Toward Online Verification of Client Behavior in Distributed Applications

Robert A. Cochran
Department of Computer Science
University of North Carolina
Chapel Hill, NC, USA
rac@cs.unc.edu

Michael K. Reiter
Department of Computer Science
University of North Carolina
Chapel Hill, NC, USA
reiter@cs.unc.edu

Abstract

Existing techniques for a server to verify the correctness of client behavior in a distributed application suffer from imprecision, increased bandwidth consumption, or significant computational expense. We present a novel method for a server to efficiently search for a code path through the client that "explains" each client message, even though the server does not know local inputs to the client that might have caused the message. This method gives rise to a precise client verification technique that consumes no additional bandwidth and that validates most legitimate client messages much faster than previous such techniques. Our technique can gain even further improvements with a minimal increase in bandwidth use. We detail this innovation and use it to verify client behavior in two client-server games, namely XPilot and TetriNET. In our best configuration, verification often keeps pace with TetriNET gameplay.

1. Introduction

In client-server applications, client misbehavior can pose dangers to the larger distributed application in a variety of ways. A manipulated client may be able to compromise the server directly if the server has an extant vulnerability. Even if the server has no such vulnerabilities, any application state for which the client is authoritative can be altered by a misbehaving client and then propagated via the server to the larger distributed application.

A common approach to defend against client misbehavior is for the server to validate client messages using a model of valid client behavior derived from the sanctioned client software. For example, Giffin et al. [12] and Guha et al. [14] developed methods to confirm that requests are consistent with a control-flow model of the client. This approach admits false negatives, however — 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 [28, 22]. Greater precision has been achieved, but with greater expense. For example, the *Ripley* system [25] replays each client on the server in order to 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. An approach by Bethea et al. [2] omits transmitting client-side inputs, thus not incurring bandwidth overheads, but then must search for whether there exist inputs that could have produced the client messages observed at the server. The resulting computational expense renders this method of verification useful primarily in an offline fashion and, even then, only after modifying test applications to constrain the search spaces they present.

In this paper 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 aforementioned approach of Bethea et al. [2] but exploits 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 Bethea et al.'s, 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

[1] More precisely, the only source of false negatives is the fidelity of modeling values returned by components with which the client software

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 Bethea et al.'s [2], however. As such, we expect our algorithm to be useful primarily as an online 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 [31, 17, 30] and due to the pressure that game developers face to minimize the bandwidth consumed by their games [19]. As such, our techniques are directly useful for cheat detection in this domain. Moreover, Hoglund and McGraw [15] 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. 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 messages traces of the highly interactive *XPilot* game without game restrictions required by previous techniques [2].

The technique that we develop here is an application of symbolic execution [3], which has been widely studied and applied for various purposes (see Section 2). 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 paper is structured as follows. We discuss related work in Section 2 and necessary background in Section 3. We present our algorithm in Section 4 and Section 5. Evaluation results for this algorithm are presented in Section 6, and we conclude in Section 7.

2. Related Work

As we will see, the approach that we take to the behavior verification problem that we study is an application of symbolic execution [3], a dynamic analysis technique that "executes" a program with some values unspecified or "symbolic", in order to derive the postconditions of the software on the symbolic state. 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. Symbolic execution has a long history of study in the security and verification communities, but we believe the optimization problem we study here to be distinct from prior work, as discussed below.

The applications of symbolic execution that are most related to our own are in debugging and diagnostics. Zamfir et al. [34] 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* [33] 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 [4, 5, 7, 32]. Viewed through the lens of this paper, 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 in this work give rise to the technique we propose, which we believe to be novel.

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* [13] 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 testing* or *dynamic* symbolic execution, include *CUTE* [21], *JPF* [27] and *Pex* [23, 1]. 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

interacts (e.g., the client OS). This will be discussed further in Section 6.

without the goal of achieving high path coverage, per se.

