256$



(b) Hint, $k = 256$



(c) Default, $k = 3790$



(d) Hint, $k = 3790$

Figure 4.4. *TetriNET* verification delays. Cross-validation over 20 traces. Boxplot at x shows verification delays for messages $msg_x, \ldots, msg_{x+9}$ in each trace (after training on the other traces). "$\Diamond$" shows the average.

To shed light on these issues, Figure 4.4 instead plots the distributions of per-message verification *delay* between the arrival of message msg_n at the server (where a server-to-client message "arrives" when it is sent) and the discovery of an execution prefix Π_n that is consistent with $msg_0, \ldots, msg_n$. Delay (Figure 4.4) differs from verification cost (Figure 4.3) by representing the fact that verification for msg_n cannot begin until after that for msg_{n-1} completes. So, for example, the rightmost boxplot in each graph provides insight into how long after the completion of the message trace (in real time) that it took for verification for the whole trace to complete.

One item to note about these graphs is that for the hint configuration with $k = 3790$ (Figure 4.4(d)), the median of the rightmost boxplot is virtually zero — i.e., the most common case is that verification kept pace with gameplay. This can occur even if verification falls behind at some point in the game, since verification commonly "catches up" after falling behind. This is illustrated, for example, in the generally downward slope of consecutive outlier points in Figure 4.4(d). That said, the cumulative effect of verification delays in the other configurations is more costly, e.g., causing verification to lag behind gameplay by more than 100 seconds by the end of a 240-message trace in the median case in the default configuration (Figure 4.4(c)).

A breakdown of verification costs for *TetriNET* is shown in Figure 4.5. In our *TetriNET* experiments, more than 50% of the verification cost is spent in KLEE, interpreting client instructions. Therefore, optimizations that interpret instructions only selectively (e.g., [23]) may be a significant optimization for our tool. The majority of the remaining time is spent in insertions and deletions on Live and in computing edit distance, both to update the edit distance for each path when a symbolic branch is reached and to compute distances between messages. A very small fraction of time in our *TetriNET* experiments is devoted to equivalent state detection (Section 4.5) or in constraint solving. In Figure 4.5, constraint solving includes not only the time spent by STP (the default solver used by KLEE), but also preprocessing techniques to make queries to STP more efficient and a canonicalization step (similar to Visser et al. [83]) to improve the hit rate on cached results for previous queries to STP. These optimizations significantly reduce the overall constraint solving time.

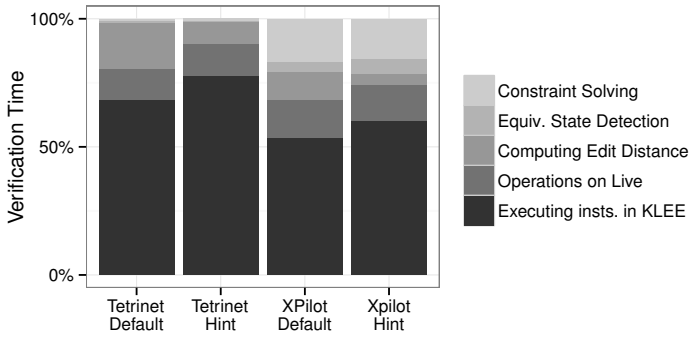


Figure 4.5. Percentage of time spent in each component of the verifier.

4.7.2 Case Study: *XPilot*

XPilot poses a significant challenge for verification because its pace is so fast. The tests described here use an *XPilot* configuration that resulted in an average of 32 messages per second. The verification costs per message vary somewhat less for *XPilot* than they did for *TetriNET*, as shown in Figure 4.6. Recall that each boxplot in Figure 4.6 represents 100×40 points, versus only 10×20 in Figure 4.3. As such, though there are larger numbers of outliers in Figure 4.6, they constitute a smaller fraction of the data points.

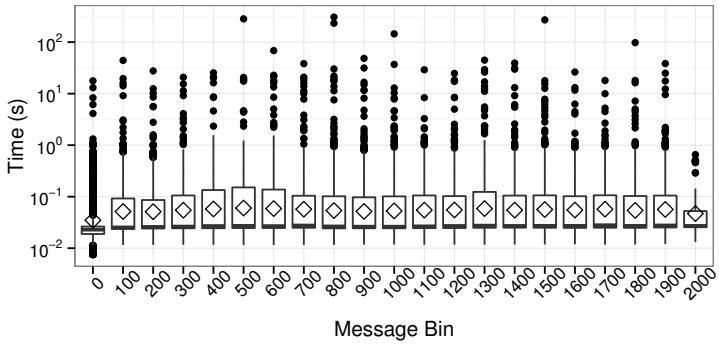
The median per-message verification cost of *XPilot* when clustering is fine-grained ($k = 475$, which implied a single execution fragment per cluster) is quite comparable to that in *TetriNET*, as can be seen by comparing Figure 4.6(c) and Figure 4.6(d) to Figure 4.3(c) and Figure 4.3(d), respectively. However, *XPilot* verification is considerably faster with coarse clustering, see Figure 4.6(a) versus Figure 4.3(a) and Figure 4.6(b) versus Figure 4.3(b). Our definition of $k = 256$ as "coarse" clustering was dictated by the goal of limiting the bandwidth overhead to one byte per client-to-server message in the hint configuration. The better performance of *XPilot* verification for coarse clustering versus *TetriNET* is at least partly because $k = 256$ is closer to fine clustering ($k = 475$) in the case of *XPilot* than it is for *TetriNET* ($k = 3790$). In the hint configuration, $k = 256$ increases bandwidth use by *XPilot* client-to-server messages by 2%, and $k = 475$ (9 bits, sent in two bytes) increases it by 4%.

Though the median per-message verification cost of *XPilot* is generally as good or better than that for *TetriNET*, the faster pace of *XPilot* makes it much more difficult for verification to keep pace with the game. This effect is shown in Figure 4.7. As shown in this figure, none of the configurations or clustering granularities permitted verification to keep up with gameplay, and the best default

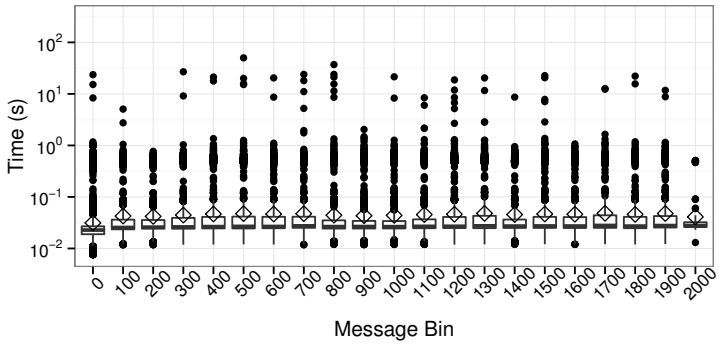
configuration ($k = 475$) included one run that required 8 minutes past the end of the trace to complete its verification (see Figure 4.7(c)). Consequently, for an application as fast-paced and as complex as *XPilot*, our algorithm does not eliminate the need to save traces for post hoc analysis.

Nevertheless, we stress that our algorithm accomplishes — even if with some delay — what is for the approach in Chapter 3 completely intractable. That is, recall that the approach in Chapter 3 utilized a restricted version of *XPilot* in which the number of user inputs per event loop iteration was artificially limited; we have removed that limitation here (see Section 4.7). With these restrictions removed, the previous approach is inherently unbounded for verifying some messages, since it seeks to eagerly find *all* paths that could explain that message, of which there could be infinitely many. Our approach, in contrast, succeeds in verifying all messages in these logs in bounded time and with per-message cost averaging under 100ms in all configurations (Figure 4.6).

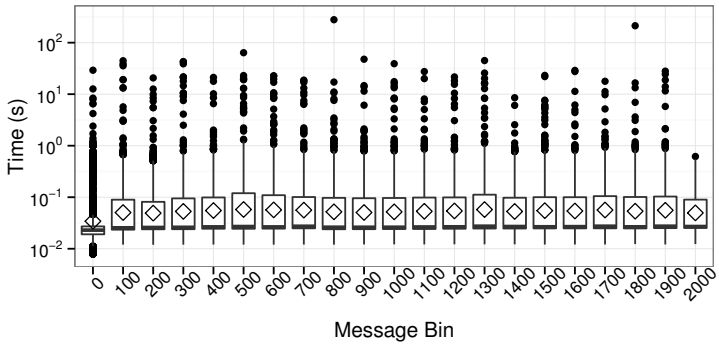
A fractional breakdown of verification costs for *XPilot* are shown in Figure 4.5. While a majority of the cost is still contributed by interpreting client instructions in KLEE, the majority is smaller in the case of *XPilot* than it was for *TetriNET*. For *XPilot*, equivalent state detection (Section 4.5) plays a more prominent role than it did for *TetriNET*, in part due to *XPilot*'s more complex memory structure. Moreover, due to the substantially more complex constraints generated by *XPilot*, constraint solving plays a much more prominent role than it did for *TetriNET*.



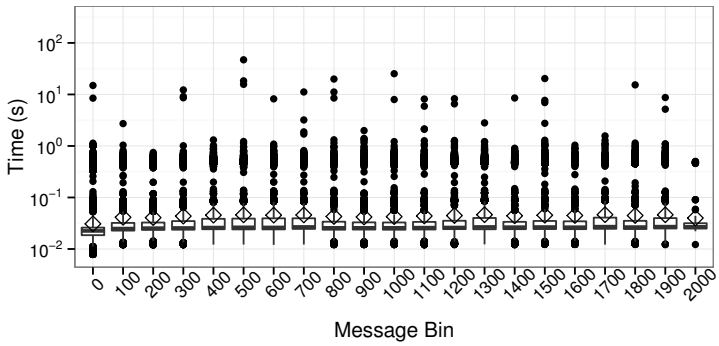
(a) Default, $k = 256$



(b) Hint, $k = 256$

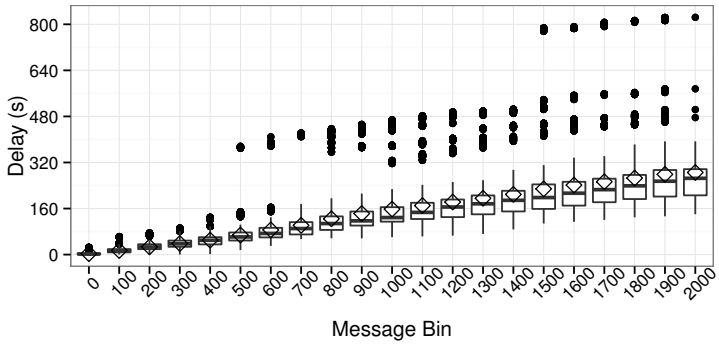


(c) Default, $k = 475$

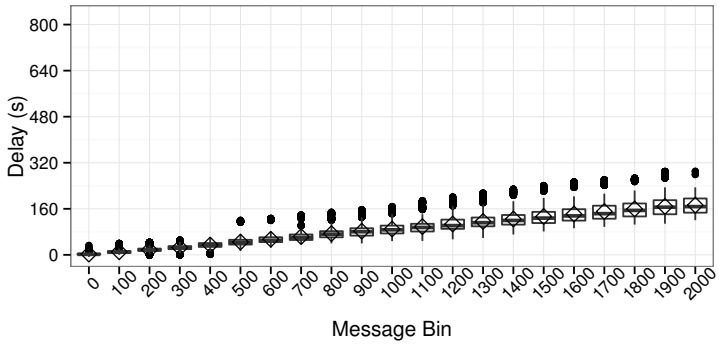


(d) Hint, $k = 475$

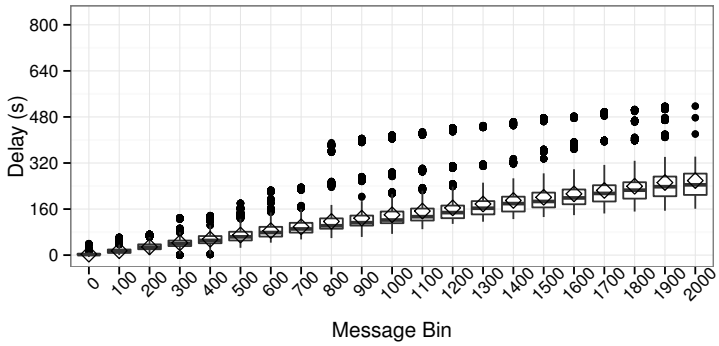
Figure 4.6. *XPilot* verification costs. Cross-validation over 40 traces. Boxplot at x shows verification cost for messages msg_x, ..., msg_{x+99} in each trace (after training on the other traces). "◊" shows the average.



