Analyzing Protocol Implementations for Interoperability

Luis Pedrosa[†] *Ari Fogel*[‡] *Nupur Kothari*[*]
Ramesh Govindan[†] *Ratul Mahajan*[*] *Todd Millstein*[‡]

University of Southern California[†] University of California, Los Angeles[‡] Microsoft[*]

Abstract— We propose PIC, a tool that helps developers search for non-interoperabilities in protocol implementations. We formulate this problem using intersection of the sets of messages that one protocol participant can send but another will reject as non-compliant. PIC leverages symbolic execution to characterize these sets and uses two novel techniques to scale to real-world implementations. First, it uses *joint symbolic execution*, in which receiver-side program analysis is constrained based on sender-side constraints, dramatically reducing the number of execution paths to consider. Second, it incorporates a search strategy that steers symbolic execution toward likely non-interoperabilities. We show that PIC is able to find multiple previously unknown non-interoperabilities in large and mature implementations of the SIP and SPDY (v2 through v3.1) protocols, some of which have since been fixed by the respective developers.

1 Introduction

Nodes in distributed systems communicate using protocols such as TCP, HTTP, SPDY, and SIP. For robust operation, it is critical that the implementations of these protocols interoperate effectively, i.e., they should be able to correctly parse and interpret messages sent by each other.

Protocol interoperability is difficult to engineer and ensure because the protocols are complex, and often contain many mandatory and optional features. Moreover, protocol standards documents can be imprecise or ambiguous, leading different developers to make different implementation choices about protocol interactions and semantics of message header formats and values. Thus, in practice, it is neither possible nor sufficient to simply "certify" that an implementation adheres to a standard. Instead, pairs of implementations must be pitted against one another to test for interoperability.

Today, protocol developers spend significant manual effort [17, 18, 30, 31, 36] in testing interoperability. They identify test inputs that can test specific features of the protocol (e.g., options negotiation). But, because the space of test inputs is large, such manual testing is often incomplete. As a result, interoperability issues continue to frustrate developers, as well as users, even years after protocols have been fully deployed [35, 39].

Our goal is to automate the search for test inputs that trigger non-interoperabilities between two implementations. In this paper, we formulate the problem of non-interoperability of protocol implementations and develop a tool called PIC (Protocol Interoperability Checker)[1] to identify non-interoperabilities. To our knowledge, PIC is the first tool that addresses this problem.

We say a message *m* exhibits a non-interoperability when certain inputs cause message *m* to be sent by one protocol participant but rejected as non-compliant by the receiver. Message *m* need not be the first message in a protocol interaction; PIC can be used to analyze the *i*-th message (e.g., a data transfer message sent after the connection establishment handshake). Conceptually, PIC characterizes the set of messages that a sender-side implementation can generate, and the associated inputs that generate those messages. Similarly, it characterizes the set of messages a receiver-side implementation considers non-compliant. Messages in the intersection exhibit non-interoperabilities. Thus, unlike manual testing where developers specify test inputs that *may* trigger non-interoperabilities, PIC uses symbolic execution [22] to *automatically* derive test inputs by analyzing protocol code.

A key challenge with this approach is that symbolic execution of protocol code, which tends to be low-level, is invariably incomplete and imprecise, and is hard to scale to real-world implementations. As originally designed, symbolic execution explores as many paths as possible, while we are interested only in exploring paths that produce elements of the sets described above. While we cannot overcome symbolic execution's inherent incompleteness, we address the scaling challenge using two novel techniques.

First, we introduce *joint symbolic execution*, in which the receiver-side symbolic execution is directed based on the sender-side analysis results, dramatically reducing the number of execution paths to consider. Specifically, the only receiver-side paths considered are those that are compatible with messages that can be sent from the sender. As we discuss later, in the absence of joint symbolic execution, an independent analysis may not even be feasible for some protocol implementations. We believe that joint symbolic execution is of general interest for scalable protocol analysis beyond the use case of searching for non-interoperabilities.

[1] PIC is available at `https://github.com/USC-NSL/SPA`

Second, PIC employs new search techniques that direct the analysis toward execution paths more likely to add new messages to the sets being characterized, and therefore to identify interoperability errors. Our experiments show that PIC's search strategies help find 25× more instances of non-interoperabilities than existing strategies within a given time.

We apply PIC to four mature implementations of two qualitatively different protocols: SIP [40] and SPDY [1]. For each protocol, we find thousands of non-interoperable messages spread across different features of the protocol. To understand the causes that lead to these, we group messages that arise from the same underlying problem (e.g., a failure to correctly validate API inputs). For SIP we find 9 distinct causes, and for SPDY we find 13 distinct causes, which fall into several high-level classes: liberal senders, conservative receivers, ambiguous specifications, specifications with optional features and so forth. We reported these to the developers, and several have been fixed.

2 Problem and Background

Protocol implementations cannot easily be validated for compliance against protocol specifications. Such specifications are typically expressed in a natural language and have inherent ambiguities, so ensuring compliance would first require formalizing the specifications in some way, a process which itself can be error prone and open to multiple interpretations. The ambiguity of protocol standards is well-documented, for instance, for BGP [26].

Therefore, instead of compliance to a specification, we focus on checking interoperability of implementations. Specifically, two implementations are considered non-interoperable when there exists at least one protocol message on which they disagree. The disagreement could be about whether the message is protocol compliant (*e.g.,*, the values of header fields are formatted correctly, or take the range of values expected by the receiver) or about what the message means. The work in this paper focuses on the former notion of disagreement. The latter notion of *semantic* interoperability poses additional challenges in extracting meaning and intention from protocol messages, which we leave to future work.

Protocol interactions. To more precisely define our notion of interoperability, consider client-server or peer-to-peer communication between two implementations. Such communication is usually triggered by a call to a protocol API function that implements a specific functionality. This API function initiates a *protocol interaction*: a sequence of message exchanges that perform the desired functionality. For example, a protocol interaction to initiate a call in SIP begins with a client *request* message, followed by a server *response* message. TCP involves several types of protocol interactions: separate API functions initiate connection setup, data transfer, and teardown. In general, a protocol interaction can involve a sequence of messages $m_1, \ldots, m_n$, some going from client to server (or one peer to another), and some the other way. Each message exchange may update the sender or receiver *state*, which consists of the values assigned to variables in the protocol implementation.

Now, consider a message m_i in a protocol interaction. The sender prepares m_i based on the contents of m_{i-1} and on its current state; the first message m_1 is prepared after processing the arguments passed to the API function that initiated the current interaction. At the receiver, m_i is first validated for protocol-compliance. If m_i does not pass these checks, it may be discarded (or an error message sent in response). If it does, subsequent steps of the interaction are invoked.

On the sender side, let S_i denote the set of all possible i-th messages that can be generated (i.e., considered protocol-compliant) given the sender's state after the first $i-1$ messages in the protocol interaction. On the receiver side, let R_i denote the set of all possible i-th messages that the receiver would consider protocol-compliant given its state after the first $i-1$ messages.

Defining non-interoperability. Let $S_i - R_i$ denote the set of messages that are in S_i but not in R_i. Then, the sender and receiver are said to exhibit a non-interoperability if $S_i - R_i \neq \varnothing$, that is, there are messages that the sender can send but the receiver will not deem compliant.

If a sender and receiver are non-interoperable, then we know that the two entities implement the specification inconsistently. However, we cannot directly determine which entity deviates from the protocol specification: non-interoperability can occur either because the sender is too liberal in interpreting the specification, the receiver is too conservative, or *both*.

