Extracting Protocol Format as State Machine via Controlled Static Loop Analysis

Qingkai Shi, Xiangzhe Xu, and Xiangyu Zhang, *Purdue University*

https://www.usenix.org/conference/usenixsecurity23/presentation/shi-qingkai

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the
32nd USENIX Security Symposium
is sponsored by USENIX.

Extracting Protocol Format as State Machine via Controlled Static Loop Analysis

Qingkai Shi
Purdue University

Xiangzhe Xu
Purdue University

Xiangyu Zhang
Purdue University

Abstract

Reverse engineering of protocol message formats is critical for many security applications. Mainstream techniques use dynamic analysis and inherit its low-coverage problem — the inferred message formats only reflect the features of their inputs. To achieve high coverage, we choose to use static program analysis to infer message formats from the implementation of protocol parsers. In this work, we focus on a class of extremely challenging protocols whose formats can be described through constraint-enhanced regular expressions and are parsed via finite state machines. Such state machines are often implemented as complicated parsing loops, which are inherently difficult to analyze via conventional static analysis. Our new technique extracts a sound state machine by regarding each loop iteration as a state and the dependency between loop iterations as state transitions. To achieve high, i.e., path-sensitive, precision but avoid path explosion, the analysis is controlled to merge as many paths as possible based on carefully-designed rules. The evaluation results show that we can infer a state machine and, thus, the message formats, in five minutes with over 90% precision and recall, far better than state of the art. We have also applied the state machines to enhance protocol fuzzers, which are improved by 20% to 230% in terms of coverage and detect ten more zero-days compared to baselines.

1 Introduction

In the era of the internet of things, any vulnerability in network protocols may lead to devastating consequences for countless devices that are inter-connected and spread worldwide. For instance, in 2020, a protocol vulnerability led to the largest ever DDoS attack that targeted Amazon Web Service, affecting millions of active users [16]. To ensure protocol security by automated analyses including fuzzing [39, 50], model checking [30, 79], verification [31], and many others, a key prerequisite is to acquire a formal specification of the message formats. However, this is a hard challenge.

There have been many works on automatically inferring the formats of network messages [49, 69, 80, 93]. However, almost all existing works are in a fashion of dynamic analysis — either network trace analysis [42, 63, 64, 74, 97, 98, 105] or dynamic program analysis [34, 35, 44, 54, 58, 71–73, 100]. The former captures online network traces and uses statistical methods including machine learning to cluster the traces into different categories and then perform message alignment and field identification. The latter runs the captured network traces against the protocol implementation and leverages the runtime control or data flows to infer message formats. Despite being useful in many applications, as dynamic analyses, they cannot infer message formats not captured by the input network traces. For instance, a recent work reported a highly precise technique but with coverage lower than 0.1 [105]. This means that it may miss message formats that are important for downstream security analysis.

To infer message formats with high coverage, we use static analysis, which does not rely on any input network traces but can thoroughly analyze a protocol parser. We target open protocols that have publicly available source code. While these protocols often have available specifications, they are usually documented in a natural language that is not machine-readable and contains inconsistencies, ambiguities, and even vulnerabilities [76]. Hence, inferring formal specifications for open protocols deserve dedicated studies. Particularly, we target a category of extremely challenging protocols, namely *regular protocols*, which have two main features. First, the format of a regular protocol can be specified by a constraint-enhanced regular expression (ce-regex), such as $(a|b)^+c$ where a, b, and c are respectively one-, two-, and four-byte variables satisfying the constraints $a \bmod 10 = 4$, $b > 3$, and $(c \gg 16) + c > 100$. Compared to a common regular expression (com-regex), the constraints in a ce-regex allow us to specify rich semantics in a network protocol. Note that a com-regex can be regarded as a simple instance of ce-regex. For instance, a com-regex $(a|b)^+c$ can be viewed as a ce-regex with the constraints $a =$ 'a', $b =$ 'b', and $c =$ 'c'. Second, the messages of a regular protocol are parsed via a finite state machine. This is common in performance-sensitive and embedded systems for the benefit of low latency [56]. That is, with a state machine, we can parse a protocol without waiting for the entire message — whenever receiving a byte, we parse it and record the current state; the recorded state allows us to continue parsing once we receive the next byte.

It is inherently challenging for static program analysis to infer the formats of a regular protocol from its parser. This is because a state machine for parsing is often implemented as a multi-path loop[1] that involves complex path interleaving that mimics the state transitions, but conventional static anal-

[1] A single-path loop contains only a single path in its loop body. A multi-path loop contains multiple paths in its loop body.

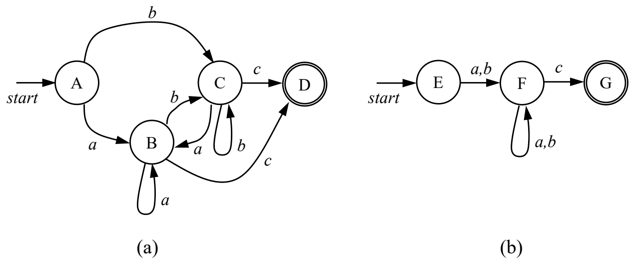


Figure 1: Example to illustrate the insight of our approach.

ysis — loop unwinding, loop invariant inference, and loop summarization — cannot handle such loops well. First, loop unwinding unrolls a loop with a limited number of iterations and, hence, will miss program behaviors beyond the unrolling times. Second, loop invariant techniques compute properties that always hold in each loop iteration. They rely on abstract interpretation for fixed point computation and, to ensure the termination, use the widening operators that often lead to significant loss of precision [20, 52, 53, 57, 60, 65, 78, 81, 86]. Third, loop summarization techniques precisely infer the input and output relations of a loop by induction. They are good at handling single-path loops [51, 84] or some simple multi-path loops [95, 103]. When used to analyze a multi-path loop that implements a state machine, they either fail to work or have to enumerate all paths in the loop body [101, 102], thus suffering from path explosion. The path explosion problem not only significantly slows down the static analysis but also leads to the explosion of states and state transitions, making the output state machine not operable.

To infer state machines from a parsing loop, our static analysis regards each loop iteration as a state and the dependency between loop iterations as state transitions. It mitigates the path explosion problem with the key insight that a state machine can be significantly compressed by merging states and state transitions. For instance, both state machines in Figure 1 represent the com-regex $(a|b)^+c$, but the one in Figure 1(b) is notably compressed. This observation guides us to design a static analysis that merges as many program paths as possible when analyzing an iteration of the parsing loop. As a result, we produce a super state, e.g., F, for a set of program paths, instead of many small states, e.g., B and C, for individual program paths. Hence, our analysis notably alleviates the path explosion problem and can infer highly compressed state machines, e.g., Figure 1(b), even from the implementation of complex state machines, e.g., Figure 1(a). As for state transitions, we record the pre- and post-condition of each loop iteration. These conditions allow us to compute the dependency between two consecutive loop iterations and are regarded as state-transition constraints. As a whole, an inferred state machine represents the message formats and can drive many security analyses.

There are three key differences between our approach and the state of the art. First, we do not assume the availability of network traces which, however, are required by existing works but could be hard to obtain [80]. Hence, our approach could be a promising alternative when high-quality network traces are not available. Second, different from many existing works that understand message formats by segmenting a message into multiple fields, we understand message formats via the parsing state machine. Such state machines allow us to specify message formats with both high precision and high coverage. As will be illustrated in §3, existing works are ineffective when handling state-machine-based parsers, thus exhibiting low precision and recall. Third, our work is also different from many previous works [23, 38, 39, 43, 46, 66, 68, 75, 87, 99, 105] that infer system state machines such as the one describing TCP's handshake mechanism. In this work, state machines are used to specify message formats. In summary, we make the following contributions.

- We developed a novel static analysis that mitigates the path-explosion problem in conventional approaches and can infer highly compressed state machines from code.
- We applied the static analysis to reverse engineering message formats. The analysis is highly precise and fast with high coverage. To the best of our knowledge, this is the first static analysis that formulates the problem of message format inference as extracting state machines.
- We implemented our approach, namely StateLifter, and evaluated it on ten protocols from different domains. StateLifter is highly efficient as it can infer a state machine or, equivalently, the message formats, in five minutes. StateLifter is also highly precise with a high recall as its inferred state machine can uncover $\geq 90\%$ protocol formats with $\leq 10\%$ false ones. By contrast, the baselines often miss $\geq 50\%$ of possible formats and may produce $\geq 40\%$ false ones. We use the inferred finite state machines to improve two state-of-the-art protocol fuzzers. The results demonstrate that, with the inferred state machines, the fuzzers can be improved by 20% to 230% in terms of coverage. We have discovered 12 zero-day vulnerabilities but the baseline fuzzers only find two of them. We also provide case studies of applying our approach to domains beyond network protocols.

2 Problem Scope

We target regular protocols, of which (1) the message formats can be described as constraint-enhanced regular expressions and (2) the messages are parsed via finite state machines (FSM). Formally, considering the equivalence of regular expression and FSM, we define a regular protocol in Definition 2.1 as an FSM enhanced by first-order logic constraints. The problem we address is to infer the FSM from the parser of a regular protocol. An FSM can be either deterministic or not. Since any non-deterministic FSM can be converted to a deterministic one, for simplicity, FSM means non-deterministic

FSM by default in this paper. Note that a non-deterministic FSM may contain multiple start states and a state may transition to multiple successor states with the same inputs.

Definition 2.1. *An FSM is a quintuple* $(\Sigma, \mathbb{S}, \mathbb{S}_0, \mathbb{F}, \delta)$ *where*

- Σ *is a set of first-order logic constraints over a byte sequence* σ^n *of length n. We use* σ_i^n *and* $\sigma_{i..j}^n$ *to represent the* $(i+1)$*th byte and a subsequence of* σ^n*, respectively. A typical constraint could be* $\sigma_1^2\sigma_0^2 > 10$*, which means that the value of a two-byte integer with* σ_1^2 *the most significant byte and* σ_0^2 *the least is larger than ten. We write* σ *as a shorthand of* σ_0^1 *and* σ^1*.*
- $\mathbb{S}$ *is a non-empty set of states;* $\mathbb{S}_0 \subseteq \mathbb{S}$ *is a non-empty set of start states;* $\mathbb{F} \subseteq \mathbb{S}$ *is a non-empty set of final states.*
- $\delta : \mathbb{S} \times \Sigma \mapsto 2^{\mathbb{S}}$ *is the transition function, meaning that when obtaining a byte sequence satisfying a constraint at a state, we will proceed to some possible states.*

By definition, a sequence of transitions from a start state to a final state defines a possible message format. For instance, $\delta(A \in \mathbb{S}_0, \sigma_1^2\sigma_0^2 > 10) = \{B\}$ and $\delta(B, \sigma = 5) = \{C \in \mathbb{F}\}$ are two transitions — one from a start state A to the state B with the constraint $\sigma_1^2\sigma_0^2 > 10$ and the other from the state B to a final state C with the constraint $\sigma = 5$. It implies a message format where the first two bytes satisfy $\sigma_1^2\sigma_0^2 > 10$ and the third byte must be 5. Such an FSM allows us to generate valid messages following the state-transition constraints.

Why Regular Protocols? In practice, the formats of a wide range of network protocols, such as HTTP and UDP, can be specified via ce-regex. This is acknowledged by many existing works, such as LeapFrog [48] that verifies protocol equivalence via FSMs, and P4 [33], a domain-specific language developed by the open networking foundation, which allows us to specify protocols via FSMs. As an example, we can specify an HTTP request using the following ce-regex: `Method Space URI Space Version CRLF ((General-Header | Request-Header | Entity-Header) CRLF)* CRLF Body?`, where each field, e.g., Method, satisfies certain constraints such as Method = 'Get' $\vee$ Method = 'Post' $\vee \cdots$.

A protocol that can be specified by ce-regex is unnecessary to be parsed by an FSM. However, an FSM parser can greatly improve the performance. Graham and Johnson [56] reported that an FSM parser can achieve over an order of magnitude performance improvement, and a hand-written FSM parser could scale better than widely-used implementations such as the Nginx and Apache web servers. The key factor contributing to this improvement is that an FSM parser can parse each byte of a network message as soon as the byte is received, without having to wait for the entire message. As an illustration, consider the FSM parser in Figure 2(a) that parses $(a|b)^*c$. Each iteration of the parser processes one byte received by the function *read_next_msg_byte()*. The parser's state, tracked by the variable *state*, allows it to continue parsing once the next byte is received. Hence, we can perform important business logic, such as preparing responses and updating system status, before a full message is received.

Due to this performance merit, regular protocols are frequently utilized in performance-critical systems, particularly in embedded systems that cannot tolerate latency. Typical examples include Mavlink [11] and MQTT [18], both of which are well-established in their respective fields. Mavlink is a standard messaging protocol for communicating with unmanned vehicles and is used in popular robotic systems such as Ardupilot [2] and PX4 [12]. MQTT, on the other hand, is a standard messaging protocol for the internet of things and is employed across various industries, such as automotive, manufacturing, and telecommunications, to name a few. In our evaluation, we include ten regular protocols designed for edge computing, musical devices, amateur radio, and others.

