Finding Trojan Message Vulnerabilities in Distributed Systems

Radu Banabic
École Polytechnique Fédérale de Lausanne
radu.banabic@epfl.ch

George Candea
École Polytechnique Fédérale de Lausanne
george.candea@epfl.ch

Rachid Guerraoui
École Polytechnique Fédérale de Lausanne
rachid.guerraoui@epfl.ch

Abstract

Trojan messages are messages that seem correct to the receiver but cannot be generated by any correct sender. Such messages constitute major vulnerability points of a distributed system—they constitute ideal targets for a malicious actor and facilitate failure propagation across nodes. We describe Achilles, a tool that searches for Trojan messages in a distributed system. Achilles uses dynamic white-box analysis on the distributed system binaries in order to infer the predicate that defines messages parsed by receiver nodes and generated by sender nodes, respectively, and then computes Trojan messages as the difference between the two.

We apply Achilles on implementations of real distributed systems: FSP, a file transfer application, and PBFT, a Byzantine-fault-tolerant state machine replication library. Achilles discovered a new bug in FSP and rediscovered a previously known vulnerability in PBFT. In our evaluation we demonstrate that our approach can perform orders of magnitude better than general approaches based on regular fuzzing and symbolic execution.

Categories and Subject Descriptors D.2.5 [*SOFTWARE ENGINEERING*]: Testing and Debugging - Symbolic execution

Keywords symbolic execution; distributed system; testing; Trojan

1. Introduction

Most reliability techniques in distributed systems are based on the assumption that failures are independent: if a node fails in some way, the impact on other nodes should be minimal, i.e., the failure should not propagate. But even if nodes are programmed independently and placed on geographically remote sites, nodes need to communicate, typically by exchanging messages. Such messages can propagate failures among nodes.

In this paper, we study what we call *Trojan messages*. These are messages that are intelligible to correct receiver nodes of a distributed system, but cannot be generated by any correct sender nodes in that system. Similar to divisions by zero or buffer overflows, Trojan messages result from absence of defensive programming and constitute a source of vulnerabilities. We know of no work that automatically discovers this type of messages.

Trojan messages often creep into the *implementation* of nodes, while being invisible in their high level *specification*. Such messages can have a major impact on the behavior of distributed systems. For example, the Amazon S3 storage system suffered several hours of downtime in 2008 [1]. The problem was caused by "a handful of messages [...] that had a single bit corrupted". Unfortunately for the system, "the message was still intelligible, but the system state information was incorrect". This allowed the initial corruption to propagate to other, correct nodes of the system, until most of the collective information in the system was corrupt. Such corner cases are difficult to identify. In order to prevent such downtime in the future, Amazon engineers added checks to "log any such messages and then reject them".

The core problem that caused the Amazon S3 downtime was the fact that nodes accepted the corrupt messages despite the fact that no correct node could have generated such messages in the respective context. We say that S3 nodes accepted Trojan messages. We argue that, if such messages exist in a system implementation, they represent an unnecessary vulnerability—servers should do what correct clients require them to do and not one bit more. Since, by definition, correct nodes cannot generate Trojan messages, it is likely that such messages will not be encountered during regular testing. Trojan messages are likely to exercise untested code paths and surface hidden, potentially undesired, behavior.

Current testing techniques face vast search spaces of possible inputs. This limits their ability to find the Trojan messages that could cause the system to fail at some point in the future. Recognizing the important distinction between specification and implementation, the most stringent operators of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2305-5/14/03…$15.00.
http://dx.doi.org/10.1145/2541940.2541984

distributed systems have started testing their live production systems with fault injection (e.g., Google organizes regular "fire drills" in which engineers intentionally cause failures in critical live systems [18]). Trojan messages are good candidates for such fault injection. However, guessing which messages are Trojan is still hard and mostly an elusive goal.

We propose Achilles, a system whose purpose is to identify Trojan messages that constitute the Achilles heel of any distributed system. Our approach is based on:

1. A new usage model for symbolic execution, aimed at discovering execution paths that accept Trojan messages.
2. A set of optimizations that enable efficient search for Trojan messages.

We applied Achilles to the File Service Protocol [12], a UDP-based file transfer protocol implementation that is distributed with some Debian versions, and to PBFT [6], an implementation of a Byzantine fault-tolerant replication protocol. We show that Achilles has high accuracy; it is orders of magnitude more efficient at finding Trojan messages than black-box fuzzing or classic symbolic execution. Achilles found in FSP a high-level semantic bug and also rediscovered a known vulnerability in PBFT—subtle bugs that would be difficult to discover using other techniques.

The rest of the paper is organized as follows: we first present the high-level idea of Achilles along with a working example of a system under test (§2). In §3 we describe the basic mechanism of Achilles and the optimizations that make it practical. We discuss the soundness and completeness of our approach in §4. Then, we describe our current prototype (§5) and evaluate Achilles on two real distributed systems (§6). Finally, we review related work (§7) and conclude (§8).

2. Achilles: A Primer

Achilles is a system designed to identify Trojan messages in a distributed system. For simplicity, we consider in our description of Achilles only client-server systems where the client generates requests and the server replies; it is straightforward to generalize the approach to peer-to-peer or other types of distributed systems.

We define Trojan messages as those that are accepted by correct server nodes in a distributed system but cannot be generated by any correct client node, as illustrated in Figure 1. By "correct" we mean nodes that execute the unaltered implementation of the distributed system, in contrast to incorrect nodes, which are nodes that have encountered a fault (e.g., memory bit flip), or are controlled by malicious users. Thus, in our definition, "correct" does not necessarily imply that nodes follow the high-level specification of the protocol, or do what the developers intended.

Achilles has two phases. In a first phase, it computes a predicate that defines all messages that can be generated by correct clients. We call this the *client predicate* P_C. The set

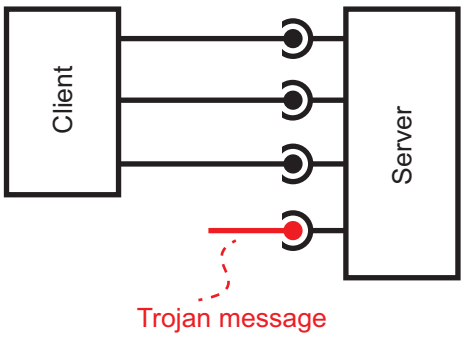

Figure 1. Trojan messages are messages that are accepted by a server but cannot be generated by any correct client.

of messages that can be generated by correct clients is $\mathcal{C} = \{msg \mid P_C(msg)\}$. In the second phase, Achilles computes a predicate that defines all messages that are accepted by the server, the *server predicate* P_S. The set of messages that are accepted as correct by correct servers is $\mathcal{S} = \{msg \mid P_S(msg)\}$. Then, the set of Trojan messages is defined as $\mathcal{T} = \mathcal{S} \setminus \mathcal{C}$, which can be expressed as $\mathcal{T} = \{msg \mid P_S(msg) \wedge \neg P_C(msg)\}$.

In essence, Achilles extracts the grammar of the communication protocol, as implemented in both the client and the server. It then operates on the two grammars, trying to determine the difference between the server's implementation of the grammar and the client's. As we describe later (§3), Achilles uses symbolic execution in order to analyze the implementation of clients and servers; it represents the extracted grammars using symbolic constraints.

2.1 Working Example

To illustrate the idea underlying our approach, consider the sample server implementation presented in Figure 2.

This is a simplified example, in which a server handles read or write requests from clients. The server first initializes its internal data structures (lines 2-3). Next, the server enters an event loop in which it handles client requests. The program receives a message and stores it in the local variable *msg* (line 5). Then, the server checks that the message sender is in a pre-configured group of known peers (line 6) and if the CRC error-detection code is correct.