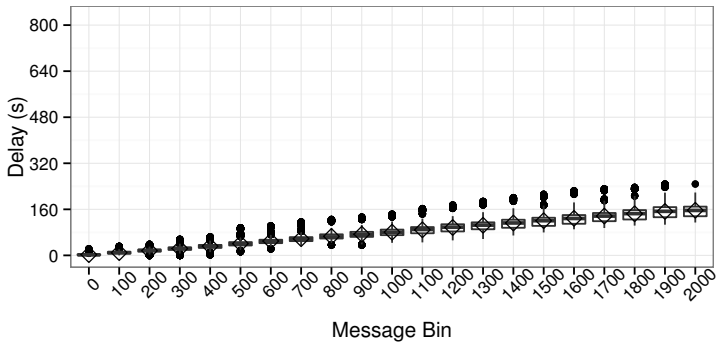
(a) Default, $k = 256$



(b) Hint, $k = 256$



(c) Default, $k = 475$



(d) Hint, $k = 475$

Figure 4.7. *XPilot* verification delays. Cross-validation over 40 traces. Boxplot at x shows verification delays for messages msg_x, ..., msg_{x+99} in each trace (after training on the other traces). "$\Diamond$" shows the average.

4.8 Summary

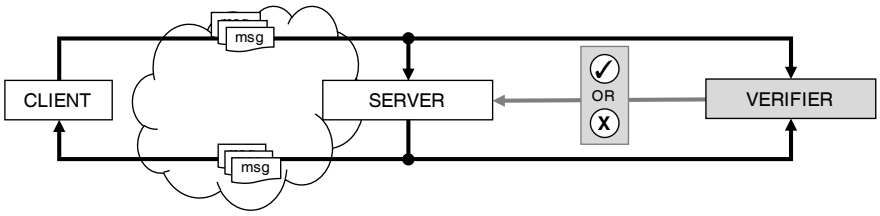
In this chapter we have presented a novel algorithm to enable a server to verify that the behavior of a client in a client-server application is consistent with the sanctioned client software. The central challenge that must be overcome in achieving this goal is that the server does not know all of the inputs to the client (e.g., user inputs) that induced its behavior, and in some domains (see [67]) the additional bandwidth utilized by sending those inputs to the server is undesirable. We therefore developed a technique by which the verifier "solves" for whether there exist user inputs that could explain the client behavior. We overcome the scaling challenges of this approach by leveraging execution history to guide a search for paths through the client program that could produce the messages received by the server. This approach enables us to achieve dramatic cost savings in the common case of a legitimate client, and by allowing minimal additional bandwidth use, we can improve performance even further. In the best configuration of our algorithm, verification of legitimate *TetriNET* gameplay often keeps pace with the game itself. In other cases, verification efficiency is adequate to practically handle client applications that previous work was forced to restrict to make its analysis tractable. We believe that the manner in which we leverage execution history can be useful in other applications of symbolic execution, as well.

CHAPTER 5: PARALLEL CLIENT VERIFICATION

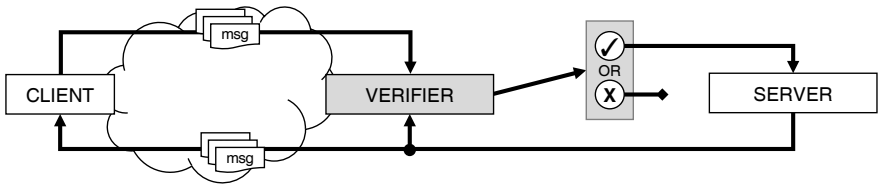
Symbolic client verification can be used to build a verifier that validates traffic received from remote clients. This technique can identify cheating players in multiplayer networked games or identify other types of malicious behavior in distributed systems. Client verification is potentially very useful, especially if the validity result can be computed quickly. Rather than logging network traffic for verification off-line, a fast verifier could be configured to validate traffic as it arrives at the server. The contributions of this chapter are motivated by the goal of keeping a window of attack that is as small as possible, or in other words, minimizing the time it takes the verifier to return a validity result for a given message. The techniques developed thus far in this dissertation have not achieved efficiency levels that would allow a verifier to be placed on the critical path of serving client requests.

At any point in time during verification, there are many possible paths of exploration; choosing the right path of exploration improves the latency of the verifier's result. In the previous chapter we demonstrated that a combination of training data and edit distance metrics can be used to guide the verifier's exploration. This approach is successful because it prioritizes exploration on paths that are more likely to lead to success. However, this algorithm alone cannot keep pace with all of our client application case studies. For a long-running or even continuous client-server session, if the average time to verify a message is greater than the average time between message arrivals, the verifier falls further and further behind. In this chapter, we demonstrate a technique to push forward on multiple paths of exploration in parallel, resulting in significant performance improvements.

The ability to keep pace with client traffic is important because it opens up new avenues for using our client verification technique. If the verification is fast enough, it can operate *in-line*, allowing each client-to-server message to be verified as being consistent with the sanctioned client software, before it is processed by the server. An in-line verification configuration would add some amount of latency to the client-to-server network traffic, but that trade-off may be very acceptable



(a) Off-line Client Verification



(b) In-line Client Verification

Figure 5.1. Off-line and in-line client verification.

in some situations, especially if the added latency is low or if the cost of invalid command attacks is very high. Figure 5.1 shows examples of off-line and in-line verifier configurations; an in-line configuration can drop a malicious message before it has time to propagate to the server. The advantages of in-line client verification include:

- Elimination of the need for logging network traffic for later verification, reducing storage costs.
- Better quality of service for other players in multiplayer games; cheaters would be removed more quickly, improving the game quality for honest players.
- Identification of invalid command messages *before* they are processed by potentially vulnerable server software, crucially saving server operators from potential damage to infrastructure or the leakage of private or sensitive data.

5.1 Goals and Background

Our goal is to rapidly identify clients that from the server's perspective are operating sanctioned client software. In this chapter we demonstrate that we can achieve verification results that can keep pace with our case study applications with a modified version of our verification algorithm. We describe modifications to our client verification technique to utilize thread-level parallelism. Although

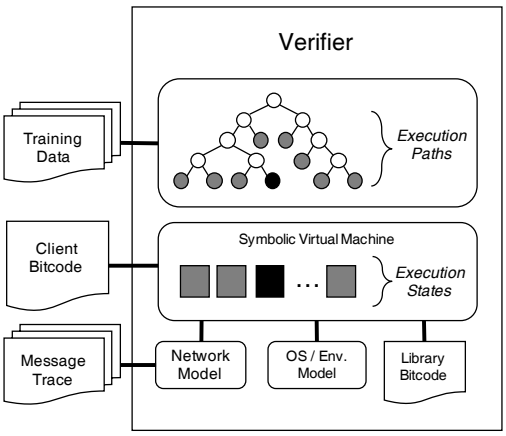


Figure 5.2. Verifier Architecture

using training data can reduce the total number of execution states that must be explored, we also wish to further reduce the time between message arrival and the verification result. Concurrent exploration can achieve this result.

The outline of the rest of this chapter is as follows. We will first review the client verification problem and describe the architecture of our verifier. We then review multi-threading primitives and describe an algorithm for parallel client verification in detail. Following that are an overview of implementation level details and then an evaluation on two client case studies, *TetriNET* and *XPilot*.

5.1.1 Client Verification Overview

We now state our assumptions, describe the architecture of the verifier and the client verification problem.

Assumptions

As in the previous chapter, we are concerned with verifying a client that generates a message trace, msg_0, msg_1, ..., msg_n, some sent by the client and some sent by the server. Furthermore, as before, we assume the software running on the remote client is single-threaded and that the verifier is provided with an ordering of messages according to the client's perspective.

Architecture and Assumptions

In previous chapters, we have described the verifier architecture shown in Figure 5.2. The verifier operates by exploring possible execution paths the client may have taken to find a path that "explains" a given message trace. If no such path can be found, the message trace is declared invalid. The verifier uses symbolic execution to explore possible client paths. To operate, the verifier requires three inputs; a message trace, $msg_0, msg_1, \ldots, msg_n$, which will be verified; the source code for some sanctioned client software, compiled into an intermediate representation (IR) that can be executed by a symbolic virtual machine; and a set of training data for the client bitcode under verification that is used to guide the selection of possible paths of execution. The message trace is incrementally fed to the verifier as it arrives at the server and the verifier outputs if the message trace it has processed thus far can be explained by an execution path in the software that the client is expected to be running.

Client Verification Definition

We formally define the task of the verifier using the same definitions as Chapter 4. The verifier's algorithm determines whether there exists an *execution prefix* of the client that is *consistent* with the messages $msg_0, msg_1, \ldots, msg_n$. Specifically, an execution prefix Π_n is a sequence of client instructions that begins at the client entry point and follows valid branching behavior in the client program. We define Π_n to be *consistent* with msg_0, msg_1, ..., msg_n, if the network SEND and RECV instructions[1] in Π_n number $n+1$ and these network instructions match msg_0, msg_1, ..., msg_n by direction—i.e., if msg_i is a client-to-server message (respectively, server-to-client message), then the i-th network I/O instruction is a SEND (respectively, RECV)—and if the branches taken in Π_n were possible. Consistency of Π_n with msg_0, msg_1, ..., msg_n requires that the conjunction of all symbolic postconditions at SEND instructions along Π_n be satisfiable, once concretized using the contents of messages msg_0, msg_1, ..., msg_i sent and received on that path.

The verifier attempts to validate the sequence msg_0, msg_1, ... incrementally, by verifying the sequence msg_0, msg_1, ..., msg_n starting from an execution prefix Π_{n-1} found to be consistent with

[1]We abbreviate call instructions to POSIX `select()`, `send()` and `recv()` system calls (or their functional equivalents) with the labels SELECT, SEND and RECV.

msg_0, msg_1, ..., msg_{n-1}, and appending to it an *execution fragment* that yields an execution prefix Π_n consistent with msg_0, msg_1, ..., msg_n. Specifically, an *execution fragment* is a nonempty sequence of client instructions (i) beginning at the client entry point, a SELECT, or a SEND in the client software, (ii) ending at a SEND or RECV, and (iii) having no intervening SEND or RECV instructions. This incremental mechanic is important to note, as it is the key step in the verification algorithm that we describe. By using an incremental construction, an entire execution prefix Π_n can be constructed if a given message trace was produced by a valid client. The verification algorithm operates as a step-by-step construction of an execution prefix, because when verification begins, the verifier will only have a single network message msg_0.

Client Verification Backtracking

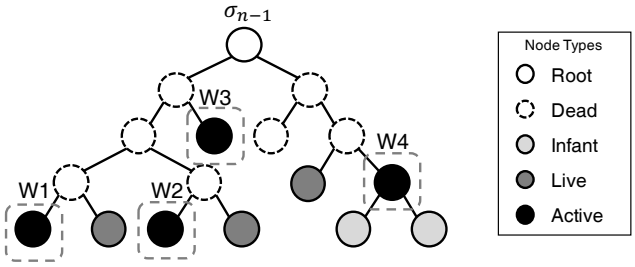
Suppose an execution prefix Π_{n-1} is consistent with a message trace up to message msg_{n-1}. It is possible for there to be no execution fragment that can be appended to produce an execution prefix Π_n that is consistent with the message trace extended by the next message to arrive, msg_n. Intuitively, a path of execution may exercise client state that precludes future client actions or outputs from occurring. The verifier incrementally constructs an execution prefix in an optimistic fashion. In many cases, there is more than one execution prefix consistent with a message trace, but the verifier only needs to find one to show that a client is valid. When the verifier cannot find an execution fragment that can be appended to Π_{n-1} to produce a Π_n consistent with msg_0, msg_1, ..., msg_n, then the verifier backtracks to find a new $\hat{\Pi}_{n-1}$ consistent with msg_0, msg_1, ..., msg_{n-1} that can be extended with an execution fragment to yield a $\hat{\Pi}_n$ consistent with msg_0, msg_1, ..., msg_n. Only after all such attempts fail can the client behavior be declared invalid, which may take substantial time. In practice, an "invalid" declaration usually comes by timeout on the verification process. Our concern in this chapter is verifying the behavior of *valid* clients quickly, irrespective of how long it requires to declare an invalid client as such.