3 Limitation of Existing Works

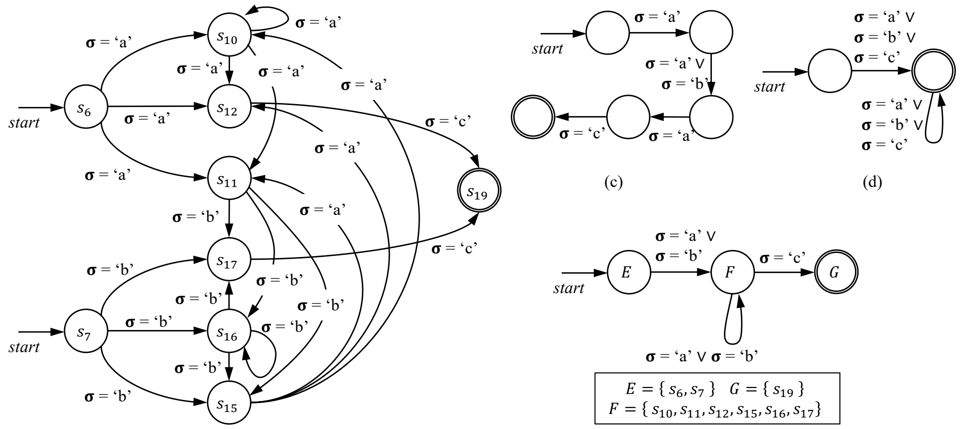
Network Protocol Reverse Engineering. Conventional techniques for format inference are either network trace analysis or dynamic program analysis. They only capture the features in input messages and are not effective for regular protocols.

(1) Network Trace Analysis (NTA). NTA does not analyze the implementation of protocols [23, 42, 63, 64, 74, 97, 98, 105]. Given a set of messages, they use statistical methods including machine learning to identify fields in a message or infer an FSM to represent message formats. The formats inferred by them strongly depend on the shape of input messages. For instance, assume that a valid message format satisfies the regular expression $(a|b)^+c$, meaning that a message can start with any combination of 'a' and 'b'. If all messages input to a typical NTA, such as REVERX [23] and NEMESYS [63, 64], start with 'aaa', it is very likely to infer an incorrect format starting with 'aaa'. In more complex cases where the format is a ce-regex, NTA cannot precisely infer constraints in the ce-regex, e.g., $a \bmod 10 = 4$, $b > 3$, and $(c \gg 16) + c > 100$. This motivates us to use program analysis so that we can precisely infer the constraints by tracking path conditions.

(2) Dynamic Program Analysis (DPA). DPA is more precise than NTA as it tracks data flows in protocols' implementation [34, 35, 44, 54, 58, 71–73, 100]. However, it shares the same limitation with NTA as the inferred formats also only capture the features of input messages. Typically, techniques like AUTOFORMAT [71] infer neither repetitive fields nor field constraints. For instance, given a set of messages such as { 'aaac', 'abac', ... } that satisfy the ce-regex $(a|b)^+c$ where $a =$ 'a', $b =$ 'b', and $c \geq$ 'c', while AUTOFORMAT will run these messages against the protocol's implementation, it does not extract conditions like $c \geq$ 'c' from the code and may produce a com-regex $a(a|b)ac$ as the format. The FSM of the com-regex is shown in Figure 2(c), which is not correct,

```
1.  void read_message_and_parse() {
2.    char state = 'A';
3.    while (1) {
4.      switch(state) {
5.      case 'A': char in = read_next_msg_byte();
6.          if (in == 'a') { state = 'B'; }
7.          else { assert(in == 'b'); state = 'C'; }
8.          break;
9.      case 'B': char in = read_next_msg_byte();
10.          if (in == 'a') { /*do nothing*/ }
11.          else if (in == 'b') { state = 'C'; }
12.          else { assert(in == 'c'); state = 'D'; }
13.          break;
14.     case 'C': char in = read_next_msg_byte();
15.          if (in == 'a') { state = 'B'; }
16.          else if (in == 'b') { /*do nothing*/ }
17.          else { assert(in == 'c'); state = 'D';}
18.          break;
19.     case 'D': exit(0);
20.     }}}
```

(a)



(b) (c) (d) (e)

Figure 2: (a) Implementation of the FSM in Figure 1(a). (b) The FSM inferred by the state-of-the-art static analysis, i.e., PROTEUS. (c) The FSM that represents the message format inferred by AUTOFORMAT. (d) The FSM that represents the message format inferred by TUPNI. (e) The FSM inferred by our approach, which is exactly the same as the compressed FSM in Figure 1(b).

inasmuch as it cannot parse messages with repetitive fields and the last transition is not labeled by the correct constraint $\sigma \geq$ 'c' and, thus, is considered to be a false transition.

TUPNI [44] handles parsing loops with the assumption that loops are used to parse repetitive fields in a network message. However, this is not true for regular protocols. For example, Figure 2 shows the implementation of the FSM in Figure 1(a). We can observe that the loop parses all fields in a message, no matter a field is repetitive, e.g., a and b, or just a single byte, e.g., c. Hence, TUPNI will produce a format like $(a|b|c)^+$ as the byte c is also handled in the loop and regarded as a repetitive field. Figure 2(d) shows the corresponding FSM, which does not represent a correct format. For example, in the inferred FSM, the incoming transitions of the final state may have the constraint $\sigma =$ 'a', but in the correct FSM shown in Figure 1, the incoming transitions of the final state are only constrained by $\sigma =$ 'c'.

Static Loop Analysis. Unlike NTA and DPA which only capture formats in their input messages, we propose to use static analysis to infer all possible formats in the form of FSM. However, we fail to find any practical static analysis that can infer such formats with high precision, recall, and speed.

(1) Loop Unwinding and Loop Invariant. Loop unwinding limits the number of loop iterations to a constant k [25, 88, 90, 104]. When analyzing the parser in Figure 2(a), it will only produce the formats of the first k bytes as each iteration analyzes one byte. Loop invariant techniques [20, 52, 53, 57, 60, 65, 78, 81, 86] do not infer FSMs, either. They compute constraints that always hold after every loop iteration. For instance, a possible invariant of the loop in Figure 2(a) could be 'a' $<$ *in* $<$ 'c'. This is far from our goal of FSM inference.

(2) Loop Summarization for FSM Inference. Chen et al.'s approach [37] assumes that an FSM parsing loop follows a simple pattern and, thus, is impractical for real-world protocol parsers. For instance, they regard a program variable as a state variable iff it is both modified in a loop iteration and referenced in future iterations. They assume such state variables have a limited number of values, e.g., the variable *state* in Figure 2 only has four possible values. This assumption is often violated in practice. A typical example is in Figure 4 where the variable *tok* satisfies their definition of state variables, but its value is not enumerable. In addition, this approach suffers from two explosion problems. First, they regard every possible combination of the state variables as a state, but the number of combinations could be explosive. For instance, if we have five state variables and each has five possible values, the resulting FSM will contain $5^5 > 3000$ states. Second, they depend on symbolic execution, which is well-known to suffer from path explosion. Such explosion problems not only make a static analysis not scalable but also significantly blow up an FSM with unnecessary states and transitions. Other approaches, e.g., [61, 91], have similar problems.

To the best of our knowledge, PROTEUS [101, 102] is the most recent and systematic approach to FSM inference. It regards every path within the body of a parsing loop as an FSM state and the dependency between two paths executed in two consecutive loop iterations as a state transition. Figure 2(b) shows the FSM inferred by PROTEUS, where s_i represents a state and also a path that goes through Line i. Each transition from s_i to s_j is labeled by the path condition of s_i. It means that if the parser executes the path s_i with its path condition, the next iteration may execute the path s_j. For instance, the state transitions from s_6 to s_{10}, s_{11}, and s_{12} are labeled by the condition $\sigma =$ 'a'. It means that if the loop executes the path s_6, of which the path condition is $\sigma =$ 'a', the loop may execute the path s_{10}, s_{11}, or s_{12} in the next iteration.

While the FSM is non-deterministic, it is absolutely correct to represent the format $(a|b)^+c$. For instance, the string 'abbc' can be parsed via the transitions $s_6 s_{11} s_{16} s_{17} s_{19}$. However, the

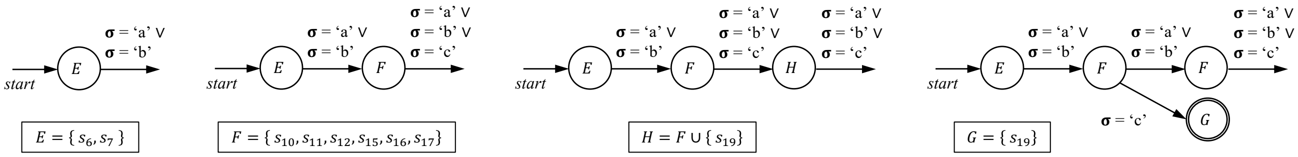


Figure 3: Basic steps of our approach.

FSM is too complex compared to the one we intend to implement, i.e., Figure 1(a). We observe that the core problem is that it enumerates all paths in the loop body as a priori but the number of paths is notoriously explosive. Thus, the resulting FSM contains an overwhelming number of states and transitions, and PROTEUS is impractical due to path explosion.

4 Technical Overview

At a high level, we follow a similar idea in terms of regarding a loop iteration as an FSM state and dependency between loop iterations as state transitions. However, unlike PROTEUS, we do not enumerate all individual paths in the loop but put as many paths as possible into a path set which, as a whole, is regarded as a single FSM state. This design simplifies the output FSM, significantly mitigates path explosion, but incurs new challenges. In what follows, we discuss two examples, one for our basic idea and the other for the detailed designs.

Basic Idea: Path Set as State. We perform a precise abstract interpretation over each iteration of the parsing loop. The basic steps of analyzing the code in Figure 2 are shown in Figure 3. In the first iteration of the parsing loop, due to the initial value of the variable *state*, we analyze the paths s_6 and s_7, depending on the condition: $\Phi_E \equiv \sigma =$ 'a' $\vee\, \sigma =$ 'b'. Thus, we create the state E to represent the path set $\{s_6, s_7\}$ and label the outgoing edge of E with the condition Φ_E.

After the first iteration, the value of the variable *state* is either 'B' or 'C'. Thus, in the second iteration, the abstract interpretation analyzes all paths in $F = \{s_{10}, s_{11}, s_{12}, s_{15}, s_{16}, s_{17}\}$ with the path condition $\Phi_F \equiv \sigma =$ 'a' $\vee\, \sigma =$ 'b' $\vee\, \sigma =$ 'c'. Hence, we create the state F with the outgoing condition Φ_F.

After the second iteration, the value of the variable *state* could be 'B', 'C', or 'D'. Thus, in the third iteration, we analyze the paths in $H = F \cup G, G = \{s_{19}\}$ with the path condition $\Phi_H \equiv \sigma =$ 'a' $\vee\, \sigma =$ 'b' $\vee\, \sigma =$ 'c'. Hence, we create the state H with the outgoing condition Φ_H.

Since the state H overlaps the state F, we split H into F and G, just as in the last graph in Figure 3. Since the state H is split, the original edge from F to H is also split accordingly. For instance, the condition from F to G is $\sigma =$ 'c' because, only when we go through the paths $s_{12}, s_{17} \in F$, of which the path condition is $\sigma =$ 'c', we can reach the path $s_{19} \in G$. The state G is a final state because it stands for the path s_{19} that leaves the parsing loop. Finally, we merge the two F states, forming a self-cycle as illustrated in Figure 2(e).

Algorithm 1: State Machine Inference.

