Analyzing Operational Behavior of Stateful Protocol Implementations for Detecting Semantic Bugs

Endadul Hoque[†], Omar Chowdhury[*], Sze Yiu Chau[†], Cristina Nita-Rotaru[‡], and Ninghui Li[†]
[†]*Purdue University, West Lafayette, Indiana, USA.* e-mail: {mhoque,schau,ninghui}@cs.purdue.edu
[*]*University of Iowa, Iowa City, Iowa, USA.* e-mail: omar-chowdhury@uiowa.edu
[‡]*Northeastern University, Boston, Massachusetts, USA.* e-mail: c.nitarotaru@northeastern.edu

***Abstract*—Network protocol implementations must comply with their specifications that include properties describing the correct operational behavior of the protocol in response to different temporal orderings of network events. Due to inconsistent interpretations of the specification, developers can unknowingly introduce *semantic bugs*, which cause the implementations to violate the respective properties. Detecting such bugs in stateful protocols becomes significantly difficult as their operations depend on their internal state machines and the complex interactions between the protocol logic. In this paper, we present an automated tool to help developers analyze their protocol implementations and detect semantic bugs violating the temporal properties of the protocols. Given an implementation, our tool (1) extracts the implemented finite state machine (FSM) of the protocol from the source code by symbolically exploring the code and (2) determines whether the extracted FSM violates given temporal properties by using an off-the-shelf model checker. We demonstrated the efficacy of our tool by applying it on 6 protocol implementations. We detected 11 semantic bugs (2 with security implications) when we analyzed these implementations against properties obtained from their publicly available specifications.**

I. Introduction

Network protocol implementations must comply with properties usually described in informal prose specifications (*e.g.*, RFC standards), which often become highly complex for stateful protocols. For instance, the specification of the Transport Layer Security (TLS) protocol [1] dictates the chronological sequence (the temporal order) in which the client and the server must exchange messages to complete a handshake before establishing a secure connection. By not complying with specifications, implementations can cause incorrect operational behavior, interoperability issues, or critical security vulnerabilities, and thus result in noncompliance. Such noncompliance instances are primarily due to *semantic bugs* [2], which cause implementations to violate the properties (high-level functional specification) of the respective protocol and behave incorrectly.

Consider the "CCS Injection" vulnerability (CVE-2014-0224), an example semantic bug, in the TLS implementation of OpenSSL [3] where it accepts a `ChangeCipherSpec` (`CCS`) message even if the `CCS` does not appear in the prescribed order. By exploiting this vulnerability, a man-in-the-middle attacker can obtain sensitive information or hijack the connection completely. Semantic bugs are not only limited to secure protocol implementations for general-purpose operating systems (*e.g.*, Linux), but prevail also in implementations of non-secure protocols developed for resource constrained devices (*e.g.*, Internet-of-Things) [4].

Detecting semantic bugs through manual inspection of a stateful protocol implementation is a cumbersome and error-prone task because shared variables and protocol states complicate the interactions between the code fragments that handle different network events (*e.g.*, arrival of a packet, occurrence of timeout). As a result, dangerous semantic bugs can remain undetected for years. For instance, the CCS Injection bug was present in OpenSSL for more than a decade. Hence, it is crucial to build automated techniques to assist developers detect semantic bugs in their protocol implementations.

Some type of semantic bugs in protocol implementations are difficult to detect automatically by applying well-known techniques like fuzzing or software model checking in a straight-forward manner. We outline three primary reasons as follows: **(C1) Silent incorrect behavior:** Many semantic bugs do not display any externally discernible erroneous effect (*e.g.*, crash) but result in silent incorrect behavior – for example, accepting a packet unexpected in a particular context. This demands a precise analysis of the *internal* interactions between the protocol logic. **(C2) Delayed discernible effects:** In case a semantic bug produces some discernible effects, they are usually exhibited far away from the actual source location of the bug. This calls for precise identifications of the buggy execution paths of the protocol implementation. **(C3) History-dependent:** These bugs are triggered during stateful processing of network events. This emphasizes the need to check temporal behavior of the protocol that are sensitive to the history and the chronological ordering of network events.