5.2 Parallel Client Verification

We now present a verification algorithm that takes advantage of thread-level parallelism. At a high level, the main work of verification is symbolically executing client code to produce an

```
struct Node {
   SymbolicState        state;
   ExecutionFragment    path;
   struct Node          child[2];
}
```

(a) Node data structure



(b) Node tree with multiple workers attached

Figure 5.3. Node data structure and node tree.

execution prefix (Section 5.1.1) for a given message sequence. The algorithm that we describe outlines the steps required to produce an execution prefix Π_n from an execution prefix Π_{n-1}. In practice, many possible paths of execution need to be explored to produce an execution fragment. Incomplete paths will be dropped upon reaching unsatisfiable states and candidate execution fragments will be dropped if the network action at the end of the fragment is not consistent with the message trace. Our parallel client verification algorithm achieves significant improvements in performance by using thread-level parallelism to explore candidate execution fragments.

5.2.1 Algorithm Definitions

We now define the data structures used by the algorithm. A state σ represents a snapshot of execution in the symbolic virtual machine, including all constraints (path conditions) and memory objects, which includes the contents (symbolic or concrete) of registers, the stack and the heap. The verifier produces state σ_n by symbolically executing the instruction sequence represented by execution prefix Π_n.

The algorithm builds and maintains binary tree, consisting of Node objects, shown in Figure 5.3(a). This tree is rooted with a state σ_{n-1} and an empty path. Figure 5.3(b) shows an example tree, with a root node containing σ_{n-1}. (The remaining node types will be described later in the algorithm description.) The two children of a node in the tree extend Π_{n-1}, the path represented by σ_{n-1}, through the next symbolic branch (i.e., branch instruction with a symbolic condition). One child state holds a constraint that maintains that the branch condition implies false, and the other child state holds a constraint that indicates that the branch condition is true. The algorithm succeeds by finding a fragment with which to extend Π_{n-1} to yield Π_n if, upon extending a path,

it encounters a network I/O instruction that yields a state with constraints that do not contradict msg_n being the network I/O instruction's message.

We define a set of training data Φ to be a set of execution fragments and associated message data, observed and collected via execution of the client prior to verification. During verification, the verifier selects a subset of execution fragments $\Phi_n \subseteq \Phi$, for verification of msg_n. The set Φ_n is used prioritize nodes in the node tree rooted at σ_{n-1}. This prioritization technique is detailed in Chapter 4.

5.2.2 Key Insights

The driving goal of our algorithm is to enable concurrent exploration of multiple states in the node tree. The sooner the algorithm can explore and then eliminate candidate paths, the sooner it can find a valid execution fragment. We designed the parallel verification algorithm to use multiple threads; it uses a single thread to manage the node tree, a single thread to select the training data subset, and several worker threads, each assigned to a single node in the node tree at time. Figure 5.3(b) shows an example assignment of four workers to multiple nodes in a node tree. In our design and experiments the number of worker threads NumWorkers is a fixed parameter provided to the verifier. Because the verification task is largely CPU-bound, it is in our experience not beneficial to use more worker threads than the number of logical CPU cores, and in some cases, fewer worker threads than cores are necessary. The main objective of our design is to get "out of the way" of the worker threads. What follows are the high level features of the design.

- **Concurrent exploration of execution fragments:** The most important contribution of the algorithm is concurrent exploration of execution fragments rooted at σ_{n-1}. This is enabled by a multi-threaded symbolic execution engine (detailed in Section 5.3). As designed, the algorithm explores only execution fragments starting from one state σ_{n-1} corresponding to one execution prefix Π_{n-1}. Although searching from multiple execution prefixes is not fundamentally impossible, our implementation of backtracking requires all worker threads to be exploring fragments in the same node tree.

- **Non-blocking node tree management:** Operations on the node tree operate asynchronously from the worker threads. This is done so that worker threads do not have to block on operations that interact with shared data structures.

- **Non-blocking training data selection:** The parallel verification algorithm extends the method of training data selection described in Chapter 4. As before, when msg_n arrives, the verifier selects from a set of execution fragments Φ, produced during training, that are deemed likely to be similar to the fragment executed by the client leading up to it sending or receiving msg_n; this subset of execution fragments is denoted Φ_n. Depending on the size of the training data and granularity of clustering, the selection of Φ_n may take a significant amount of time. The parallel verification algorithm asynchronously selects Φ_n in a separate thread so that the exploration from σ_{n-1} is not blocked and can begin before Φ_n is available. Once Φ_n has been computed, the node selection algorithm switches from naive breadth-first search to an edit-distance search using the selected subset of execution fragments (Φ_n).

5.2.3 Multi-threading primitives

The description of our algorithm requires the definition of primitives used in the dynamic multi-threading model [27]. These primitives are **spawn**, **sync** and **atomic**. The keyword **spawn** must precede a procedure call and has semantics such that the next statement may continue execution concurrently with the spawned procedure. In other words, **spawn** indicates the creation of a child thread that will execute the declared procedure until completion, at which point the child thread will terminate. The second keyword **sync** denotes that the parent procedure will wait to proceed to the next statement until all spawned child threads have finished execution. The keyword **atomic** is necessary because, when executing in a multi-threaded environment, multiple threads may access the same memory locations at once. In order for concurrent threads to safely change such a value, a thread that loads and modifies a variable should execute these operations atomically, in isolation from the rest the threads. We use the keyword **atomic** in our algorithm description to indicate that an operation or sequence of operations must execute with the appearance to other threads of occurring instantaneously. Note that this is a simplification of the underlying implementation,

which may make use of compare-and-swap operations, mutexes and condition variables to achieve thread-safe memory operations.

Algorithm Parallel Client Verification

```
300: procedure ParallelVerify(σ_{n-1}, msg_n, Φ)
301:     Active ← 0                                          ▷ Number of active threads
302:     atomic Q_R ← makeNodeQueue()                        ▷ Nodes for worker threads to execute
303:     atomic Q_A ← makeNodeQueue()                        ▷ Nodes created by worker threads
304:     atomic Valid ← ⊥                                    ▷ Will be set to a valid path if successful
305:     atomic Finished ← false                             ▷ Instructs all threads to halt
306:     Φ_n ← {}                                            ▷ Training for set msg_n
307:     spawn TrainingSelector(msg_n, Φ, Φ_n)
308:     spawn NodeScheduler(σ_{n-1}, Φ_n, Q_R, Q_A, Active, Finished)
309:     for 1 to NumWorkers do
310:         spawn VerifyWorker(msg_n, Q_R, Q_A, Active, Finished, Valid)
311:     sync                                                ▷ Wait for all child threads
312:     return Valid                                        ▷ Return result, ⊥ or σ_n
```

Figure 5.4. Main procedure for parallel client verification.

5.2.4 Details of parallel verification algorithm

The algorithm for verifying a client-to-server message using thread-level parallelism is shown in Figure 5.4. This algorithm, denoted ParallelVerify, takes as input the symbolic state σ_{n-1} resulting from execution of Π_{n-1} from the client entry point on message trace $msg_0, \ldots, msg_{n-1}$, the next message msg_n; and the complete training fragment set Φ. Its output is either an execution fragment that can be appended to Π_{n-1} to make Π_n that is consistent with msg_0, ..., msg_n, or no solution ($\perp$). The latter case indicates failure and, more specifically, that there is no execution prefix that can extend Π_{n-1} to make Π_n that is consistent with msg_0, ..., msg_{n-1}. This will induce backtracking to search for another $\hat{\Pi}_{n-1}$ that is consistent with msg_0, ..., msg_{n-1}, which the verifier will then try to extend to find a Π_n consistent with msg_0, ..., msg_n.

The algorithm operates in a parent thread that spawns NumWorkers $+ 2$ child threads; this includes one thread to select training data, one thread to manage scheduling of nodes for execution, and NumWorkers worker threads to explore candidate execution fragments. We outline in detail the operation of algorithm ParallelVerify below.

Initialization: First, variables that will be passed to the algorithm sub-procedures are initialized (Lines 301–305). This includes a counter Active that tracks the number of active workers (301), two empty Node queues Q_R and Q_A (301–303), an object Valid that will be set to a valid execution fragment if the algorithm is successful (304) and a boolean Finished that will be set to **true** when the algorithm reaches a stop condition (305).

Training Set Selection Thread: Next, on line 307, a thread is spawned for TrainingSelector. This procedure selects a set of training fragments $\Phi_n \subseteq \Phi$ from the set of all training fragments Φ based on the contents of the message msg_n. The details of this algorithm are outlined in Chapter 4. The important change here is that this algorithm is run in separate thread. We will outline the intuition behind this choice below.

Node Selection Thread: The algorithm then spawns a procedure that manages the selection of nodes to execute next (308). The parameters to NodeScheduler are training fragment set Φ_n, node queues Q_R and Q_A, counter Active and boolean Finished. Procedure NodeScheduler is spawned in a child thread so that operations on the node tree do not block state exploration.

Verification Worker Threads: Finally, the algorithm spawns a fixed number of threads that execute the VerifyWorker procedure (310). This procedure takes as parameters: network message msg_n, node queues Q_R and Q_A, counter Active, boolean Finished and execution path Valid.

Termination: Execution is blocked at the call **sync** until all child threads have exited due to a termination condition, namely Finished $=$ **true**. After all child threads have finished the algorithm returns the result Valid, which is either set by a worker thread to an execution fragment π, that when appended to Π_{n-1} is consistent with msg_0, ..., msg_n or on failure, remains undefined $\perp$ as initialized.

Algorithm Parallel Client Verification (sub-procedures)

```
400: procedure NodeScheduler(σ_{n−1}, Φ, Q_R, Q_A, Active, Finished)
401:     nd ← makeNode()
402:     nd.path ← ⟨⟩; nd.state ← σ_{n−1}
403:     Live ← {nd}
404:     while Finished = false do
405:         if |Live| > 0 and |Q_R| < (NumWorkers − Active) then
406:             nd ← SelectNode(Live, Φ_n)
407:             atomic enqueue(Q_R, nd)
408:         while |Q_A| > 0 do
409:             atomic Live ← Live ∪ {dequeue(Q_A)}
410:         if |Live| = 0 and |Q_R| = 0 and |Q_A| = 0 and Active = 0 then
411:             Finished ← true
412:
413: procedure VerifyWorker(msg_n, Q_R, Q_A, Active, Finished, Valid)
414:     while Finished = false do
415:         if |Q_R| > 0 then
416:             atomic Active ← Active + 1
417:             atomic nd ← dequeue(Q_R)
418:             if nd ≠ ⊥ then
419:                 π ← nd.path ; σ ← nd.state
420:                 while isNetInstr(σ.next) = false and isSymbolicBranch(σ.next) = false do
421:                     π ← π ‖ ⟨σ.next⟩
422:                     σ ← execStep(σ)
423:                 if isNetInstr(σ.next) = true then
424:                     if ((σ.constraints ∧ σ.next.msg = msg_n) ⇏ false) then
425:                         Finished ← true
426:                         Valid ← σ                                   ▷ Success!
427:                 else if isSymbolicBranch(σ.next) = true then
428:                     nd.child_0.state ← [ execStep(σ) | σ.next.cond ↦ false ]
429:                     nd.child_0.path ← π ‖ ⟨nd.child_0.state.next⟩
430:                     if nd.child_0.state.constraints ⇏ false then
431:                         atomic enqueue(Q_A, nd.child_0)
432:                     nd.child_1.state ← [ execStep(σ) | σ.next.cond ↦ true ]
433:                     nd.child_1.path ← π ‖ ⟨nd.child_1.state.next⟩
434:                     if nd.child_1.state.constraints ⇏ false then
435:                         atomic enqueue(Q_A, nd.child_1)
436:             atomic Active ← Active − 1
```

Figure 5.5. Sub-procedures for parallel client verification.

5.2.5 Details of parallel verification algorithm sub-procedures

We now describe in further detail the sub-procedures used by ParallelVerify and that are shown in Figure 5.5. Recall that the parent procedure ParallelVerify spawns NumWorkers $+ 2$ child threads which run the procedures TrainingSelector, NodeScheduler and VerifyWorker. We omit the details of TrainingSelector here, it is described in Chapter 4.

Management of nodes in NodeScheduler

ParallelVerify spawns a single thread to run the procedure NodeScheduler, shown starting on Line 400 of Figure 5.5. This procedure manages the selection of nodes to execute next and maintains the flow of nodes between worker threads. There are two queues of nodes, a "ready" queue Q_R and an "added" queue Q_A. These queues are shared between the worker threads and the NodeScheduler thread. Worker threads pull nodes from Q_R and push new nodes onto Q_A. There is only one scheduler thread and one or more worker threads producing and consuming nodes from the queues Q_R and Q_A; hence, Q_R is a single-producer-multi-consumer queue and Q_A is a multi-producer-single-consumer queue. The goal of NodeScheduler is to keep Q_A empty and Q_R "full," and we will define what we mean by this below. Nodes are in one of four possible states, either actively being explored inside VerifyWorker, stored in Q_R, stored in Q_A or stored in Live. A node at the front of Q_R is the highest priority node not currently being explored. The nodes in Q_A are "infant" nodes that have been created by VerifyWorker threads and need to be added to Live. The remaining nodes stored in Live are the candidate nodes.