1 **Procedure** `infer_state_machine` ($\mathbb{E}_{init}$)
2 $(S, \mathbb{E}_S) =$ abstract_interpretation $(\mathbb{E}_{init})$;
3 *Worklist* $= \{(S, \mathbb{E}_S)\}$; *FSM* $= \emptyset$;
4 **while** *Worklist not empty* **do**
5 $(S, \mathbb{E}_S)$ = *Worklist*.pop();
6 $(S', \mathbb{E}_{S'}) =$ abstract_interpretation $(\mathbb{E}_S)$;
7 add $(S, \mathbb{E}_S, S')$ into *FSM*;
`/* splitting operations */`
8 **foreach** *state X that should be split* **do**
9 split X into $X_1, X_2, \ldots$;
10 replace $(X, \mathbb{E}_X, Y) \in$ *FSM* with $(X_i, \mathbb{E}_{X_i}, Y)$;
11 replace $(Y, \mathbb{E}_Y, X) \in$ *FSM* with $(Y, \mathbb{E}_Y, X_i)$;
12 **assume** S' is split into S'_i, or $S' \equiv S'_i$ if S' is not split;
13 **if** $\nexists (S'_i, \mathbb{E}_{S'_i}, *) \in$ *FSM, where* $*$ *means any state* **then**
14 add $(S'_i, \mathbb{E}_{S'_i})$ into *Worklist*;
`/* merging operations */`
15 merge states that represent the same path set into one state;
16 **foreach** *pair of states* (X, Y) *such that there are multiple transitions* $(X, \mathbb{E}_{X1}, Y), (X, \mathbb{E}_{X2}, Y), \cdots \in$ *FSM* **do**
17 $\mathbb{E}_X =$ merge$(\mathbb{E}_{X1}, \mathbb{E}_{X2}, \ldots)$;
18 replace all $(X, \mathbb{E}_{Xi}, Y)$ with $(X, \mathbb{E}_X, Y)$ in *FSM*;
19 **if** $\forall \mathbb{E}_{Xi}.\mathbb{E}_X \not\equiv \mathbb{E}_{Xi}$ **then** add $(X, \mathbb{E}_X)$ into *Worklist*;
20 **return** *FSM*;

Algorithm Framework. Algorithm 1 sketches out our approach. Its parameter is the initial program environment $\mathbb{E}_{init}$, which provides necessary program information such as the initial path condition and the initial value of every program variable before entering a parsing loop. Line 2 analyzes the first iteration of the parsing loop and outputs the analyzed path set as well as the resulting program environment, i.e., $(S, \mathbb{E}_S)$. Line 3 initializes the FSM and a worklist.

The FSM is represented by a set of state transitions. Each transition is a triple $(S, \mathbb{E}_S, S')$ and describes the analyses of two consecutive iterations of the parsing loop — one analyzes the path set S and outputs $\mathbb{E}_S$; the other uses $\mathbb{E}_S$ as the precondition, which lets us analyze the path set S'. Each item in the worklist is the analysis result from an iteration of the parsing loop, i.e., $(S, \mathbb{E}_S)$. We use the worklist to perform a fixed-point computation. That is, whenever we get a new pair $(S, \mathbb{E}_S)$ that has not been included in the FSM, we add it to the worklist, because using a new $\mathbb{E}_S$ as the initial program environment may result in new analysis results from the parsing loop.

Lines 5-7 continue the analysis of the next loop iteration and add the new state transition to the FSM. Lines 8-11 split a state into multiple sub-states, just like we split the state H in

Figure 3. Lines 12-14 update the worklist by adding $(S'_i, \mathbb{E}_{S'_i})$ if the pair has not been included in the FSM. Line 15 merges the states that represent the same path set, just like that we merge the two states F in the last example. If the procedure above yields multiple but non-equivalent transitions between a pair of states, e.g., $(X, \mathbb{E}_{Xi\geq 1}, Y)$, Lines 16-19 merge them into one, $(X, \mathbb{E}_X, Y)$. If $\mathbb{E}_X \equiv \mathbb{E}_{Xi}$, we do not need to add $(X, \mathbb{E}_X)$ to the worklist, because the resulting transition $(X, \mathbb{E}_{Xi}, Y)$ has been in the FSM. Otherwise, $(X, \mathbb{E}_X)$ should be added to the worklist for further computation.

The details of the merging operation will be discussed later in §5, but it is sound and also guarantees the convergence of a fixed-point computation. That is, while we keep merging transitions from X to Y whenever a new transition between the two states is produced, the merging operation ensures that we will not endlessly generate new transitions from X to Y. Instead, it will converge, i.e., reach a fixed point.

Controlled State Splitting and Merging. The previous example shows the power of regarding multiple paths as a single state, which mitigates the path explosion problem and produces compressed FSMs. However, we observe that we cannot arbitrarily put all possible paths in a single state. Otherwise, invalid FSMs may be generated or the algorithm performance may be seriously degraded. Thus, we establish dedicated rules to control state splitting and merging. They are implemented into two key operations in Algorithm 1, namely `split` and `merge`. Next, we informally discuss them in three parts: (1) we list the rules of splitting and merging states; (2) we use a detailed example to show how these rules are used; and (3) we briefly justify the rationale behind the rules.

(1) Splitting and Merging Rules. We establish the following rules to split a state or merge multiple states.

- Splitting Rule (**SR1**): If two states represent overlapping path sets, we split them into multiple disjoint path sets. This rule has been illustrated in Figure 3 where the state H is split into F and G, so that we can reuse the state F.
- Splitting Rule (**SR2**): If a state represents a path set that includes both loop-exiting paths and paths that go back to the loop entry, we split it into a final state containing the exiting paths and a state containing the others. Otherwise, it will be hard to decide if an FSM terminates.
- Splitting Rule (**SR3**): If a state represents a path set where a variable is defined recursively in some paths, these paths should be isolated from others. For example, the paths s_{12} and s_{13} in Figure 4 define the variable *tok* in two manners. The path s_{13} defines the variable *tok* recursively based on its previous value. Hence, we put the two paths s_{12} and s_{13} in different path sets.
- Merging Rule (**MR1**): Given a set of states that represent the same path set with the same path conditions, we merge them into a single state. This rule has been illustrated in Figure 3 where we merge the two states F.
- Merging Rule (**MR2**): Given a sequence of transitions between a pair of states, we merge them into a single transition either by induction or, if induction fails, via a widening operator from classic abstract interpretation. Let us use the following examples to illustrate.
  - Given multiple transitions between a pair of states where the transition constraints form a sequence such as $\sigma = 1$, $\sigma = 2$, $\sigma = 3$, ..., we can apply inductive inference [22] to merge them into a single state transition with the constraint $\sigma = k$, meaning the kth transition constraint.
  - If the transition constraints are $\sigma = 0$, $\sigma = 3$, $\sigma = 1$, ..., we cannot inductively merge them as before. Instead, we merge them into $0 \leq \sigma \leq 3$ using the classic widening operator from interval-domain abstract interpretation [40]. This merging operation is sound but may lose precision.
- Merging Rule (**MR3**): To ensure the validity, i.e., a state transition does not refer to inputs consumed by previous transitions, we perform this rule after Algorithm 1 terminates. That is, given two consecutive transitions, e.g., $\delta(A, \Phi_A) = \{B\}$ and $\delta(B, \Phi_B) = \{C\}$, they are valid by definition iff Φ_A and Φ_B respectively constrain two consecutive but disjoint parts of an input stream. If the inputs constrained by Φ_A and Φ_B overlap, we either (1) replace the transition constraints with Φ'_A and Φ'_B such that $\Phi'_A \wedge \Phi'_B \equiv \Phi_A \wedge \Phi_B$ and neither Φ'_A nor Φ'_B refers to previous inputs, or (2) merge the transitions, yielding $\delta(A, \Phi_A \wedge \Phi_B) = \{C\}$ if Φ'_A and Φ'_B cannot be computed.

Theorem 1 (Convergence). *The splitting and merging rules guarantee the convergence of Algorithm 1.*

Proof. Given a parsing loop that contains n program paths in the loop body, SR1 ensures that we split these paths into at most n disjoint path sets. Thus, Algorithm 1 generates at most n states. While we may generate different transitions between a pair of states, Algorithm 1 leverages MR1-2 to merge them by conventional inductive inference [22] or interval-domain abstract interpretation [40], until a fixed point is reached. Thus, we compute at most one fixed-point transition between each pair of states. Since both the inductive inference and abstract interpretation converge, Algorithm 1 converges after generating at most n states and n^2 fixed-point state transitions. □

(2) Detailed Example. Figure 4 shows a common but complex case in protocol parsers. It looks for a nonempty token between the symbol '^' and the symbol ':'. The token *tok* is initialized to be an empty string and is reset when the input is '^' (Line 12). If the input character is a letter, the character is appended to *tok* (Line 13). If the input character is ':', it will check if the token *tok* is a nonempty keyword (Line 10).

Figure 4(a). Since the variable *state* and the variable *tok* are respectively initialized as *TOK* and an empty string, in the

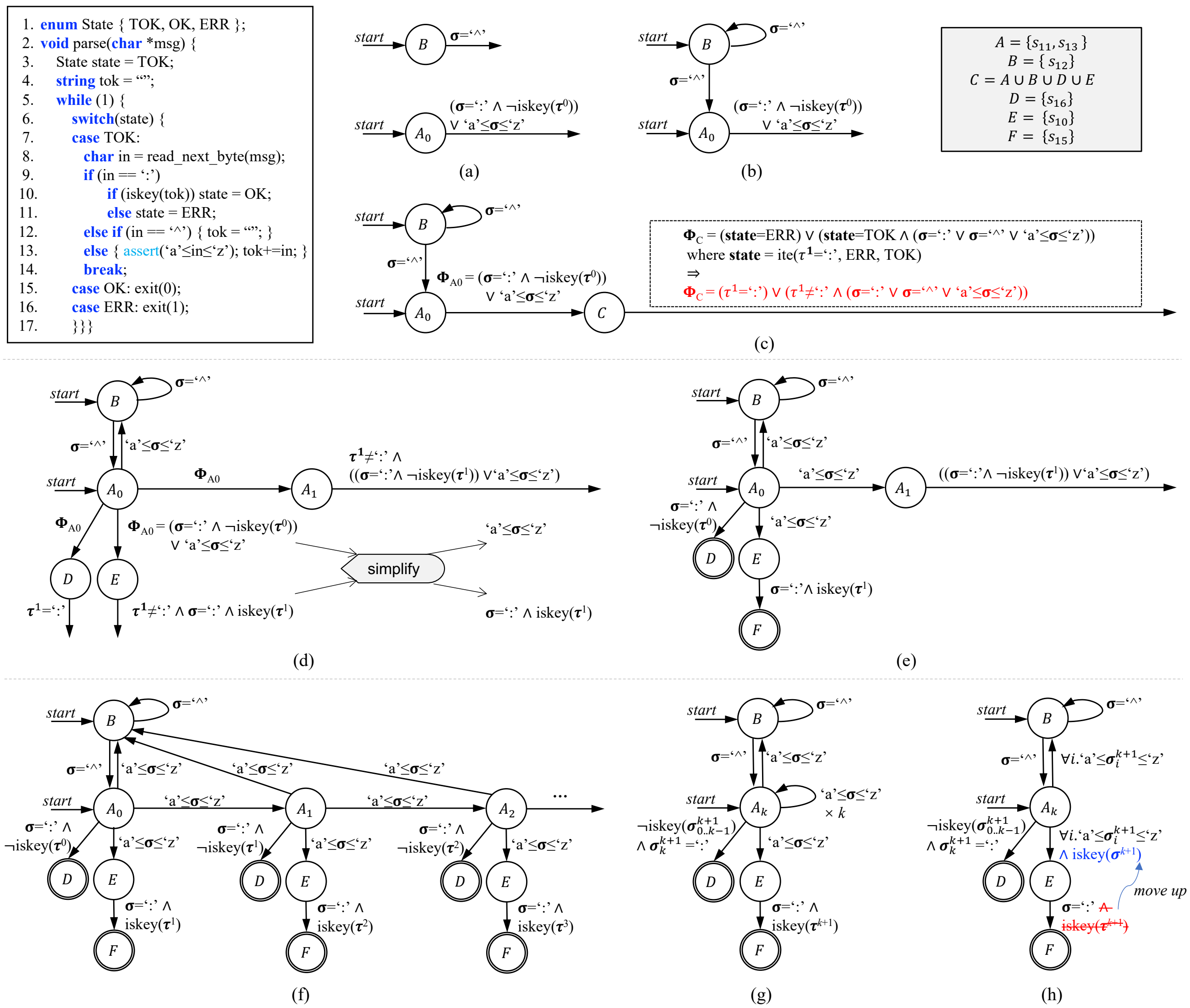


Figure 4: A detailed example. (a)-(h) The steps of FSM inference.

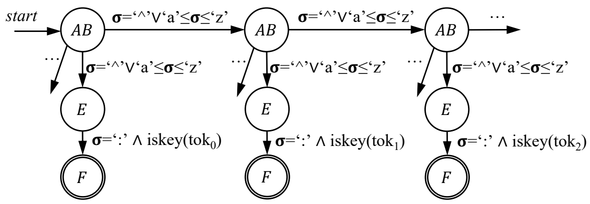
first iteration, we analyze the paths s_{11}, s_{12}, and s_{13} as other paths are infeasible. By SR3, the paths s_{12} and s_{13} cannot be in the same state. Thus, we create the states $A_0 = \{s_{11}, s_{13}\}$ and $B = \{s_{12}\}$. The outgoing constraint of each state is the path constraint, where we use the symbol τ^n to represent the input byte stream of length n before the current loop iteration. In the first iteration, *tok* is an empty string and denoted as τ^0.

Figure 4(b). The first iteration creates two states, $A_0 = \{s_{11}, s_{13}\}$ and $B = \{s_{12}\}$. If we follow the state B, i.e., the first iteration runs the path s_{12}, the code only resets the variable *tok* and, after the reset, it is like we never enter the loop. Hence, in the second iteration, we analyze the paths in $A_0 \cup B$ again just as in the first iteration. By MR1, we reuse the state A_0 and the state B. That is, we add a self-cycle on the state B and a transition from the state B to the state A_0.