Prior work on detecting bugs in protocol implementations used fuzzing [5], [6] to detect the CCS Injection bug. However, fuzzing tools are limited to detecting only bugs with externally discernible effects. They cannot detect bugs that have silent incorrect behavior, nor can they point out the location of the bug. Tools specifically designed for network protocols like [7]–[9] applied software model checking directly on protocol implementations to detect bugs that violate state invariants (*i.e.*, properties that must hold in all reachable states of the program, irrespective of any temporal behavior). Thus, these tools could not detect bugs causing violations of properties that depend on the chronological ordering of network events.

In this paper, we aim to automatically analyze a given protocol implementation and detect semantic bugs that violate properties describing the temporal behavior of the protocol; hence, we call them *temporal properties*. We observe that semantic bugs often lie in the code fragments that handle how the protocol implementation reacts to network events (*e.g.*, by changing the protocol's internal state or by sending a response). Such reactions of the protocol are described as finite state machines (FSMs) in informal prose specifications, either explicitly (*e.g.*, DHCP, TCP) or implicitly (*e.g.*, Telnet, TLS). While implementations intend to closely follow the specified FSMs, the informal descriptions of the FSMs often leave room for inconsistent interpretations giving rise to errors related to state machines. Hence, in our analysis, we concentrate on the event handling portions of the implementation, which encompasses the implemented FSM. Our approach allows us to detect semantic bugs that have silent incorrect behavior (**C1**) or delayed discernible effects (**C2**) as the implemented FSM allows us not only to observe the internal (possibly silent) interactions of the protocol but also to identify buggy execution paths. Our approach also allows us to detect bugs that are history dependent (**C3**) by reasoning about the temporal properties of the protocol related to the implemented FSM since semantic bugs often manifest during stateful processing of events.

Given an event-driven implementation of a stateful network protocol, we show, in this paper, that it is possible to automatically extract the implemented FSM from the source and use it to detect semantic bugs violating given temporal properties. We design and develop CHIRON,[1] an automated tool that enables a developer to check whether a protocol implementation violates the (user provided) desired temporal properties and thus detect the corresponding semantic bugs. In a nutshell, CHIRON's approach consists of two steps: (a) extracting the implemented FSM from the source (which we call the `E-FSM`), and (b) model checking the `E-FSM` against desired temporal properties.

For detecting semantic bugs, we must analyze the protocol FSM implemented in the protocol source by precisely capturing the relevant implementation details along with the chronological network events. Clearly, manual extraction is impractical and error-prone. Therefore, we devise an FSM extraction technique that takes as input the protocol source (written in C) along with some meta-information provided by the developer and outputs an approximated protocol FSM (*i.e.*, `E-FSM`) implemented in the source. Our technique is based on *symbolic execution* [10], which precisely simulates a program's execution with symbolic inputs and explores all possible execution paths. However, to circumvent the path-explosion problem often plaguing off-the-shelf symbolic execution tools, we devise a protocol state aware path exploration technique that dynamically prunes redundant execution paths.

We observe that the temporal properties of a protocol typically involves stateful processing of an arbitrarily long sequence of network events. Writing these properties as code snippets (perhaps, in C) is invariably ponderous as the snippets need to explicitly maintain the history of the protocol execution. Instead, we use *temporal logic formulas* [11] as they provide the flexibility of expressing temporal properties in a concise and fine-grained fashion. To check whether the `E-FSM` violates the temporal properties, we use an off-the-shelf model checker. In case of a violation, the model checker generates a counterexample (CEX) as evidence. A CEX is an execution of the protocol demonstrating the violation. Due to the abstractions in our analysis, the generated CEX may not be *realizable* in an actual execution of the protocol. Hence, we devise a validation technique to rule out such false CEXs.