If the message passes the general validation tests, the server reads the *request* field to get the message type—either a *READ* or a *WRITE* request. If the request is not one of the two, the message is discarded (line 29). For either of the request types, the server validates the address field and handles the respective request.

The client shown in Figure 3 follows the same protocol as the server. It reads user requests from standard input, validates the data (lines 5-7) and computes corresponding requests for the server.

The sample system has a Trojan message. The server validates that the requested address is below the maximum

```
1  #define DATASIZE 100
2  peers = initializePeers();
3  data = new int[DATASIZE];
4  while(TRUE) {
5    msg = receiveMessage();
6    if (!isInSet(msg.sender, peers))
7      continue;
8    if (!isValidCRC(msg, msg.CRC))
9      continue;
10   switch (msg.request) {
11     case READ:
12       if (msg.address >= DATASIZE)
13         continue;
14       //Security vulnerability: forgot to check
15       //          address < 0
16       sendMessage(msg.sender, REPLY,
17                   data[msg.address]);
18       continue;
19     case WRITE:
20       if (msg.address >= DATASIZE)
21         continue;
22       if (msg.address < 0)
23         continue;
24       data[msg.address] =
25                   msg.value;
26       sendMessage(msg.sender, ACK);
27       continue;
28     default:
29       continue};
30   }
31 }
```

Figure 2. A simple server that handles requests from clients. The server accepts Trojan messages, as it does not correctly validate the address field of read requests.

DATASIZE, however, for *READ* requests it does not ensure that the address is greater than zero. The client, however, validates the input from the user before contacting the server. Therefore, no correct client can generate *READ* messages with negative offsets, although they are accepted by the server. Thus, any *READ* message with a negative address is a Trojan message.

In this example, the Trojan message can lead to a potential privacy leak, as attackers can read from negative offsets in the data array and discover, for instance, the list of peers that communicate with the server.

3. Design

At a high level, Achilles works as follows. In a first phase, it obtains the client predicate P_C. Then, it pre-processes P_C to eliminate redundancy and to pre-compute structure information for the next phase (details follow in §3.3). In the second phase, Achilles computes the server predicate P_S, as well as incrementally discovering Trojan Messages. As an optimization, our implementation does not compute the entire formula of P_S before computing Trojan messages, but rather searches for Trojan messages incrementally, as it builds P_S.

In this section, we first give an overview of how Achilles uses symbolic execution in order to extract the grammar

```
1  #define DATASIZE 100
2  peerID = getPeerID();
3  operationType = readFromKeyboard();
4  address = readFromKeyboard();
5  if (address >= DATASIZE)
6    exit(1);
7  if (address < 0)
8    exit(1);
9  //Client only sends addresses in [0,100)
10 if (operationType == READ) {
11   msg = new ReadMessage();
12   msg.sender = me;
13   msg.request = READ;
14   msg.address = address;
15   msg.CRC = computeCRC(msg);
16   sendMessage(server, msg);
17 }
18 if (operationType == WRITE) {
19   value = readFromKeyboard();
20   msg = new WriteMessage();
21   msg.sender = me;
22   msg.request = WRITE;
23   msg.address = address;
24   msg.value = value;
25   msg.CRC = computeCRC(msg);
26   sendMessage(server, msg);
27 }
```

Figure 3. A simple client that generates messages. Correct clients validate the address field, therefore cannot expose the bug in the server.

from both the client and the server (§3.1). We then describe how Achilles finds Trojan messages efficiently using off-the-shelf constraint solvers (§3.2). We present optimizations that enable Achilles to handle large client predicates (§3.3). Finally, we describe how Achilles handles local state in the client and in the server (§3.4).

3.1 Symbolic Execution and Message Grammars

Achilles uses symbolic execution [17] to extract the grammars of messages that can be sent/received by an implementation of a distributed system. Symbolic execution is a technique that systematically explores execution paths in a system. Rather than exercising a system with concrete inputs, a symbolic execution engine provides "symbolic" data, which can conceptually take any value. The symbolic execution engine then executes the system under test and interprets expressions symbolically. At each branching point that depends on symbolic data, the symbolic execution engine invokes a Satisfiability Modulo Theories (SMT) solver to assess the feasibility of each branch, based on the current state of the system. When a branch is deemed feasible, the execution engine follows the respective branch and also updates the symbolic data, keeping track of the constraints that need to be satisfied such that the branch is feasible. If both paths are feasible, the execution engine forks the exploration and follows both branches.

At every step of the symbolic execution, the engine keeps track of several possible execution states of the system, each

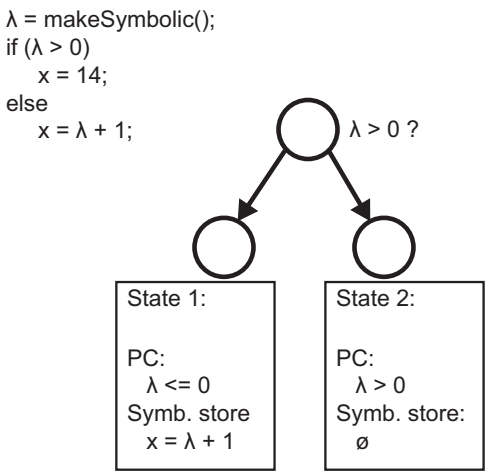

Figure 4. An example of symbolic execution for a small piece of code.

```
∃ symb_PeerID , symb_Address , symb_Value :

message.sender = symb_PeerID
⋀ message.request = READ
⋀ message.address = symb_Address
⋀ message.CRC = CRC(message)
⋀ symb_Address < 100
⋀ symb_Address >= 0
⋁
message.sender = symb_PeerID
⋀ message.request = WRITE
⋀ message.address = symb_Address
⋀ message.value = symb_Value
⋀ message.CRC = CRC(message)
⋀ symb_Address < 100
⋀ symb_Address >= 0
```

Figure 5. The client predicate $P_C(message)$ of the client in Figure 3, as discovered by the symbolic execution phase of Achilles.

corresponding to a possible execution path. Figure 4 shows a sample symbolic execution of a small piece of code. A symbolic execution state consists of a *symbolic store*, which maps variables from the system under test to expressions on symbolic inputs, and the set of *path constraints*, which represent the conditions on symbolic input that must hold for the respective execution path to be feasible.

In Achilles, we represent message grammars as sets of constraints obtained during symbolic execution. The server predicate P_S is expressed by the set of path constraints that correspond to execution paths that handle accepted messages. The client predicate P_C is expressed by the symbolic messages that the client sends over the network on different execution paths, along with their respective path constraints.

Client Predicate. In a distributed system, clients receive "local" inputs, parse the inputs, generate corresponding messages conforming to the specification of the protocol (or, more precisely, to the client's implementation of that specification), and finally send the messages to the server. The client predicate is a representation of all valid messages that can be *sent*.

In order to obtain the set of all possible messages that can be generated by a client in a distributed system, we conceptually need to provide the client with all possible "local" inputs and then capture the messages it sends over the network. We start the client in a symbolic environment such that any local system call that the client emits in order to read input is intercepted, and the result is replaced with symbolic data. The symbolic execution engine keeps track of how the symbolic data flows through the client and, when the client attempts to send a message over the network, Achilles captures and stores the contents of the message and the corresponding path constraints. The message payload may contain concrete data, but also expressions on symbolic data.

Thus, the predicate P_C is the disjunction of predicates $path_{C_1} \vee \cdots \vee path_{C_n}$, where each $path_{C_i}$ is a conjunction of path constraints and symbolic expressions for an execution path that sends a message. We call each such predicate $path_{C_i}$ a *client path predicate*.