Figure 4(c). If we follow the state A_0, i.e., the first iteration runs the paths in $A_0 = \{s_{11}, s_{13}\}$, the second iteration will analyze the paths in $C = \{s_{10}, s_{11}, s_{12}, s_{13}, s_{16}\}$. Thus, we create the state C and add the transition from A_0 to C. The outgoing transition of C is the path condition of all paths in C.

Figure 4(d). By SR1 and SR2, we split the state C into four sub-states $A_1, B, D = \{s_{16}\}$, and $E = \{s_{10}\}$. We reuse the state B but create a new state A_1 because the states A_0 and A_1 have different post-conditions. We then replace the state C with the four sub-states. The transition constraint from the state A_0 to each sub-state is the original constraint from the state A_0 to the state C. The outgoing constraint of each sub-state is the constraint of paths represented by the sub-state. For instance, for the sub-state $D = \{s_{16}\}$, its path condition is $state = \text{ERR}$ where the value of *state* is $ite(\tau^1 = \text{‘:’}, \text{ERR}, \text{TOK})$, meaning that if the previous input is ‘:’, $state = \text{ERR}$ and, otherwise, $state = \text{TOK}$. Thus, the outgoing constraint of D is $\tau^1 = \text{‘:’}$.

The incoming and outgoing constraints of a state can be cross-simplified. For instance, the outgoing constraint of E



tok_0=ite($\boldsymbol{\tau}_0^1$ =‘^’, “”, $\boldsymbol{\tau}_0^1$)
tok_1=ite($\boldsymbol{\tau}_1^2$=‘^’, “”, ite($\boldsymbol{\tau}_0^2$ =‘^’, “”, $\boldsymbol{\tau}_0^2$) +$\boldsymbol{\tau}_1^2$)
tok_2=ite($\boldsymbol{\tau}_1^3$=‘^’, “”, ite($\boldsymbol{\tau}_1^3$=‘^’, “”, ite($\boldsymbol{\tau}_0^3$ =‘^’, “”, $\boldsymbol{\tau}_0^3$) +$\boldsymbol{\tau}_1^3$)+$\boldsymbol{\tau}_1^3$)

Figure 5: Violation of SR3.

includes $\tau^1 \neq$ ‘:’. This means that the incoming constraint of E satisfies $\sigma \neq$ ‘:’, and thus, can be simplified to ‘a’ $\leq \sigma \leq$ ‘z’.

Figure 4(e). We continue a similar analysis of the next iteration from the states D, E, or A_1 because they have undetermined target states. From the state $D = \{s_{16}\}$, since the path s_{16} exits the loop, we stop the analysis and mark the state D as a final state. Similarly, we can find the final state F.

Figure 4(f) and Figure 4(g). If we continue the analysis from the state A_1, we will find a repetitive state sequence, i.e., A_0, A_1, A_2, and so on. We use MR2 to inductively merge them into A_k as shown in Figure 4(g). The merged state A_k means the $(k+1)$th state A. Thus, the self-cycle on A_k loops k times and each time consumes an input satisfying ‘a’ $\leq \sigma \leq$ ‘b’. For state transitions, e.g. the one from E to F, since the constraints between them in Figure 4(f) form the sequence: $\sigma =$ ‘:’ $\wedge\, \text{iskey}(\tau^1)$, $\sigma =$ ‘:’ $\wedge\, \text{iskey}(\tau^2)$, $\sigma =$ ‘:’ $\wedge\, \text{iskey}(\tau^3)$ and so on, the transition constraint from E to F in Figure 4(g) is summarized as $\sigma =$ ‘:’ $\wedge\, \text{iskey}(\tau^{k+1})$.

Figure 4(h). To ensure that a state transition does not refer to symbols in previous transitions, we merge the incoming and outgoing constraints of the state A_k and E by MR3, yielding the final FSM in Figure 4(h). The inferred FSM is correct. For instance, given a string “^^^abcd:” where we assume “abcd” is a keyword, the FSM can parse it by the transitions $BBBA_kEF$. That is, the transitions $BBBA_k$ consumes the prefix “^^^” and the transition from A_k to E consumes the keyword “abcd” by instantiating the induction variable $k = 4$. Finally, the transition from E to F consumes the colon.

(3) Consequences of Violating Rules. As stated in the proof of Theorem 1, SR1, MR1, and MR2 contribute to the convergence of the algorithm. Violating these rules may make the algorithm not terminating. SR2 and MR3 ensure the validity of an FSM by definition. That is, SR2 distinguishes final states from other states, and MR3 ensures that a state transition does not refer to symbols in previous transitions.

Particularly, SR3 facilitates the use of induction in MR2. Figure 5 shows the case where we do not use SR3 and, thus, merge the states A and B. In this case, after each iteration, the variable *tok* may be either reset or recursively defined, depending on if the previous input is ‘^’. In result, the value sequence of the variable *tok*, as shown in Figure 5, cannot be summarized as an expression parameterized by an induction variable k. According to MR2, to merge such repetitive states, we have to rely on widening operators, which are sound but imprecise [40]. Recall that, in Figure 4(f) where SR3 is used, the value of *tok* is a sequence of τ^1, τ^2, τ^3, and so on. Thus, we can precisely summarize its value as τ^{k+1} via MR2.

5 Formalizing the Approach

In this section, the notation $a[b/c]$ returns the expression a after using b to replace all occurrences of c in a. We use sat(ϕ) and unsat(ϕ) to mean that the constraint ϕ is satisfiable and not. An ite(v_1, v_2, v_3) formula returns v_2 and v_3 if the condition v_1 is true and false, respectively. We use a simplification procedure [47], $\phi_1' = \text{simplify}(\phi_1, \phi_2)$, to simplify ϕ_1 but keep the equivalence of ϕ_1 and ϕ_1' in terms of $\phi_2 \Rightarrow (\phi_1 \equiv \phi_1')$.

Abstract Language. For clarity, we use a C-like language in Figure 6 to model a parser that implements an FSM via a do-while loop. We use a do-while loop as it is a general form of loops with initialization, i.e., $\mathcal{S};\textbf{while}(1)\{\mathcal{S};\}$. The statements could be assignments, binary operations, read statements that read the next byte of a message to parse, exit statements that exit the loop, and branching statements that are uniquely identified by the identifier κ. To use our approach, users manually annotate the statement reading the inputs, e.g., the read function. The rest is fully automated. Although we do not include function calls or returns for simplicity, our system is interprocedural as a call statement is equivalent to assignments from the actual parameters to the formals, and a return statement is an assignment from the return value to its receiver. The language abstracts away pointer operations because the pointer analysis is not our technical contribution and, in the implementation, we follow existing works to resolve pointer relations [104]. We do not assume nested loops for simplicity as we focus on the outermost loop that implements the FSM. In practice, we observe that inner loops often serve for parsing repetitive fields in a network message rather than implementing the FSM. Hence, in the implementation, we follow traditional techniques to analyze inner loops [51, 84].

Abstract Domain. An abstract value of a variable represents all possible concrete values that may be assigned to the variable during program execution. The abstract domain specifies the limited forms of an abstract value. In our analysis, the abstract value of a variable v is denoted as $\tilde{v}$ and defined in Figure 7. An abstract value could be a constant value c and a byte stream of length k, i.e., σ^k and τ^k, which respectively represent the input byte stream read in the current loop iteration and the previous iterations. The symbols τ_i^n, $\tau_{i..j}^n$, and τ are defined similarly as σ_i^n, $\sigma_{i..j}^n$, and σ. An abstract value can also be a first-order logic formula over other abstract values. To ease the explanation, we only support binary and ite formulas. Especially, we also include an interval abstract value to mean a value between two constants. As discussed later in Algorithm 3, such interval abstract values allow our analysis

Parser $\mathcal{P}$	:=	**do** { $\mathcal{S}$; } **while**(1);	
Statement $\mathcal{S}$	:=	$v_1 \leftarrow v_2$	**::assign**
		$\mid v_1 \leftarrow v_2 \oplus v_3$	**::binary**
		$\mid v_1 \leftarrow$ *read*()	**::read**
		$\mid$ *exit*()	**::exit**
		$\mid$ **if**$_\kappa$ (v) $\{\mathcal{S}_1;\}$ **else** $\{\mathcal{S}_2;\}$	**::branching**
		$\mid \mathcal{S}_1;\mathcal{S}_2$	**::sequencing**

$$\oplus \in \{\wedge, \vee, +, -, >, <, =, \neq, \dots\}$$

Figure 6: Language of target programs.

Abstract Value $\tilde{v}$	:=	c	**::constant value**
		$\mid \sigma^k$	**::current input of length** k
		$\mid \tau^k$	**::previous input of length** k
		$\mid \tilde{v}_1 \oplus \tilde{v}_2$	**::binary formula**
		$\mid \mathrm{ite}(\tilde{v}_1, \tilde{v}_2, \tilde{v}_3)$	**::if-then-else formula**
		$\mid \mathrm{int}(c_1, c_2)$	**::interval**

Figure 7: Abstract values.

to fall back to conventional interval-domain abstract interpretation [40], in order to guarantee convergence and soundness.

Abstract Interpretation. The abstract interpretation is described as transfer functions of each program statement. Each transfer function updates the program environment $\mathbb{E} = (\mathbb{I}, \phi)$. Given the set $\mathbb{V}$ of program variables and the set $\tilde{\mathbb{V}}$ of abstract values, $\mathbb{I} : \mathbb{V} \mapsto \tilde{\mathbb{V}}$ maps a variable to its abstract value. The constraint ϕ captures the skeletal path constraint, which stands for a path set executed in a single loop iteration. We say ϕ is a skeletal path constraint because it is in a form of conjunction or disjunction over the symbols κ or $\neg\kappa$, e.g., $\kappa_1 \wedge (\kappa_2 \vee \neg\kappa_2)$, where each symbol κ uniquely identifies a branch and is not evaluated to its branching condition. The real path constraint is denoted by the uppercase Greek letter $\Phi = \phi[\mathbb{I}(\kappa)/\kappa]$ where each κ is replaced by its abstract value. We list the transfer functions in Figure 8, which describe how we analyze a loop iteration, i.e., the procedure `abstract_interpretation` in Algorithm 1. In these transfer functions, we use $\mathbb{E} \vdash \mathcal{S} : \mathbb{E}'$ to describe the environment before and after a statement.

To initialize the analysis of a loop iteration, we set the initial environment to $\mathbb{E} = (\mathbb{I}, \phi)$, which is obtained from the previous iteration, and assume that abstract values in $\mathbb{I}$ use the symbols τ_i^k and $\sigma_{i'}^{k'}$. This means that the previous iteration depends on an input stream of length $k + k'$, in which k bytes from iterations before the last iteration and k' bytes from the last iteration. For the current iteration, all $k + k'$ bytes are from previous iterations. Hence, we rewrite all σ to τ.

The rules for assignment, binary operation, read, and exit are straightforward, which update the abstract value of a variable. The sequencing rule says that, for two consecutive statements, we analyze them in order. The branching rule states how we handle conditional statements. In the branching rule, $(\mathbb{I}, \phi)$ represents the environment before a branching statement. $(\mathbb{I}_1, \phi \wedge \phi_1)$ and $(\mathbb{I}_2, \phi \wedge \phi_2)$ are program environments we respectively infer from the two branches. At the joining point, we either use the analysis results of one branch if the other branch is infeasible, or merge program environments from both branches. When merging results from both branches, variables assigned different values from the two branches are merged via the *ite* operator. Path constraints are merged via disjunction with the common prefix pulled out.

Algorithm 2: Splitting Rules (SR1-3).

1 **Procedure** `split` $((S_1, \mathbb{E}_{S_1}, S_2), (S_2, \mathbb{E}_{S_2}, S_3))$
2   **assume** $\mathbb{E}_{S_1} = (\mathbb{I}_{S_1}, \phi_{S_1})$ and $\mathbb{E}_{S_2} = (\mathbb{I}_{S_2}, \phi_{S_2})$;
3   **assume** S_2 is split into two sub-states $S_{21}, S_{22}, \dots$;
4   **let** $\Phi_{S_{2i}} = \phi_{S_{2i}}[\mathbb{I}_{S_2}(\kappa)/\kappa]$;
5   **let** $\mathbb{I}_{S_{2i}} = \mathbb{I}_{S_2}[\mathrm{simplify}(\tilde{v}, \Phi_{S_{2i}})/\tilde{v}]$;
6   **let** $\mathbb{E}_{S_{2i}} = (\mathbb{I}_{S_{2i}}, \phi_{S_{2i}})$;
7   replace input transitions with $(S_1, \mathbb{E}_{S_1}, S_{2i}), (S_{2i}, \mathbb{E}_{S_{2i}}, S_3)$;

Abstract Finite State Machine. We use a graph structure to represent an FSM. That is, an FSM is a set of labeled edges. Each edge is a triple $(S, \mathbb{E}_S, S')$ where $\mathbb{E}_S = (\mathbb{I}_S, \phi_S)$, meaning a transition from the state S to the state S' with the transition constraint $\phi_S[\mathbb{I}_S(\kappa)/\kappa]$. In the triple, $\mathbb{E}_S$ is the resulting program environment after analyzing the path set S in a loop iteration. Next, we formally describe the other two key procedures, i.e., `split` and `merge`, in Algorithm 1.