We implemented CHIRON and demonstrated its efficacy by applying it to a total of 6 implementations of 3 protocols: one secure protocol (TLS [1]) and two non-secure protocols (Telnet [12] and DHCP [13]). For TLS, we used the implementation from the OpenSSL library developed for general purpose operating systems (*e.g.*, Linux). CHIRON's general approach allows us to apply it to protocols implemented for Internet-of-Things (IoT) devices. Therefore, for Telnet and DHCP, we used several implementations from two separate TCP/IP protocol stacks designed for IoT devices: uIP [14] and FNET [15]; they are widely used but have not been extensively studied. To evaluate, we used 6 properties for TLS, 11 for Telnet, and 7 for DHCP; all are derived from their respective RFCs and documentation. We discovered 11 semantic bugs in total, 2 of which have security implications.

Contributions. Our work shares a common vision with similar efforts advocating the application of formal methods to improve the security of systems [16] and makes the following technical contributions:

- We present an automated tool, CHIRON, to help developers detect semantic bugs in their protocol implementations when analyzed against the given temporal properties.
- We devise a technique that automatically extracts the `E-FSM` from the source of a stateful, event-driven protocol with minimal user guidance.
- We show the efficacy of CHIRON by testing 6 implementations of 3 protocols against 24 properties and by uncovering 11 semantic bugs, 2 of which have critical security implications.

II. Problem and Background

Detecting semantic bugs in a protocol implementation by checking for compliance with its *full* specifications is a long-standing challenging problem due to two reasons. First, protocol specifications (*e.g.*, RFCs), are not usually formalized in any form, leading to inherent ambiguities and leaving room for multiple interpretations [7], [17]. Secondly, a protocol implementation running on one host, interacts with other (possibly, remote) peer(s) to achieve its goals. Performing a joint analysis of all the peers has an amplified complexity due to diverse and independent implementations of the same protocol available in the wild.

We focus on checking the temporal properties that prescribe the correct operational behavior of the protocol in response to network events (*e.g.*, arrival of a packet, timeout). Semantic

[1]In Greek mythology, CHIRON was considered to be the wisest centaur

bugs violating such properties are due to *logical flaws* in the execution flow of the protocol implementation, which are different from low-level errors (*e.g.*, null dereferencing, memory leak). We analyze the implementation of only one peer (*e.g.*, client) of the protocol while considering the other peer (*e.g.*, server) as symbolic. Instead of *proving satisfaction*, we intend to *find violations* of the given properties in the protocol implementation—a common practice in software model checking [7], [18].

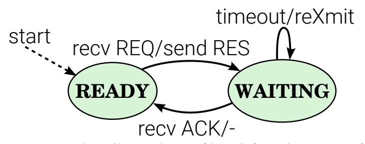
Protocol implementations. Interactions between protocol implementations rely on either client-server or peer-to-peer communication. Regardless of how they communicate, a protocol implementation typically follows the *event-driven programming* paradigm in which the implementation is centered on executing appropriate protocol logic (known as *event handlers*) in response to the occurred network events, causing the flow of the protocol execution to be determined by these events.

Such an implementation usually contains a main loop (called the *event loop*) that listens for a list of network events (*e.g.*, arrival of a packet, connection). These events are predefined by the underlying protocol stack (*e.g.*, TCP/IP) and often abstracted away from the developer via a socket API library. Some of these events notify about the incoming data (*e.g.*, a protocol message). The event handler that is invoked to process an event may send a message as a response by utilizing the underlying socket function (*e.g.*, send for TCP/IP). The handler may also update the protocol state, which consists of the values assigned to the variables encoding the semantic state of the protocol. We call these variables **state** variables.

In this paper, we focus on analyzing the source code of an event-driven implementation of a stateful protocol. For a client-server protocol (*e.g.*, TLS), we require the source code of the client (or server) implementation if the developer wants to analyze the client (or server) since we analyze only one end of the protocol in isolation. We also make no assumption about how the other peer would behave since we consider it as symbolic. In addition, we expect the provided source to have the event loop inside a function, which we call the **event dispatcher** function. As will be seen later, this function serves as the entry point for our analysis. Should there exist no explicit event dispatcher function (common for embedded systems), a test harness can easily be added to create one.