For the example client in Figure 3, Achilles discovers the predicate presented in Figure 5. Names that begin with *symb_* indicate symbolic data automatically inserted by Achilles. There are two main execution paths in the client that lead to messages being sent. One path corresponds to a *READ* request, while the other corresponds to a *WRITE* request. In both cases, the *message.request* header contains concrete data. This is because there is no data flow dependency between the *message.request* header and any symbolic input; the values for the two paths only differ because of control flow dependencies.

All other headers except *message.request* contain symbolic data. The *message.value* and *message.sender* headers, for example, contain the respective unconstrained symbolic inputs. The *CRC* header contains an expression on all other symbolic inputs (in Figure 5, the expression is summarized by the CRC function, but in real deployments, the expression contains the full chain of operations that transform the symbolic inputs). Finally, the *address* header contains constrained symbolic data. There are no operations directly performed on *symb_Address*; however, the execution path that leads to the message imposes constraints on the possible values: *symb_Address* needs to have values between $[0, 100)$ in order for the message to be sent. This is reflected in the path constraints that are captured and stored by Achilles.

In distributed systems, messages usually have clearly differentiated fields. Developers might be interested to check only a subset of those fields for Trojan messages (e.g., only check the contents of the *address* field). We support this selective approach in Achilles (§5.2) and also refer to techniques that automatically avoid complex authentication or encryption functions (§7). Since the grammar extraction

phase of Achilles is based on vanilla symbolic execution, Achilles can benefit from a variety of approaches that enhance symbolic execution.

Server Predicate. In a distributed system, the server receives messages from a client, parses the message, and executes the operation requested by the client. The server should discard messages that do not follow the specification, and only execute operations corresponding to valid messages. The server predicate P_S is a representation of all valid messages that can be *received*.

Each message received by the server triggers the execution of code to handle it. For simplicity, we refer to the sequence of executed instructions as the *execution path* triggered by the message. The execution path starts after the return of the *receive* instruction and ends when the message processing is complete—either the server exits, or it listens for new events.

After receiving a message, the server decodes it and parses its fields, checking if the message conforms to the specification of the protocol (or, more precisely, to the server node's implementation of that specification). In order to infer the set of valid messages, Achilles first classifies execution paths in the system under test as either *accepting* or *rejecting*. *Accepting* execution paths are those that are triggered by messages that pass the initial parsing stages of the system under test and cause the server to perform an action. Conversely, *rejecting* execution paths are those that are triggered by messages that are rejected by the server. Achilles automates the classification of paths, using some simple observations on the behavior of the server. Human operators of Achilles can also manually place tags in the code in order to speed up the analysis (details follow in §5.2).

In order to obtain the server predicate, Achilles executes the server node symbolically, feeding it an unconstrained symbolic message. The server analyzes the message and branches into different execution paths, one for each type of message in the protocol specification. Symbolic execution enumerates all these paths, keeping track of the constraints that make each path feasible.

We define the server predicate P_S as the disjunction of all path constraints corresponding to *accepting* execution paths. Thus, P_S represents all possible messages that can trigger *accepting* execution paths in the server.

Similarly to P_C, the server predicate P_S is obtained using vanilla symbolic execution. Therefore, Achilles can benefit from any optimization that enhances the performance of symbolic execution engines.

In the example in Figure 2, we define *rejecting* execution paths as those that do not send a reply to the client (reach lines 7, 9, 13, 21, 23 or 29 of the server code), and *accepting* execution paths as those paths that send a reply (reach line 17 or 26). The server predicate, as discovered by our technique, is presented in Figure 6.

```
isInSet(message.sender, peers) = TRUE
∧ isValidCRC(message, message.CRC) = TRUE
∧ message.request = READ
∧ message.address < 100
∨
isInSet(message.sender, peers) = TRUE
∧ isValidCRC(message, message.CRC) = TRUE
∧ message.request = WRITE
∧ message.address < 100
∧ message.address ≥ 0
```

Figure 6. The server predicate $P_S(message)$ of the server in Figure 2, as discovered by the symbolic execution phase of Achilles.

3.2 Finding Trojan Messages

Trojan messages are, by definition, all messages in $\mathcal{S} \setminus \mathcal{C}$. This is equivalent to $\mathcal{T} = \{msg \mid P_S(msg) \wedge \neg P_C(msg)\}$. In Achilles, Trojan messages are computed incrementally, as it builds up the formula of P_S. This point is essential, as clients are usually less complex than servers. The extraction of the client predicate P_C is easier than the extraction of the server predicate P_S.

For every explored execution path in the server, Achilles keeps track of a list of client path predicates that contain messages that can trigger the respective path, as shown in Figure 7. At every branch point encountered during the symbolic execution of the server, Achilles checks which of the client messages can still trigger each path, and whether the paths can be triggered by any Trojan messages. As soon as an execution path cannot be triggered by any Trojan messages, it is dropped from the exploration. Therefore, by construction, any execution path in the server that reaches an *accepting* marker has Trojan messages. For such execution paths, Achilles outputs a symbolic expression and a concrete example of the Trojan message.

Note that Trojan messages might be exclusive on execution paths (e.g., the accepting path on the right in Figure 7), or they might be bundled with other, non-Trojan messages (e.g., the accepting path on the left). The latter case means that classic symbolic execution by itself does not alleviate the problem of finding Trojan messages by much—it weeds out execution paths with dropped messages from those with accepted messages, but Trojan messages may be anywhere among the execution paths that handle accepted messages. In the example in Figure 2, the Trojan messages (those with $msg.address < 0$) are on the same execution path with valid messages (those that satisfy $msg.address >= 0$ and $msg.address < DATASIZE$); symbolic execution by itself would not point out the presence of a potential bug. This is also the case of the wildcard bug found by Achilles in FSP (§6.3) and the vulnerability found by Achilles in PBFT (§6.3).

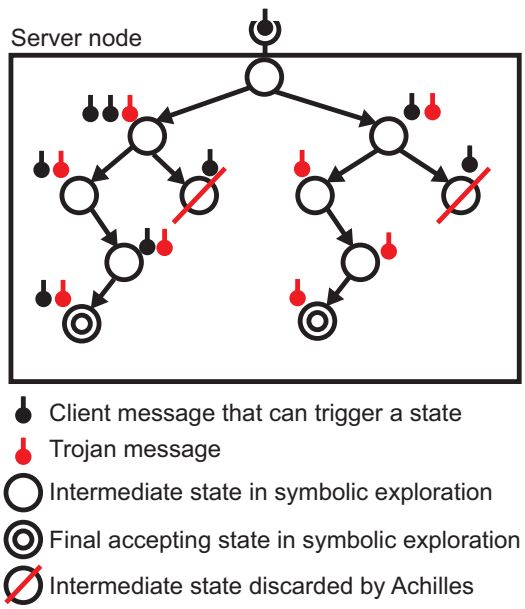

Figure 7. Symbolic execution of a server node in Achilles. Using the predicate collected from the client, Achilles restricts the server exploration to paths that can be triggered by Trojan messages.

We optimize the discovery of Trojan messages by adapting Achilles to the particularities of symbolic execution, as described in the following subsections.

Constraint Solving. Predicates P_C and P_S are disjunctions of path predicates. Each path predicate $path \in P$ is a conjunction of constraints and assignments. A predicate defines a class of messages generated by a client, or accepted by a server. In order to compare two predicates $path_1$ and $path_2$, Achilles must combine the two in a single formula and then call an SMT solver in order to check satisfiability.