Upon initialization, the procedure NodeScheduler creates a root node and adds it to a set of nodes called Live (401–403). After initialization, the procedure NodeScheduler has three cases of execution. First, the condition on line 405 checks if Live is non-empty and if there are more worker threads waiting to execute than nodes in Q_R, i.e., the queue Q_R is not "full"; if this condition is true, we call SelectNode (see Chapter 4 for details) and atomically append the result to Q_R. Second, if Q_A is non-empty, its members are dequeued and added to Live (408–409). Finally, the condition on line 410 is only true if no worker threads are active, both queues are empty and there are no remaining states to execute; when this condition is met, all paths of exploration rooted at σ_{n-1} have been exhaustively explored and a termination condition has been met. The boolean Finished will

be set to **true**, forcing all threads to exit. The parent thread executing ParallelVerify will return $\perp$ in this case.

Building execution fragments in VerifyWorker

Shown starting on Line 413 of Figure 5.5, the procedure VerifyWorker does the main work of client verification: stepping execution forward in the state σ of each node. Like NodeScheduler, the procedure VerifyWorker runs inside of a while loop until the value of Finished is no longer equal to **false** (414). Recall that the parent procedure ParallelVerify spawns multiple instances of VerifyWorker. Whenever there is a node on the queue Q_R, the condition on line 415 will be true and the procedure calls dequeue atomically. Note that even if $|\mathsf{Q}_R| = 0$, multiple instances of VerifyWorker may call dequeue, but only one will return a node, the rest will retrieve undefined ($\perp$) from dequeue.

If nd is not undefined, the algorithm proceeds to execute the state nd.state and extend the associated path nd.path up to either the next network instruction (SEND or RECV) or the next symbolic branch (a branch instruction that is conditioned on a symbolic variable). The first case, stepping execution on non-network / non-symbolic-branch instructions, executes in a while loop on lines 420–422. The current instruction is appended to the path and the procedure execStep is called, which symbolically executes the next instruction in state σ. These lines, are where the majority of the computation work is done by the verifier (see Figure 4.5 in Chapter 4) and viewed as the "hot" path of the verification algorithm. The ability to concurrently step execution on multiple states is where the largest performance benefits of parallelization are achieved. Note that calls to execStep may invoke branch instructions, but these are non-symbolic branches. In the second case, if the next instruction is SEND or RECV and if the constraints σ.constraints accumulated so far with the symbolic state σ do not contradict the possibility that the network I/O message σ.next.msg in the next instruction σ.next is msg_n (i.e., $(\sigma.\mathsf{constraints} \wedge \sigma.\mathsf{next.msg} = msg_n) \not\Rightarrow$ **false**, line 423), then the algorithm sets the termination value (Finished $=$ **true**) and sets the return value of the parent function (Valid $\leftarrow \pi \parallel \langle\sigma.\mathsf{next}\rangle$). All other threads of execution now exit because Finished $=$ **true** and the parent procedure ParallelVerify will return Valid, which is now an execution fragment that meets the verifier's goals successfully.

In the final case, (isSymbolicBranch(σ.next) $=$ **true**), the algorithm is at a symbolic branch. Thus, the branch condition contains symbolic variables and cannot be evaluated as true or false

in isolation. Using symbolic execution, the algorithm evaluates both the true branch and the false branch by executing σ.next conditioned on the condition evaluating to **false** (denoted [execStep(σ) | σ.next.cond $\mapsto$ **false**] in line 428) and conditioned on the branch condition evaluating to **true** (432). In each case, the constraints of the resulting state are checked for consistency (430, 434). If either state is consistent, it is atomically placed onto Q_A (431, 435).

5.2.6 Algorithm summary

Let us return to Figure 5.3(b) from earlier, which depicts a node tree rooted at σ_{n-1} during the verification of msg_n. The node colored white with a solid outline represents the *root* node with state σ_{n-1}. The nodes colored white with dashed outlines, are the *dead* nodes and represent intermediate states that no longer exist. A node is dead when it does not reach a success condition or exits the main if block of VerifyWorker (starting on line 418) without generating any child nodes. Nodes colored black are the *active* nodes and are currently being explored by worker threads. Nodes colored dark gray are in the set Live. If there are worker threads that are ready to process a node, the highest priority live nodes are in Q_R, otherwise live nodes and are either in the set Live. Nodes colored light grey are the *infant* nodes and are in Q_A. We can see that worker *W4* recently hit a symbolic branch condition and created two child nodes which were added to Q_A. The other workers are likely executing lines 420–422.

5.3 Multi-threaded KLEE

Supporting the algorithm described in the previous section requires a multi-threaded symbolic execution engine. To our knowledge, no current symbolic execution engines support multi-threaded execution in a single process. Since our previous verification tool was built upon KLEE, we modified it to support multi-threaded execution. There were many small changes made to the code to make it thread-safe, but some of the major changes include: making the internal reference count class thread safe, tracking statistics with per-thread objects, adding critical sections around shared resources, and creating separate solver stacks and memory allocators for each worker thread. As a side benefit, this multi-threaded implementation of KLEE can be used outside of client verification and should enable improved performance for other symbolic execution tasks.

5.4 Evaluation

We now evaluate the performance of the parallel verification algorithm. Recall that by design, our verification algorithm has no false positives — i.e., if a message trace is declared to be inconsistent with the sanctioned client software, then it really is. The only source of false negatives arises from the limited fidelity of the constraints used to model values returned by components with which the client software interacts. In these experiments we make the assumption that the environment modelling is sound. We limit our evaluation to *legitimate* message traces since our approach is designed to quickly validate legal behavior in clients that have an unbounded number of execution paths for exploration. However, there is no reason that a version of the parallel algorithm presented in this chapter could not be applied to the exhaustive client verification problem from Chapter 3.

The experiments in this chapter use a multi-threaded version of our verification tool, built upon KLEE version 1.0[17], LLVM 3.4[60] and STP (rev-940)[34]. The experiments were run on system with 256GB of RAM and 32 3.2GHz processor cores. To evaluate performance, we applied our parallel algorithm to verify behavior of legitimate clients of two open-source games from our previous case studies, namely *XPilot* and *TetriNET*, which are described in previous chapters. Evaluation of our parallel verification algorithm requires message traces for both training and testing. For *TetriNET*, we generated 20 traces of manual gameplay, each of 240 messages in length (which corresponds to roughly 6.5 minutes of gameplay). For *XPilot*, we generated 40 traces of manual gameplay, each consisting of 2100 messages (roughly 70 seconds of gameplay). For our experiments we use the same message traces and training data from the evaluation section from Chapter 4. Note however that these experiments were performed on a system with a different CPU that has a 10% higher clock rate.

For each client type, we ran three cross validation experiments over the message traces with either 1, 8 or 16 worker threads. The training fragments were clustered with cluster parameter values $k = 3790$ and $k = 475$ for *TetriNET* and *XPilot*, respectively. These parameters ensured a single training fragment per cluster and corresponds to the "default" configuration used in Chapter 4. These experiments do not include results using the "hint" configuration, because as we will see, when using the parallel verification technique, client-side modification to provide hints to the verifier is not necessary. Although the experiments in our evaluation use the same data, the single

worker experiments in this chapter gain the benefits of separate threads for the NodeScheduler and TrainingSelector procedures and the increased clock rate of our experiment platform.

In our evaluation we observe two data points for each msg_n in a message trace, verification *cost* and verification *delay*. The verification cost for message msg_n, denoted $cost(n)$, accounts for all time spent in ParallelVerify(σ_{n-1}, msg_n, Φ). This is measured as the wall-clock time that the algorithm takes to determine that msg_n is valid.[2] Note that the verification time represents work done concurrently and doesn't represent the total summation of CPU time across multiple cores. Our second data point for evaluation is the verification *delay*, which is the delay in time between the arrival of message msg_n at the server (where a server-to-client message "arrives" when it is sent) and the discovery of an execution prefix Π_n that is consistent with msg_0, ..., msg_n. Delay differs from verification time by representing the fact that verification for msg_n cannot begin until after verification for msg_{n-1} completes. The methods for calculating verification cost and delay are formally defined in Section 4.7 of Chapter 4.

5.4.1 Case Study: *TetriNET*

	Cost (s)			Delay (s)		
NumWorkers	16	8	1	16	8	1
Min	0.0007	0.0003	0.0001	0.0008	0.0003	0.0001
Max	12.1548	16.6219	100.8736	13.1708	22.7357	337.01
Median	0.0368	0.0370	0.0321	0.0489	0.0484	17.946
Mean	0.1287	0.1622	1.5859	0.3975	0.6361	57.137
Std. Dev.	0.7613	1.0367	8.8912	1.4841	2.3314	77.718

Figure 5.6. Summary statistics for *TetriNET* results

Figure 5.6 shows summary statistics for the parallel verification algorithm on the *TetriNET* experiments and shows both verification costs and delays for 16-worker, 8-worker and 1-worker configurations. These results were obtained by a 20-fold cross validation of the *TetriNET* traces; i.e., in each test, one of the traces was selected for testing, and the remainder were used for training. Each row in Figure 5.6 is a measure of values (costs or delays) observed over all 20 experiments. The mean verification cost per message, regardless of the number of threads used, is easily beneath the inter-message delay of roughly 1.6s. As an optimization over results in Chapter 4, we can see

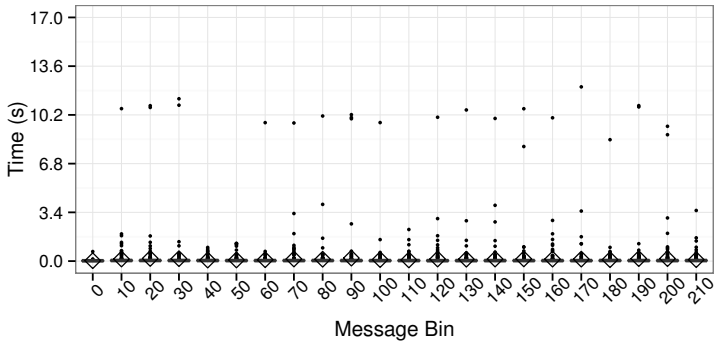
[2]If backtracking occurs, that time is included in the verification time of the associated message index, i.e., the verification time for msg_n may include the total wall clock time of more than one instance of ParallelVerify.

that parallelization of the verification algorithm allows us to drop the mean verification cost from 1.59s using a single worker thread to 130ms when using the 16-thread configuration. If the verifier was placed in-line, this might be an acceptable overhead on average to add to the processing of network data. However the maximum verification time of 12.15s is greater than the average inter-message delay of 1.6s and would not be an acceptable amount of latency to add to the client-server communication. Looking at the Delay columns in Figure 5.6 we can see that the max delay is 25X less in the 16-worker configuration than in the 1-worker configuration.

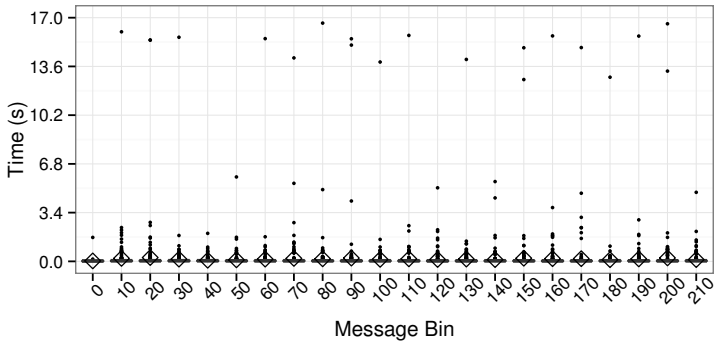
Figure 5.7 shows the distribution of verification cost per message, binned into ten-message bins, across all 20 traces. The boxplot labeled "0" shows the distribution of verification times for messages $msg_0, \ldots, msg_9$ in the 20 traces. In each boxplot, the "box" shows the first, second (median) and third quartiles, and with whiskers extending to ± 1.5 times the interquartile range. Additional outlier points are shown as dots. Overlaid on each boxplot is a diamond ($\Diamond$) that shows the average of the data points. In Figure 5.7 we can see how additional worker threads reduce the verification costs. We can also see that it is the outlier points that dominate the overall time spent in the verifier, with bands at 90s, 14s and 10s for the three configurations. Comparing the 1-worker and 16-worker configurations, the performance multiplier is roughly 9, rather than 16. Our implementation does not achieve a "perfect" speed-up when adding additional workers and there are several factors that may be at play. First, a single outlier can represent verification in multiple node trees due to backtracking. Although 2.5% of *TetriNET* messages required backtracking, there is not a direct correlation between existence of backtracking and higher verification cost. Nevertheless, a round with backtracking cannot exploit multiple workers as efficiently as possible because early in the exploration of a node tree because there are more fewer live nodes than workers available. Second, there is some overhead in our implementation and as we add more threads there is a added cost to access shared resources. We attempted to minimize this as much as possible in our design.