An example of a semantic bug. To illustrate the conceptual ideas behind our approach with a simple semantic bug, we use a fictitious but plausible example of a protocol (see Fig.1). The FSM specification (Fig.1(a)) describes that the protocol must start at the READY state, move to WAITING after receiving a REQ, and stay there until it receives an ACK. However, the implementation of the protocol (Fig.1(b)) violates the property when it fails to move back to READY after receiving an ACK due to a semantic bug (*i.e.*, the incorrect assignment to `state` at line 7). CHIRON extracts the underlying FSM implemented in the actual source and reasons about the temporal behavior of the protocol with respect to the extracted FSM.

Problem definition. Given an event-driven protocol implementation I and a temporal property φ, CHIRON aims to check whether I violates φ and detect the underlying semantic bug. CHIRON achieves this objective by extracting the implemented FSM (`E-FSM`) from I and model checking it against φ. CHIRON also ensures that each reported semantic bug is indeed reproducible in an actual execution of I.



(a) The protocol FSM described in the specification

```
1  void handle_recv_message(Packet_Ty *p) {
2    if (p->type == REQ && state == READY){
3      send_res(); /* send RES */
4      state = WAITING;
5    }
6    else if (p->type == ACK && state ==
         WAITING){
7      state = WAITING; /* Semantic bug */
8    }
9    else { /* ignore the packet */}
10 }
```

(b) Simplified code snippet from the implementation

Fig. 1: The implementation of our example protocol manifests a semantic bug (at line 7) violating with the specification

Extracted protocol FSM (`E-FSM`). We define an `E-FSM` M as a tuple $\langle Q, Ev, A, \mathcal{V}, q_I, R\rangle$. Q is a finite set of states $\{q_0, \ldots, q_n\}$, and $q_I \in Q$ is the initial state of the `E-FSM` M. We use Ev to denote a finite set of network events (*e.g.*, recv, which means the arrival of a network packet) and A to denote a finite set of high-level **actions** the protocol can perform utilizing the underlying socket functions of the protocol stack (*e.g.*, send_ack, which means sending an acknowledgment). Let $\mathcal{V}$ be a finite set of program variables. We assume that Ev, A, and $\mathcal{V}$ are pairwise disjoint.

Let R be the **transition** relation such that $R \subseteq Q \times Ev \times \mathcal{C} \times 2^A \times Q$, where $\mathcal{C}$ is a set of transition conditions such that each element $c \in \mathcal{C}$ is a quantifier-free first order logic formula [19] – with the theories of equality, bit vector, and array – over $\mathcal{V}$. If $\mathcal{V} = \{x, y\}$, then c can be, for instance, $x \geq 0 \land x + y \neq 10$. Since transitions are conditioned on the variables in $\mathcal{V}$, we call them **conditional** variables. In addition, each atomic formula of the transition condition (*e.g.*, $x \geq 0$) is called an **atom**. Given a transition $\langle q_a, \text{recv}, \text{buf\_len} \neq 0 \land \text{buf}[0] = 255, \{\text{send\_ack}\}, q_b\rangle$, it signifies that if the protocol implementation is currently at state q_a, the event recv is triggered, the receive buffer is not empty (*i.e.*, buf_len $\neq 0$), and buf's first byte is 255, then the implementation performs the action send_ack and moves to state q_b.

III. Design of CHIRON

In this section, we first give an overview of CHIRON's design followed by the details of its major components.