In order to combine a client and a server path predicate, Achilles adds a conjunction between their respective path constraints, and also adds an equality constraint between the value of the message generated in the client and the value of the message received in the server. In essence, the combination between a server and a client path predicate, $path_S$ and $path_C$, represents messages *msg* such that:

- msg_S satisfies the server predicate
- msg_C satisfies the client predicate
- $msg_S = msg_C = msg$

SMT solvers, such as STP [13] or Z3 [11], verify the satisfiability of a set of constraints and expressions on symbolic variables. Furthermore, SMT solvers can compute a concrete assignment of the symbolic variables that satisfies a set of expressions and constraints. For example, an SMT solver can determine that the set of constraints $\lambda > 0 \wedge \lambda < -5$ is unsatisfiable. However, $\lambda > 0 \wedge \lambda < 5$ is satisfiable and $\lambda = 3$ satisfies the constraints.

In terms of symbolic expressions, the definition of the set of Trojan messages $\mathcal{T}$ can be written as the set of messages that satisfy the server predicate, but not the client predicate:

$$\mathcal{T} = \{msg \mid P_S(msg) \wedge \neg P_C(msg)\}$$

P_C contains an existential quantifier; negating it produces a universal quantifier. This makes solving the expression above difficult using current SMT solvers. Z3 has limited support for quantifiers, using heuristics and patterns to eliminate quantification. Achilles calls Z3, attempting to solve the predicate and check for the presence of Trojan messages. However, the heuristics may fail; in this case, Z3 cannot answer whether an expression is satisfiable or not. In the following subsection, we discuss how we under-approximate the negation of client path predicates and eliminate the universal quantifier.

Negating Path Predicates. Recall that a client path predicate $path_C \in P_C$ represents all possible messages that can be generated on a particular execution path. Let $negate(path_C)$ be an operator that generates a predicate representing all possible messages that *cannot* be generated on that path.

Figure 8 shows the expression of READ messages generated by the sample client in Figure 3. There are three symbolic variables in the message, $\lambda_{PeerID}, \lambda_{Address}$ and λ_{CRC}, and one concrete variable, containing the value $READ$. λ_{PeerID} is the unconstrained value of the $symb_PeerID$ input. $\lambda_{Address}$ is the value of the $symb_Address$ input, but it is subjected to the restrictions in the path constraint. Finally, λ_{CRC} is a more complex operation on $\lambda_{PeerID}, \lambda_{Address}$ and $READ$.

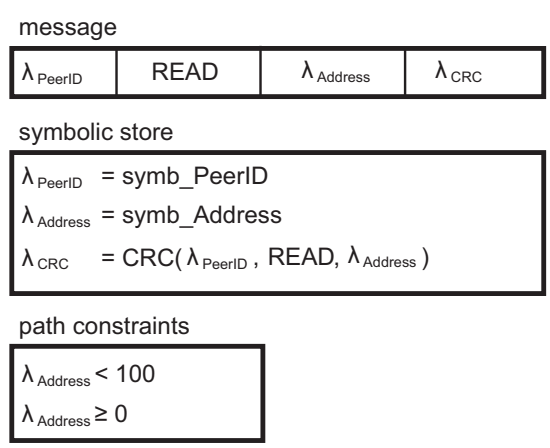

Figure 8. The expression of READ messages generated by the sample client.

Achilles under-approximates the true *negate* operator as follows. Achilles' custom operator computes the negation of a path predicate as a disjunction of the negation of each individual value in the message buffer. In order to negate a value, there are two cases:

1. If the value is a concrete value C, then replace the respective value with a new symbolic value λ and add the constraint $\lambda \neq C$.
2. If the value is an expression on symbolic variables, then negate the set of constraints that influence the respective variables. If there are no constraints available, then abandon the negation of the current value.

For the example in Figure 8, the negation of the path predicate is a representation of the set of message where:

- the *request* field is different from *READ* or
- the *address* field is smaller than 0 or greater than 100 or
- the *CRC* field is the CRC function of a message where the *address* is smaller than 0 or greater than 100 or its *request* is different from READ.

In constraint form, this is written as:

$$(message.request = \lambda_{req}) \wedge \lambda_{req} \neq READ$$
$$\vee (message.address = \lambda_{Address})$$
$$\wedge (\lambda_{Address} \geq 100 \vee \lambda_{Address} < 0)$$
$$\vee (message.CRC = CRC(READ, \lambda_{Address}, \lambda_{PeerID}))$$
$$\wedge (\lambda_{Address} \geq 100 \vee \lambda_{Address} < 0)$$

We discuss the soundness and completeness of the custom *negate* operator in a dedicated section (§4).

3.3 Handling Large Client Predicates.

The server analysis phase of Achilles is a search problem—find all possible execution paths that accept Trojan messages. Symbolic execution enumerates the possible execution paths in the server, with the added precondition that there exists at least one message that satisfies the path constraints in the server and the conjunction of all negated client path predicates. The negation operator allows checking for Trojan messages with a single query to the constraint solver; however, this query can be complex.

Achilles needs a strategy to efficiently handle the thousands of expressions that can appear in a client predicate. The key idea is to keep track of which path predicates can trigger an execution path explored in the server.

The P_S extraction phase of Achilles incrementally adds constraints to each $path_S \in P_S$, as it explores execution paths. Rather than just checking whether there are Trojan messages on the current path (if $path_S \wedge negate(path_{C_1}) \wedge \cdots \wedge negate(path_{C_n})$ is satisfiable), Achilles also checks whether $path_S \wedge path_{C_i}$ is satisfiable, for each $path_{C_i} \in P_C$. Whenever the latter is not satisfiable, Achilles drops $negate(path_{C_i})$ from the solver query that checks for Trojan messages—if $path_S \wedge path_{C_i}$ is false, then $path_S \wedge negate(path_{C_i})$ is implicitly true (this is because $path_S$ and $path_{C_i}$ are both satisfiable, by construction). Once dropped, $path_{C_i}$ will never be checked again on the respective execution path; $path_{C_i}$ does not change, while $path_S$ only becomes more constrained, meaning that the conjunction can never become feasible in the future. As the symbolic execution engine explores longer execution paths, the number of client path predicates that can trigger the path decreases, and the size of the SMT query that checks for Trojan messages becomes smaller (we evaluate the effect of this optimization in our experiments §6.4).

In order to further optimize the search for Trojan messages, Achilles also exploits the process through which path predicates are generated. The intuition behind this optimization is that messages generated by the client are similar to each other. Two client path predicates can represent the same values for a certain field in a message, e.g., $path'_C = (msg.x = 1) \wedge (msg.y = 2)$ and $path''_C = (msg.x = 1) \wedge (msg.y = 7)$ have the same values for field x. In this case, if $path'_C$ is dropped due to a new server constraint on field x, then $path''_C$ can also be dropped without further checking.

Achilles pre-computes a data structure that stores information about relations between client path predicates. The data structure can be seen as a three-dimensional matrix $differentFrom$, where $differentFrom[i][j][field] = TRUE$ means that there exists at least one message $msg_i \in path_{C_i}$ such that there is no message $msg_j \in path_{C_j}$ with $msg_i.field = msg_j.field$. The data structure is computed by applying the *negate* operator for each field of messages, between each pair of client path predicates. The data structure is only computed for message fields that are independent, meaning that they do not appear in constraints with other fields. As can be seen in our evaluation (§6), this precomputation is fast; moreover, it is trivially parallelizable.

While symbolically executing a server node, Achilles uses the information from $differentFrom$ at every point where the exploration forks. If a branching point depends on field a from the received message (and not on any other fields), and the new constraints of the branching point make $path_{C_i} \wedge path_S$ no longer satisfiable, then Achilles can drop $path_{C_i}$ from the current state, but also all path predicates $path_{C_j}$ such that $differentFrom[j][i][a] = FALSE$. In other words, if $path_{C_i}$ no longer holds, due to the additional checks on field a, then all path predicates $path_{C_j}$ can be dropped, because they cannot have any additional values for field a.