Figure 5.8 plots the distributions of per-message verification *delay* between the arrival of message msg_n at the server and the discovery of an execution prefix Π_n that is consistent with $msg_0, \ldots, msg_n$. Delay (Figure 5.8) differs from verification cost (Figure 5.7) by representing the fact that verification for msg_n cannot begin until after that for msg_{n-1} completes. So, for example, the rightmost boxplot in each graph provides insight into how long after the completion of the message trace (in real time) that it took for verification for the whole trace to complete. We note that for the multi worker

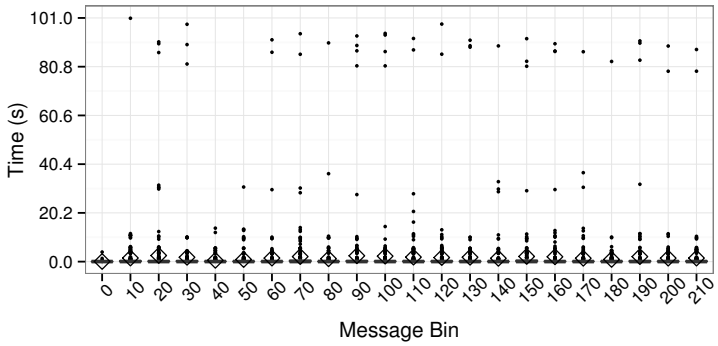
configurations of 8 and 16 workers, verification is able to keep pace with gameplay and never accumulates delay over the course of verification. The median of the rightmost boxplot is virtually zero. Even if verification falls behind at some point in the game, it always catches up because of the gap between message arrival times. This indicates that the verifier needs only a fixed sized buffer of network messages to manage a long-running verification session. We see that in Figure 5.8c, which shows the single worker configuration, verification delay lags behind message arrival times by more than 60s by the end of a 240-message trace in the average case.



(a) NumWorkers $= 16$

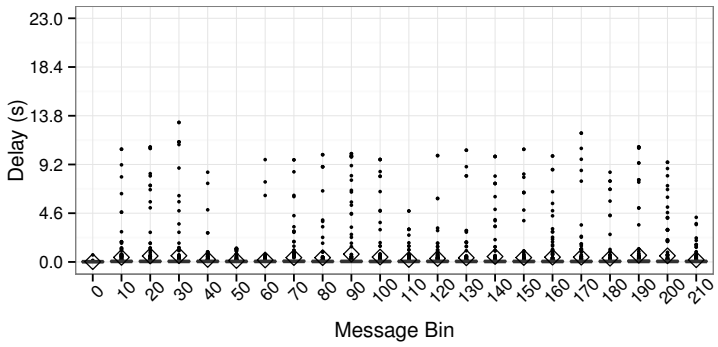


(b) NumWorkers $= 8$

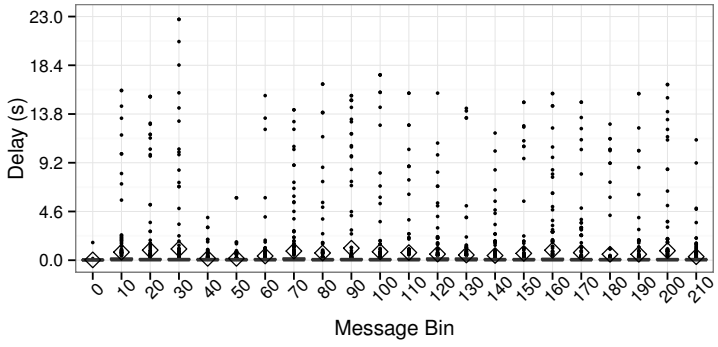


(c) NumWorkers $= 1$

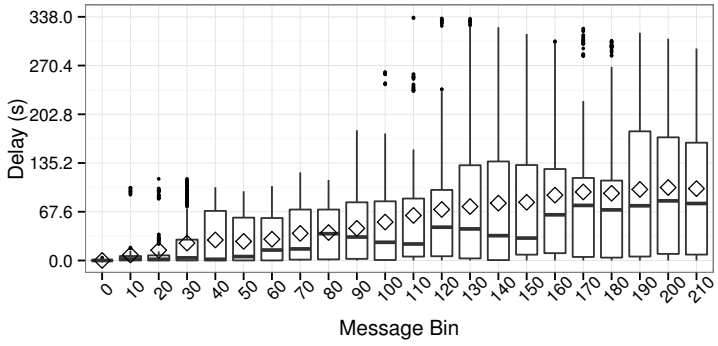
Figure 5.7. *TetriNET* parallel verification costs. Cross-validation over 20 traces. Boxplot at x shows verification costs for messages $msg_x, \ldots, msg_{x+9}$ in each trace (after training on the other traces). "$\Diamond$" shows the average.



(a) NumWorkers $= 16$



(b) NumWorkers $= 8$



(c) NumWorkers $= 1$

Figure 5.8. *TetriNET* parallel verification delays. Cross-validation over 20 traces. Boxplot at x shows verification delays for messages $msg_x, \ldots, msg_{x+9}$ in each trace (after training on the other traces). "$\Diamond$" shows the average.

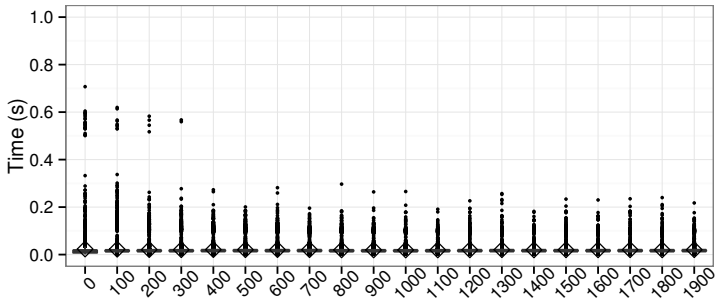
5.4.2 Case Study: *XPilot*

	Cost (s)			Delay (s)		
NumWorkers	16	8	1	16	8	1
Min	0.0004	0.0004	0.0001	0.0025	0.0022	0.0023
Max	0.7071	0.4966	5.4948	1.1990	1.5294	122.85
Median	0.0164	0.0168	0.0122	0.0205	0.0250	29.869
Mean	0.0205	0.0213	0.0658	0.0754	0.0791	32.740
Std. Dev.	0.0237	0.0198	0.1364	0.1313	0.1248	24.187

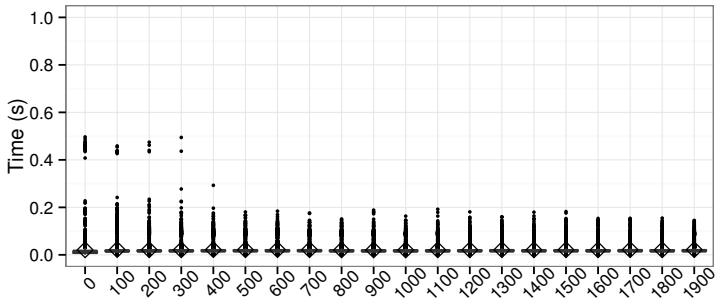
Figure 5.9. Summary statistics for *XPilot* results

Figure 5.9 shows the summary statistics of the experiments on *XPilot*, for verification cost and delay – using 16-worker, 8-worker and 1-worker configurations. *XPilot* poses a different challenge than *TetriNET* for verification because the message rate is so fast. The tests described here use an *XPilot* configuration that resulted in an average of 32 messages per second or an average inter-message time of 31.5ms. Nevertheless, with our parallel verification technique, we can achieve a mean verification cost of only 20ms and return a verification result in less than 0.71s for all messages in our experiments. The multi-worker configurations are both more than $3\times$ faster on average than the single worker configuration. For the 16-worker configuration, the verifier is on average only 75ms behind (1.19s in the worst case), whereas when using only a single-worker, verification delay is on average 32.7s and over 2 minutes in the worst case. These results demonstrate that in-line verification is feasible and would add minimal latency in the average case. However, adding 1.19s of latency in the worst case would not be acceptable for fast-paced gameplay.

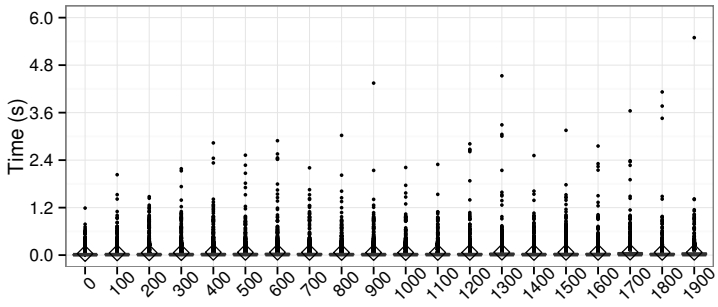
In Figure 5.10, the verification costs for *XPilot* are shown for the three worker configurations across the message traces. Each boxplot in Figure 5.10 represents 100×40 points. Despite a mean verification cost of 75ms when using a single worker thread, the fast pace of *XPilot* makes it difficult for verification to keep pace with the game. This effect is shown in Figure 5.11(c). However, by increasing the number of worker threads, we can see that in the 16-worker and 8-worker configurations, verification delay never significantly falls behind and could use only a fixed buffer of messages for verification in long running sessions.



(a) NumWorkers $= 16$

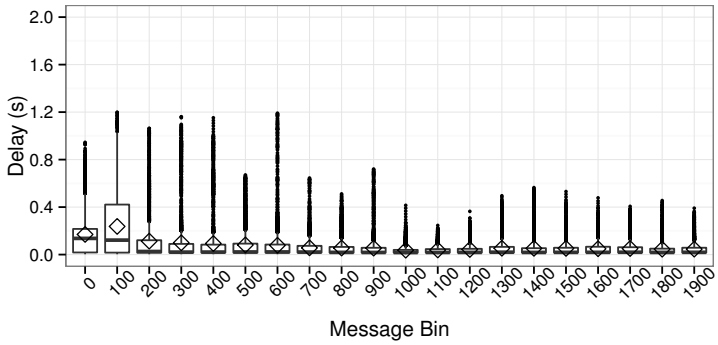


(b) NumWorkers $= 8$

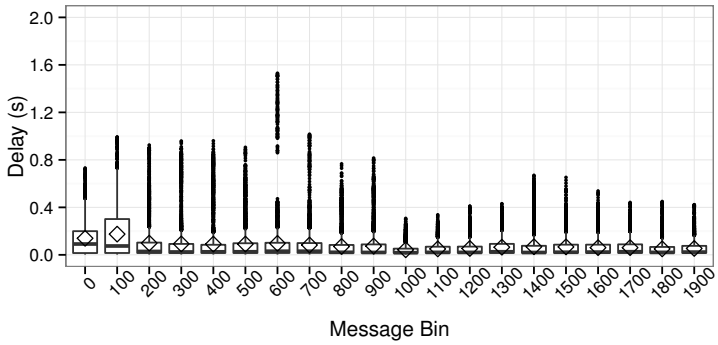


(c) NumWorkers $= 1$

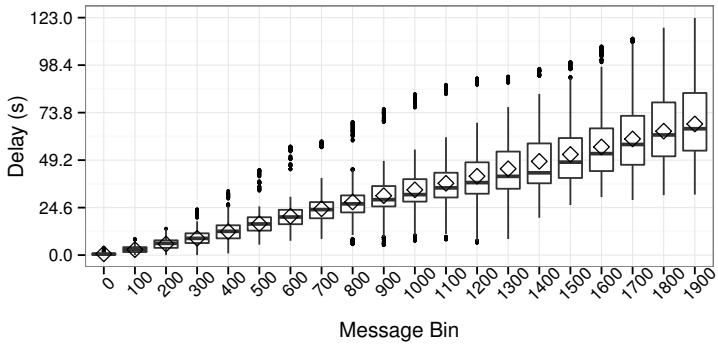
Figure 5.10. *XPilot* parallel verification costs. Cross-validation over 40 traces. Boxplot at x shows verification costs for messages $msg_x, \ldots, msg_{x+99}$ in each trace (after training on the other traces). "$\Diamond$" shows the average.