An example. Figures 1 and 2 list the code for NetCalc, a simple networked calculator protocol that we use to illustrate our approach later. (Functions with `pic_*` names are explained in §3.) At the client, the `Compute()` API function initiates an interaction with the server, sending a message containing an operator and two operands. At the server, the `handleMessage()` function processes the received message. This example has two non-interoperabilities: the client incorrectly expects the server to implement multiplication, and the client does not test for division-by-zero while the server does. So, the set $S_i - R_i$ consists of all messages sent by the client with the multiply operator or messages with zero divisor.

Interoperability testing today. The current practice for testing interoperability of protocol implementations is largely manual and *ad hoc*. Developers of different implementations participate in physical or virtual interoperability events. They test how well two implementations

```
1  Compute(char operator[8], int32 operand1, int32 operand2) {
2    // buffer, buffer size, name
3    pic_api_input(operator, 8, "operator");
4    pic_api_input(operand1, 4, "operand1");
5    pic_api_input(operand2, 4, "operand2");
6
7    byte[] message = new byte[9];
8    if (operator == "plus")
9      message[0] = 0;
10   else if (operator == "minus")
11     message[0] = 1;
12   else if (operator == "divide")
13     message[0] = 2;
14   else if (operator == "multiply")
15     message[0] = 3;
16   else
17     throw exception;
18   message[1..4] = operand1;
19   message[5..8] = operand2;
20
21   // buffer, length, buffer size, name
22   pic_msg_output(message, 9, 9, "message");
23   sendMessage(message);
24 }
25
26 testHarness() {
27   pic_api_entry();
28   char o[8];
29   Compute( o, 0, 0 ); // Dummy arguments
30 }
```

Figure 1: *NetCalc client. The pseudo-code parses API inputs and populates a message before sending it. PIC annotations declare API inputs (l. 3-5), and message outputs (l. 22). A simple test harness initiates a protocol interaction (l. 26-30).*

```
1  handleMessage(byte[] query) {
2    // buffer, buffer size, name
3    pic_msg_input(query, 9, "message");
4
5    int32 operand1 = query[1..4];
6    int32 operand2 = query[5..8];
7    switch (query[0]) {
8      case 0:
9        print(operand1 + operand2);
10       break;
11     case 1:
12       print(operand1 - operand2);
13       break;
14     case 2:
15       if (operand2 == 0)
16         throw exception;
17       print(operand1 / operand2);
18       break;
19     default:
20       throw exception;
21   }
22   pic_valid_path();
23 }
24
25 void testHarness() {
26   pic_msg_handler_entry();
27   handleMessage(new byte[9]);
28 }
```

Figure 2: *NetCalc server. The pseudo-code parses and validates the incoming message and acts on it. PIC annotations are used to declare the message input (l. 3) and the point in the code where the message has been considered valid (l. 22). Invalid messages generate an exception and are detected by not reaching the validity assertion. A simple test harness allocates a message and launches the message handler (l. 25-28).*

interoperate in order to uncover undesirable interactions resulting from code bugs or ambiguities in the standard.

During testing, developers specify and execute a series of interoperability tests on each pair of participating implementations. This is done by configuring those implementations to communicate with each other in a specific manner, by selecting a network topology for the test, and (optionally) by injecting failures or packet losses. Finally, the participants document testing results in event reports [17, 18, 30, 31, 36].

There are two conceptually different components to specifying an interoperability test: which protocol interaction (e.g., connection setup and termination, options negotiation, data transfer, and control commands) to test, and what specific test inputs to use. Specifying the protocol interaction is conceptually straightforward, since there is usually a small number of such features. However, specifying good test inputs (e.g., the call parameters in a SIP connection setup) is difficult. The space of potential protocol inputs can be very large (e.g., all possible URLs). To detect a non-interoperability, the specified test inputs must generate a message in $S_i - R_i$. Today the developer gets essentially no help in this task, resorting to a random search, perhaps guided by intuition, experience, and an understanding of potential ambiguities in the specification. Once a non-interoperability is found, developers converge on a mutually consistent reading of the specification in order to fix it.

Further, given the size of the search space, developers are able to consider only a very small fraction of the possible inputs; as one of the interoperability reports acknowledges, "the test parameters were limited" [18]. As a result, many corner cases go undiscovered during testing and are discovered *after* implementations are deployed in production [35, 39].

This paper explores methods to *derive* test inputs that generate messages in $S_i - R_i$ using program analysis, and instantiates these methods in a tool called PIC. Program analysis simultaneously removes a large burden on developers and improves the effectiveness of interoperability testing via targeted search and increased coverage.

3 PIC Design and Implementation

In this section, we first describe PIC's high-level approach and then detail its design.

3.1 PIC Approach

Our goal is to develop methods that, with low developer effort, can uncover non-interoperabilities in real-world protocol implementations. We seek methods that are *independent of a specific protocol implementation or type* and can therefore be applied to a wide range of protocols.

Symbolic execution. We use *symbolic execution* [22], a program analysis technique that simulates the execution of code using a symbolic value $\sigma_{\mathtt{x}}$ to represent the value of each variable `x`. As the symbolic executor runs, it updates the symbolic store that maintains information about program variables. For example, after the assignment `y = 2*x` the symbolic executor does not know the exact value of `y` but has learned that $\sigma_{\mathtt{y}} = 2\sigma_{\mathtt{x}}$. At branches, symbolic execution uses a constraint solver to determine the value of the guard expression, given the information in the symbolic store. The symbolic executor then only explores branches when the corresponding boolean guard is satisfiable. For example, it will explore both then and else branches of an if-then-else statement if the condition can be either true or false given the symbolic state of the system. In this way, a tree of possible

program execution paths is produced. Each path is summarized by a *path constraint* that is the conjunction of branch choices made to go down that path.

Key insight. In a protocol implementation, the set S_i of messages that can be sent and the set R_i of messages that would be accepted as the i-th message in a protocol interaction *can be succinctly represented by the symbolic values of the message header fields* generated by the sender and accepted by the receiver, when they are both symbolically executed. Thus, $S_i - R_i$ can be computed by determining if there exist concrete values for the message header fields that would match the symbolic constraints on the sender, but not on the receiver (and vice versa).

In theory, we can use an existing symbolic execution tool to compute the sets S_i and R_i, in order to then produce the set $S_i - R_i$ of non-interoperabilities. In practice, however, we face two difficulties. First, symbolic execution is invariably incomplete, since real code can have a large (often unbounded) number of possible executions due to loops, recursion, etc. Thus, what we are bound to get are subsets $S_i' \subset S_i$ and $R_i' \subset R_i$. Using the difference of these subsets $S_i' - R_i'$ to determine whether $S_i - R_i \neq \varnothing$ will lead to many false positives, since many relevant elements of R_i may be missing from R_i'.

To address this limitation, we recast our analysis to ask a question that can be answered precisely even with incomplete sets. Instead of trying to compute R_i, we compute $\neg R_i$, the set of messages that the receiver rejects. Given the limitations of symbolic execution, we will actually obtain some set $\hat{R}_i$ such that $\hat{R}_i \subset \neg R_i$. This results in computing $S_i' \cap \hat{R}_i \subset S_i - R_i$, by which non-interoperabilities are found when $S_i' \cap \hat{R}_i \neq \varnothing$, i.e., there are messages that are generated by the sender but rejected by the receiver. This formulation of interoperability is mathematically identical to the original one when the sets are complete but trades false positives for false negatives in the presence of partial information. This is more useful, since any message that belongs to both S_i' and $\hat{R}_i$ is in fact non-interoperable but does, however, limit PIC's ability to certify interoperability as $S_i' \cap \hat{R}_i = \varnothing \not\Rightarrow S_i - R_i = \varnothing$.

The second difficulty is that, as originally designed, symbolic execution attempts to cover as many paths as possible, without regard to where they lead. Given the already difficult task of scaling symbolic execution to protocol code, as well as the inherent incompleteness described above, such a blind search is less likely to produce useful results than one directed toward likely non-interoperabilities. Later in this section, we describe our techniques to address this difficulty.