For the sample client predicates in Figure 5, $differentFrom[1][2][request] = TRUE$, because client predicate 1 is satisfied by an assignment $message.request = READ$, while predicate 2 is not. However, $differentFrom[1][2][address] = FALSE$, because the $message.address$ field of the $READ$ request is not satisfied by any other values than those that also satisfy the address of the *WRITE* request.

3.4 Local State

Distributed systems often keep local state across several rounds of messages and change their behavior based on the contents of the local state. For example, Paxos [19] uses three phases to achieve consensus. In each phase, Paxos

nodes accept different types of messages—the predicate P_S depends on the local state of the nodes. Thus, Achilles needs a mechanism to control the local state of the analyzed nodes. Achilles provides several alternatives to support local state.

Concrete Local State. The analysis phase of Achilles can be started at any point in the execution of a distributed system. Achilles is implemented on top of S2E [8], a symbolic execution platform that can run whole systems: operating systems, libraries and user applications. This enables the possibility of running the distributed system concretely up to some point, implicitly building up concrete local state.

This mode of operation is useful when developers are interested in the behavior of their implementation in a specific scenario. For example, in the case of basic Paxos, developers might be interested in what happens when a Paxos Acceptor has just entered the second phase, with proposed value 7. It should only validate *Accept* messages for value 7—any other message is a Trojan message.

Another example where Concrete Local State is useful is the Amazon S3 bug [1], described in §1. The Trojan message that caused the outage incorrectly reported a high failure rate in the system. The message was not necessarily Trojan in all possible scenarios—there may be scenarios where there are indeed a lot of failures in the system. However, the message was Trojan in the concrete scenario in which it occurred. By building concrete state in a deployment with few failures, Achilles could discover that no correct client node can report high failure rates, yet the servers accept such messages.

Constructed Symbolic Local State. As a generalization of the Concrete Local State mode, Achilles is able to pass symbolic messages between nodes of a distributed system, such that the nodes build up symbolic local state. The Constructed Symbolic Local State mode can capture entire sets of possible concrete execution scenarios. However, due to the additional symbolic variables and constraints, the Constructed Symbolic Local State mode can encounter difficulties with complex data structures, like any symbolic execution tool.

Consider again the example of Paxos. In order to be confident that there are no Trojan messages in the second phase of the protocol, developers need to re-use Achilles for every combination of concrete values in the local state. Using Concrete Local State, this entails re-running Paxos with different proposed values (1, 2, etc.) and then applying Achilles for each scenario. Alternatively, with Constructed Symbolic Local State, developers can run Paxos once, with a symbolic proposed value. Paxos nodes will store the symbolic value in their local state. Thus, Achilles can be applied a single time.

Over-approximate Symbolic Local State. The third mode of operation allows distributed system developers to place annotations that describe the local state. Developers can insert annotations in the code of the distributed system nodes, or insert them directly in the binary at runtime, using S2E plugins that monitor execution. Essentially, developers can manually define memory as symbolic local state and specify constraints on the stored values.

In the Paxos example, developers can annotate the functions that deal with local state in order to directly return symbolic values. This approach is useful when the implementation uses complex data structures to store local state, as it makes it possible for developers to reduce the burden on SMT solvers.

4. Soundness and Completeness

The purpose of Achilles is to discover Trojan messages, i.e., messages that are accepted by a server but cannot be generated by any correct client. Achilles can have both false positives and false negatives and is therefore not guaranteed to be sound or complete. False positives, however, can be kept under control by the operator of Achilles, as described below. We believe this makes Achilles a useful testing tool for distributed system developers, who can incorporate the messages discovered by Achilles in fault injection testing.

4.1 False Positives

We say that Achilles has false positives when there exists a message m that is falsely identified as Trojan. This can happen either when m is rejected by the server, or when m can be generated by a correct client.

Achilles relies on symbolic execution in order to extract predicates. Symbolic execution incrementally builds up an expression of an entire program by systematically exploring feasible execution paths. As long as symbolic execution does not completely explore all paths, the expression under-approximates the program. When the client is under-approximated, it might happen that a message m can only be generated on the execution paths that were not yet explored — this leads to false positives in Achilles.

In our experiments in §6, we did not encounter any false positives of Achilles. We bounded the maximum messages size in both the client and server, allowing symbolic execution to complete. We also restricted symbolic execution to only some fields of a message. Essentially, we restrict symbolic execution to only a subset of the input space, as has also been proposed by Person et al. [21].

Achilles generates concrete values for the Trojan messages it discovers, as well as symbolic expressions, allowing testers to inject the concrete messages in a real deployment (such as in live fire drills) and check the effect of the messages, weeding out harmless messages. While this concretization does not disambiguate between Trojan and non-Trojan messages, it helps discover any adverse effects of the suspected Trojan messages.

As future work, we imagine using the expressions that define Trojan messages to guide a new symbolic execution of the client node; this approach is similar in spirit to the abstraction refinement in CEGAR [9]. As shown by ESD [23] or Demand-driven symbolic execution [2], this focused sym-

bolic execution is significantly faster than "blind" exploration, and can help eliminate false positives in Achilles.

It is worth pointing out that Achilles does not also encounter false positives due to inaccuracy in Achilles' implementation of the *negate* operator. The implementation is strictly an under-approximation of the ideal *negate* operator. For each negated expression generated by the *negate*, we use the SMT solver to check if there is any common solution between the original expression and its negation; whenever there is such a solution, we discard the negated expression, eliminating any false positives that could have occurred.

4.2 False Negatives

We say that Achilles has false negatives when there exists a message m that is not reported as a Trojan message, although it is actually accepted by the server and cannot be generated by any correct client.

False negatives can occur when the symbolic execution of the server does not exhaust all execution paths. If a message m cannot be generated by any correct client, but is accepted by the server on an execution path not explored by Achilles, then m is an undiscovered Trojan message.

False negatives can also occur due to the fact that our implementation of the *negate* operator under-approximates the real negate. As described in the Design section (§3.2), we use two alternative approaches to compute the negation: one approach relies on quantifier support in the Z3 SMT solver, while the other approach relies on tweaked constraints that are solved using the STP SMT solver. Z3 uses heuristics to eliminate quantifiers, and may fail to determine satisfiability. When this happens, Achilles falls back to customized constraints sent to STP. Achilles' customized *negate* is under-approximate; for example $negate((\lambda = 2 * x) \wedge (x > 0))$ produces $((\lambda = 2 * x) \wedge (x \leq 0))$, even though -1 and 1 are also values for λ that do not satisfy the initial expression. In our evaluation (§6), though, we found that although under-approximate, Achilles' customized implementation of *negate* can find Trojan messages in real distributed systems.

5. Implementation

In this section, we describe some implementation details of our Achilles prototype. We built Achilles on top of the S2E symbolic execution platform [8]. S2E can run whole systems; this means that S2E allows running the implementation of distributed system nodes in their real environment, enabling Achilles to also discover Trojan messages that are due to third party system components, such as libraries. S2E is also highly extensible through a rich plugin API, simplifying the development of Achilles.

5.1 Capturing System Calls

Achilles uses the *LD_PRELOAD* mechanism in Linux to intercept calls to the standard *libc* library. Thus, Achilles can insert symbolic data automatically by overwriting calls that read inputs, replacing concrete values with symbolic data.

Achilles also intercepts calls to network operations, in order to automatically mark server execution paths as *accepting* or *rejecting*. By default, we consider that any execution path that sends a reply to the user is *accepting*, while any execution path that waits for new messages is *rejecting*. The assumption behind this is that servers have a single event loop to listen for messages; restarting the loop means that the previous message was processed. This can be trivially extended to handle other common error signaling mechanisms (e.g., 4xx status codes in HTTP). Of course, human operators can also provide additional domain knowledge by manually placing *accept* or *reject* markers in the system under test. For example, if the target protocol uses encryption, it makes sense for the human operator to place an *accept* marker before the encryption of the reply.