Aside from the behavior verification that we study here, an orthogonal defense against client compromise is to strip clients of authoritative state. In this approach, any state that could affect the integrity of the larger distributed application is instead managed at the server, outside the reach of direct manipulation by the client. Tools such as *Swift* [9] automatically identify such important state for placement at the server. This approach, however, is known to increase the bandwidth consumed by interactive applications such as distributed games, owing to the need for every access to authoritative state to reach the server (e.g., [19, p. 112]). Another defense is to augment the client with monitoring software (e.g., [10, 18, 20, 11, 16]), but this approach begs the question of how to defend the monitoring software from compromise and, in some domains, has suffered resistance from the user community (e.g., [29]).

3. Background and Goals

As discussed in Section 1, our goal is to build a verifier to detect a client in a distributed application 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,[2] and previous work described an efficient method for the client to inform the server of the order in which the client processed these messages [2]. As such, the message trace $msg_0, msg_1, \ldots$ is known to the server and, so, the verifier.

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 paper 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[3]) 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. Training

The algorithm we present in this paper to meet the goals described in Section 3 incorporates a training phase that is used to configure the verifier.

4.1. 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

[2] We do not consider the loss of client-to-server messages here, though previous work [2] provided an efficient method to recover from such losses that we can employ equally well.

[3] In this paper, we abbreviate call instructions to POSIX `select()`, `send()` and `recv()` system calls (or their functional equivalents) with the labels SELECT, SEND and RECV. Our techniques apply to software written to other interfaces, of course, but would require some of our definitions to be adapted accordingly.

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. Algorithm

As we will discuss in Section 5, 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 5, 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 existing techniques [2].

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).

5. 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 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 [2], and this exhaustive search will generally be costly. However, for the applications we consider in Section 6, legitimate clients rarely require backtracking. Combined with optimizations to backtracking that we describe in Section 5.3, 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).