3.2 PIC Architecture

The PIC workflow has four stages, shown in Figure 3. Its input is the intermediate code representation generated from annotated source (*i.e.,* LLVM [2] bitcode).

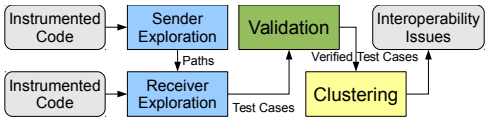

Figure 3: *PIC Analysis Work-Flow*

Annotation	Description
pic_api_entry()	Marks calling function as test harness for API handler.
pic_message_handler_entry()	Marks calling function as test harness for message handler.
pic_api_input(var, size, name)	Annotates buffer as API input.
pic_msg_output(var, length, bufSize, name)	Annotates buffer as message output.
pic_msg_input(var, size, name)	Annotates incoming message buffer of specified size.
pic_msg_input_size(var, name)	Annotates variable specifying size.
pic_valid_path()	Marks code point at which message is considered valid.

Table 1: *PIC Annotations*

Analysis annotations. As with today's interoperability tests, a developer using PIC has to specify (a) the protocol interaction to test for interoperability, (b) what constitutes a non-interoperability, and (c) the API input parameters. These are specified using code annotations (Table 1). In a crucial departure from today's practice, the developer does *not* need to specify concrete values for the test inputs: rather, PIC derives test inputs that trigger non-interoperabilities as described below.

The `pic_api_entry()` and `pic_msg_handler_entry()` annotations identify test harnesses at the client (lines 26-30, Figure 1) and server (lines 25-28, Figure 2), respectively. The `pic_api_input()` annotation (lines 3-5, Figure 1), specifies which inputs are of interest; other inputs are bound to a single concrete value during analysis (limiting its scope to trade completeness for complexity). In Figure 1, these annotations cover all arguments to the client API, but in practice, some of them can be omitted, focusing the analysis only on the specified inputs. Finally, two annotations convey the semantics of non-interoperability. `pic_msg_output()` on the client indicates the codepoint at which a message is transmitted (line 22 in Figure 1). `pic_valid_path()` on the server indicates the codepoint at which a message is considered protocol-compliant (line 22, Figure 2).

Symbolic execution of sender and receiver. The annotated sources are compiled into LLVM bitcode and fed to PIC. The first stage of PIC uses the sender-side annotations to analyze the sender code and generates *paths*, defined by a path constraint and the symbolic value of the resulting message. The second stage *uses these paths*, in a technique we call *joint symbolic execution*, to perform a similar analysis of the receiver, using receiver-side annotations. This results in a set of path constraints that represent non-interoperabilities, which are subsequently passed to a *constraint solver*. The solver determines a satisfying *model*: an assignment of concrete values to symbols that satisfies the formula. As such, the output is a set of concrete instances of non-interoperabilities.

The first two stages use the KLEE [9] symbolic execution engine, modified to use a novel guided search strategy (described below). Being based on KLEE and LLVM, PIC can analyze any language that can be translated to LLVM, which currently includes many popular languages such as C, C++, Objective-C, and C#.

Validation using concrete execution. The third stage removes false positives from the output of the second stage. False positives can arise because the first two phases may produce path constraints that do not in fact represent feasible paths to the target program points. This happens due to the conservative nature of symbolic execution in the face of code that is either not available (e.g., system calls and external libraries) or that cannot be analyzed precisely (e.g., complex heap manipulations).

Therefore, we concretely execute the protocol code, to rule out infeasible paths. The annotations used to declare API inputs are now used to inject concrete input values directly into the running program. At the receiver, whether a validity annotation is reached is now used to confirm or refute the non-interoperability. While a test harness is used to exercise the protocol API during the symbolic execution stage, in the validation stage the application may be used in a setting closer to an actual deployment, including the use of more elaborate network topologies, if needed. The validation stage scales well in our experience: as we show later, it is feasible to validate every produced concrete non-interoperability in our evaluation (some generate 70,000+ non-interoperabilities). n
Clustering. To help developers interpret the results, the final stage clusters the potentially many non-interoperability instances produced post-validation. PIC provides an extensible technique for this purpose.

3.3 Joint Symbolic Execution

Independent symbolic execution, in which S'_i and $\hat{R}_i$ are independently computed, can miss many interoperability bugs, for two reasons. First, the receiver-side analysis will waste a lot of time analyzing messages that the sender would never send, for example because it performs its own validation. Second, without any coordination, the sender and receiver are likely to explore different subsets of the space of possible messages, and by definition interoperability bugs will only surface in the intersection of these subsets.

PIC's joint symbolic execution modifies the receiver-side symbolic execution as follows. Initially, symbolic execution proceeds normally, up until the point where a message is received. The symbolic execution state is then forked multiple ways, one for each sender path. The path constraint on each of these forked states is then manipulated, AND-ing the path constraints from the respective sender path as well as *connecting constraints* that bind the sent message to the received message. Binding is done by declaring a byte-for-byte equality of the received message buffer to the symbolic value established for the message buffer on the sender. Symbolic exploration then proceeds as usual. By construction then, the output of the receiver-side analysis is a set of path constraints that characterize non-interoperabilities (i.e., $S'_i \cap \hat{R}_i$).

Joint symbolic execution has the effect of driving receiver-side symbolic execution only along paths consistent with messages the sender can send. This technique solves the two problems of independent symbolic execution described above. Any non-compliant message that the sender would never send is not explored by the receiver. More importantly, any symbolic message from the sender that represents an interoperability bug will *definitely* be explored by the receiver, without needing to be independently discovered. For example, in NetCalc, if the symbolic executor has only enough resources to produce a path for one of the two non-interoperabilities on each side, independent symbolic execution could produce different ones on each side, with the result that neither interoperability is detected (since the intersection would be empty). For these reasons, as we show in §4, joint symbolic execution is significantly more effective than independent symbolic execution.

3.4 Guiding Symbolic Execution, To and Fro

The original goal of symbolic execution is to explore as many code paths as possible for the purpose of program testing. Real-world protocol implementations have a huge number of paths, and a direct application of symbolic execution can be extremely inefficient.

In our setting, we require a form of guided symbolic execution that preferentially explores paths that satisfy certain properties of interest. Specifically, interoperability testing requires two different kinds of guidance. On the sender side, we need to explore paths that reach one of the few program points where a message is sent, which we call *convergent exploration*. This goal is similar to the notion of *directed symbolic execution* described by Ma *et al.* [28]. On the receiver, we need to explore paths that *avoid* a few points where the message is considered valid, which we call *dispersive exploration*. To our knowledge, there is no prior work on achieving this goal.[2]

Convergent and dispersive exploration. To achieve both convergent and dispersive exploration in a common framework, we reduce the problem of directed symbolic execution to a specialized instance of graph search. Indeed, choosing which paths to further explore is analogous to picking which node to visit next in a graph traver-

[2] One could avoid dispersive exploration by annotating *invalidity* assertions at the receiver and then doing convergent exploration. But that greatly increases the annotation burden, thereby increasing the possibility of erroneous annotations. One of the SPDY implementations we explore requires 2 validity assertions but 28 invalidity assertions.

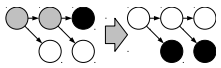

Figure 4: ***Dispersive symbolic execution preprocessing.*** *The black dispersive target on the left is converted to two convergent ones on the right using a reverse control-flow analysis (gray).*

sal. Convergent exploration is a matter of using search strategies for efficiently reaching a target set of program points. This translates into computing a *distance function* that determines an ordering of paths to explore. Different distance functions embody distinct search strategies.