Achilles uses system call interception in order to implement the Constructed Symbolic Local State mode. In order to avoid unnecessary forking in device drivers, Achilles intercepts network operations that send symbolic data and reroutes them through custom channels (currently, nodes are deployed within the same S2E instance and messages are rerouted through shared memory).

5.2 Annotations

Developers can use annotations to speed up Achilles. The annotations can be inserted in the code of the distributed system, or can be injected in the system at runtime, using S2E's plugin support.

mark_accept is a server-side annotation that marks an execution path as *accepting*, triggering the check for any Trojan messages.

mark_reject is a server-side annotation that marks an execution path as *rejecting*.

function_start and *function_end* are used in order to over-approximate the behavior of a function. The operator can add constraints between the two markers, in order to impose constraints on return values.

drop_path is used in conjunction with the *function_* markers in order to impose constraints on the return values.

return_symbolic is used in conjunction with the *function_* markers in order to set the return value of a function.

make_symbolic makes a system variable symbolic; it can be used in order to mark local state as symbolic.

For example, a developer can over-approximate the $getPeerID()$ function from the client code in Figure 3, as shown in Figure 9. The over-approximation bypasses the code of the function completely and returns a new symbolic value constrained to the interval $[0, 10]$. In our evaluation (§6) we used annotations to bypass cryptography and authentication code and speed up predicate extraction.

Achilles supports a mask to "hide" certain message fields from the analysis. The symbolic execution of the server still branches on the hidden values; however, Achilles does not check for Trojan messages involving those fields. The mask increases the signal-to-noise ratio of Achilles by hiding un-

```
1 int getPeerID() {
2   function_start();
3   int toRet = makeSymbolic();
4   if ( toRet < 0 ) drop_path();
5   if ( toRet > 10 ) drop_path();
6   return_symbolic(toRet);
7   function_end();
8   ...   //actual code of getPeerID
```

Figure 9. An example of using Achilles annotations in order to over-approximate a function to return values $[0, 10]$

interesting results, and also makes the analysis faster, since it reduces the workload of the SMT solver (Achilles applies the mask before calling the SMT solver).

6. Evaluation

In this section, we evaluate the ability of Achilles to analyze an implementation of a distributed system and extract Trojan messages. After a description of our experimental setup (6.1), we answer the following questions:

- How accurate is Achilles at finding Trojan messages in real distributed systems (§ 6.2)?
- What is the impact of Trojan messages discovered by Achilles on real distributed systems(§ 6.3)?
- How does Achilles manage large symbolic expressions (§ 6.4)?

6.1 Setup

We ran experiments on a 16-core (dual Intel E5-2690) machine with 256 GB of RAM running Ubuntu Linux 11.04.

Achilles gathered the client and server predicates by running individual system nodes in the S2E [7] platform. The S2E virtual machine ran 32-bit Ubuntu Linux 10.04.

We applied Achilles to FSP 2.8.1b26 [12], a UDP-based file transfer protocol, and to the latest version of PBFT [6], a Byzantine-fault-tolerant replication system.

FSP is an implementation of a file transfer protocol. An FSP deployment consists of a server and several client utilities, which emulate well-known UNIX core utilities, such as *cp*, *mv*, *rm*, etc. The FSP implementation of these utilities parses command-line arguments (usually a target file path and a set of flags), tweaks the path to follow some protocol-specific rules (e.g., always start paths with '/'), and finally generates a corresponding command for the server. The server parses the command, performs the corresponding action on its local file system, and replies to the client.

An FSP command message has the following fields:

- *cmd* - 1-byte identifier of the requested action
- *sum* - 1-byte checksum
- *bb_key* - 2-byte message key
- *bb_seq* - 2-byte message sequence number
- *bb_len* - 2-byte length of file path
- *bb_pos* - 4-byte position of block in a file
- *buf* - arbitrary-length payload (file path + file data)

In our evaluation, we approximated the values of *sum*, *bb_key*, *bb_seq* and *bb_pos*: we added annotations in both the client and the server in order to bypass the checks on these fields, such that the client writes a predefined constant value and the server checks that value (in §7 we survey some approaches that can alleviate the problem of checksums, authentication or encryption). We focused our evaluation on how FSP parses file paths. We started the FSP clients with symbolic command line arguments (of fixed length) and inserted symbolic data in any system calls from the client. In the FSP server, we set *accept* markers at the point where it invokes system calls to make changes to its local file system, as requested by the command received from the client.

PBFT is a Byzantine fault-tolerant replication protocol implementation. PBFT clients send requests to a set of replicas. The replicas ensure total order among requests from all clients, and forward them to an upper layer application.

A PBFT client request has the following fields:

- *tag* - 2-byte identifier of the message type
- *extra* - 2-byte flags field (1 bit per flag)
- *size* - 4-byte length of message
- *od* - 16-byte message digest
- *replier* - 2-byte identifier of responsible replica
- *command_size* - 2-byte length of command
- *cid* - 2-byte client id
- *rid* - 2-byte request id
- *command* - arbitrary-length command payload
- *MAC* - list of message authenticators, signed with a private key for each replica

In our evaluation, we approximated the values of the digest and authenticator fields: we used annotations in both the client and server in the same way as for FSP, replacing the digest and authenticators with predefined constant values. In the PBFT replica, we also over-approximated local state: the replicas keep an internal data structure to track previous requests from a given client, which we over-approximate with unconstrained symbolic values. We started a PBFT client and generated a request with symbolic *extra*, *replier*, *rid*, *cid*, and *command*. We set a fixed length for the *command*, list of authenticators, and for the overall message. We considered a message to be accepted when the replica generates a *Pre_prepare* message for the client request, initiating the agreement protocol.

6.2 Accuracy of Achilles

The purpose of this experiment is to quantify the accuracy of Achilles at finding Trojan messages. We compute the overlap between the expression of Trojan messages computed by

Achilles and a known set of Trojan messages in the protocol under test. Ideally, the overlap should be perfect—the representation generated by Achilles should cover all known Trojan messages, and no known non-Trojan messages.

We evaluated the accuracy of Achilles on real Trojan messages in FSP. Achilles discovered that the FSP server accepts messages where the file path length is less than the length reported in the message header (more details in §6.3).

To enable symbolic execution to complete, we restricted the FSP clients and servers to only handle file paths with length less than 5. For this scenario, we can mathematically compute how many different types of Trojan messages exist: there is one Trojan message for reported length 1 (the file path is empty), two Trojan messages for reported length 2 (file path is empty or has length 1), three Trojan messages for reported length 3 and four Trojan messages for reported length 4. We analyze eight FSP clients that have a single file path as argument. Therefore, there are in total $(1+2+3+4)*8 = 80$ Trojan messages that differ in the reported length of the file path, request type or true length of the path.

The bounded path size enabled symbolic execution to run to completion. The total testing time for Achilles to find all Trojan messages was approximately 1 hour, which was split as follows:

- Gathering the client predicate took 3 minutes
- Preprocessing the client predicate took 15 minutes.
- Analyzing the server took 45 minutes.

Figure 10 shows the percentage of the known Trojan messages discovered by Achilles, as a function of server analysis time. Achilles produced the first Trojan message after 20 minutes of server analysis and discovered all Trojan messages in 43 minutes. Achilles did not produce any false positives. The figure shows how Achilles produces Trojan messages incrementally, as it analyzes the server; even if the analysis is interrupted before completion, Achilles produces valuable results.