5.1. Basic Verification Algorithm

The verification algorithm takes as input an execution prefix Π_{n-1} consistent with msg_0, ..., 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, ..., 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 5.4; 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 verify (σ+_{n−1}, msg_n, Φ_n)
101. nd ← makeNode()
102. nd.path ← ⟨⟩; nd.state ← σ+_{n−1}
103. Live ← {nd}
104. while (|Live| > 0) {
105.    nd ← arg min_{nd′∈Live} min_{φ∈Φ_n} min_{φ′⊑φ} editDist(nd′.path, φ′)
106.    Live ← Live \ {nd}
107.    σ ← nd.state; π ← nd.path
108.    while (σ.next ≠ ⊥ and
                isNetInstr(σ.next) = false and
                isSymbolicBranch(σ.next) = false)
109.       π ← π ‖ ⟨σ.next⟩; σ ← execStep(σ)
110.    if (isNetInstr(σ.next) = true and
           ((σ.constraints ∧ σ.next.msg = msg_n) ⇏ false))
111.       return π ‖ ⟨σ.next⟩        // success!
112.    else if (isSymbolicBranch(σ.next) = true) {
113.       nd.child_0 ← makeNode()
114.       nd.child_0.path ← π ‖ ⟨σ.next⟩
115.       nd.child_0.state ← [ execStep(σ) |
                                σ.next.cond ↦ false ]
116.       if (nd.child_0.state.constraints ⇏ false)
117.          Live ← Live ∪ {nd.child_0}
118.       nd.child_1 ← makeNode()
119.       nd.child_1.path ← π ‖ ⟨σ.next⟩
120.       nd.child_1.state ← [ execStep(σ) |
                                σ.next.cond ↦ true ]
121.       if (nd.child_1.state.constraints ⇏ false)
122.          Live ← Live ∪ {nd.child_1}
123.    }
124. }
125. return ⊥        // failure
```

Figure 1. Basic verification algorithm, described in Section 5.1

Details of basic verification algorithm The algorithm for verifying a client-to-server message is summarized more specifically in Figure 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 5.4). 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, 113, and 118 in Figure 1. Each node has fields path, state, and children child_0 and child_1. The root node nd is initialized with nd.path $= \langle\rangle$ and nd.state $= \sigma^+_{n-1}$ (102). Initially only the root is created (101–102) and added to a set Live (103), which includes the nodes that are candidates for extending. The algorithm executes a **while** loop (104–124) while Live includes nodes (104) and the algorithm has not already returned (111). If the **while** loop exits because Live becomes empty, then the algorithm has failed to find a suitable execution fragment and $\perp$ is returned (125).

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 105, where $\phi' \sqsubseteq \phi$ denotes that ϕ' is a prefix of ϕ. The selected node is then removed from Live (106) since any node will be extended only once. The state σ of this node (107) is then executed forward one step at a time ($\sigma \leftarrow \text{execStep}(\sigma)$, line 109) and the execution path recorded ($\pi \leftarrow \pi \parallel \langle\sigma.\text{next}\rangle$, where $\parallel$ denotes concatenation) until this stepwise execution encounters the client exit ($\sigma.\text{next} = \perp$, line 108), a network I/O instruction ($\text{isNetInstr}(\sigma.\text{next}) = \textbf{true}$), or a symbolic branch ($\text{isSymbolicBranch}(\sigma.\text{next}) = \textbf{true}$). In the first case ($\sigma.\text{next} = \perp$), execution of the main **while** loop (104) continues to the next iteration. In the second case ($\text{isNetInstr}(\sigma.\text{next}) = \textbf{true}$) 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., $(\sigma.\text{constraints} \wedge \sigma.\text{next.msg} = msg_n) \not\Rightarrow \textbf{false}$, line 110), then the algorithm returns successfully since $\pi \parallel \langle\sigma.\text{next}\rangle$ is an execution fragment that meets the verifier's goals (111).