A. *Overview*

CHIRON's workflow consists of three major steps as shown in Fig.2. Given the protocol source and some meta-information (configuration files), the FSM extractor (❶) extracts the `E-FSM`

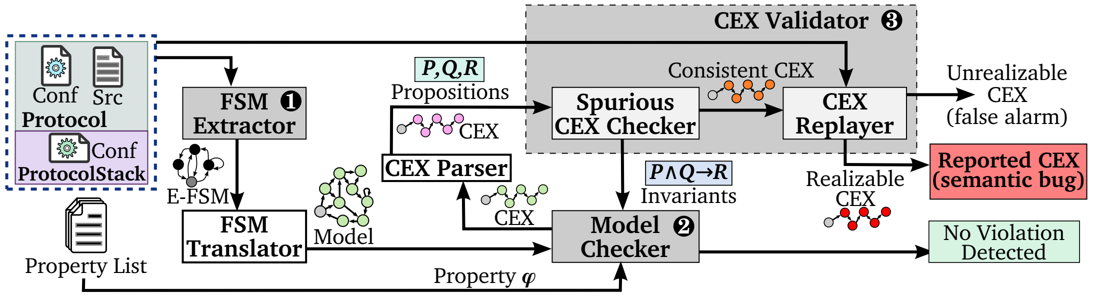


Fig. 2: CHIRON's workflow

(say M) of the protocol by leveraging symbolic execution. The details will be described in §III-B. Appendix A presents a brief introduction to symbolic execution.

Next, CHIRON utilizes a model checker (❷) that takes M and a temporal property (φ) of the protocol expressed as a propositional linear temporal logic (pLTL) formula [20]. Thus CHIRON tries to find an execution of M that violates φ. If a violation is detected, the model checker generates a counterexample (**CEX**) as evidence, which is essentially an execution of the protocol demonstrating the violation.

To employ any pLTL model checker with CHIRON, the FSM translator takes M as input and translates it to a high-level modeling language (*e.g.*, SMV [21]). Note that the conditions associated with the transitions of M are QF-FOL formulas over conditional variables, *e.g.*, $(x > 0) \wedge (x + y = 5)$. As pLTL model checkers expect the transition conditions to be boolean formulas, the translator maps each unique atom (*e.g.*, $x > 0$) in the conditions to a unique **propositional variable** (say, P) and stores the mappings in an **atom-proposition map** file, which is used later to automatically translate a temporal property φ in pLTL (see §III-C).

Due to the incompleteness of symbolic execution (*e.g.*, for unrolling loops for a small number of iterations to achieve termination [22], [23]) and due to the level of abstraction used in our analysis, the CEX generated by the model checker may not always be realizable in an actual execution of the given implementation. To rule out such **unrealizable** CEXs (*i.e.*, false alarms), CHIRON uses a 2-step CEX validation technique (❸). If a CEX passes both validation steps, CHIRON reports the CEX to the user as a **realizable** CEX which points out the underlying semantic bug causing the violation of φ (§III-D).

One additional module is employed by CHIRON: the CEX parser which interfaces the model checker with the CEX validator. It parses a CEX generated by the model checker and uses the atom-proposition map file to replace each proposition in the CEX with its corresponding atom. For instance, if the map file contains the mapping $P \mapsto \{x > 0\}$ and the CEX has $P \mapsto \mathsf{false}$, then the parsed CEX becomes $(x > 0) \mapsto \mathsf{false}$. The CEX validator inspects this parsed CEX (§III-D).

B. FSM Extraction

We now describe our `E-FSM` extraction algorithm.

Required meta-information. CHIRON needs the protocol source and two configuration files (meta-information) from the developer for extracting the `E-FSM`: one is specific to the protocol stack and the other is specific to the implementation.

The configuration file specific to the protocol stack contains information about all possible network events (*e.g.*, arrival of a packet, new connection, disconnection) that the protocol reacts to. For instance, if the protocol under analysis runs on top of TCP, CHIRON requires the list of network events that TCP can trigger. Obtaining such a list of network events for the underlying protocol stack is a one-time effort because the same list can be reused for other protocols developed for the same protocol stack. Such a list of events is often standardized for the protocol stacks of popular operating systems (*e.g.*, Linux, Windows) and well-documented for newer stacks such as uIP and FNET for IoT devices.