Dispersive exploration, on the other hand, is not so straightforward. A potential approach is to negate the distance function, but that merely guides exploration away from undesirable points. Absent further guidance, this could easily get stuck in endless loops going nowhere. Instead, we conduct an initial pass on the CFG (control flow graph) to map dispersive targets to be avoided into convergent targets to be reached (Figure 4), in effect deriving invalidity assertions from the user specified validity ones. This preprocessing stage starts with a reverse control-flow analysis, finding all points that can reach the dispersive target. Any remaining points cannot reach the dispersive target and are subsequently designated as targets for a convergent exploration.

Fast search. By default, symbolic execution engines use depth-first-search (DFS), which is memory-efficient, but is unaware of the targets. DFS makes early branching decisions, leading to a point deep in the control-flow graph. Once there, it exhaustively explores nearby branches, before backtracking. DFS can waste a lot of time in this localized exploration, or if it finds a non-interoperability, it will find many instances or small variants of the same underlying non-interoperability before discovering a qualitatively different non-interoperability (§4.5).

Ma *et al.* [28] propose three new strategies: call-chain backward symbolic execution (CCBSE), shortest-distance symbolic execution (SDSE), and mixed-chain backward symbolic execution (Mix-CBSE), a hybrid of the first two. CCBSE works backward from the target point, performing (forward) symbolic execution from the nearest enclosing function, and repeats this process backwards until reaching the entry point. This goal-directed approach does not work well when there are a large number of possible targets, as is the case with dispersive exploration, since it requires managing and prioritizing among the several independent analyses for each target.

SDSE, which is similar to the technique in ESD [44], is analogous to greedy best-first search (GreedyBestFS), based on a control-flow distance metric. Unlike CCBSE, it can naturally accommodate multiple targets. But we found that, because of its greediness, it sometimes suffers from stubbornly sticking to potentially bad early branching decisions. This can lead to local minima. An example is the case where one branch of a conditional has a shorter control-flow distance to the target than the other, but may require significantly more symbolic execution (e.g., unraveling loops).

In PIC, we therefore use a strategy based on the A* heuristic search algorithm [41]. A* is a variant of best-first-search that also *considers the cost of the path traversed so far*. This approach allows the search to quickly exit local minima since local exploration increases the path cost, making other paths more attractive. Further, since the control-flow distance metric can be just as easily calculated for many points as for a few, this heuristic permits both convergent and dispersive exploration.

Basic distance heuristic. A good distance heuristic is key to the efficient use of A*. A* does not allow overestimating and significantly under-estimating approximates a breadth-first search, potentially leading to state explosion. The basic distance heuristic that we use is based on a work-list based inter-procedural analysis on the CFG (done before symbolic execution) [12, 37, 38]. In the CFG, nodes represent program points and directed edges connect each node to its possible control-flow successors. Some edges connect two program points in the same function, while others represent function calls and returns. We assign a distance to each node, intuitively representing distance to a target node. For simplicity in the discussion we assume a single target.

Consider a node n, that is an ancestor of the target node. If the path from n to the target does not traverse a function return, then the distance of n is defined to be one more than the minimum distance of any of n's direct successors in the CFG. We call this an *absolute* distance.

However, the distance metric is less clear for program points within functions that are called *and returned from* on the path toward the target, since different call sites to these functions can have very different distances to the target. One could naively work with the minimal distance from any call site to statically compute an absolute distance but, without the context of a call stack, this could significantly underestimate distances during earlier calls. Therefore, for such functions our algorithm computes a *relative* distance for each program point, which is simply the distance to a return point in the function, and we later compute a final distance metric for these nodes on the fly during symbolic execution. Specifically, to compute the distance metric for a node reached during symbolic execution, we traverse the current call stack backward, summing all of the relative distances encountered and stopping when reaching the first absolute distance, which is included in the final sum (Figure 5).

Return normalization. The basic distance metric above works well under most circumstances, but we encountered a problem that occurs frequently in input-parsing

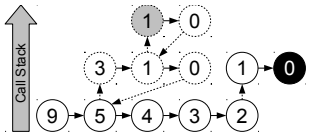

Figure 5: ***Calculating inter-procedural distances:*** *absolute distances from each program point to the black target are calculated on the direct call path (solid circles) and relative distances (dashed circles) are computed to exit points in called functions. The final distance from the gray node is calculated as as the sum of relative distances along the stack up to and including the first absolute one:* $1+1+5=7$.

```
1  int entry( char *input ) {
2    if (checkInput(input) == SUCCESS)
3      target();
4      return SUCCESS;
5    } else {
6      return FAIL;
7    }
8  }
9
10 int checkInput( char *input ) {
11   int i = 0;                    // Distance: 6
12   while (input[i] == ' ')       // Distance: 5
13     i++;                        // Distance: 6
14   if (input[i] != 'g')          // Distance: 4
15     return FAIL;                // Distance: 3
16   if (input[i+1] != 'o')        // Distance: 3
17     return FAIL;                // Distance: 2
18   if (input[i+2] != 'o')        // Distance: 2
19     return FAIL;                // Distance: 1
20   if (input[i+3] != 'd')        // Distance: 1
21     return FAIL;                // Distance: 0
22
23   return SUCCESS;               // Distance: 0
24 }
```

Figure 6: *Example showing the importance of equally prioritizing all exits.*

code, where the particular return point from a function affects the ability of the path to eventually reach the target. Figure 6 shows an example. The `entry` function reaches the target and calls function `checkInput` along the way. A naive approach would guide execution within `checkInput` to the nearest return instruction. However, it's clear from the code that only the instruction that returns `SUCCESS` can actually lead to the target point; the other return points signal an error that gets propagated back up the call stack until the message is rejected as invalid.

Augmenting the CFG analysis with the necessary *value sensitivity* to address this problem would require a much more sophisticated and computationally intensive algorithm (akin to symbolic execution itself!). Instead, we modify our metric for return points to prioritize them equally. This will not prevent exploration from taking the wrong ones at some point, but it will mitigate the pathological case in Figure 6 by exploring each return point *before* unraveling the loop on line 12 one more iteration, instead of unraveling all iterations with the first return before trying the next one. This adaptation is achieved by taking into account each exit point's depth, *i.e.,* its minimum control-flow distance from the entry point, in relation to the function's maximum depth. Using this notion, instead of initializing each exit point's distance to 0, each one is assigned a custom distance metric calculated as $maxDepth - depth(node)$, effectively making

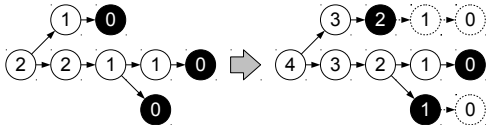

Figure 7: ***Normalizing the costs to exit nodes:*** *the initial cost of exit nodes (black) is adjusted when computing relative distances to avoid favoring any particular one. This is equivalent to adding virtual NOPs (dashed) to early returns so that all return statements appear as if they were at the same depth.*

them all look equidistant (Figure 7). The resulting relative distances are annotated in Figure 6.

3.5 State Initialization

PIC provides support for finding non-interoperabilities on a message that occurs somewhere "in the middle" of a protocol. For instance, a developer may want to detect non-interoperabilities for a data transfer, which occurs only after proper connection establishment. For this, PIC provides a lightweight form of message record and replay. During test runs of a protocol, PIC automatically logs the concrete values of messages as well as client and server API inputs at each protocol interaction. Suppose there are m messages in a protocol interaction, and a developer wishes to analyze the k-th message ($k \leq m$). The developer invokes PIC with the log and k as inputs, and PIC begins symbolic execution by feeding in the first $k-1$ messages as concrete values. This sets up the initial state for symbolically executing the k-th message, which proceeds exactly as described above.

This approach enables PIC to symbolically explore the space of k-th messages, predicated on the sets of $k-1$ concrete recorded messages. In future work, we intend to explore scalable iterative joint symbolic execution, exploring all possible combinations of k messages.