Finally, in the third case ($\text{isSymbolicBranch}(\sigma.\text{next}) = \textbf{true}$), the algorithm explores the two possible ways of extending π, namely by executing σ.next conditioned on the branch condition evaluating to **false** (denoted $[\ \text{execStep}(\sigma) \mid \sigma.\text{next.cond} \mapsto \textbf{false}\ ]$ in line 115) and conditioned on the branch condition evaluating to **true** (120). In each case, the constraints of the resulting state are checked for consistency (116, 121) and the consistent states are added to Live (117, 122).

5.2. Refinements

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 105). 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 [24] that tests whether $\mathsf{editDist}(s_1, s_2) \leq t$ and, if so, computes $\mathsf{editDist}(s_1, s_2)$ 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 $\mathsf{nd}' \in \mathsf{Live}$ such that for some $\phi \in \Phi_n$ and $\phi' \sqsubseteq \phi$, $\mathsf{editDist}(\mathsf{nd}'.\mathsf{path}, \phi') \leq t$. 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 $\mathsf{editDist}(\mathsf{nd}'.\mathsf{path}, \phi)$, it is possible to reuse intermediate results from a previous calculation of $\mathsf{editDist}(\mathsf{nd}'.\mathsf{path}, \phi')$ 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 $\mathsf{editDist}(\mathsf{nd}'.\mathsf{path}, \phi)$ for the execution fragments Φ_n that share the prefix represented by the interior node. In a similar way, the calculation of $\mathsf{editDist}(\mathsf{nd}'.\mathsf{path}, \phi)$ can reuse intermediate results from the $\mathsf{editDist}(\mathsf{nd}.\mathsf{path}, \phi)$ calculation, where $\mathsf{nd}'.\mathsf{path}$ extends $\mathsf{nd}.\mathsf{path}$. 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 1 assembles each path π instruction-by-instruction (lines 108–109), the paths $\mathsf{nd}'.\mathsf{path}$ and fragments Φ_n are not represented as strings of instructions for the purposes of the edit distance calculation in line 105. 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 6, yielding expensive edit-distance calculations. Instead, $\mathsf{nd}'.\mathsf{path}$ 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.

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 6), 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.

Selecting nd In each iteration of the main **while** loop 104–124 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 117, 122) and extracting a node of minimum weight (line 105), 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 6, it was not uncommon for Live to grow to tens of thousands of elements.

Memory management The verification algorithm, upon traversing a symbolic branch, creates new symbolic states to represent the two possible outcomes of the branch (lines 115 and 120). Each state representation includes the program counter, stack and address space contents. While KLEE [6] (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.

5.3. Backtracking and Equivalent State Detection