(1) Splitting Rules (SR1-3). Splitting a state consists of two steps — splitting the path set the state represents and recomputing its outgoing program environment.

SR1 splits two overlapping path sets S_1 and S_2 into at most three subsets, respectively represented by $\phi_{S_1} \wedge \neg\phi_{S_2}$ that means paths in the first set but not in the second, $\phi_{S_1} \wedge \phi_{S_2}$ that means paths shared by the two sets, and $\neg\phi_{S_1} \wedge \phi_{S_2}$ that means paths not in the first set but in the second. We create a state for each of the three skeletal constraints if it is satisfiable. SR2 and SR3 isolate some special paths from a path set. Given the path set S_1 and the paths S_2 to isolate, we create two states represented by $\phi_{S_1} \wedge \neg\phi_{S_2}$ and $\phi_S \wedge \phi_{S_2}$, respectively.

After a state is split into multiple sub-states, we recompute the outgoing program environment for each sub-state. Algorithm 2 and Figure 9 show the splitting procedure, where we assume we split the state S_2 into multiple sub-states S_{2i} and split its outgoing transition $(S_2, \mathbb{E}_{S_2}, S_3)$ into $(S_{2i}, \mathbb{E}_{S_{2i}}, S_3)$. The splitting procedure consists of two steps. First, Line 4 in Algorithm 2 computes the real path constraint according to the skeletal path constraint of each sub-state. Second, Line 5 recomputes each abstract value under the new path constraint. Basically, this step is to remove values from unreachable branches. For instance, assume $\mathbb{I}_{S_2}(v) = \mathrm{ite}(\tilde{v}_1, \tilde{v}_2, \tilde{v}_3)$, meaning that after analyzing the path set S_2, the abstract value of the variable v is either $\tilde{v}_2$ or $\tilde{v}_3$, depending on if the branching condition $\tilde{v}_1$ is true. If paths in the subset S_{21} ensures $\tilde{v}_1 = \text{true}$, we then rewrite the abstract value as $\mathbb{I}_{S_{21}}(v) = \tilde{v}_2$.

(2) Merging Rules (MR1). MR1 merges two equivalent states. Lines 13-14 of Algorithm 1 implements this rule. We show the idea in Figure 10, where we assume $S_1' \equiv S_1$ and $\mathbb{E}_{S_1'} \equiv \mathbb{E}_{S_1}$. In this case, we merge S_1 and S_1', but do not compute the next states using $\mathbb{E}_{S_1'}$ because we have already computed them

$$\frac{\mathbb{I}\text{ uses }\tau_i^k,\sigma_{i'}^{k'}}{\mathbb{I},\phi\vdash:\mathbb{I}[\tau_i^{k+k'}/\tau_i^k][\tau_{k+i'}^{k+k'}/\sigma_{i'}^{k'}],\text{true}}\ \textbf{Init}\qquad \frac{\mathbb{I}(v_2)=\tilde{v}_2}{\mathbb{I},\phi\vdash v_1\leftarrow v_2:\mathbb{I}\cup(v_1,\tilde{v}_2),\phi}\ \textbf{Assign}\qquad \frac{\mathbb{I}(v_2)=\tilde{v}_2\quad \mathbb{I}(v_3)=\tilde{v}_3}{\mathbb{I},\phi\vdash v_1\leftarrow v_2\oplus v_3:\mathbb{I}\cup\{(v_1,\tilde{v}_2\oplus\tilde{v}_3)\},\phi}\ \textbf{Binary}$$

$$\frac{\mathbb{I}\text{ uses }\sigma_i^k}{\mathbb{I},\phi\vdash v_1\leftarrow \mathit{read}():\mathbb{I}[\sigma_i^{k+1}/\sigma_i^k]\cup\{(v_1,\sigma_k^{k+1})\},\phi}\ \textbf{Read}\qquad \frac{}{\mathbb{I},\phi\vdash \mathit{exit}():\mathbb{I},\phi}\ \textbf{Exit}\qquad \frac{\mathbb{I}_1,\phi_1\vdash \mathcal{S}_1:\mathbb{I}_2,\phi_2\quad \mathbb{I}_2,\phi_2\vdash \mathcal{S}_2:\mathbb{I}_3,\phi_3}{\mathbb{I}_1,\phi_1\vdash \mathcal{S}_1;\mathcal{S}_2:\mathbb{I}_3,\phi_3}\ \textbf{Sequencing}$$

$$\frac{\mathbb{I}(v)=\tilde{v}\quad \mathbb{I}\cup\{(\kappa,\tilde{v})\},\phi\wedge\kappa\vdash\mathcal{S}_1:\mathbb{I}_1,\phi\wedge\phi_1\quad \mathbb{I}\cup\{(\kappa,\tilde{v})\},\phi\wedge\neg\kappa\vdash\mathcal{S}_2:\mathbb{I}_2,\phi\wedge\phi_2}{\mathbb{I},\phi\vdash \textbf{if}_\kappa\ (v)\ \{\mathcal{S}_1;\}\ \textbf{else}\ \{\mathcal{S}_2;\}:\begin{cases}\mathbb{I}_1,\phi\wedge\phi_1 & \text{unsat}(\phi[\mathbb{I}(\kappa')/\kappa']\wedge\neg\tilde{v})\\ \mathbb{I}_2,\phi\wedge\phi_2 & \text{unsat}(\phi[\mathbb{I}(\kappa')/\kappa']\wedge\ \tilde{v})\\ \{(u,\text{ite}(\tilde{v},\tilde{u}_1,\tilde{u}_2):(u,\tilde{u}_1)\in\mathbb{I}_1\wedge(u,\tilde{u}_2)\in\mathbb{I}_2)\},\phi\wedge(\phi_1\vee\phi_2) & \textit{otherwise}\end{cases}}\ \textbf{Branching}$$

Figure 8: Transfer functions as inference rules for analyzing a loop iteration.

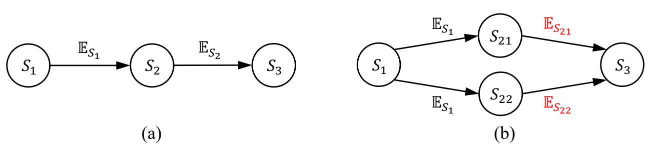


Figure 9: SR1-3. (a) Before splitting. (b) After splitting.

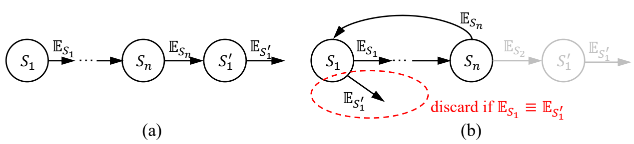


Figure 10: MR1. (a) Before merging. (b) After merging.

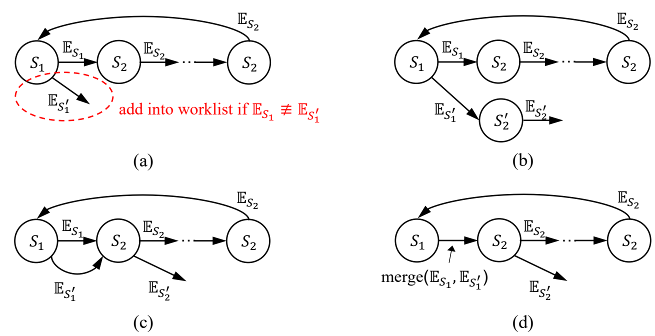


Figure 11: MR2. (a) Add $(S_1, \mathbb{E}_{S_1'})$ into worklist if $\mathbb{E}_{S_1'} \not\equiv \mathbb{E}_{S_1}$. (b) Generate $(S_2', \mathbb{E}_{S_2'})$ using $\mathbb{E}_{S_1'}$ as the precondition. (c) Merge S_2 and S_2'. (d) Merge transitions between S_1 and S_2.

using its equivalent counterpart $\mathbb{E}_{S_1}$. Thus, Algorithm 1 does not add $(S_1', \mathbb{E}_{S_1'})$ to the worklist at Lines 13-14.

(3) Merging Rules (MR2). MR2 merges two states that represent the same path sets but have non-equivalent outgoing program environments. Let us consider the example in Figure 11 to understand how Algorithm 1 deals with this case. Figure 11(a) is the same as Figure 10(b) except that we assume $\mathbb{E}_{S_1'} \not\equiv \mathbb{E}_{S_1}$. In this situation, we add $(S_1, \mathbb{E}_{S_1'})$ to the worklist (see Lines 13-14 in Algorithm 1). When $(S_1, \mathbb{E}_{S_1'})$ is popped out, we will perform abstract interpretation using $\mathbb{E}_{S_1'}$ as the initial program environment (see Lines 5-6 in Algorithm 1). Assume the abstract interpretation produces $(S_2', \mathbb{E}_{S_2'})$ where $S_2' \equiv S_2$ as illustrated in Figure 11(b). In Figure 11(c), we merge S_2 and S_2', yielding multiple non-equivalent transitions between S_1 and S_2. Lines 16-19 in Algorithm 1 merge such transitions, yielding Figure 11(d). If the merged environment, i.e., merge$(\mathbb{E}_{S_1}, \mathbb{E}_{S_1'})$ equals $\mathbb{E}_{S_1}$ or $\mathbb{E}_{S_1'}$, we do not add $(S_1, \text{merge}(\mathbb{E}_{S_1}, \mathbb{E}_{S_1'}))$ to the worklist because it has been in the FSM. Otherwise, the pair $(S_1, \text{merge}(\mathbb{E}_{S_1}, \mathbb{E}_{S_1'}))$ will be added to the worklist for further computation.

A naïve merging procedure is shown in Algorithm 3, which utilizes the traditional interval abstract domain to guarantee soundness and convergence. Lines 3-4 convert each abstract value to an interval, int(c_{min}, c_{max}), by solving two optimization problems via an SMT solver. Basically, solving the optimization problems respectively produces the minimum and maximum solutions, c_{min} and c_{max}, of the abstract value $\tilde{v}$ with respect to the path constraint. Lines 5-6 merge the interval values via the traditional widening operator [40]. As proved by Cousot and Cousot [40], the widening operator ensures convergence and soundness, which, in our context, means that it ensures the convergence and soundness of computing a fixed-point transition between two states. Nonetheless, the naïve merging procedure could result in a significant loss of precision because both the computation of intervals (Lines 3-4) and the merging of intervals (Lines 5-6) over-approximate each abstract value. Thus, before using the interval abstract domain to merge transitions, we always try an induction-based solution, which is discussed below.

The induction-based solution is sound and does not lose precision [22]. In the solution, we delay the transition merging operation until the number of transitions between a pair of states reaches a predefined constant. For instance, in Figure 12(a), we do not merge transitions until the number of transitions between each pair reaches 3. Given a list of transitions between a pair of states, we can then perform the inductive inference in two steps – guess and check. For instance, in Figure 12(a), assume $\mathbb{I}_{11}(v) = \sigma + 1$, $\mathbb{I}_{12}(v) = \sigma + 2$ and $\mathbb{I}_{13}(v) = \sigma + 3$. As shown in Figure 12(b), we then inductively "guess" the kth abstract value of the variable v as $\mathbb{I}_{1k}(v) = \sigma + k$. To check the correctness of $\mathbb{I}_{1k}(v) = \sigma + k$, as shown in Figure 12(c), we rerun the abstract interpretation using $\mathbb{E}_{S_{nk}}$ as the initial program environment, if in the resulting

Algorithm 3: Merging Rules (MR2).

Procedure `merge` ($\mathbb{E}_1$, $\mathbb{E}_2$)
- **assume** $\mathbb{E}_1 = (\mathbb{I}_1, \phi)$ and $\mathbb{E}_2 = (\mathbb{I}_2, \phi)$;
- **let** $\Phi_{S_1} = \phi_{S_1}[\mathbb{I}_{S_1}(\kappa)/\kappa]$; $\mathbb{I}_1 = \mathbb{I}_1[\mathtt{interval}(\tilde{v}, \Phi_{S_1})/\tilde{v}]$;
- **let** $\Phi_{S_2} = \phi_{S_2}[\mathbb{I}_{S_2}(\kappa)/\kappa]$; $\mathbb{I}_2 = \mathbb{I}_2[\mathtt{interval}(\tilde{v}, \Phi_{S_2})/\tilde{v}]$;
- **foreach** v such that $\mathbb{I}_1(v) = \tilde{v}_1 \wedge \mathbb{I}_2(v) = \tilde{v}_2$ **do**
  - **let** $\mathbb{I}(v) = \mathrm{widen}(\tilde{v}_1, \tilde{v}_2)$;
- **return** $(\mathbb{I}, \phi)$;

Procedure `interval` ($\tilde{v}$, Φ)
- **let** c_{min} = minimize $\tilde{v}$ with respect to Φ by SMT solver;
- **let** c_{max} = maximize $\tilde{v}$ with respect to Φ by SMT solver;
- **return** $\mathrm{int}(c_{min}, c_{max})$;

Procedure `widen` ($\mathrm{int}(a_1, b_1)$, $\mathrm{int}(a_2, b_2)$)
- **let** $c_1 = \mathrm{ite}(a_1 > a_2, -\infty, a_1)$; **let** $c_2 = \mathrm{ite}(b_1 < b_2, +\infty, b_1)$;
- **return** $\mathrm{int}(c_1, c_2)$;

Algorithm 4: Merging Rules (MR3).