3.6 Clustering Non-interoperabilities

During exploration, it is not uncommon for PIC to find many non-interoperable messages or API inputs that stem from the same underlying issue (e.g., improper validation of an API input). It can be difficult for a developer to manually sift through each of these. We implemented an optional analysis step that allows developers to classify the resulting instances into separate clusters.

The developer uses this capability as follows. She first picks one instance and defines a clustering function for it. The function indicates whether a given instance belongs to the cluster. It can use as input any attribute of the instance, such as the protocol API that was called, conditions on the API input, or even a regular expression on the message content. Then, she runs the clustering function(s) defined thus far, at the end of which there are several unclassified test cases. She picks one of these, and repeats the steps above until no unclassified test cases remain. In one of our evaluations, PIC produced over 17,000 non-interoperability instances, which

were eventually clustered into 6 or 7 groups. The developer effort is proportional to the number of clusters, not the number of instances and, from our experience, is not significant. The most elaborate such function we developed copied a state-machine from the original code and detected a particular transition. With such an iterative process, clustering becomes intertwined with debugging and can be seen as a means of filtering instances of non-interoperabilities that have already been classified. We have left automating this process to future work.

3.7 Stage Pipelining and Parallelism

Despite our optimizations, analyzing real protocol implementations can take a significant amount of time. Our implementation uses pipelining between the sender-side and receiver-side analyses and between the receiver-side and the validation stages. It also uses parallelism within the receiver-side analysis stage and the validation stage. We omit the details of these optimizations for brevity.

4 Evaluation

To evaluate PIC, we selected two different protocols to uncover non-interoperabilities: the Session Initiation Protocol (SIP), a signaling protocol for Internet telephony systems; and SPDY, a widely-deployed protocol for accelerating Web transfers. SIP was chosen due to its prior history of interoperability problems [35, 39], and SPDY was chosen because it is a recently developed, but rapidly evolving, protocol that is already implemented in the latest versions of browsers and servers and deployed on many major content providers. There is one additional important difference between the two protocols: whereas SIP headers are human readable text-based messages, SPDY is a binary protocol where messages have a more rigid structure. That PIC is able to analyze both classes of protocols demonstrates its generality.

We analyzed two SIP implementations (eXoSIP and PJSIP), and two versions each of two SPDY implementations (spdylay and nginx), using the following procedure. We first defined use cases, defining the client-server roles and assumptions regarding network state (*e.g.,* a SPDY client fetching content from a server on localhost). We then created simple test harnesses that essentially exercised a concrete example of a protocol interaction matching the selected use case (*e.g.,* invoking the SPDY request API with a partially symbolic URL in the form of "http://127.0.0.1/*"). Using a debugger, we then followed the code as it processed API and message inputs in order to determine where the message was sent and received and where the message passes validation and begins to be handled. These steps helped determine where to annotate the code. When analyzing later messages in an interaction, we used the annotations to record and replay inputs and messages from a test run, before initiating symbolic execution. After this, the analysis proceeded as described in §3.1.

Implementation	Library	Annotations	Test Harness
eXoSIP-3.6.0	43990	38	209
PJSIP-1.12	83916	21	-
spdylay-0.3.7	22739	23	420
spdylay-1.3.1	25495	23	420
nginx-1.5.5	99446	4	-
nginx-1.7.4	104364	4	-

Table 2: *Source lines of code for library, annotations and test harnesses.*

Table 2 shows that PIC scales to implementations that involve tens of thousands of SLoC. Moreover, we see that the effort of adding the annotations is small, especially compared to the size of the original code base. For example, for nginx, since we analyzed only one message, only 4 annotations were needed. For PJSIP and nginx, a separate test harness was not needed as we could simply re-use its command-line application. As an aside, although PIC was designed for protocol developers, we (the authors) did not develop any of the analyzed code; that we were able to find significant non-interoperabilities in these large implementations is an example of how automating the search for test inputs can reduce reliance on developer insight and intuition.

4.1 SPDY results

We evaluate two implementations of SPDY. The first is a modular SPDY stack called spdylay [42], for which we analyze versions v0.3.7 and v1.3.1. spdylay offers both client and server functionality. The second implementation is spdylay, a popular open-source Web server, for which we analyze versions v1.5.5 and v1.7.4. The former version supports only SPDY v2, but the latter includes support for v3 and v3.1.

Our analyses explore all 4 client-server combinations across these versions. Certain components of SPDY (data encryption and compression) are challenging for symbolic execution, since analyzing them can be analogous to reversing one-way functions. State-of-the-art techniques usually treat these as uninterpreted functions and then use developer-supplied invariants (*e.g.,* $Decrypt(Encrypt(data,key),key) = data$) to simplify the resulting expressions (*e.g.,* cancel out the encryption once the client has decrypted the cypher-text with the same key). Since KLEE doesn't support this kind of evaluation, we abstracted these into identity functions during analysis (but not during validation). This of course limits PIC's ability to detect non-interoperabilities that originate in these functions, but was necessary to enable analysis of the remaining protocol code.

Our experiments focused on the Stream Creation (SYN_STREAM) message and the Stream Response (SYN_REPLY) message (*the second message in the protocol interaction*). The non-interoperabilities discovered for the latter message are a subset of those for the former, so we focus on describing results for the former.

The results are summarized in Table 3. To obtain

#	Inputs	spdylay 0.3.7 spdylay 0.3.7	spdylay 0.3.7 nginx 1.5.5	spdylay 1.3.1 spdylay 1.3.1	spdylay 1.3.1 nginx 1.7.4	Cause	Verdict
A	hName="\x1"	✘				The client insufficiently validates API input, allowing control characters in header names. The server checks for and rejects these characters. The client API should fail if illegal characters are found.	Liberal Sender
B	hValue1="\x1"			✘		The spdylay server now validates header characters more carefully. The new validation was not applied on the client.	
C	hValue1=""		✘	✘		Empty header values still seems to escape client-side validation.	
D	path="/%%"		✘		✘	spdylay doesn't percent encode illegal characters.	
E	version= "HTTP/0.9"		✘		✘	nginx rejects any HTTP version prior to 1.0 if used in SPDY.	Conservative Receiver
F	hValue1="\n"		✘		✘	nginx doesn't allow either CR or LF characters in header values.	
G	path="/.."		✘		✘	nginx carefully parses the path hierarchy and prevents breaking out from the web root.	
H	hName=":"				✘	nginx only allows the the ":method", ":path", ":version", ":host", and ":scheme" headers to start with ":".	
J	hValue1="" hValue2=""	✘				In some circumstances, the client allows empty header values, whereas the server doesn't. An "off-by-one" error masks the check on the server when only the last header value is empty. The server is otherwise correct in disallowing empty header values, but should also check the last header. The client API should fail if empty values are found.	Bug
K	path="/%00"		✘			nginx specifically rejects the correctly encoded NUL character in the path.	Optional Feature
L	method= "TRACE"		✘			nginx doesn't allow the TRACE HTTP method.	
M	cliVersion=3		✘			This version of nginx doesn't yet support SPDY/3 and fails to parse messages. It also drops the message instead of responding with the specified error.	Unsupported Version
N	cliVersion=2 srvVersion=3	✘		✘		In the spdylay API the server application must specify the protocol version (usually negotiated by SSL). Messages processed in the wrong context are invalid. The version specified in incoming messages should be used instead.	

Table 3: *Interoperability issues in SPDY.*

these results, we constrained the PIC analysis to explore a subset of the protocol inputs for SPDY, namely the client version or `cliVersion`, the server version `srvVersion`, header names and values `hName` and `hValue`, and the HTTP `path`, `method`, and `version`. PIC found several tens of thousands of instances of non-interoperabilities, which we classify into 12 clusters; for brevity, we omit exact counts per cluster. A cross indicates that the cluster manifested in the corresponding client-server combination.