As discussed at the start of Section 5, if verify(σ^+_{n-1}, msg_n, Φ_n) returns $\perp$ (line 125), 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 111, 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 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 2. Toy example 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.[4] 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] 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.

5.4. 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). We consider two possibilities for populating Φ_n in this paper.

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 6, 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 6, we use $\beta = 8$.)

The distance measure between messages that we use in our evaluation in Section 6 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, 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.

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. (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.[5] For the applications we evaluate in Section 6, 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). While sending a hint does increase bandwidth cost, it does so minimally; e.g., in Section 6, 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.

6. Evaluation

To evaluate our technique, we built a prototype that uses the KLEE [6] 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 5.4, 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*, for two reasons. First, performance is the dimension on which our algorithm offers a contribution over the most closely related previous research [2]. Second, 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 5.3). Similarly, the only

[5] 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 6, however, backtracking usually incurred more verification cost even in this case.

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 6.1). 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 previous work [2] 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 [30].

6.1. Applications

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 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.

The previous work by Bethea et al. that leveraged *XPilot* to evaluate its techniques found it necessary to modify the *XPilot* client in various small ways to make its analysis tractable (see [2, Section 5.2]). We used this modified version in our evaluations, as well, though to illustrate certain improvements enabled by our technique, we reverted an important modification made there. Specifically, Bethea et al. 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, caused symbolic execution to explore arbitrary numbers of corresponding input-processing loop iterations. By inserting bounds, Bethea et al. rectified this problem but introduced a potential source of false positives, if the deployed client software is not modified in the same way. In our evaluation, 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. Aside from highlighting the strength of our technique, removing these bounds renders the Bethea et al. approach to verification intractable.

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 Bethea et al. to cap the number of inputs in a single *XPilot* event-loop iteration. As such, in their experimentation with *TetriNET*, Bethea et al. 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 [2]. We emphasize that none of these restrictions are employed in our evaluation, and again the ability of our algorithm to verify the behavior of a *TetriNET* client in its unconstrained form illustrates its strengths.

6.2. Results