Procedure `merge` ($(S_1, \mathbb{E}_{S_1}, S_2), (S_2, \mathbb{E}_{S_2}, S_3)$)
- **assume** $\mathbb{E}_{S_1} = (\mathbb{I}_{S_1}, \phi_{S_1})$ and $\mathbb{E}_{S_2} = (\mathbb{I}_{S_2}, \phi_{S_2})$;
- **let** $\Phi_{S_1} = \phi_{S_1}[\mathbb{I}_{S_1}(\kappa)/\kappa]$; $\Phi_{S_2} = \phi_{S_2}[\mathbb{I}_{S_2}(\kappa)/\kappa]$;
- **let** $\Phi_{S_1} = \mathrm{simplify}(\Phi_{S_1}, \Phi_{S_2})$; $\Phi_{S_2} = \mathrm{simplify}(\Phi_{S_2}, \Phi_{S_1})$;
- **if** Φ_{S_2} does not use any symbol τ **then return**;
- **assume** $\Phi_{S_1} = f(\sigma^k)$;
- **if** $\Phi_{S_2} = g(\sigma^l) \wedge h(\tau^m)$ **then**
  - **let** $\Phi_{S_1} = f(\sigma^k) \wedge h(\tau^m)[\sigma^k_{i-m+k}/\tau^m_{i \geq m-k}][\tau^{m-k}_i/\tau^m_{i<m-k}]$;
  - **let** $\Phi_{S_2} = g(\sigma^l)$;
- **else if** $\Phi_{S_2} = g(\sigma^l) \vee h(\tau^m)$ **then**
  - split the state S_2 as shown in Figure 13(c-d) and recursively call this procedure.
- **else**
  - **let** $\Phi_{S_1} = f(\sigma^k)[\sigma^{k+l}_i/\sigma^k_i]$; $\Phi_{S_2} = g(\sigma^l, \tau^m)[\sigma^{k+l}_{k+i}/\sigma^l_i]$;
  - **if** $m \geq k$ **then**
    - **let** $\Phi = \Phi_{S_1} \wedge \Phi_{S_2}[\sigma^{k+l}_{i-m+k}/\tau^m_{i \geq m-k}][\tau^{m-k}_i/\tau^m_{i<m-k}]$;
  - **else let** $\Phi = \Phi_{S_1} \wedge \Phi_{S_2}[\sigma^{k+l}_{k-m+i}/\tau^m_i]$;
  - merge transitions into one from S_1 to S_3 constrained by Φ;

program environment, the abstract value of v is $\sigma + (k+1)$, it means the summarized value $\mathbb{I}_{1k}(v) = \sigma + k$ is correct. This guess-and-check procedure follows the procedure of mathematical induction [85] and, thus, is correct.

(4) Merging Rules (MR3). MR3 ensures the validity of FSM by eliminating state transitions that refer to inputs consumed by previous transitions. It is performed after an FSM is produced by Algorithm 1. Algorithm 4 and Figure 13 demonstrate how it works on two transitions, one is from the state S_1 to the state S_2 and consumes k bytes, i.e., σ^k; the other is from the state S_2 to the state S_3, consumes l bytes, i.e., σ^l, and, meanwhile, constrains m bytes consumed by previous transitions, i.e., τ^m. First, for conjunctive constraints, e.g., $g(\sigma^l) \wedge h(\tau^m)$ in Figure 13(a), we only need to move the constraint $h(\tau^m)$ to the previous transition and perform constraint rewriting. Such rewriting does not change the semantics of the transition constraint but just lets it follow the definitions of σ and τ. Second, for disjunctive constraints, e.g.,

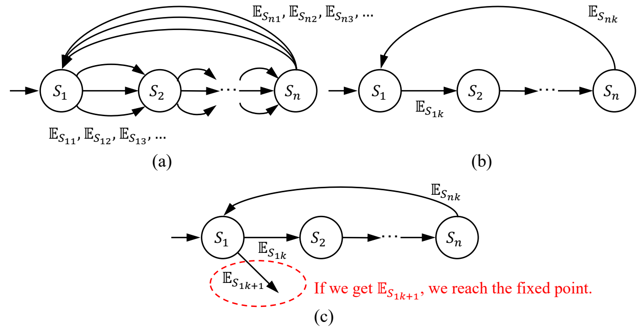


Figure 12: MR2 via induction. $\mathbb{E}_{S_{ij}} = (\mathbb{I}_{S_{ij}}, \phi_{S_i})$. (a) Delay merging. (b) Guess. (c) Fixed-point computation.

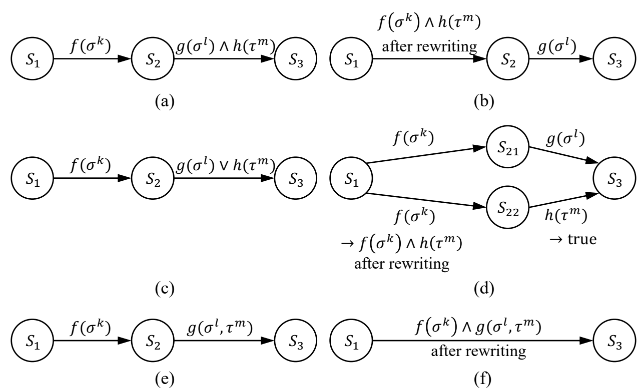


Figure 13: MR3. Eliminating τ in (a-b) conjunctive constraints, (c-d) disjunctive constraints, and (e-f) constraints where τ cannot be isolated by disjunction or conjunction.

$g(\sigma^l) \vee h(\tau^m)$ in Figure 13(c), we split the state S_2 to eliminate the disjunctive operator as shown in Figure 13(d) and then use the method for conjunction discussed above. Third, for constraints that cannot isolate τ-related sub-formulas via disjunction or conjunction, as shown in Figure 13(f), we merge the transitions into one.

Theorem 2 (Soundness and Completeness). *Given a program in the language defined in Figure 6, Algorithm 1 is sound using the aforestated splitting and merging rules. It is complete if the interval domain is never used during the analysis.*

Proof. The full proof can be found in the appendix. We sketch out the proof in what follows. Algorithm 1 computes a state transition $(S, \mathbb{E}, S')$ in three ways. First, it computes a transition based on the inference rules in Figure 8. The inference rules model the exact semantics of each program statement and, thus, do not introduce any approximation into $\mathbb{E}$. Second, a transition may be split by Algorithm 2, which applies a sound and complete simplification procedure [47]. Third, when multiple transitions exist between a pair of states, we merge them into one by either inductive inference or the widening operator of the interval domain. The inductive inference is sound and complete [22] and the widening operator is sound but not complete [40]. □

Discussion. We propose a static analysis that can infer an FSM from a parsing loop. While it is undecidable to check if an input loop intends to implement an FSM, as discussed in Theorem 2, given any loop in our abstract language, our approach guarantees to output a sound result. Nevertheless, the implementation in practice shares some common limitations with general static analysis. For instance, our static analysis is currently implemented for C programs and does not handle virtual tables in C++. We focus on source code and do not handle inline assembly. For libraries without available source code, e.g., crc16() and md5(), which are widely used to compute checksums or encrypt messages, we manually model these APIs. A common limitation shared with the state of the art is that, if the code implements a wrong FSM, the FSM we infer will be incorrect, either. Nevertheless, we will show that our approach is promising via a set of experiments.

6 Evaluation

On top of the LLVM compiler framework [67] and the Z3 theorem prover [45], we have implemented STATELIFTER for regular protocols written in C. The source code of a protocol is compiled into the LLVM bitcode and sent to STATELIFTER for FSM inference. In STATELIFTER, LLVM provides facilities to manipulate the code and Z3 is used to represent abstract values as symbolic expressions and solve path constraints.

Research Questions. First, we compare our approach to the state-of-the-art static analysis, i.e., PROTEUS [101, 102]. Second, we compare STATELIFTER to dynamic techniques, including REVERX [23], AUTOFORMAT [71], and TUPNI [44]. Third, to show the security impacts, we apply STATELIFTER to fuzzing and applications beyond protocols.

Benchmarks. Our approach is designed to work on the C code that implements the FSM parsing loop for regular protocols. We do not find any existing test suite that contains such C code. Thus, we build the test suite. To this end, we search the Github for regular protocols implemented in C language via the keywords, "protocol parser", "command parser", and "message parser", until we found the ten in Table 1. These protocols include text protocols such as ORP and binary protocols such as MAVLINK. They are widely used in different domains in the era of the internet of things. For example, ORP allows a customer asset to interact with Octave edge devices. MAVLINK is a lightweight messaging protocol for communicating with drones. TINY specifies the data frames sent over serial interfaces such as UART and telnet. SML defines the message formats for smart meters. RDB is a protocol for communicating with Redis databases. MQTT is an OASIS standard messaging protocol for IoT devices. MIDI is for musical devices and KISS is for amateur radio.

Environment. All experiments are conducted on a Macbook Pro (16-inch, 2019) equipped with an 8-core 16-thread Intel Core i9 CPU with 2.30GHz speed and 32GB of memory.

Table 1: Sizes of the Inferred State Machines

Protocols	STATELIFTER		PROTEUS	
	#states	#transitions	#states	#transitions
ORP [10]	5	8	42	92
MAVLINK [11]	42	197	-	-
IHEX [4]	15	63	-	-
BITSTR [7]	22	75	-	-
TINY [15]	14	54	151	872
SML [6]	32	89	-	-
MIDI [17]	19	81	765	3812
MQTT [19]	28	87	105	581
RDB [14]	22	57	-	-
KISS [5]	6	12	24	142

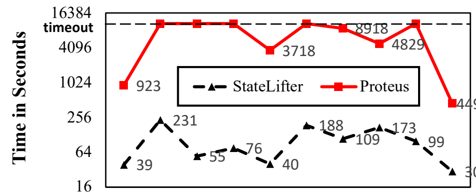


Figure 14: Time cost. The X-axis lists the ten protocols.

6.1 Against Static Inference Techniques

Our key contribution is a static analysis that fends off the path-explosion problem when inferring an FSM. To show the impacts of our design, we run both STATELIFTER and the state-of-the-art technique, PROTEUS [101, 102], against the benchmark programs on a 3-hour budget per program. The time cost of each analysis is shown in Figure 14. As illustrated, PROTEUS cannot complete many analyses within the time limit due to path explosion. By contrast, all our analyses finish in five minutes, exhibiting at least 70× speedup. Since both PROTEUS and STATELIFTER perform path-sensitive analysis, they have the same precision and recall when both of them succeed in inferring the FSM for a protocol, e.g., ORP. We detail the results of precision and recall in §6.2.

Table 1 shows the size of the inferred FSMs. Observe that the FSMs inferred by our approach are much (4×-40×) smaller than those inferred by PROTEUS. It demonstrates that our design not only significantly mitigates the path explosion problem but also infers highly compressed FSMs, which are expected to be easier to use in practice.

6.2 Against Dynamic Inference Techniques

Dynamic analysis is orthogonal to static analysis. Thus, in general, they are not comparable. Nevertheless, for the purpose of reference rather than comparison, we evaluate three dynamic analyses, including REVERX [23], AUTOFORMAT [71], and TUPNI [44]. REVERX is a black-box approach that learns an FSM from input messages without analyzing the code. It instantiates general automata induction techniques like L* [21] and is specially designed for protocol format inference. AUTOFORMAT and TUPNI are white-box methods that rely on

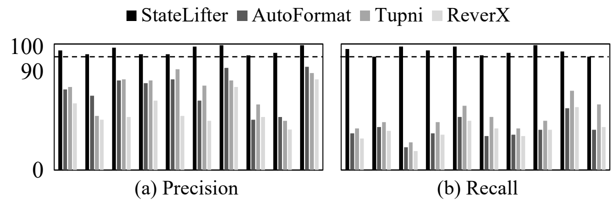


Figure 15: Precision and recall. The X-axes list the protocols.

dynamic dataflow analysis. They generate message formats in BNF, which can be easily converted to FSMs. Given that all analyses complete within a few minutes, our focus is primarily on examining their precision and recall. In an extended version of this paper [89], we detail how we compute precision and recall. Intuitively, the precision is the ratio of correct state transitions to all inferred transitions; and the recall is the ratio of correct transitions to all transitions in the ground truth.