(a) NumWorkers $= 16$



(b) NumWorkers $= 8$



(c) NumWorkers $= 1$

Figure 5.11. *XPilot* parallel verification delays. Cross-validation over 40 traces. Boxplot at x shows verification delays for messages $msg_x, \ldots, msg_{x+99}$ in each trace (after training on the other traces). "$\Diamond$" shows the average.

Cost(s)		
	with TrainingSelector thread	without TrainingSelector thread
Min	0.0003	0.0003
Max	16.0291	17.1220
Median	0.0349	0.0221
Mean	0.1804	0.1833
Std.Dev.	1.0507	1.2745

(a) *TetriNET*

Cost(s)		
	with TrainingSelector thread	without TrainingSelector thread
Min	0.0007	0.0006
Max	0.4312	0.4469
Median	0.0158	0.0157
Mean	0.0170	0.0172
Std.Dev.	0.0184	0.0182

(b) *XPilot*

Figure 5.12. Summary of verification cost in seconds for *TetriNET* and *XPilot*, with and without a TrainingSelector thread, using NumWorkers $= 8$.

5.4.3 Evaluation of NodeScheduler and TrainingSelector Threads

The parallel algorithm presented in this chapter divides and distributes the verification computation into separate threads across multiple CPU cores. Recall in addition to a variable number of VerifyWorker threads, the algorithm uses two additional threads; one for the NodeScheduler procedure (Figure 5.5) and one for the TrainingSelector procedure (detailed in Chapter 4). In this section, we will examine the performance impact of computing the work done by the procedures NodeScheduler and TrainingSelector in separate threads as opposed to an alternative arrangement where the work is divided amongst all VerifyWorker threads.

In our current experiments and implementation, the usage of a NodeScheduler thread does not provide a measurable performance benefit. The increased granularity of the workload is offset by the small overhead of the implementation. Nevertheless, this design provides a cleaner separation of work for the implementation and we predict verification of different client software in the future may benefit from the architecture.

Using a separate thread for the TrainingSelector procedure does however have a measurable performance impact. The procedure TrainingSelector is provided a network message msg_n and a set of execution fragments Φ, produced during training and returns a subset of execution fragments Φ_n, used by the NodeScheduler to inform node selection. The details of the TrainingSelector algorithm are described in Chapter 4. In our experimental results thus far, each thread has exclusive access to a CPU core. However, the algorithm is designed so that the TrainingSelector method can be called synchronously by removing **spawn** on Line 307 of the ParallelVerify procedure in Figure 5.4. We evaluated this design decision on the case studies *XPilot* and *TetriNET* with a new experimental setup where the TrainingSelector procedure is called synchronously before the VerifyWorker threads are spawned. Figure 5.12 shows a summary of the verification costs in seconds for the case studies using NumWorkers $= 8$. The mean verification cost in seconds increases measurably for both *XPilot* and *TetriNET* when the verifier is executed without a TrainingSelector thread.

5.5 Evaluation of Optimization Techniques

The performance of parallel symbolic verification is dependent on many factors. We have shown that a parallel implementation with multiple worker threads decreases the cost of verification over the single-threaded implementation outlined in the previous chapters. Using symbolic execution as the basis for a verification method has required several optimizations to the underlying implementation, which have been described in previous chapters. In this section we review these optimization techniques and evaluate their impact and interaction with parallel symbolic execution in our case studies.

Several optimization methods were developed and utilized to improve the performance of symbolic execution for verification. KLEE was designed with techniques for reducing the size and number of queries that are sent to the underlying constraint solver. One of these techniques is a cache of solver queries and their respective solver outputs. Before a constraint formula or query is tested for satisfiability via STP, KLEE checks a *query cache* and if the cache contains a hit, the potentially expensive solver operation can be avoided. If there is a miss, the solver must be instantiated and the result is added to the cache afterwards. In our implementation, each worker thread operates a separate solver chain, which means that KLEE query caches are not shared between

workers. Additionally, the caches are empty upon starting verification of a message sequence. In the previous chapters, two methods have been demonstrated that enable better utilization of these caches: canonicalization (Section 4.7.1) and constraint pruning (Section 3.2.3). The canonicalization method is used to rewrite the variable names in a solver query before the query cache is checked. The constraint pruning optimization is used to eliminate variables and formula constructs that have not been concretized but can be shown to never be relevant to future branch conditions; variables can be pruned that are no longer in the scope of execution and are independent from the current scope.

In this section we evaluate the impact and interactions of parallel symbolic verification with and without the canonicalization and constraint pruning optimizations. The experiment operates with the verifier in three configurations. The first configuration, which we call AllOpt, is the same configuration used in Section 5.4, and utilizes both canonicalization and constraint pruning. The second configuration, NoCanon, is the same as the former but disables the canonicalization of queries before the query cache is checked. The third and final configuration, NoPrune, is the same as AllOpt but with constraint pruning disabled. Additionally, we are only verifying a *single* game play log from each of our case studies, *XPilot* and *TetriNET*. Despite using a single log, the experimental results are representative of the full case study data sets. The use of a single log allows easier characterization of the interactions of the optimizations with symbolic client verification. All of the experiments were performed with NumWorkers $= 16$. If the experiments are configured with NumWorkers $= 1$, disabling canonicalization or constraint pruning causes the verification to take several hours. In addition to demonstrating the benefit of constraint pruning and canonicalization, these results are also useful to characterize the workloads of our case studies in terms of the symbolic execution optimizations of KLEE and may be of use to others.

5.5.1 Impact of Optimizations on Cost and Delay

We now examine the performance interactions of canonicalization and constraint pruning with parallel symbolic verification. We start with an experiment to determine the impact of these optimizations on our key evaluation metrics, cost and delay. Figure 5.13 shows an overview of the cost and delay times per message during the verification of a single game play trace from the *TetriNET* and *XPilot* case studies. In Figure 5.13(a) we can see that for *TetriNET*, either disabling

	Cost (s)			Delay (s)		
	AllOpt	NoCanon	NoPrune	AllOpt	NoCanon	NoPrune
Min	0.0016	0.0014	0.0013	0.0016	0.0014	0.0013
Max	1.4875	3.3591	4.7334	1.4925	3.3690	4.7390
Median	0.0384	0.0376	0.0760	0.0504	0.0467	0.1281
Mean	0.0835	0.0986	0.2214	0.1387	0.1919	0.4207
Std. Dev.	0.1970	0.3225	0.5242	0.2655	0.5062	0.7522

(a) *TetriNET*

	Cost (s)			Delay (s)		
	AllOpt	NoCanon	NoPrune	AllOpt	NoCanon	NoPrune
Min	0.0009	0.0007	0.0009	0.0027	0.0086	0.0090
Max	0.6268	0.7627	8.2392	1.1417	55.0549	1158.8995
Median	0.0171	0.0327	0.3217	0.0266	35.5776	421.1561
Mean	0.0198	0.0559	0.6080	0.1084	29.5545	537.8276
Std. Dev.	0.0256	0.1213	0.7888	0.2019	16.8834	410.7422

(b) *XPilot*

Figure 5.13. Verification costs and delays for *TetriNET* and *XPilot* for a single representative log.

the canonicalization optimization (NoCanon) or disabling the constraint pruning optimization (NoPrune) adversely affects the mean values for cost and delay. The benefit of these optimizations is even more prevalent in the *XPilot* results shown in Figure 5.13(b). Disabling canonicalization increases the mean verification cost by a factor of five and due to the rapid rate at which *XPilot* messages are transmitted, disabling canonicalization introduces more than a $200\times$ increase in the mean verification delay. Also, for *XPilot*, constraint pruning is even more important for efficient verification; the mean verification delay increases from roughly a 0.10 seconds with all optimizations to over 8 minutes without constraint pruning.

5.5.2 Impact of Optimizations on Solver Queries

We can better understand how the canonicalization and constraint pruning optimizations impact cost and delay in parallel symbolic verification by looking at how these optimizations change the properties of the formulas sent as queries to the constraint solver (STP).

Query Cache Hit Rate

Figure 5.14 shows the overall hit rates of the query cache in the single log verification experiment and can begin to explain why the optimizations make verification more efficient. With all optimizations (AllOpt), both *TetriNET* and *XPilot* have high hit rates of 99.89% and 97.13% respectively.

	TetriNET	*XPilot*
AllOpt	0.9989	0.9713
NoCanon	0.6860	0.3326
NoPrune	0.9989	0.9618

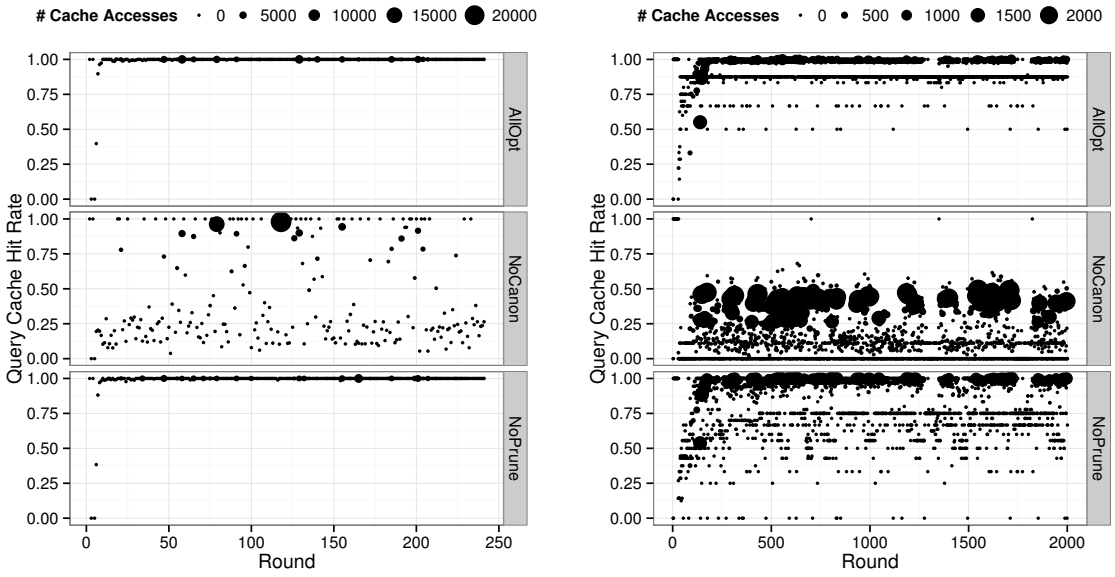
Figure 5.14. Overall query cache hit rates for a single log selected from each of the *TetriNET* and *XPilot* case studies.

In other words, of all queries to the constraint solver stack, less than 0.12% and 3.87% (*TetriNET* and *XPilot*, respectively) actually result in a potentially expensive call to to the constraint solver STP. Without the canonicalization optimization (NoCanon), the effectiveness of the cache is reduced significantly for *TetriNET* and even more so for *XPilot*, dropping the hit rate to 33.26%.

Canonicalization has a large impact on the query cache hit rate because it reduces the space of variable names that can be found in the constraint cache. During verification, when a symbolic variable is generated (e.g., to represent an unknown user input value), in addition to representing a region of memory, the variable is given a unique name. The symbolic variable name may be used in a formula to represent any constraints on any associated symbolic memory region. Verification in our case studies often generates constraints that have the same formula structure, but with different variable names. By canonicalizing the variable names before the query cache is utilized, the chance of a cache hit becomes much more likely and the expensive constraint solver query can be avoided. The need for canonicalization highlights a key difference between the use of symbolic execution for client verification versus more traditional uses of symbolic execution; client verification exercises the same paths repeatedly but with slightly different contexts. Unlike canonicalization, constraint pruning does not significantly improve the cache hit rate for the case studies; the NoPrune experiment shows the same hit rate for *TetriNET* and only slightly lower hit rate for *XPilot*.