To gain insight into the underlying issues that cause non-interoperabilities, based on our reading of the specification, we classified the non-interoperability clusters into 5 qualitatively different sub-categories, discussed below. Because PIC has a general definition of non-interoperability (§2), it can discover non-interoperabilities that stem from many different underlying issues.

Liberal sender. A long-standing guideline for protocol developers has always been: be conservative in what you send, and liberal in what you receive. Non-interoperabilities can arise when senders are more liberal than receivers. For example, the spdylay client permits control characters in header names, but the spdylay server does not (**A** in Table 3). We reported this error to the spdylay developer, who fixed it in the newer version of spdylay. However, in fixing this error, spdylay added newer code to validate control characters in headers and values, but the developer appears to have forgotten to validate values for control characters on the client side, introducing a new non-interoperability where there was none previously (**B**). When we contacted the developer about this new bug, he was hesitant to fix it because he thought clients using the spdylay library may already be leveraging the library's lax checking of control characters; in effect, the developer seems inclined to preserve "bug compatibility." This fear of breaking compatibility once non-interoperability has been released into the wild motivates systematic checking prior to the release.

Similarly, the spdylay client is liberal in permitting empty header values, while the server rejects requests with empty headers (**C**). In analyzing the test cases for this non-interoperability, and discussing them with the spdylay developer, we discovered an implementation error, which we describe below. Finally, the spdylay client does not escape non-ASCII characters correctly in the path, which the nginx server appropriately rejects (**D**).

Implementation error. This non-interoperability (**J**) between spdylay client and server in the older version is more subtle and is unlikely to have been found by manually designed test inputs. The spdylay client allows empty values in name-value pairs, and the server usually checks for these and correctly skips them except in one corner case. The assumption that the beginning of a header value cannot happen at the last position of the decompressed packet payload masks the check, in what looks like an "off-by-one" error. The spdylay developer fixed it in the newer version of spdylay.

Conservative receiver. Non-interoperability can also arise when receivers are more conservative than what the specification requires, either because the specification is ambiguous, or for security reasons. One such non-interoperability is between spdylay and nginx. Although the nginx web server supports HTTP v0.9, it disallows tunneling HTTP 0.9 over SPDY (**E**). This non-interoperability is subtle because it occurs within a tunneled protocol, and demonstrates the power of the kind of systematic analysis that PIC performs. The SPDY specification does not require servers to prevent HTTP 0.9 tunneling within SPDY: the nginx developers appear to have made an undocumented assumption that clients are unlikely to

be using SPDY to tunnel HTTP 0.9. While this may be true, it is another instance of the benefit of systematic analysis to uncover such undocumented assumptions.

A second non-interoperability in this category occurs because nginx prevents paths that traverse up the directory hierarchy using "/.." (**G**): this is a security feature designed to prevent attackers from breaking out of the web root and accessing files from elsewhere in the filesystem hierarchy. The nginx code for determining this involves a fairly sophisticated state machine and PIC was able to symbolically traverse the state machine to uncover the non-interoperability.

A third non-interoperability in this category occurs with SPDY v3 on nginx. This version of SPDY merges normal HTTP headers with new ones added for tunneling, requiring tunneling headers to be prefixed with ':' (**H**). The specification is silent on whether other headers may be colon-prefixed, and nginx conservatively only permits a specific set of headers to be colon-prefixed.

Finally, nginx conservatively rejects header values containing carriage returns and linefeeds; the SPDY specification is silent on this point (**F**).

Optional features. Non-interoperabilities can also occur because specifications permit optional features. One example in this category is that a spdylay client can generate a SPDY request with an HTTP TRACE method (a valid HTTP method defined in the spec), which nginx does not support in either of its versions because of a cross-site scripting vulnerability (**L**). A second non-interoperability in this category is that spdylay permits generation of URL paths with a NUL character (escaped using '%', **K**). The specification for URI generation permits receivers to be conservative and reject paths with NUL characters in them; however, URIs with arbitrary binary characters (including the NUL character) are used in RESTful APIs, so some Web servers permit them.

Unsupported versions. The first non-interoperability in this category occurs between a spdylay client and server. Spdylay permits server applications to specify a version number, and incoming messages from clients are processed in that context, even if a client specified a different version number (**M**). While this is a relatively obvious error, with a simple fix—ensure that incoming messages are processed using the embedded version number—it was still surprising to see this error in a stack against which a client, a server, and a proxy have been developed. In discussions with the developer, it became clear there is an undocumented assumption that spdylay will be used with SSL, which negotiates the protocol version out-of-band. PIC, being a systematic tool, can unearth such undocumented assumptions.

A second non-interoperability, between the older spdylay and the older nginx, occurs because the older nginx does not support SPDY v3 (**N**). While this doesn't match a colloquial notion of interoperability, it matches our definition: spdylay generates a v3 request, but nginx rejects that message. As expected, this disappears in the newer nginx which supports SPDY v3.

Discussion. Several of the spdylay non-interoperabilities have been fixed. We have communicated the nginx non-interoperabilities to the developers and are awaiting their feedback. To quantify the complexity of these non-interoperabilities, we counted the number of path constraints minus the connecting constraints (§3.3) in the results reported by PIC: all of these non-interoperabilities contained *between 60 and 80 path constraints*. Roughly speaking, a blind search for these non-interoperabilities would have required searching a space of at least 2^{60} message-header combinations. Developer intuition can likely reduce this search space, but that alone is not sufficient. That is why we find many non-interoperabilities in our analysis, even when the client and server code was developed by the same developer (spdylay).

4.2 Session Initiation Protocol (SIP) results

SIP includes several messages for features such as establishing, answering, forwarding, and terminating calls; sending instant messages; and subscribing to events (*e.g.,* user presence). For our analysis we chose two mature and well-known SIP stacks: eXoSIP, an extension of the GNU oSIP library, as sender and receiver, and PJSIP, as a receiver. Our experiments with SIP implementations bring out two capabilities of PIC not highlighted above: PIC can be used to analyze later messages in a protocol interaction, and to iteratively uncover non-interoperabilities that reveal themselves only after existing non-interoperabilities are fixed.

Table 4 shows the 9 clusters of non-interoperabilities generated while running eXoSIP as client and PJSIP as server. We also ran eXoSIP as client and server, which does not exhibit any of these non-interoperabilities.

The discovered non-interoperabilities span different message types: call establishment (INVITE), feature discovery (OPTIONS) and event subscription (SUBSCRIBE). For each of these messages, we analyzed interoperability for three headers: `from`, `to`, and `event`.