To drive the dynamic analyses, we randomly generate one thousand valid messages as their inputs. By contrast, StateLifter does not need any inputs and, thus, provides a promising alternative to state of the art especially when the input quality cannot be guaranteed. The precision and recall of the inferred FSMs are plotted in Figure 15. It shows that we achieve over 90% precision and recall while the others often generate over 40% false or miss 50% true transitions. This is because they depend on input messages and cannot handle FSM parsing loops well. StateLifter also reports a few false transitions or misses some true ones as it inherits some general limitations of static analysis (see §5).

6.3 Security Applications

Protocol Fuzzing. AFLNet [82] accepts a corpus of valid messages as the seeds and employs a lightweight mutation method. Thus, we create a corpus, where each message is generated by solving the transition constraints. BooFuzz [9] directly accepts the message formats as its input and automatically generates messages. Thus, we respectively input the formats inferred by StateLifter, ReverX, AutoFormat, and Tupni to BooFuzz. The experiments are performed on a 3-hour budget and repeated 20 times to avoid random factors. As shown in Figure 16, since we can provide more precise and complete formats, fuzzers enhanced by StateLifter achieve 1.2×-3.3× coverage. Meanwhile, we detect twelve zero-day bugs while the others detected only two of them. We provide an example of these bugs in an extended version of this paper [89]. All bugs are exploitable as they can be triggered via crafted messages. Thus, they may pose notable threats to software security. For example, we identified four vulnerabilities in the official implementation of ORP [10], which is used for connecting Octave edge devices to the cloud [1].

Beyond Protocols. FSMs are widely used in other domains. In an extended version of this paper [89], we provide a case study of applying StateLifter to autopilot systems.

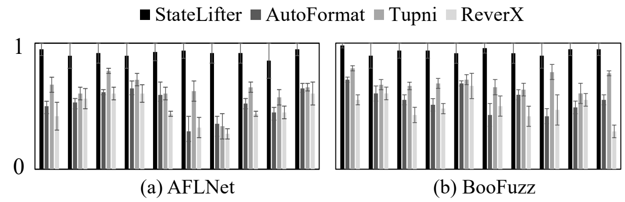


Figure 16: The X-axes list the protocols. The Y-axes denote the coverage normalized to 1 with a 95% confidence interval.

7 Related Work

Static Analysis for Protocol Reverse Engineering. While almost all existing works for inferring formats use dynamic analysis, Lim et al. [70] proposed a static analysis that is different from StateLifter in two aspects. First, it infers the formats of output messages whereas we focus on received messages. Second, it cannot handle loops that implement complex state machines and all loops are assumed to process repetitive fields in a message. StateLifter does not assume this. Rabkin and Katz [83] statically infer input formats in key-value forms, particularly for program configuration rather than networks. Shoham et al. [92] infer valid API sequences rather than message formats as state machines. Existing static analysis for reverse engineering focuses on security protocols, which, different from message formats, infers an agreed sequence of actions performed by multiple entities [24].

Applications of Protocol Reverse Engineering. Formal message formats are important for protocol fuzzing. Mutation-based fuzzers use formats to generate the seed corpus [36, 50, 55, 59, 82, 94]. Generation-based fuzzers directly use the formats to generate messages for testing [3, 8, 9, 13, 26]. Protocol model checking and verification also need formal protocol specifications [27–32, 41, 77, 79, 96]. Blanchet [32] specifies a protocol by Horn clauses and applies their technique to verify TLS models [29]. Beurdouche et al. [28] use Frama-C [62] to verify TLS implementations. Tamarin [77] uses a domain-specific language to establish proofs for security protocols and applies to 5G AKA protocols [27, 41]. Some works verify TCP components via symbolic analysis [30, 31, 79]. Udrea et al. [96] use a rule-based static analysis to identify problems in protocols. All these works assume the existence of formal specifications or manually build them. We push forward the study of automatic specification inference and can infer message formats with high precision, recall, and speed.

8 Conclusion

We present a static analysis that infers an FSM to represent the format of regular protocols. We significantly mitigate the path-explosion problem via carefully designed path merging and splitting rules. Evaluation shows that our approach achieves high precision, recall, and speed. Fuzzers supported by our work can achieve high coverage and discover zero-day bugs.

Acknowledgments

We thank the anonymous reviewers and shepherd for their valuable comments and suggestions. This work was supported in part by DARPA VSPELLS HR001120S0058, NSF1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] All-in-one edge-to-cloud solution. https://www.sierrawireless.com/octave/.

[2] Ardupilot. https://ardupilot.org/.

[3] Gitlab protocol fuzzer community edition (a.k.a. the Peach fuzzer). https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[4] Intel hex file parser. https://github.com/sfyip/Intel-HEX-file-parser.

[5] Kiss protocol (keep it simple stupid) parsing library for amateur radio. https://github.com/memoryhole/libkiss.

[6] Low memory C++ library to parse smart message language (SML) data from smart meters. https://github.com/olliiiver/sml_parser.

[7] Lua bitstring parsing and creation library based on Erlang bit syntax. https://github.com/luaforge/bitstring/.

[8] The network protocol fuzzer that we will want to use. https://github.com/OpenRCE/sulley.

[9] Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz.

[10] Octave resource protocol. https://github.com/SierraWireless/octave-orp/.

[11] Official reference C/C++ library for the mavlink v2 protocol. https://github.com/mavlink/c_library_v2/.

[12] Open source autopilot. https://px4.io/.

[13] A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley.

[14] Redis protocol parser. https://github.com/p5-RedisDB/perl-RedisDB-Parser.

[15] A simple library for building and parsing data frames for serial interfaces (like UART/RS232). https://github.com/MightyPork/TinyFrame.

[16] AWS shield threat landscape review: 2020 year-in-review. https://aws.amazon.com/blogs/security/aws-shield-threat-landscape-review-2020-year-in-review/.

[17] A C library for parsing midi messages. https://github.com/binarynate/midi-message-parser.

[18] MQTT: The standard for IoT messaging. https://mqtt.org/.

[19] Zero-copy, interruptible MQTT protocol parser and serialiser written in C. https://github.com/deoxxa/mqtt-protocol-c/.

[20] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*, 267(1):3–16, 2010.

[21] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[22] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15(3):237–269, 1983.

[23] Joao Antunes, Nuno Neves, and Paulo Verissimo. Reverse engineering of protocols from network traces. In *Working Conference on Reverse Engineering*, pages 169–178. IEEE, 2011.

[24] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26(1):99–123, 2014.

[25] Domagoj Babic and Alan J. Hu. Calysto: Scalable and precise extended static checking. In *International Conference on Software Engineering*, pages 211–220. IEEE, 2008.

[26] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: Toward a stateful network protocol fuzzer. In *International Conference on Information Security*, pages 343–358. Springer, 2006.

[27] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *ACM Conference on Computer and Communications Security*, pages 1383–1396. ACM, 2018.

[28] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *IEEE Symposium on Security and Privacy*, pages 535–552. IEEE, 2015.

[29] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the tls 1.3 standard candidate. In *IEEE Symposium on Security and Privacy*, pages 483–502. IEEE, 2017.

[30] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 265–276. ACM, 2005.

[31] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Hol specification and symbolic-evaluation testing for tcp implementations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66. ACM, 2006.

[32] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[33] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[34] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *ACM Conference on Computer and Communications Security*, pages 621–634. ACM, 2009.

[35] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM Conference on Computer and Communications Security*, pages 317–329. ACM, 2007.

[36] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Symposium on Network and Distributed System Security*, pages 1–15. Internet Society, 2018.

[37] Yongheng Chen, Linhai Song, Xinyu Xing, Fengyuan Xu, and Wenfei Wu. Automated finite state machine extraction. In *ACM Workshop on Forming an Ecosystem Around Software Transformation*, pages 9–15. ACM, 2019.

[38] Chia Yuan Cho, Domagoj Babić, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *ACM Conference on Computer and Communications Security*, pages 426–439. ACM, 2010.

[39] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, pages 110–125. IEEE, 2009.

[40] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[41] Cas Cremers and Martin Dehnel-Wild. Component-based formal analysis of 5g-aka: Channel assumptions and session confusion. In *Symposium on Network and Distributed System Security*, pages 1–15. Internet Society, 2019.

[42] Weidong Cui, Jayanthkumar Kannan, and Helen Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 199–212. USENIX, 2007.

[43] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H Katz. Protocol-independent adaptive replay of application dialog. In *Network and Distributed System Security Symposium*, pages 1–15. Internet Society, 2006.

[44] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *ACM Conference on Computer and Communications Security*, pages 391–402. ACM, 2008.

[45] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[46] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *USENIX Security Symposium*, pages 193–206. USENIX, 2015.

[47] Isil Dillig, Thomas Dillig, and Alex Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *International Static Analysis Symposium*, pages 236–252. Springer, 2010.

[48] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. Leapfrog: Certified equivalence for protocol parsers. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 950–965. ACM, 2022.

[49] Julien Duchene, Colas Le Guernic, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques*, 14(1):53–68, 2018.

[50] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, pages 330–347. Springer, 2015.

[51] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 23–33. ACM, 2011.

[52] Denis Gopan and Thomas Reps. Lookahead widening. In *International Conference on Computer Aided Verification*, pages 452–466. Springer, 2006.

[53] Denis Gopan and Thomas Reps. Guided static analysis. In *International Static Analysis Symposium*, pages 349–365. Springer, 2007.

[54] Rahul Gopinath, Björn Mathis, and Andreas Zeller. Mining input grammars from dynamic control flow. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–183. ACM, 2020.

[55] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security*, 10(8):239, 2010.

[56] Robert David Graham and Peter C. Johnson. Finite state machine parsing for internet protocols: Faster than you think. In *IEEE Security and Privacy Workshops*, pages 185–190. IEEE, 2014.

[57] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *International Conference on Computer Aided Verification*, pages 634–640. Springer, 2009.

[58] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *International Conference on Automated Software Engineering*, pages 720–725. ACM, 2016.

[59] Zhicheng Hu, Jianqi Shi, Yanhong Huang, Jiawen Xiong, and Xiangxing Bu. Ganfuzz: a gan-based industrial network protocol fuzzing framework. In *ACM International Conference on Computing Frontiers*, pages 138–145. ACM, 2018.

[60] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. Abstract acceleration of general linear loops. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 529–540. ACM, 2014.

[61] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin R. B. Butler, Antonio Bianchi, and Dave Jing Tian. Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. In *IEEE Symposium on Security and Privacy*, pages 2212–2229. IEEE, 2022.

[62] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[63] Stephan Kleber, Henning Kopp, and Frank Kargl. Nemesys: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *USENIX Workshop on Offensive Technologies*, pages 1–13. USENIX, 2018.

[64] Stephan Kleber, Rens W. van der Heijden, and Frank Kargl. Message type identification of binary network protocols using continuous segment similarity. In *IEEE Conference on Computer Communications*, pages 2243–2252. IEEE, 2020.

[65] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M Wintersteiger. Loop summarization using state and transition invariants. *Formal Methods in System Design*, 42(3):221–261, 2013.

[66] Patrick LaRoche, A Nur Zincir-Heywood, and Malcolm I Heywood. Network protocol discovery and analysis via live interaction. In *European Conference on the Applications of Evolutionary Computation*, pages 11–20. Springer, 2012.

[67] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75:1–75:12. IEEE, 2004.

[68] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: an automated script generation tool for honeyd. In *Annual Computer Security Applications Conference*, pages 203–214. IEEE, 2005.

[69] Xiangdong Li and Li Chen. A survey on methods of automatic protocol reverse engineering. In *International Conference on Computational Intelligence and Security*, pages 685–689. IEEE, 2011.

[70] Junghee Lim, Thomas Reps, and Ben Liblit. Extracting output formats from executables. In *Working Conference on Reverse Engineering*, pages 167–178. IEEE, 2006.

[71] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed System Security Symposium*, pages 1–15. Internet Society, 2008.

[72] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Reverse engineering input syntactic structure from program execution and its applications. *IEEE Transactions on Software Engineering*, 36(5):688–703, 2010.

[73] Min Liu, Chunfu Jia, Lu Liu, and Zhi Wang. Extracting sent message formats from executables using backward slicing. In *International Conference on Emerging Intelligent Data and Web Technologies*, pages 377–384. IEEE, 2013.

[74] Jian-Zhen Luo and Shun-Zheng Yu. Position-based automatic reverse engineering of network protocols. *Journal of Network and Computer Applications*, 36(3):1070–1077, 2013.

[75] Chris McMahon Stone, Sam L. Thomas, Mathy Vanhoef, James Henderson, Nicolas Bailluet, and Tom Chothia. The closer you look, the more you learn: A grey-box approach to protocol state machine learning. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2265–2278. ACM, 2022.

[76] Stephen McQuistin, Vivian Band, Dejice Jacob, and Colin Perkins. Parsing protocol standards to parse standard protocols. In *the Applied Networking Research Workshop*, pages 25–31. ACM, 2020.