Query Cache Hit Rate Per Message

Figure 5.15 shows the same cache hit rate values shown in Figure 5.14 but *per message*. Figure 5.15 illustrates the change in the query cache hit rate as verification progresses through the message log as well as the volume of cache accesses needed to verify each message. The query cache hit rate is shown on the y-axis and the message index on the x-axis. Each plotted dot represents the effective hit rate of the query cache over all queries during the creation of Π_n from Π_{n-1}. The number of cache accesses needed to construct each Π_n is represented exactly by the area of each circle. The



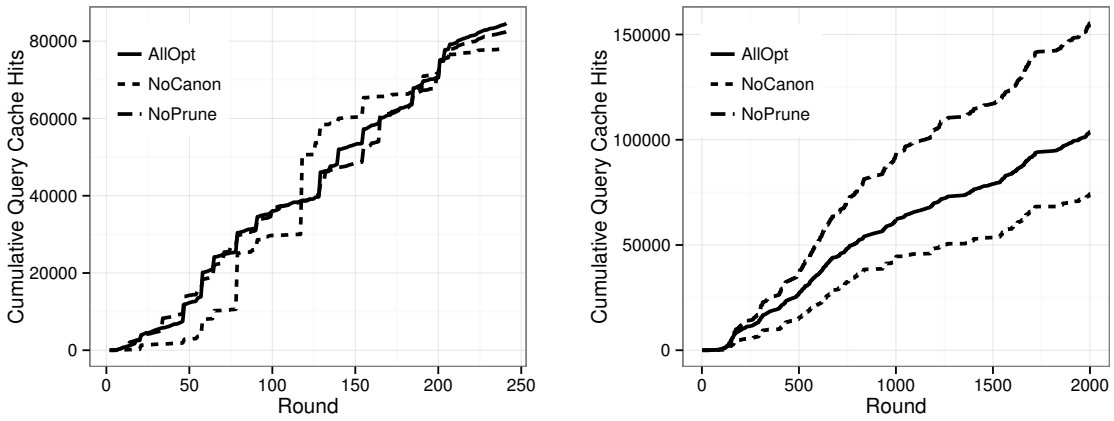
(a) *TetriNET* (b) *XPilot*

Figure 5.15. Query cache hit rates per message for the verification of a single representative game play log from each of the case studies. The area of each circle is scaled relative to the number of cache accesses represented.

legend shows several example circle areas and the corresponding number of cache accesses each area represents. The three configuration types are indicated on the right side of each plot.

Figure 5.15(a) shows the *TetriNET* experiment. The AllOpt and NoPrune results are very similar; after verification of a few messages, the query cache contains enough entries to produce a very high cache hit rate. However for the NoCanon configuration, we can observe that without canonicalization the query cache does not contain entries that match the formulas with new variables names and is not effective, even towards the end of the message log.

The *XPilot* results are shown in Figure 5.15(b). The AllOpt configuration for *XPilot* shows that verification proceeds for around 30 messages before the cache hit rate improves significantly. The NoPrune configuration is similar, but with intermittent poor performance due to the disabling of constraint pruning. Additionally, noticeable bands appear in the results because of the variety of message types. The NoCanon plot for *XPilot* shows the significance of the canonicalization optimization, both the query cache hit rate and the volume of cache accesses are affected with negative impacts on performance.

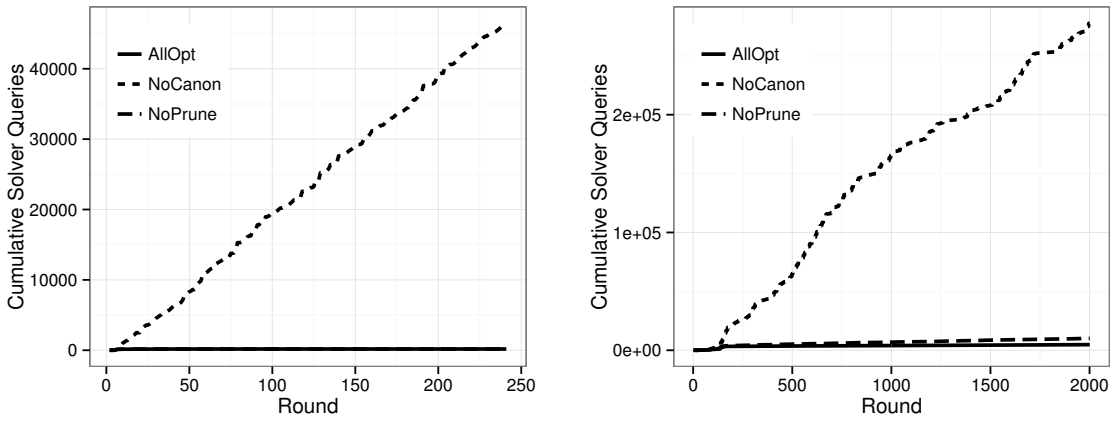


(a) *TetriNET* (b) *XPilot*

Figure 5.16. The cumulative number query cache hits over the verification of a single game play log from *XPilot* and *TetriNET*.

Cumulative Query Cache Hits and Solver Queries

We now examine how the query cache is exercised by the three configurations in terms of the cumulative number of hits in the query cache and the cumulative number of misses that lead to actual calls to the constraint solver. Figure 5.16 shows the cumulative number of query cache hits for *TetriNET* and *XPilot* over the verification of a single representative game play log. By the time the last message is verified, the *TetriNET* experiment results in over $80,000$ cache hits amongst the 16 worker threads, while the *XPilot* experiment generates over $100,000$ cache hits. The cumulative number of final cache hits is about the same for each configuration of *TetriNET*. However for *XPilot* there is a significant difference; the NoPrune configuration has approximately twice as many query cache hits as NoCanon. The NoPrune configuration produced more cache hits overall, even more than AllOpt, because the overall search was less efficient and interacted with the parallel verification. Under NoPrune, each verify worker has a more complex workload with larger constraints and therefore the time to discover the correct execution prefix increases. The additional time needed allows other worker threads to search additional paths, thus increasing the number of query cache hits. This aligns with the results for the NoPrune configuration in Figure 5.15(b); the relative query cache hit rate is slightly less than AllOpt, but the overall number of cache accesses (indicated by the size of the circles) is larger.



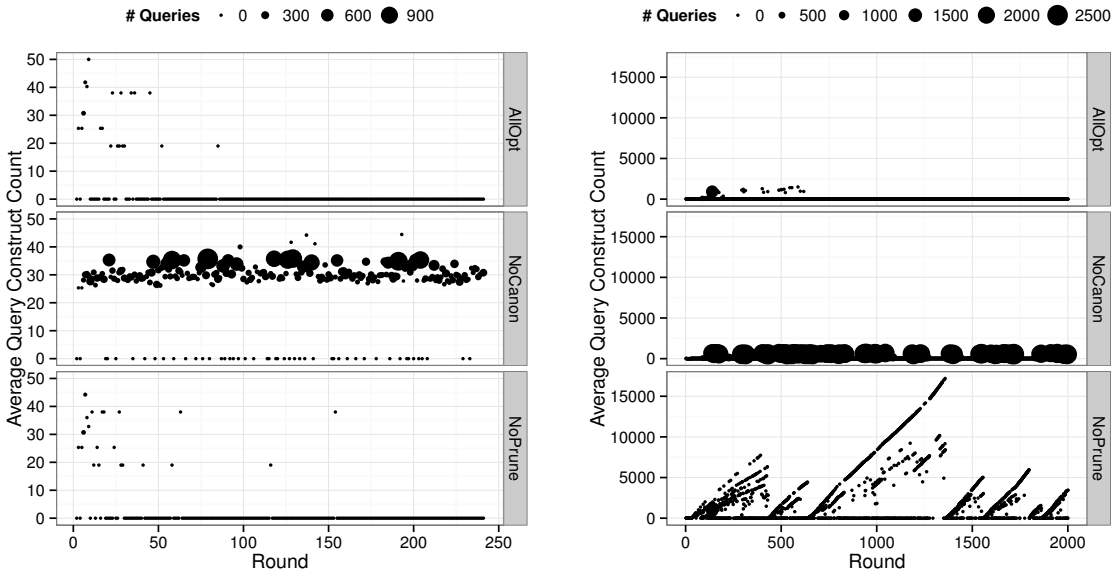
(a) *TetriNET* (b) *XPilot*

Figure 5.17. The cumulative number of queries to the solver (STP) over the verification of a single game play log from *XPilot* and *TetriNET*.

A query is only made to the constraint solver *after* a query cache miss has occurred. Therefore, the number of misses in the caches is exactly equal to the number of actual queries made to the constraint solver. Figure 5.17 shows the cumulative number of queries to the constraint solver (STP) over the verification of single game log for both *TetriNET* and *XPilot* using the three different configurations. Here it is very evident that for both case studies, the canonicalization optimization significantly reduces the overall number of expensive queries to STP. If we combine the overall numbers for *TetriNET* from Figure 5.17(a) and Figure 5.16(a), we can see that NoCanon had a significantly greater number of solver queries, including queries resulting in a cache hit. Again, this is due to an interaction with workers operating in parallel; the time to discover a valid execution prefix increases when more time is spent in the constraint solver. Therefore, other worker threads explore additional paths, leading to a greater number of client instructions executed across all worker threads. The impact of this interaction is examined further in the next section (5.5.3).

Complexity of Solver Queries

In addition to the number of queries to the constraint solver, we also characterize the complexity of these queries in terms of the size of the queried constraint. A constraint formula consists of binary and unary operators on variables or concrete values; each of those operators is counted as a single construct. The metric we use for size is a measure that corresponds to the number constructs in the query. Each circle plotted in Figure 5.18 summarizes the complexity of the constraint formulas sent



(a) *TetriNET* (b) *XPilot*

Figure 5.18. Average query construct count per message verified in a single representative game play log from *TetriNET* and *XPilot*. The area of each circle is scaled relative to the number of solver queries represented.

to the solver by showing the average number of constructs in the solver queries during the creation of Π_n from Π_{n-1}. Additionally, the area of each circle represents the number of queries sent to the constraint solver. Note that the data shown in Figure 5.18 characterizes the query formulas sent to the constraint solver (STP) after a query cache miss has occurred.

As we have seen, the lack of canonicalization leads to poor cache utilization; the effects of which are twofold for the NoCanon configuration in Figure 5.18(a). First, the average query construct count (on the y-axis) is greater than AllOpt across the message sequence. Without canonicalization, fewer queries are located in the query cache and thus more queries are sent to the constraint solver and these queries have a greater query construct count. Second, the total number of queries is greater (area of circles) because poor cache utilization leads to a greater number of client instructions executed; this is examined further in the next section (5.5.3).

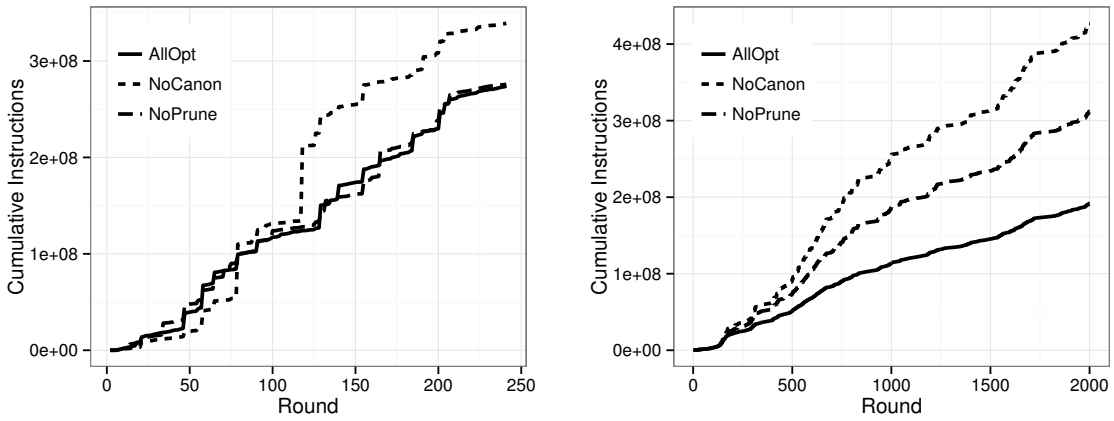
The *XPilot* experiment in Figure 5.18(b) reveals similar results for the NoCanon configuration. The average query construct count (on the y-axis) is greater than AllOpt across the message sequence because larger queries are not located in the cache and the total number of queries is larger (area of circles) because a greater number of client instructions are executed. In the NoPrune configuration,

there is a behavior where the average number of constructs increases with each creation of Π_n from Π_{n-1}, with a periodic "reset" of the average constraint complexity. This behavior is seen when constraint pruning is disabled because independent constraints that contain symbolic variables that are generated for Π_n, may have a relationship with a symbolic variable generated during the creation of Π_{n-1}. For example, suppose a symbolic variable represents the current time and is fully symbolic. This variable will be strictly greater than a similar variable that represents an earlier instance of the time. These constraints do not grow throughout the entire gameplay session because of high level game events that cause, for example, a reset of certain game state values. By using the constraint pruning techniques that are described in the previous chapter, the complexity of solver queries can be dramatically reduced and can be seen clearly in Figure 5.18(b) by comparing AllOpt and NoPrune.