The configuration file specific to the protocol implementation consists of: (i) the names of the state variables that encode the protocol's state (*e.g.*, state in struct SSL); (ii) the name of the event dispatcher function (*i.e.*, the entry point to the event handling code); (iii) the names of the conditional variables (*e.g.*, the packet buffer buf), which should be considered in our analysis as variables that will use symbolic values; and (iv) a list of ⟨action_name, identifier⟩ pairs for each high-level action performed by the protocol (*e.g.*, sent_client_hello). Requiring users to provide such meta-information is a *common practice* to conduct symbolic execution of network protocols [22], [24]. Note that CHIRON's FSM extraction is agnostic to how the meta-information is obtained. Automatically extracting the meta-information with higher accuracy is left as future work.

The provided action list is not necessarily an exhaustive list of all protocol actions. In fact, the developer is free to specify the necessary actions relevant to the analysis. Hence, we provide a simple API (`void set_action (int identifier);`) that the developer can leverage to annotate the source code to indicate each desired protocol action. One of the benefits of this list is that it makes the `E-FSM` easy to comprehend. In addition, selecting conditional variables may seem difficult. Therefore, we apply a rule of thumb and that is to select any input or environment variables (*e.g.*, packet buffer) that can influence the state transition of the protocol.

Algorithm. Our algorithm (Algorithm 1) constructs the `E-FSM` M by searching for new FSM states and transitions in a breadth-first search (BFS) manner. While BFS is memory inefficient, it can find new states in a shorter time and can

Algorithm 1: FSM Extraction Algorithm

```
Input: The protocol source $S$, implementation specific
       configuration $C$, protocol stack configuration $N$
Output: The implemented FSM $M$ of the protocol
1  Queue $W_e \leftarrow \emptyset$; FSM $M \leftarrow \emptyset$;
2  Create initial program state $e_0$;
3  $q_0 \leftarrow \mathsf{ExtractFsmState}(e_0)$;  $//q_0$ : initial protocol state
4  $W_e.\mathsf{enqueue}(e_0)$;
5  $M.Q \leftarrow \{q_0\}$;   $//M.Q$ : set of states
6  $M.q_I \leftarrow q_0$;   $//M.q_I$ : initial state
7  $M.R \leftarrow \emptyset$;   $//M.R$ : set of transitions
8  Mark state $q_0$ as old;
9  while $W_e \neq \emptyset$ do
10     $e_i \leftarrow W_e.\mathsf{dequeue}()$;
11     $q_i \leftarrow \mathsf{ExtractFsmState}(e_i)$;
12     foreach Event $\tau \in \mathsf{EventList}$ do
13         $S_q \leftarrow \mathsf{SymbolicExecution}(S, C, N, e_i, \tau)$;
14         foreach $\langle e_j, c, a\rangle \in S_q$ do
15             $q_j \leftarrow \mathsf{ExtractFsmState}(e_j)$;
16             if $q_j$ is not old then
17                 $W_e.\mathsf{enqueue}(e_j)$;
18                 $M.Q \leftarrow M.Q \cup \{q_j\}$;
19                 Mark state $q_j$ as old ;
20             if Transition $\langle q_i, \tau, c, a, q_j\rangle$ is not old then
21                 $M.R \leftarrow M.R \cup \{\langle q_i, \tau, c, a, q_j\rangle\}$;
22                 Mark transition $\langle q_i, \tau, c, a, q_j\rangle$ as old ;
23 return $M$
```

also be parallelized. Our algorithm begins by constructing the initial program state e_0, which contains initialized values for both protocol state variables and other global variables. It then extracts the initial protocol state q_0 from e_0 using the function ExtractFsmState, which returns the underlying protocol state consisting of the values assigned to the state variables in the given program state. It marks q_0 as seen (*i.e.*, old) and adds it to M. It then adds e_0 to a working queue W_e.