[77] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*, pages 696–701. Springer, 2013.

[78] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

[79] Madanlal Musuvathi and Dawson Engler. Model checking large network protocol implementations. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 1–14. USENIX, 2004.

[80] John Narayan, Sandeep Shukla, and Charles Clancy. A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys*, 48(3):1–26, 2015.

[81] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to generate disjunctive invariants. In *International Conference on Software Engineering*, pages 608–619. ACM, 2014.

[82] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLnet: A greybox fuzzer for network protocols. In *International Conference on Software Testing, Validation, and Verification*, pages 460–465. IEEE, 2020.

[83] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *International Conference on Software Engineering*, pages 131–140. ACM, 2011.

[84] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 225–236. ACM, 2009.

[85] Diana Schmidt and Hans Hermes. *Introduction to mathematical logic*. Springer, 1973.

[86] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *International Conference on Computer Aided Verification*, pages 703–719. Springer, 2011.

[87] Maxim Shevertalov and Spiros Mancoridis. A reverse engineering tool for extracting protocols of networked applications. In *Working Conference on Reverse Engineering*, pages 229–238. IEEE, 2007.

[88] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 693–706. ACM, 2018.

[89] Qingkai Shi, Xiangzhe Xu, and Xiangyu Zhang. Extracting protocol format as state machine via controlled static loop analysis. *arXiv preprint arXiv:2305.13483*, 2023.

[90] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Path-sensitive sparse analysis without path conditions. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 930–943. ACM, 2021.

[91] Takahiro Shimizu, Norihiro Yoshida, Ryota Yamamoto, and Hiroaki Takada. Symbolic execution-based approach to extracting a micro state transition table. In *ACM SIGSOFT International Workshop on Testing, Analysis, and Verification of Cyber-Physical Systems and Internet of Things*, pages 1–6. ACM, 2019.

[92] Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

[93] Baraka D. Sija, Young-Hoon Goo, Kyu-Seok Shim, Huru Hasanova, and Myung-Sup Kim. A survey of automatic protocol reverse engineering approaches, methods, and tools on the inputs and outputs view. *Security and Communication Networks*, 2018(8370341):1–17, 2018.

[94] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *ACM Conference on Computer and Communications Security*, pages 1492–1504. ACM, 2016.

[95] Jan Strejček and Marek Trtík. Abstracting path conditions. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 155–165. ACM, 2012.

[96] Octavian Udrea, Cristian Lumezanu, and Jeffrey Foster. Rule-based static analysis of network protocol implementations. In *USENIX Security Symposium*, pages 193–208. USENIX, 2006.

[97] Yipeng Wang, Xingjian Li, Jiao Meng, Yong Zhao, Zhibin Zhang, and Li Guo. Biprominer: Automatic mining of binary protocol features. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 179–184. IEEE, 2011.

[98] Yipeng Wang, Xiaochun Yun, Zubair Shafiq, Liyan Wang, Alex Liu, Zhibin Zhang, Danfeng Yao, Yongzheng Zhang, and Li Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *IEEE International Conference on Network Protocols*, pages 1–10. IEEE, 2012.

[99] Yipeng Wang, Zhibin Zhang, Danfeng Daphne Yao, Buyun Qu, and Li Guo. Inferring protocol state machine from network traces: a probabilistic approach. In *International Conference on Applied Cryptography and Network Security*, pages 1–18. Springer, 2011.

[100] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *European Symposium on Research in Computer Security*, pages 200–215. Springer, 2009.

[101] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. Proteus: Computing disjunctive loop summary via path dependency analysis. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 61–72. ACM, 2016.

[102] Xiaofei Xie, Bihuan Chen, Liang Zou, Yang Liu, Wei Le, and Xiaohong Li. Automatic loop summarization via path dependency analysis. *IEEE Transactions on Software Engineering*, 45(6):537–557, 2019.

[103] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. S-looper: Automatic summarization for multipath string loops. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 188–198. ACM, 2015.

[104] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363. ACM, 2005.

[105] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. Netplier: Probabilistic network protocol reverse engineering from message traces. In *Symposium on Network and Distributed System Security*, pages 1–18. Internet Society, 2021.

A Soundness and Completeness

Algorithm 1 uses a worklist for fixed-point computation. The worklist is a set of $(S, \mathbb{E}_S)$ where S is a path set that we analyze in a loop iteration and $\mathbb{E}_S = (\mathbb{I}_S, \phi_S)$ is the resulting program environment. In the algorithm, whenever we create a new $(S, \mathbb{E}_S)$ (Line 14), or $(S, \mathbb{E}_S)$ in the FSM does not reach the fixed point (Line 19), we add it to the worklist. Given each item $(S, \mathbb{E}_S)$ popped from the worklist, we create a state transition $(S, \mathbb{E}_S, S')$. Hence, in what follows, we prove Theorem 2 in three steps, respectively discussing (1) the soundness and completeness of items in the worklist, i.e., $(S, \mathbb{E}_S)$, (2) the soundness and completeness of state transitions, i.e., $(S, \mathbb{E}_S, S')$, and (3) the soundness and completeness of the FSM, which is a set of state transitions.

Lemma 1 (Soundness of $(S, \mathbb{E}_S)$)**.** *For each variable v, $\mathbb{I}_S(v)$ returns a sound abstract value that over-approximates all possible concrete values of the variable v.*

Proof. In Algorithm 1, the pair $(S, \mathbb{E}_S)$ in the worklist may come from three places: ❶ the ones produced by the abstract interpretation (Line 14 if we have $S' \equiv S'_i$, meaning that we actually do not split the state); ❷ the ones produced by splitting (Line 14); and ❸ the ones produced by merging (Line 19). Next, we explain that in each case, any abstract value $\mathbb{I}_S(v)$ in the program environment is sound.

❶ Figure 8 shows a standard dataflow analysis for our abstract language model, i.e., Figure 6. The analysis models the exact semantics of each program statement. For instance, if the abstract values of the variables v_1 and v_2 are respectively $v_1^\sharp$ and $v_2^\sharp$, the result of $v_1 \oplus v_2$ will be $v_1^\sharp \oplus v_2^\sharp$. Hence, each inference rule in Figure 8 is sound and complete. Given that each inference rule is sound and complete, the analysis of each loop iteration is also sound and complete. Therefore, the resulting program environment is sound and complete, meaning that the abstract interpretation does not introduce any over- and under-approximation into the program environment.

❷ As shown in Algorithm 2, when splitting a state S to multiple states S_i, we rewrite each abstract value in $\mathbb{I}_S$ via a simplification procedure to build $\mathbb{I}_{S_i}$. This simplification procedure [47] only rewrites a formula by removing abstract values from unreachable paths and, thus, does not introduce any over- and under-approximation into the program environment. For instance, assume $\mathbb{I}_S(v) = \text{ite}(\tilde{v}_1, \tilde{v}_2, \tilde{v}_3)$, meaning that after analyzing the path set S, the abstract value of the variable v is either $\tilde{v}_2$ or $\tilde{v}_3$, depending on if the branching condition $\tilde{v}_1$ is true. If paths in the subset $S_i \subseteq S$ ensures $\tilde{v}_1 = \text{true}$, we then rewrite the abstract value as $\mathbb{I}_{S_i}(v) = \tilde{v}_2$.

❸ As shown in Algorithm 3, when merging two program environments, we first convert them into constant intervals and then use the widening operator to merge them. Both the conversion and widening operations are sound [40]. However, they are not complete because the two operations may introduce over-approximation into the abstract values. For instance, we may widen two intervals $[1,3]$ and $[8,10]$ to $[1,+\infty]$ which includes a large number of values, e.g., 5, not in the original intervals. □

In the worklist algorithm, an FSM is a set of transitions, $(S, \mathbb{E}_S, S')$, which is actually $(S, \mathbb{E}_S)$ together with the the path set S' analyzed in the next loop iteration. Intuitively, if we have state transitions $(S, \mathbb{E}_S, S'_1), (S, \mathbb{E}_S, S'_2), (S, \mathbb{E}_S, S'_3), \dots \in FSM$, it means that after executing a path $s \in S$ in a loop iteration, we will execute a path $s' \in \bigcup S'_i$ in the next loop iteration. Next, we discuss the soundness of $(S, \mathbb{E}_S, S')$ as follows.

Lemma 2 (Soundness of $(S, \mathbb{E}_S, S')$)**.** *If in a concrete execution, two consecutive loop iterations respectively execute two paths in the loop body, e.g., s and s', there must exist a state transition $(S, \mathbb{E}_S, S') \in FSM$ such that $s \in S$ and $s' \in S'$.*

Proof. By Lemma 1, the output environment of analyzing the path set $s \in S$ is sound, meaning that each abstract value in $\mathbb{I}_S$ over-approximates values in the concrete path s. Due to the over-approximation, using $\mathbb{E}_S$ as the initial environment, the next loop iteration must analyze a path set S' that includes s'. If S and S' are not further split into sub-states in Algorithm 1, we have $(S, \mathbb{E}_S, S') \in FSM$. Hence, the lemma is proved.

If S and S' are split into smaller sub-states, e.g., $s \in S_i$ and $s' \in S'_i$, Lines 9-11 in Algorithm 1 say that we still preserve the connections between S_i and S'_i. Hence, we have $(S_i, \mathbb{E}_{S_i}, S'_i) \in FSM$. The lemma is also proved. □

Given that the transitions inferred by Algorithm 1 is sound, we discuss the soundness of the whole FSM below.

Lemma 3 (Soundness of *FSM*)**.** *If a network message can be accepted by the loop under analysis, it can also be accepted by our inferred FSM.*

Proof. If the original program can accept an input message, then the input message will execute a sequence of paths, e.g., $(s_1, s_2, \dots, s_n)$, such that s_i is a path in the loop body and is executed in the ith loop iteration, and s_n is a path ending with an *exit* statement. Assuming that the exact path constraint (i.e., path constraint without over- and under-approximation) of each path s_i is Γ_{s_i}, we can write the exact path constraint of the whole input message as $\bigwedge_{i=1}^n \Gamma_{s_i}$.

By Lemma 2, for each pair of path (s_i, s_{i+1}), we can find a state transition $(S_i, \mathbb{E}_{S_i}, S_{i+1})$ such that $s_i \in S_i$ and $s_{i+1} \in S_{i+1}$. By Lemma 1, $\mathbb{E}_{S_i}$ is sound, meaning that the state transition from S_i to S_{i+1} is constrained by a sound path constraint Φ_{S_i} such that $\Gamma_{s_i} \Rightarrow \Phi_{S_i}$. Therefore, $\bigwedge_{i=1}^n \Gamma_{s_i} \Rightarrow \bigwedge_{i=1}^n \Phi_{S_i}$. This means that the input message also satisfies $\bigwedge_{i=1}^n \Phi_{S_i}$. Thus, the state transitions from the state S_1 to the state S_n can consume the whole input message.

Finally, due to SR2, S_n is a final state. Hence, our inferred FSM also accepts the input message. □

The completeness of our inferred FSMs can be discussed in a similar manner as below.

Lemma 4 (Completeness of *FSM*)**.** *Assuming we do not use any interval domain during our analysis, the inferred FSM is complete — if a message can be accepted by our inferred FSM, it can also be accepted by the loop under analysis.*

Proof. As discussed in the proof of Lemma 1, we only introduce over-approximation into the program environment in the third case when the interval domain is used. Hence, $(S, \mathbb{E}_S)$ is complete if the interval domain is never used. In this case, each state transition in the FSM, i.e., $(S, \mathbb{E}_S, S')$, is constrained by the exact path constraint. That is, we have $\forall s \in S, \Phi_s \Leftrightarrow \Gamma_s$ where Φ_s and Γ_s respectively denote the inferred and the exact path constraints of the path s. The transition constraint is denoted by $\Phi_S = \bigvee_{s \in S} \Phi_s$.

If our inferred FSM can accept a message, then there is a sequence of state transitions $(S_1, S_2, \dots, S_n)$ that can consume the message. That is, the message satisfies $\bigwedge \Phi_{S_i}$, i.e.,

$$\bigvee_{s_1 \in S_1} \Phi_{s_1} \wedge \bigvee_{s_2 \in S_2} \Phi_{s_2} \wedge \bigvee_{s_3 \in S_3} \Phi_{s_3} \wedge \cdots \wedge \bigvee_{s_n \in S_n} \Phi_{s_n}.$$

We can then pick one path s_i from each path set S_i such that the network message satisfies $\bigwedge \Phi_{s_i}$. Since $\Phi_s \Leftrightarrow \Gamma_s$ as discussed before, the network message also satisfies $\bigwedge \Gamma_{s_i}$. This means the loop under analysis can consume the network message via the path sequence $(s_1, s_2, \dots, s_n)$.

Finally, since S_n is a final state, SR2 ensures that $s_n \in S_n$ ends with a loop-exiting statement, meaning that the loop under analysis accepts the network message. □