Most of the SIP non-interoperabilities fall into a *liberal sender* category. The eXoSIP sender permits control characters in the `from` (**P**, **S**), `to` (**Q**, **T**) and `event` fields (**W**), which the PJSIP server rejects. The eXoSIP sender also permits `to` header inputs where the URI schema is confused with other parts of the URI (such as the display name); PJSIP rejects these malformed URIs (**Q**). The eXoSIP developer acknowledged these non-interoperabilities, but pointed out that eXoSIP is a library that performs minimal input validation. However, this appears to place an undue burden on the developer who

Message	#	Inputs	Cause	Verdict
INVITE (Call Initiation)	**P**	from="<\x2@127.0.0.1:5061>"	eXoSIP insufficiently validates API input, allowing control characters in the "from" field. PJSIP is unable to parse the resulting URI. The application should prevent such chars from being used.	Liberal Sender
	Q	to="sip\":c@127.0.0.1:5060>"	The eXoSIP parser seems to confuse the SIP URI scheme with other parts of the URI (e.g., the display-name). PJSIP is unable to parse the resulting URI. A robust scheme detection mechanism in the application could prevent such issues.	
	R	from="\"\"<si@127.0.0.1:5061>"	eXoSIP seems to allow the SIP URI scheme ("sip:") to be omitted, whereas PJSIP requires it. The server is unable to parse the resulting URI. The application should check for the absence of the scheme and add one when necessary.	
OPTIONS (Feature Discovery)	**S**	from="\"\"\x2<@127.0.0.1:5061>"	eXoSIP reuses code for the "from" field in both OPTIONS and INVITE messages. Invalid characters are still not checked for.	
	T	to="<SIP\":@127.0.0.1:5060>"	eXoSIP reuses code for the "to" field in both OPTIONS and INVITE messages. The URI scheme is still misinterpreted.	
	U	from="@127.0.0.1:5061>"	eXoSIP reuses code for the "from" field in both OPTIONS and INVITE messages. The URI scheme is still optional.	
OPTIONS Response (Feature Discovery)	**V**	header=" "	eXoSIP affords significant flexibility in generating new headers in response messages, but doesn't validate application input on the header name, allowing control characters to mangle the message. PJSIP is unable to parse the resulting message. The application should be careful not to inject syntactically incorrect headers.	
SUBSCRIBE (Event Subscription)	**W**	event="\r\n"	eXoSIP doesn't validate application input on the event header, allowing control characters to mangle the message. PJSIP is unable to parse the resulting message. The eXoSIP API should fail if illegal characters are found.	
	X	event="aaa"	After implementing simple character checking semantics, PIC creates valid tokens. PJSIP, however, implements more advanced event semantics and only allows subscription to known event types.	Optional Feature

Table 4: *Interoperability issues in SIP with spdylay as client and spdylay as server.*

uses the eXoSIP library to understand the details of the protocol standard. Furthermore, eXoSIP validates some inputs but not others, with no documented guidance for developers on what input validation is left to the application and what eXoSIP performs. In these circumstances, PIC can be used by library or application developers to discover the types of input validation that need to be performed at the application level to avoid non-interoperability issues when used with another implementation in production. eXoSIP also exhibits another liberal sender non-interoperability, omitting the "sip:" URI scheme (**R**, **U**). This truncated URI is permitted by eXoSIP, so it appears to be an undocumented assumption that permits a deviation from the standard for the case when eXoSIP clients talk to eXoSIP servers.

Later messages in a protocol interaction. We used PIC to analyze the OPTIONS response; *this experiment exercised PIC's ability to analyze deeper messages (§3.5).* In our experiment a valid OPTIONS request was generated in PJSIP and replayed into eXoSIP prior to analysis. When eXoSIP receives an OPTIONS request, it returns an OPTIONS response with a set of features specified by the application; however, as a liberal sender, eXoSIP does not validate this application input (**V**).

Iterative testing for non-interoperabilities. SIP allows applications to subscribe for specific "events" (e.g., status changes in a buddy list). The eXoSIP stack permits control characters in event names, which PJSIP rejects. We then fixed this non-interoperability in eXoSIP, and re-ran PIC. This time, PIC generated *another* non-interoperability caused by the fact that although now eXoSIP generates syntactically correct event tokens, it does not check if these match valid event tokens that are known to the remote end (**X**). In this case, the PJSIP server returns an error in response to the SUBSCRIBE message; correctly so, since supporting new client-defined features is an *optional feature* in the protocol. Given enough time, analyzing the unpatched eXoSIP would have found this issue. However, applying fixes such as this one can accelerate the process of finding deeper issues.

4.3 Joint vs. Independent Symbolic Execution

We compared joint and independent symbolic execution on our protocol implementations. On the newer versions of spdylay (as client) and nginx (as server), we find that *joint symbolic execution produces over 100,000 paths in half a day, while independent symbolic execution produces none in the same time.* This performance gap rendered independent symbolic execution completely ineffective for these implementations. On SIP and the older version of spdylay (client and server), independent symbolic execution is able to find the non-interoperabilities, but is slower than joint symbolic execution by orders of magnitude. There appears to be a performance cliff for independent symbolic execution with the newer versions of spdylay and with nginx, which support multiple SPDY versions and are more complex implementations. Beyond this cliff, independent symbolic execution cannot be used, and joint symbolic execution is needed.

4.4 Impact of Search Strategy

Figure 8 compares three search strategies from §3.4: depth-first-search (DFS), best-first-search (GreedyBestFS, which is equivalent to SDSE from [28]), and our customized A*. It also evaluates A* without the return normalization heuristic. Without a reference implementation to work with, we are unable to compare with CCBSE and Mix-CCBSE. The results in Figure 8 indicate a clear performance advantage of A* with return normalization: within an hour, it discovers 25× more test inputs than state-of-the-art approaches. Further, within this time, A* found 5 of the 8 clusters of non-interoperabilities affecting the older nginx, while both A* without return normalization and GreedyBestFS detect only 3, and DFS none.

For this particular scenario, A* without return normalization performed as well as GreedyBestFS. This suggests that, for these implementations, early returns represent the most important cause of local minima. In other

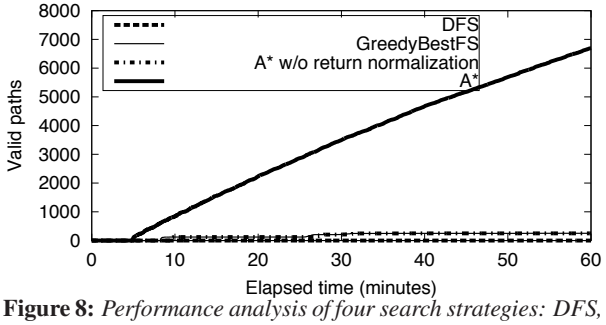

Figure 8: *Performance analysis of four search strategies: DFS, GreedyBestFS, and A* with and without return normalization. The plot illustrates the number of non-interoperabilities produced over time, while analyzing spdylay with nginx. The trials were performed on 10 servers totaling 216 CPU cores.*

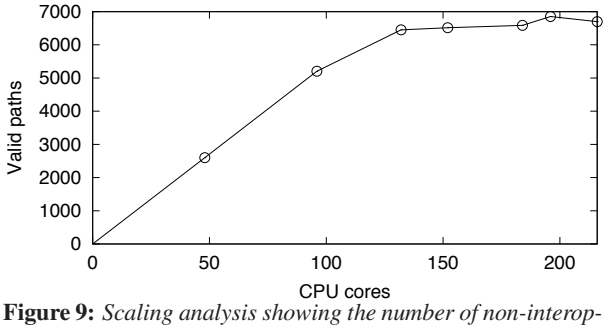

Figure 9: *Scaling analysis showing the number of non-interoperabilities found after 1 hour of analyzing spdylay with nginx, while varying the number of cores.*

settings, we believe A*'s local minima avoidance could outperform GreedyBestFS, even without return normalization.

4.5 Performance Micro-benchmark

To micro-benchmark PIC's performance, we measure the computation time for the main analysis stages, in isolation, for the newer spdylay against the newer nginx. This setup represents the most complex protocol combination we have tested to date. The results show that the client-side analysis took just over 5 hours to generate 60,000 paths, while the server took a little over half a day to generate more than twice that number.

We also ran a scaling analysis in Figure 9. The server-side analysis is pleasantly parallelizable as each client-path is independent but the client-side uses one core and bottlenecks the server at around 132 cores. Scaling out symbolic execution has been done [8], but we leave integrating such an approach for future work.

5 Related Work

Protocol analysis: Prior work has used high-level specifications, using finite state machines, higher-order logic [3, 4], or domain-specific languages [19] to perform a formal verification of protocols. While such specifications are powerful tools to reason about protocol behavior, they do not ensure correctness of implementation. PIC focuses on protocol implementations rather than manually derived formal specifications. Researchers have also explored the use of explicit-state model checkers to find bugs in protocol implementations [20, 21, 27, 32, 33, 43]. To our knowledge, model checking has not been used to discover non-interoperabilities. Like model checking, PIC faces similar challenges in path explosion and uses execution steering techniques to scale the analysis.