The algorithm then processes each program state $e_i \in W_e$, one at a time in FIFO manner, until W_e is empty. It first extracts the associated protocol state q_i from e_i and then applies all events to e_i. For each event τ, it symbolically executes the implementation starting from the event dispatcher function by simulating the occurrence of τ. The symbolic execution returns all possible paths where each path is summarized by its associated program state e_j, path constraints on conditional variables c, and actions a performed by the protocol. For each possible path and its associated $\langle e_j, c, a\rangle$, the algorithm first extracts the protocol state q_j from e_j. If q_j is new, it inserts q_j into M, marks q_j as seen, and adds e_j to W_e. Finally, it checks whether it has seen the transition $q_i \xrightarrow{\tau,c,a} q_j$. If not, the algorithm adds it to M and marks it as seen. Once W_e becomes empty, the algorithm returns M as the E-FSM.

Protocol state driven path exploration. Traditional symbolic execution of a protocol implementation using an off-the-shelf symbolic execution tool can lead to path explosion as the tool aims to explore as many (possibly new) program states as possible to achieve a high code-coverage of the given implementation, often within a limited resource budget. Consequently, such exploration techniques are not suitable for our FSM extraction as they are not tailored to give precedence to the program states that encode new protocol states. We customize the path exploration technique of symbolic execution by restricting the exploration to only those program states that contain new protocol states (line 10 and 17 of Algorithm 1). Thus, we avoid the exploration of redundant execution paths.

C. Protocol Properties

Due to its proficiency in capturing the relative temporal order of events succinctly, we use pLTL to express the desired protocol behavior. Requiring developers to formulate the desired temporal properties in pLTL is very inconvenient. Instead, we envision developers to write the temporal properties in SALT (Structured Assertion Language for Temporal Logic) [25]. SALT is close to a high-level programming language, containing constructs for frequently occurring property patterns. In addition to being user-friendly and permitting users to express properties using regular expressions, SALT properties can be automatically translated to optimized pLTL formulas using its compiler. All the protocol properties we examined were written in SALT and then converted to pLTL.

We demonstrate the expressiveness of SALT by considering the following property from [25]: "*In a normal case, connection open event must be followed by zero or more receive data events and a connection close event unless a reset event occurs, which makes the requirement trivially satisfied*". This can be expressed in SALT as: `assert / connection_open; receive_data*; connection_close/ accepton reset`. Its pLTL counterpart is expressed as:
$(\texttt{connection\_open} \vee \texttt{reset}) \wedge ((\bigcirc\ ((\texttt{receive\_data} \vee \texttt{reset})\ \mathcal{U}\ (\texttt{connection\_close} \vee \texttt{reset}))) \vee \texttt{reset})$.

For model checking, we need the properties in pLTL format using propositions (see § III-A) while the pLTL generated by the SALT compiler contains a semantic name for each atom (*e.g.*, `connection_open`). We automatically translate the pLTL generated by SALT compiler to the pLTL format required by the model checker by utilizing the atom-proposition map (say, $\mathcal{M}_{\mathsf{AP}}$) generated by our FSM translator and an additional map ($\mathcal{M}_{\mathsf{AE}}$) between atoms and their semantic names. We require the developer to provide the map $\mathcal{M}_{\mathsf{AE}}$.

D. Validating CEX*s*

The CEX generated by the model checker can be unrealizable (*i.e.*, false alarms) due to the following reasons: **(i)** the abstraction caused by the replacement of atoms with propositions during FSM translation and **(ii)** the incompleteness of symbolic execution and the limited granularity of user input to FSM extraction. We use a 2-step CEX validation technique where Step 1 and 2 rule out the unrealizable CEXs generated due to reason (i) and (ii) respectively.