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).

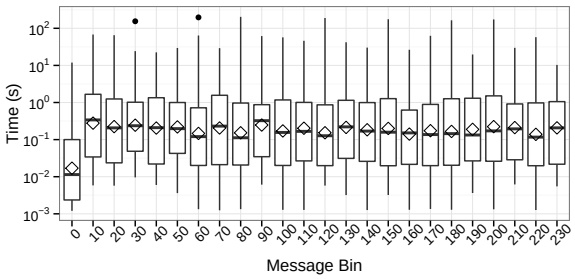
TetriNET Figure 3 shows *TetriNET* verification costs. Figure 3 includes plots for both the default and hint configurations, as well as for clustering parameter values $k = 256$ and $k = 3790$; the latter case ensured a single execution fragment per cluster.

The numbers represented in Figure 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 3 shows the distribution of verification time per message, binned into ten-message bins, across all 20 traces. So, for example, the boxplot labeled "0" shows the distribution of verification times for messages $msg_0, \ldots, msg_9$ in the 20 traces. The data point for message msg_n accounts for all time spent in $\mathsf{verify}(\sigma^+_{n-1}, msg_n, \Phi_n)$ and any immediately preceding preprocessing step (see Section 5.1), 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.

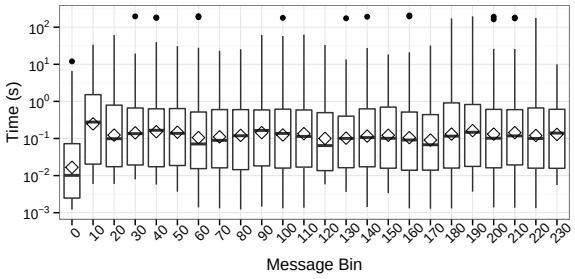
Several things are worth noting about Figure 3. In all cases, the distribution of verification times is largely unaffected by the message index, i.e., where in the trace the message appears. This confirms that our implementation is mostly free from sources of increasing verification expense as traces 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 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 delay of roughly 1.6s. That said, there are two issues that require further exploration. First, there are messages that induce verification times 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.

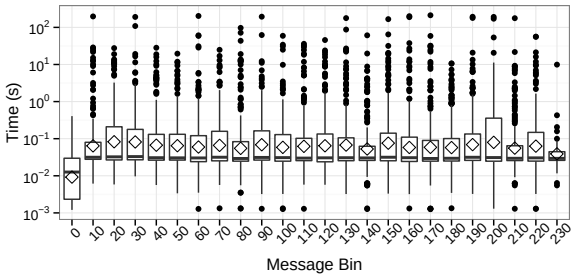
To shed light on these issues, Figure 4 instead plots the distributions of per-message verification *delay* between the arrival of message msg_n at the server (where a server-to-client message "arrives" when it is sent) and the discovery of an execution prefix Π_n that is consistent with $msg_0, \ldots, msg_n$. Delay (Figure 4) differs from verification time (Fig-



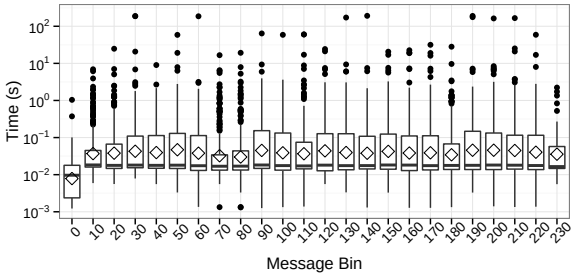
(a) Default, $k = 256$



(b) Hint, $k = 256$

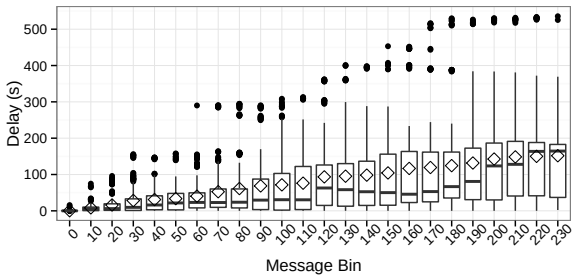


(c) Default, $k = 3790$

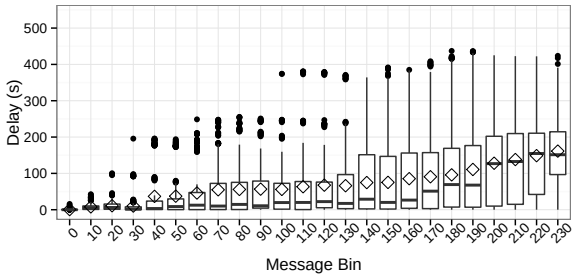


(d) Hint, $k = 3790$

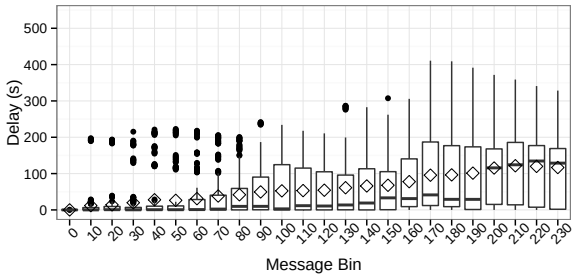
Figure 3. *TetriNET* **verification times. Cross-validation over** 20 **traces. Boxplot at** x **shows verification times for messages** msg_x**,** $\ldots$**,** msg_{x+9} **in each trace (after training on the other traces). "◇" shows the average.**



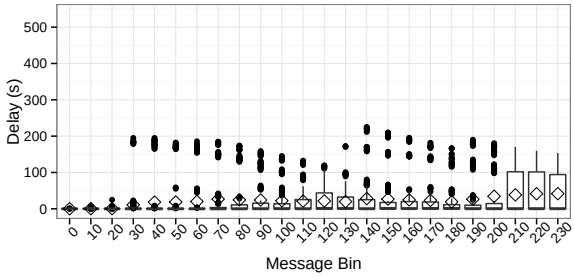
(a) Default, $k = 256$



(b) Hint, $k = 256$



(c) Default, $k = 3790$



(d) Hint, $k = 3790$

Figure 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). "◇" shows the average.**

ure 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.

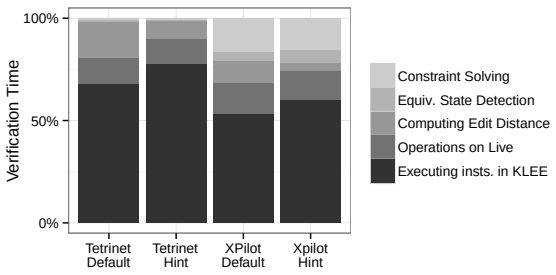


Figure 5. Percentage of time spent in each component of our algorithm.

One item to note about these graphs is that for the hint configuration with $k = 3790$ (Figure 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(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(c)).

A breakdown of verification costs for *TetriNET* is shown in Figure 5. In our *TetriNET* experiments, more than 50% of the verification time is spent in KLEE, interpreting client instructions. Therefore, optimizations that interpret instructions only selectively (e.g., [8]) 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 5.3) or in constraint solving. In Figure 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 (borrowed from Bethea et al. [2, Section 4.4]) and a canonicalization step (borrowed from Visser et al. [26]) to improve the hit rate on cached results for previous queries to STP. These optimizations significantly reduce the overall constraint solving time.