5.5.3 Impact of Optimizations on Instructions Executed and Memory Usage

We now examine the impact of canonicalization and constraint pruning on two additional performance metrics in the verifier. The number of *client* instructions executed during symbolic execution by the verifier and the memory utilization of the verifier.

The number of client instructions executed characterizes the work done by the symbolic execution component of the verifier. The lower bound on the number of client instructions executed up to verification of message n is the length of the execution prefix Π_n. In practice though, the number of client instructions executed by the verifier is much greater than the lower bound. In fact, achieving this lower bound is likely not possible without knowledge of the remote client state and remote user inputs. A key objective in the design of our verifier was to reduce the total number of client instructions executed during the creation of Π_n from Π_{n-1}, thus lowering the verification cost. Chapter 4 outlines techniques to reduce the number of client instructions executed using edit distance based node selection and training data. Executing client instructions is not, however, the only work of the verifier. Figure 4.5 in Chapter 4 gives a high-level view of the time spent in each component of the verifier and shows that a significant percentage of the verifier's computation is spent doing work other than executing instructions, e.g., solving constraints. Canonicalization and constraint pruning do not *directly* affect the number of client instructions executed, but may decrease the time of constraint solving. In fact, when the verifier is configured with NumWorkers $= 1$, under

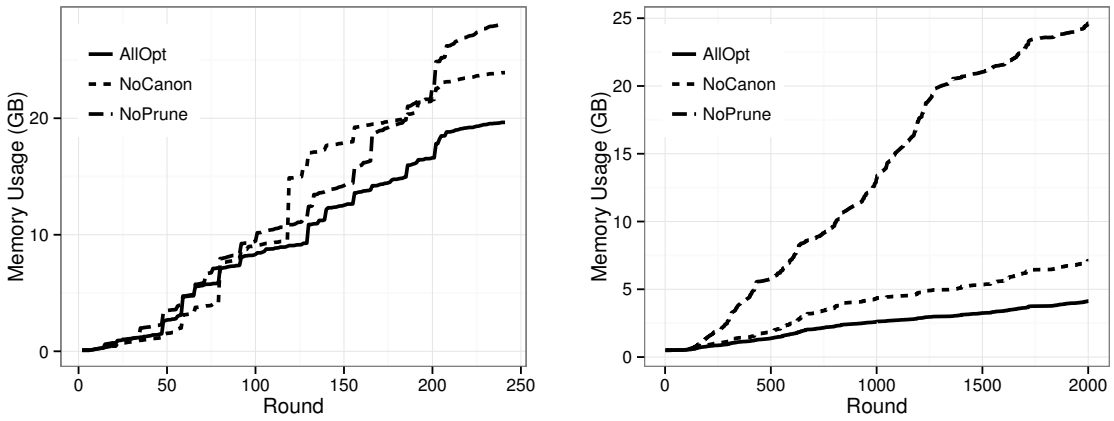


(a) *TetriNET* (b) *XPilot*

Figure 5.19. Cumulative number of client instructions executed per message during during verification of a single representative game play log from the *TetriNET* and *XPilot* case studies.

each of the configurations AllOpt, NoCanon and NoPrune, the verifier will execute the exact same number of client instructions. However, when NumWorkers > 1, any change to the performance of the verifier can affect the total number of client instructions executed amongst the verify worker threads. The interaction between the performance of constraint solving and the total number of client instructions executed is complex. For example, by decreasing the time spent constraint solving, nodes may be "revealed" more quickly in the node tree, leading to quicker utilization of all VerifyWorker threads, and therefore a greater number of instructions executed. Conversely, more expensive constraint solving may also lead to an increased number of client instructions executed. This is because if the time taken to create Π_n by one VerifyWorker thread is increased, other VerifyWorker threads may execute additional client instructions. Another important point regarding the relationship between verification cost and client instructions executed is that when the verifier is configured with NumWorkers > 1, verification costs should be lower than a verifier configured with NumWorkers $= 1$, as seen in our earlier evaluation, despite potentially executing a greater number of client instructions.

Figure 5.19 shows the cumulative number of client instructions executed over the verification of a single representative game play log from *TetriNET* and *XPilot* under the three configurations. In the AllOpt configuration, for *TetriNET* and *XPilot*, the total number of client instructions executed is 2.7×10^8 and 1.9×10^8 respectively. As before, the verifier was configured with NumWorkers $= 16$ for these results.



(a) *TetriNET* (b) *XPilot*

Figure 5.20. Cumulative overall memory utilization in gigabytes during verification of a single representative game play log for the *TetriNET* and *XPilot* case studies.

The *TetriNET* client contains more possible execution paths than the *XPilot* client; therefore worker utilization is high when a message is verified. With high worker utilization, disabling canonicalization or constraint pruning increases the overall cost of executing a client instruction on average and sometimes decreases the total number of client instructions executed. This explains the regions in Figure 5.19(a), where the NoCanon or NoPrune lines are below the AllOpt line. During verification of the *XPilot* client, utilization of the VerifyWorker threads is not always 100%, therefore in the NoCanon and NoPrune configurations, the increased verification cost of a single message enables better utilization of all VerifyWorker threads. Better thread utilization allows exploration of additional paths and execution of additional client instructions. Figure 5.19(b) shows the increase in cumulative client instructions executed for NoCanon and NoPrune. If more client instructions are executed, there is a corresponding increase in the number of solver queries, which helps explain why disabling canonicalization and constraint pruning led to an increase in the number of cache hits and solver queries seen in Figure 5.16 and Figure 5.17.

5.6 Summary

In this chapter we described a parallel client verification algorithm that vastly reduces verification costs for our case study applications, *XPilot* and *TetriNET*. Improving the verification cost using a parallel algorithm pays large dividends when accumulated over time; in some cases the

delays drop from 2 minutes to less than 2 seconds. To achieve these results, we designed and implemented a multi-threaded client verification algorithm that uses multiple threads to enable concurrent exploration of execution paths. These results open the door to using symbolic client verification on the critical path of serving client requests and could provide a new method for preventing malicious clients from disrupting gameplay and damaging server infrastructure.

CHAPTER 6: CONCLUSION

We have presented a technique to detect any type of malicious behavior that causes a remote client to exhibit behavior, as seen by the server, that is inconsistent with the sanctioned client software and the client state known at the server. Our technique 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 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. We have also presented a verification technique that validates legitimate client behavior (as being consistent with the sanctioned client software) sufficiently fast to keep pace with the application itself as demonstrated in two case studies in the context of online games. The parallel implementation of symbolic client verification could be used to prevent malicious messages from ever reaching a vulnerable server if used to verify client messages before they are processed. Our technique for verification operates without encumbering the application with substantially more bandwidth use and without sacrificing accuracy.

BIBLIOGRAPHY

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

[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381. Mar. 2008.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An experiment in public-resource computing. *Commun. ACM*, 45(11), 2002.

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

[5] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the Conference on Advances in Cryptology*. 2013.

[6] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer, 2013.

[7] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: CoqâĂŹArt: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

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

[9] D. Bethea, R. A. Cochran, and M. K. Reiter. Server-side verification of client behavior in online games. *ACM Transactions on Information and System Security*, 14(4), Dec. 2011.

[10] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM conference on Computer and Communications Security*, 2010.

[11] M. Blanton, Y. Zhang, and K. B. Frikken. Secure and verifiable outsourcing of large-scale biometric computations. *ACM Transactions on Information and System Security*, 16(3), 2013.

[12] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. In *International Conference on Reliable Software*, pages 234–245, 1975.

[13] P. Bright. Valve dns privacy flap exposes the murky world of cheat prevention, Feb. 2014. http://arstechnica.com/gaming/2014/02/valve-dns-privacy-flap-exposes-the-murky-world-of-cheat-prevention.

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

[15] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys)*. ACM, 2011.

[16] J. Caballero, Z. Liang, P. Poosankam, and D. Song. Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration. In *Recent Advances in Intrusion Detection, 12th International Symposium, RAID 2009*, volume 5758 of *Lecture Notes in Computer Science*, pages 161–181. 2009.

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

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

[19] 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 *ACM International Conference on Advances in Computer Entertainment Technology*, June 2006.

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

[21] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[22] A. Chi, R. A. Cochran, M. Nesfield, M. K. Reiter, and C. Sturton. Server-side verification of client behavior in cryptographic protocols. *CoRR*, abs/1603.04085, 2016.

[23] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.

[24] S. G. Choi, J. Katz, R. Kumaresan, and C. Cid. Multi-client non-interactive verifiable computation. In *Proceedings of the 10th Conference on Theory of Cryptography*, 2013.

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

[26] R. A. Cochran and M. K. Reiter. Toward online verification of client behavior in distributed applications. In *Proceedings of the 20th ISOC Network and Distributed System Security Symposium*, 2013.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[28] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

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

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

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

[32] D. Fiore, R. Gennaro, and V. Pastro. Efficiently verifiable computation on encrypted data. In *Proceedings of the Conference on Computer and Communications Security*, 2014.

[33] 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`.

[34] 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.

[35] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the Conference on Advances in Cryptology*, 2010.

[36] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[37] J. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *ISOC Symposium on Network and Distributed System Security*, Feb. 2004.

[38] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, Aug. 2002.

[39] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *2005 ACM Conference on Programming Language Design and Implementation*, pages 213–223, June 2005.

[40] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *40th ACM Symposium on Theory of Computing*, May 2008.

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

[42] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, 2015.

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

[44] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[45] S. Hawkins, consultant for Sega of America. Quoted [67, p. 182], 2003.

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

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

[48] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *22nd IEEE Conference on Computational Complexity*, June 2007.

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

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

[51] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21, 2009.

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

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

[54] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[55] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–394, July 1976.

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

[57] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[58] L. Lamport. Byzantizing paxos by refinement. In *International Symposium on Distributed Computing*, 2011.

[59] R. Lanciani. Gambling and cheating in ancient rome. *The North American Review*, 155(428):97–105, 1892.

[60] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2004 International Symposium on Code Generation and Optimization*, Mar. 2004.

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

[62] M. M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ACM, 2010.

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

[64] A. Meer. Valve offers free game after 12,000 false steam bans, July 2010. http://www.gamesindustry.biz/articles/valve-offers-free-game-after-12-000-false-bans.

[65] 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.

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

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

[68] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.

[69] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip:detecting the unexpected in distributed systems. In *In Proceedings of USENIX Symposium of Networked Systems and Distributed Systems*, 2006.

[70] D. Ricketts, V. Robert, D. Jang, Z. Tatlock, and S. Lerner. Automating formal proofs for reactive systems. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*, 2014.

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

[72] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Software Engineering Notes*, 30:263–272, Sept. 2005.

[73] J. H. Siddiqui and S. Khurshid. Parsym: Parallel symbolic execution. In *Proceedings of the 2nd International Conference on Software Technology and Engineering (ICSTE)*. IEEE, 2010.

[74] N. Skrupsky, P. Bisht, T. Hinrichs, V. N. Venkatakrishnan, and L. Zuck. TamperProof: A server-agnostic defense for parameter-tampering attacks on web applicatoins. In *3rd ACM Conference on Data and Application Security and Privacy*, Feb. 2013.

[75] D. Spohn. Cheating in online games, 2005. http://internetgames.about.com/od/gamingnews/a/cheating.htm.

[76] Surian and AnAkIn. Statistics: Check if a steamid is banned or not. http://vacbanned.com/view/statistics.

[77] K. Tan, J. McHugh, and K. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Information Hiding, 5th International Workshop, IH 2002*, pages 1–17, 2003.

[78] N. Tillmann and J. D. Halleux. Pex: White box test generation for .NET. In *2nd International Conference on Tests and Proofs*, pages 134–153, 2008.

[79] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of London Mathematics Society*, 2(42):230–265, 1936.

[80] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3), Mar. 1985.

[81] Valve. Valve anti-cheat system.
https://support.steampowered.com/kb_article.php?ref=7849-Radz-6869.

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

[83] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *20th ACM International Symposium on the Foundations of Software Engineering*, FSE, pages 58:1–11, 2012.

[84] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. *SIGSOFT Software Engineering Notes*, 29:97–107, July 2004.

[85] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[86] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *9th ACM Conference on Computer and Communications Security*, Nov. 2002.

[87] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2), Feb. 2015.

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

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

[90] 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.

[91] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

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

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

[94] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

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

[96] A. Yao. How to generate and exchange secrets. In *27th Symposium on Foundations of Computer Science*, 1986.

[97] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, Mar. 2010.

[98] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *5th European Conference on Computer Systems*, pages 321–334, Apr. 2010.

[99] P. Zave. Using lightweight modeling to understand chord. *ACM SIGCOMM Computer Communication Review*, 42(2), 2012.