Symbolic execution: Godefroid *et al.* [15] and Cadar and Engler [10] developed a general technique for combining symbolic execution with concrete execution to generate test inputs. Since then, researchers have built mature tools using this approach [9, 16], have used it for finding program errors [6, 11], and have enhanced the basic technique in various ways [14, 28, 29]. Others have focused on making symbolic execution more efficient either by directing the search process [13, 28, 34, 44], or by merging states to reduce the search space [24]. We have compared PIC's approach to existing directed symbolic execution techniques (§4.4). State merging, on the other hand, is an orthogonal approach that could potentially be useful in merging paths exchanged during joint symbolic execution, and we have left to future work an exploration of this technique. Further, researchers have used symbolic execution to analyze protocols, but for properties other than interoperability: to determine *equivalence* between two implementations playing the role of the same network service [5, 25], to uncover manipulation attack vectors [23], and to discover bugs in layered server implementations [7]. While this body of work uses tools similar to those used by PIC, it focuses on fundamentally different problems. PIC's focus on interoperability between senders and receivers is unique to the best of our knowledge and motivates new techniques.

6 Conclusion

We presented PIC, which discovers interoperability problems in real protocol implementations. It uses program analysis to infer the sets of messages that one implementation can send but the other rejects. To scale the analysis, it uses joint symbolic execution, in which the receiver-side analysis is seeded by results from the sender. This technique was crucial for PIC and may be generally useful for analyzing interacting protocol implementations. On mature implementations of two protocols, PIC found thousands of instances of non-interoperabilities, across multiple message types and fault causes. Many of the issues have been acknowledged as undesirable by developers and some have already been fixed.

Acknowledgements: We thank the NSDI reviewers and our shepherd Petros Maniatis for helpful feedback on this paper. This work is supported in part by the US National Science Foundation (grant CNS-1161595) and by Cisco Systems.

References

[1] SPDY Protocol - Draft 3.1. http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1.

[2] The LLVM Compiler Infrastructure Project. http://llvm.org/.

[3] BHARGAVAN, K., OBRADOVIC, D., AND GUNTER, C. A. Formal verification of standards for distance vector routing protocols. *J. ACM 49*, 4 (July 2002).

[4] BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. SIGCOMM '05.

[5] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. SS'07.

[6] BRUMLEY, D., POOSANKAM, P., SONG, D. X., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy* (2008).

[7] BUCUR, S., KINDER, J., AND CANDEA, G. Making Automated Testing of Cloud Applications an Integral Component of PaaS. In *APSys 2013*.

[8] BUCUR, S., URECHE, V., ZAMFIR, C., AND CANDEA, G. Parallel symbolic execution for automated real-world software testing. EuroSys '11.

[9] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI'08.

[10] CADAR, C., AND ENGLER, D. R. Execution generated test cases: How to make systems code crash itself. In *SPIN* (2005), vol. 3639, Springer.

[11] CADAR, C., TWOHEY, P., GANESH, V., AND ENGLER, D. Exe: A system for automatically generating inputs of death using symbolic execution. In *CCS* (2006).

[12] FÄHNDRICH, M., REHOF, J., AND DAS, M. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI 2000*.

[13] GE, X., TANEJA, K., XIE, T., AND TILLMANN, N. DyTa: Dynamic symbolic execution guided with static verication results. In *ICSE 2011, Demonstration*.

[14] GODEFROID, P. Compositional dynamic test generation. POPL '07.

[15] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: directed automated random testing. In *PLDI* (2005).

[16] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *NDSS 2008*.

[17] GUNASINGHE, H. SCIM interop event at IETF 83rd meeting. http://hasini-gunasinghe.blogspot.com/2012/03/scim-interop-event-at-ietf-83rd-meeting.html, Mar. 2012.

[18] HALEPLIDIS, E., OGAWA, K., WANG, W., AND SALIM, J. H. Implementation report for forwarding and control element separation (ForCES). RFC 6053 http://tools.ietf.org/html/rfc6053, Nov. 2010.

[19] Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour, 1989.

[20] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: finding liveness bugs in systems code. NSDI'07.

[21] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: language support for building distributed systems. PLDI '07.

[22] KING, J. C. Symbolic execution and program testing. *CACM 19*, 7 (1976).

[23] KOTHARI, N., MAHAJAN, R., MILLSTEIN, T., GOVINDAN, R., AND MUSUVATHI, M. Finding Protocol Manipulation Attacks. In *SNAP* (August 2011).

[24] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient state merging in symbolic execution. In *PLDI 2012*.

[25] KUZNIAR, M., PERESINI, P., CANINI, M., VENZANO, D., AND KOSTIC, D. A SOFT way for OpenFlow switch interoperability testing. In *CoNEXT* (2012).

[26] LABOVITZ, C., AHUJA, A., ABOSE, A., AND JAHANIAN, F. An Experimental Study of Delayed Internet Routing Convergence. In *SIGCOMM 2000*.

[27] LEE, H., SEIBERT, J., KILLIAN, C., AND NITA-ROTARU, C. Gatling: Automatic attack discovery in large-scale distributed systems. NDSS 2012.

[28] MA, K.-K., KHOO, Y. P., FOSTER, J. S., AND HICKS, M. Directed symbolic execution. In *SAS* (September 2011), vol. 6887 of *Lecture Notes in Computer Science*.

[29] MAJUMDAR, R., AND XU, R.-G. Directed test generation using symbolic grammars. ASE '07.

[30] MASINTER, L. WebDAV interop report. http://www.webdav.org/users/masinter/interop/report.html, July 1999.

[31] MOME interoperability testing event. http://www.ist-mome.org/events/interop/, July 2005.

[32] MUSUVATHI, M., AND ENGLER, D. R. Model checking large network protocol implementations. NSDI'04.

[33] MUSUVATHI, M., PARK, D. Y. W., CHOU, A., ENGLER, D. R., AND DILL, D. L. Cmc: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002).

[34] PERSON, S., YANG, G., RUNGTA, N., AND KHURSHID, S. Directed incremental symbolic execution. In *PLDI 2011*.

[35] RAO, A., AND SCHULZRINNE, H. Real-world SIP interoperability: Still an elusive quest. http://www.sipforum.org/component/option,com_docman/task,doc_view/gid,124/, 2007.

[36] RCS VoLTE interoperability event 2012. http://www.msforum.org/interoperability/RCSVoLTE.shtml, Oct. 2012.

[37] REHOF, J., AND FÄHNDRICH, M. Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In *POPL 2001*.

[38] REPS, T. W. Program analysis via graph reachability. In *International Symposium on Logic Programming*.

[39] ROSENBERG, J. Basic level of interoperability for session initiation protocol (SIP) services (BLISS) problem statement. Internet draft http://tools.ietf.org/html/draft-ietf-bliss-problem-statement-04, Mar. 2009.

[40] ROSENBERG, J., SCHULZRINNE, H., CAMARILLO, G., JOHNSTON, A., PETERSON, J., SPARKS, R., HANDLEY, M., AND SCHOOLER, E. SIP: Session Initiation Protocol. RFC 3261, June 2002.

[41] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence - A Modern Approach (Third Edition)*. Pearson Education, 2010.

[42] TSUJIKAWA, T. Spdylay - SPDY C Library. http://spdylay.sourceforge.net/.

[43] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. NSDI '09.

[44] ZAMFIR, C., AND CANDEA, G. Execution synthesis: A technique for automated software debugging. In *EuroSys 2010*.