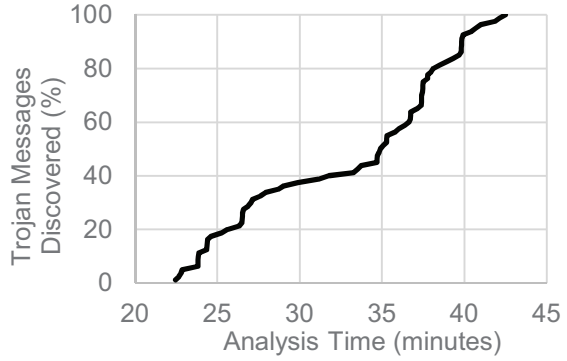

Figure 10. Percentage of real Trojan messages in FSP discovered by Achilles, as a function of time.

We compared Achilles to classic symbolic execution of a server node. We ran the FSP server in vanilla S2E, under the same conditions as Achilles (same bounds, annotations and approximation). Table 1 summarizes the results. Achilles discovers all Trojan messages and produces no false negatives. Classic symbolic execution also found all Trojan messages in 2 minutes, but with low signal-to-noise ratio. Trojan messages are "hidden" among valid messages, and it is left to the developer to sift among the results.

	Achilles	Classic symbolic exec.
True Positives	80	80
False Positives	0	7,520

Table 1. Results obtained by Achilles in 1 hour, compared to classic symbolic execution

As expected, classic symbolic execution of the server is faster than Achilles, since it performs fewer computations (it does not need to combine server and client constraints). However, classic symbolic execution cannot identify Trojan messages among other, valid messages. Even worse, in the case of FSP, all Trojan messages are on the same server execution paths as non-Trojan messages (as opposed to having execution paths that only contain Trojan messages and execution paths that only contain non-Trojan messages). This makes it difficult for the human operators to sift through results and discover Trojan messages—they need to investigate not only each execution path, but also each message on each execution path. This is a difficult task; as we also discuss in the Design section (§3.2), current SMT solvers are not designed to enumerate all values that satisfy a given constraint and are inefficient at doing so.

We ran the same experiment for PBFT. Achilles discovered a single type of Trojan message (more details in §6.3). Surprisingly, PBFT replicas make few checks on the data received from clients. They verify that request ids are recent and have not already been handled, verify that the client id is in a set of known clients and also check if the flags field marks the request as read-only.

Due to the simplicity of checks on the client request fields, Achilles completed the PBFT analysis in just a few seconds. The Trojan message discovered by Achilles appears on all execution paths in the server; however, just like for FSP, it is bundled on the same execution paths with valid messages. Thus, classic symbolic execution cannot easily identify the Trojan message among the valid messages, while Achilles identifies it accurately.

We also make a theoretical comparison between Achilles and a naive black-box fuzzing tool. There are other, more sophisticated fuzzers, which use various heuristics to improve performance. However, we are not aware of any fuzzer that optimizes towards anything similar to Trojan messages, so it would be difficult to obtain an unbiased comparison.

We first measured the maximum throughput that a fuzzing tool could achieve on FSP on our testbed: 75,000 tests per minute. Then, we compute the probability of discovering a Trojan message at random. In order to be fair, we only fuzz the same message fields that are analyzed by Achilles and by classic symbolic execution. There are 8 bytes relevant to the Trojan messages: *cmd*, *bb_len* and *buf*. There are 8 relevant possible values for the *command* field (corresponding to commands with a single file path argument). There are 4 relevant values for the *length* field. The FSP server only accepts printable characters in the file path (ASCII codes 33 to 126). Thus, there are $94 * 94 * 94 * 8 = 6.6$ million messages that exhibit a Trojan message for length 1 (first character in the message is '\0', second must be '\0' due to server checks, other three can be any ASCII character with code 33-126). In total, there are 66 million Trojan messages out of the $256^8 = 1.8 * 10^{19}$ possible bit combinations. This means that the expected number of Trojan messages found in 1 hour of fuzzing is 0.00001. Fuzzing also produces 4.5 million non-Trojan messages, which correspond to false positives.

Our experiments show that Achilles is significantly more efficient at discovering Trojan messages than fuzzing or classic symbolic execution. While classic symbolic can find all messages accepted by the server faster than Achilles, it cannot disambiguate between Trojan and non-Trojan messages. Black-box fuzzing is several orders of magnitude worse than symbolic execution.

6.3 The Impact of Trojan Messages

FSP: The Wildcard Character. Achilles discovered a bug in FSP related to the handling of the '*' wildcard character. In essence, FSP clients always expand the '*' wildcard before sending a command to the server. There is no possibility of escaping the '*' character, thus correct clients cannot send a command with '*' in a source file path to the server. The server, however, accepts any printable character, including '*' in the file paths it receives. This leads to an interesting behavior where it is possible to create files containing '*' on the server, but not possible to delete them directly.

In UNIX environments, shells use a simple form of pattern matching in order to expand file paths using wildcard characters (this simple pattern matching is referred to as globbing). For example, the command *'rm file*'* will delete all files with name prefixed by *'file'* from the current directory. The '*' character is a wildcard that can be expanded to any sequence of characters. The expansion of the wildcard is handled directly by the shell, before invoking the *rm* utility. Suppose the current directory has files *'file1'*, *'file2'* and *'file3'* and that the user executes the command *'rm file*'*. The shell will initially parse the command and expand *'rm file*'* into *'rm file1 file2 file3'*. Then the shell invokes *'rm'* with the three command line arguments.

The FSP implementation emulates the regular UNIX globbing behavior. When parsing command-line arguments, the FSP client will first try to expand any *source* file path containing a wildcard. Unlike most shells, however, the FSP globbing does not allow the user to escape any wildcard character. *Destination* file paths, however, are not globbed; for example the *'mv file1* file2*'* will rename any file that matches the pattern *'file1*'* to the literal string *'file2*'*. The FSP server handles wildcards like any regular character.

Trojan messages in FSP can lead to an interesting scenario, where it is possible to create a file called *'file*'* on the server, but it is then difficult to actually remove the respective file. A file called *'file*'* can be created by a user of FSP (e.g., *'mv file file*'*), by a malicious third party that has access to the server, or even by a bit flip that appears on the client during a command execution (a single bit flip can convert the ASCII 'j' character into '*').

Once the file named *'file*'* appears on the server, it is difficult to remove. Calling *'rm file*'* would remove *'file*'*, but would also remove any other file prefixed by *'file'*, including a potentially valuable *'fileWithAllMyBankAccounts'*. Similarly, *'mv file* fileToDelete'* would rename all files prefixed by *'file'* to *'fileToDelete'*, removing all but one of the original files. Calling *'rm file*'* would delete all files prefixed by *'file\'*, since there is no escape character in the FSP globbing.

It is interesting to point out that this bug discovered using Achilles is a semantic bug, which makes it difficult to detect automatically using other approaches. While there are techniques to detect buffer overflows or divisions by zero automatically, the FSP bug cannot be found unless the developers write a complex (and potentially buggy) specification of correct behavior. This is one of the main benefits of finding Trojan messages.

FSP: Mismatched String Lengths. Achilles also discovered that the FSP server does not check whether the file path lengths it receives in commands match the actual lengths. It accepts messages that have the actual file path length smaller than the value in the message's *length* field (i.e., there is a *'\0'* character in the file path). The bug allows malicious users to send an additional arbitrary payload to the server.

PBFT: The MAC Attack. Achilles discovered that PBFT replicas (servers) accept client requests without checking their authentication code. This leads to a known vulnerability of the protocol, known as the MAC attack [10].