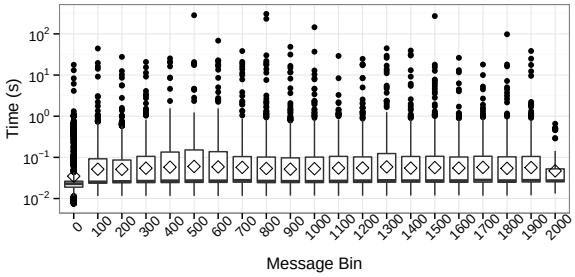
XPilot *XPilot* poses a significant challenge for verification because its pace is so fast. The tests described here use an *XPilot* configuration that resulted in an average of 32 messages per second. The verification times per message vary somewhat less for *XPilot* than they did for *TetriNET*, as shown in Figure 6. Recall that each boxplot in Figure 6 represents 100×40 points, versus only 10×20 in Figure 3. As such, though there are larger numbers of outliers in Figure 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 6(c) and Figure 6(d) to Figure 3(c) and Figure 3(d), respectively. However, *XPilot* verification is considerably faster with coarse clustering, see Figure 6(a) versus Figure 3(a) and Figure 6(b) versus Figure 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%.

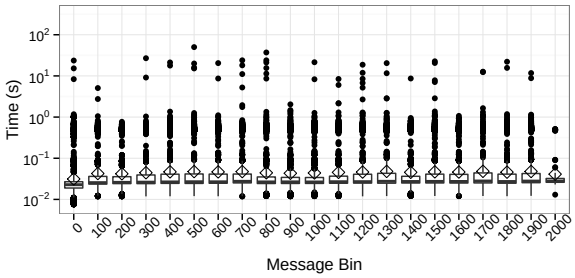
Though the median per-message verification cost of *XPilot* is generally as good or better than that for *TetriNET*, the faster pace of *XPilot* makes it much more difficult for verification to keep pace with the game. This effect is shown in Figure 7. As shown in this figure, none of the configurations 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 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 most closely related previous work [2] completely intractable. That is, recall that Bethea et al. 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 6.1). With these restrictions removed, the Bethea et al. 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 6).

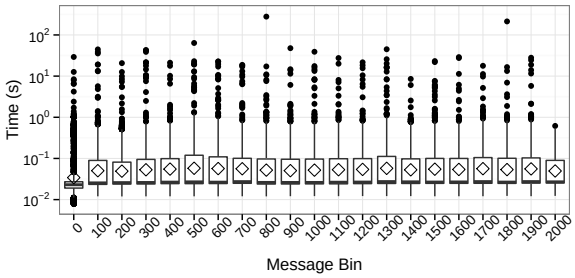
A fractional breakdown of verification times for *XPilot* are shown in Figure 5. While a majority of the cost is still contributed by interpreting client instructions in KLEE, the majority is smaller in the case of *XPilot* than it was for *TetriNET*. For *XPilot*, equivalent state detection



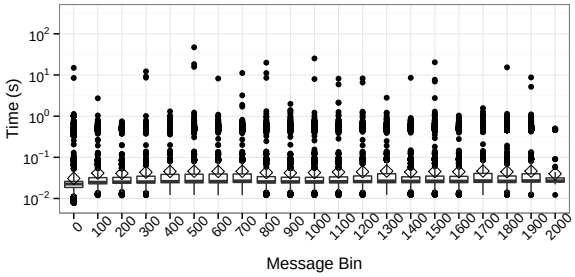
(a) Default, $k = 256$



(b) Hint, $k = 256$

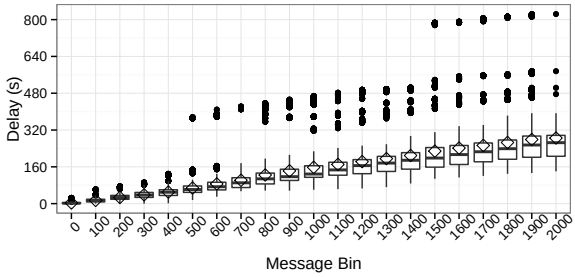


(c) Default, $k = 475$

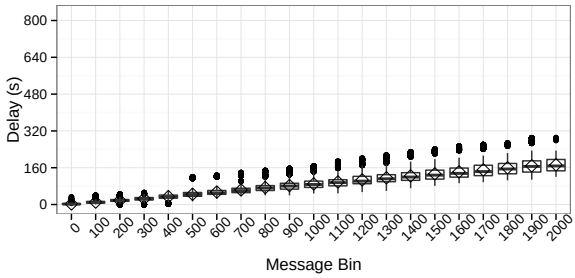


(d) Hint, $k = 475$

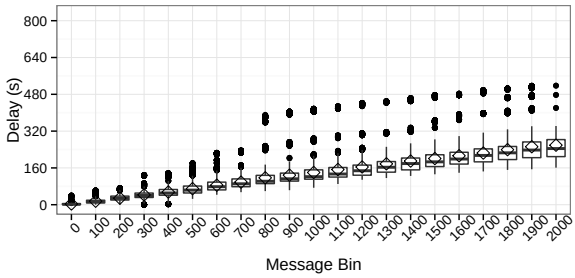
Figure 6. *XPilot* **verification times. Cross-validation over** 40 **traces. Boxplot at** x **shows verification times for messages** msg_x**, ...,** msg_{x+99} **in each trace (after training on the other traces). "◇" shows the average.**



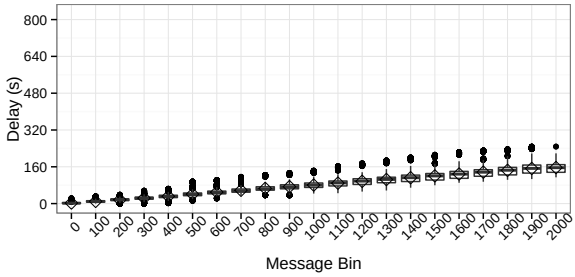
(a) Default, $k = 256$



(b) Hint, $k = 256$



(c) Default, $k = 475$



(d) Hint, $k = 475$

Figure 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). "◇" shows the average.**

(Section 5.3) 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*.

7. Conclusion

In this paper 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 [19]) 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.

Acknowledgements This work was supported in part by NSF grants 0910483 and 1115948 and by a gift from Intel. We are grateful to Darrell Bethea for comments on a draft of this paper.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

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

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.