Clients can send messages with incorrect authenticators. The first replica to receive the client request does not verify any of the authenticators. If forwards the message to other replicas, which discover the incorrect authenticator, but cannot know whether the original client or the first replica have corrupted the message. In order to guarantee progress, they initiate an expensive recovery protocol, which impacts performance. This allows incorrect nodes to have a significant impact on the performance of the system. A node whose private key was corrupted due to a memory error will always produce incorrect MAC authenticators and will trigger recovery. Alternatively, a malicious client can also corrupt

its own messages in order to trigger the expensive recovery mechanism and slow down the system, affecting the service of other, correct clients.

This vulnerability is an interesting example of the subtle effects that Trojan messages can have on a system.

6.4 Handling Large Client Predicates

In this section, we analyze internal details of Achilles. We look at how the optimizations described in §3.3 enable Achilles to handle large client predicates.

Figure 11 shows the number of client path predicates Achilles keeps for each execution path analyzed during the FSP server exploration, as a function of the length of the path. The figure shows results for the paths as they are incrementally generated—many of the points in the figure represent incomplete paths. A point at length 7 with 1,000 matching predicates means that there exists an execution path in the server which encountered 6 branching points that depend on symbolic data, and can be triggered by messages in 1,000 different path predicates. Recall from §3.2 that Achilles uses these path predicates to search for any existing Trojan messages—the more path predicates in a state, the more complex the check.

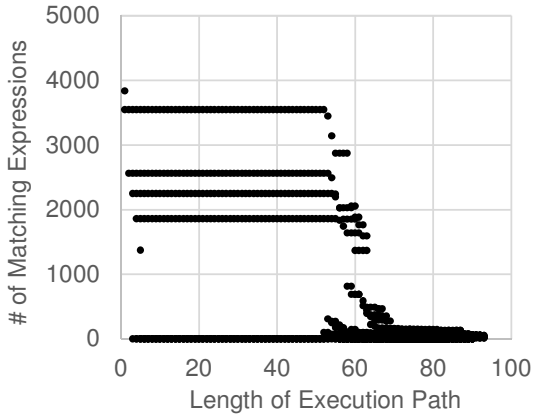

Figure 11. Number of client path predicates that can trigger each execution path in the FSP server, as a function of the length of the path.

The number of path predicates decreases as the length of paths increases—longer execution paths become more and more specialized, and can be triggered by a smaller group of messages. This makes the check for Trojan messages easier, as Achilles makes fewer calls to the solver.

To quantify the effect of optimizations, we compared Achilles to a non-optimized implementation of symbolic constraint differencing. We ran unmodified S2E on both the FSP client and server, and then computed Trojan messages a posteriori. The total execution time of the non-optimized implementation was 2 hours and 15 minutes, compared to the 1 hour and 3 minutes required by Achilles.

7. Related Work

To the extent of our knowledge, there is no previous work that directly targets Trojan messages.

Caballero and Song [5] use dynamic symbolic execution in order to reverse engineer the message format and semantics of an undocumented protocol. In this paper, we assume that Achilles is used by the distributed system developers, in order to detect corner cases in their implementation. Thus, the purpose is not to produce a documentation of the protocol, but concrete corner cases that trigger incorrect behavior.

Caballero et al. [4] used symbolic execution to find bugs in malware. They faced the challenge of handling encryption and obfuscation techniques used in the malware, and developed a technique that only solves subsets of path constraints and then restitches the individual solutions to obtain a complete input. Achilles is more targeted, focusing on a specific type of bugs that cannot be found with other techniques. The work by Caballero et al. can be integrated with Achilles to enable the predicate extraction to handle complex operations without human operator support.

Achilles analyzes distributed system nodes in isolation and then composes the results of the analyses. This is similar to the Compositional Dynamic Test Generation proposed by Patrice Godefroid [14]. The predicate P_C of a client node can be seen as a functional summary, which is then integrated in the analysis of a server node. In the case of Achilles, though, we do not use the summary directly, as we are actually interested in the negation of the summary.

The idea of analyzing distributed nodes in isolation was also explored by Guerraoui and Yabandeh [15]. In their work, the authors proposed model checking individual nodes in isolation, separated from the complexity of network message interleaving. In Achilles, we do not analyze nodes in isolation for performance reasons, but in order to catch specific bugs.

Maurer and Brumley [20] use tandem symbolic execution of the pre- and post-patch versions of a given program. Their approach compares the behaviors of the two versions and checks that the patch only affects the buggy executions it was meant to fix. A similar approach is taken by Person et al. [21] to characterize code changes. This is similar in spirit to the way Achilles compares the predicate of the client and the predicate of the server in order to check for Trojan messages.

In Achilles, we currently ignore the order in which messages are received, focusing strictly on the effect of message contents. Achilles could be enhanced by techniques such as MODIST [22] to also consider alternative orderings. Alternatively, the distributed system under test can be deployed with DDOS [16], which ensures deterministic behavior.

Cloud9 [3] is a symbolic execution tool that can analyze distributed systems. Similar to Achilles' Constructed Local State, Cloud9 reroutes symbolic messages through alternative channels.

8. Conclusion

Trojan messages represent a type of bug specific to distributed systems—an artifact of different implementations of the same protocol specification. Trojan messages occur in real distributed systems and can cause unpredictable behavior: from low performance in the case of Amazon S3 and PBFT, to loss of data in the case of FSP.

We propose Achilles, a system designed to efficiently discover Trojan messages in distributed system implementations. Achilles uses dynamic white-box analysis in order to extract the grammar of messages accepted by servers, and the grammar of messages that can be generated by clients. Achilles then computes the set of Trojan messages as the difference between the two. We described the basic principles behind Achilles, as well as the optimizations that make it practical.

We applied Achilles to real distributed systems, discovering bugs that are difficult to find with alternative techniques. Our evaluation shows that Achilles is efficient at discovering Trojan messages; it discovered all instances of a Trojan message in FSP, while producing no false positives.

Acknowledgments

We are indebted to the anonymous reviewers, and to Jonas Wagner and Volodymyr Kuznetsov for their insightful feedback and generous help in improving this paper. We are grateful to the SNF (Sinergia Project No. CRSII2_136318/1) and to IBM for supporting our work.

References

[1] Amazon s3 availability event: July 20, 2008. Retrieved on 2013-07-20. http://status.aws.amazon.com/s3-20080720.html.

[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[3] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[4] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input Generation via Decomposition and Re-Stitching: Finding Bugs in Malware. In *Proceedings of the 17th ACM Conference on Computer and Communication Security*, Chicago, IL, October 2010.

[5] J. Caballero and D. Song. Automatic Protocol Reverse-Engineering: Message Format Extraction and Field Semantics. *Computer Networks*, 57(2):451 – 474, 2013.

[6] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Symp. on Operating Sys. Design and Implem.*, 1999.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[8] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems*, 30(1), 2012. Special issue: Best papers of ASPLOS.

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5):752–794, 2003.

[10] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Symp. on Networked Systems Design and Implem.*, 2009.

[11] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[12] File service protocol. Retrieved on 24-Jul-2013. http://fsp.sourceforge.net/.

[13] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Intl. Conf. on Computer Aided Verification*, 2007.

[14] P. Godefroid. Compositional dynamic test generation. In *Symp. on Principles of Programming Languages*, 2007.

[15] R. Guerraoui and M. Yabandeh. Model checking a networked system without the network. In *Symp. on Networked Systems Design and Implem.*, 2011.

[16] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble. Ddos: taming nondeterminism in distributed systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[17] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[18] K. Krishnan. Weathering the unexpected. *Queue*, 10(9):30:30–30:37.

[19] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[20] M. Maurer and D. Brumley. Tachyon: tandem execution for efficient live patch testing. In *USENIX Security Symp.*, 2012.

[21] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Symp. on the Foundations of Software Eng.*, 2008.

[22] 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 *Symp. on Networked Systems Design and Implem.*, 2009.

[23] C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *ACM EuroSys European Conf. on Computer Systems*, 2010.