Step 1 (Spurious CEX checking). The propositions used in the translated E-FSM can have, possibly nontrivial, dependencies among each other with respect to their corresponding

```
void telnetd_appcall(void *ts) {
  if(uip_connected())/*A new connection?*/
  {
    if(!connected){
      s.state = STATE_NORMAL;
      connected = 1;
      ... /* other initialization code */
    }
    else{/* reject the new connection */}
  }
  ...
}
```

Fig. 3: Code snippet to explain an unrealizable CEX

atoms. The pLTL model checker is unaware of any such dependency. A CEX is a state sequence $s_1, \ldots, s_n$, where every state s_i maps each proposition to either true or false. The spurious CEX checker (see Fig. 2) inspects whether the truth assignments in each CEX state (s_i) agree with their atom-semantics (not the semantic names) using an SMT solver. If the assignments agree, the CEX is considered to be a **consistent** CEX. If they do not agree, we guide the model checker with an invariant that instructs it not to explore s_i in the future.

Consider the following proposition-atom mappings: $p \mapsto \{x > 0\}, q \mapsto \{x + y = 5\}, r \mapsto \{y < 5\}$. Let $p \wedge q \wedge r$ be a transition condition and s_i be a CEX state that assigns the values false, true, and true to the propositions p, q, and r, respectively. It is evident that if $(y < 5)$ and $(x + y = 5)$ are both true, $x > 0$ cannot be false. So the truth assignment $\neg(x > 0) \wedge (x+y = 5) \wedge (y < 5)$ makes the CEX unsatisfiable. The following invariant will be automatically generated to guide the model checker: $q \wedge r \rightarrow p$ (*i.e.*, p must be assigned true when both q and r are assigned true).

Step 2 (Replaying CEX). Due to the incompleteness of symbolic execution (*e.g.*, loop unrolling) and the limited granularity of user input (*e.g.*, incomplete input to Algorithm 1) to our analysis, the generated CEX may not be reproducible in an actual execution. Consider the Telnet server (Telnetd) implementation in Contiki-2.4 [26] as an example. One of the properties that the implementation is expected to comply with is: "*when the server has an ongoing connected session with a client, the server must reject any further connection requests.*" Due to a semantic bug in the implemented FSM, the server mistakenly accepts a new connection while there is an ongoing session, and thus, the implementation violates the property. We will explain this violation later in detail (see § VI-B).

Now consider the code snippet of the **patched** Telnetd implementation shown in Fig. 3. It fixes the bug by adding a guard variable connected so that the server accepts a new connection only if it is not already connected. Now, suppose the developer mistakenly assumes the protocol FSM state to be composed of only one variable (*i.e.*, s.state) and excludes connected. Since connected is neither a state variable nor marked as a conditional variable, the extracted E-FSM M lacks information about connected and has transition(s) that would allow M to accept multiple connections at a time. As a result, the property of allowing only one connection at a time would appear to be violated since the model checker would be able to generate a consistent CEX that would not be ruled out in Step 1. However, it is evident from Fig. 3 that such a CEX is not realizable in any actual execution of the protocol as connected guards the violation of the property.

To rule out such unrealizable CEXs, we concretely replay each consistent CEX and monitor the execution of the protocol implementation. The replay execution is guided by the CEX through concrete values obtained from an SMT solver. We monitor the performed actions and the changes of states during the execution and check whether it matches the CEX. If the protocol execution agrees with the CEX, we report this as a realizable CEX which reproduces the underlying semantic bug.

IV. Implementation

We now describe the implementation of CHIRON's components and present an optimization technique to preemptively rule out spurious transitions.

E-FSM extractor. We implemented the FSM extractor of CHIRON on top of the KLEE symbolic execution engine [27], which we use to symbolically execute the protocol source. We also implemented our path exploration technique. Our implementation is about 3.5 KLoC of C++ code in addition to the original KLEE code base.

FSM translator and model checker. We choose the NuSMV-2.5.4 symbolic model checker [28] because symbolic model checkers tend to support models with a large state space [21]. We implemented the FSM translator in C++ (400 LoC), which performs the following three steps: (i) parses the intermediate representation (*i.e.*, XML) of the E-FSM, (ii) generates the atom-proposition mapping file, and (iii) translates the E-FSM into the SMV modeling language.

Spurious CEX checker and CEX replayer. We implemented the spurious CEX checker in C++ (600 LoC) utilizing the libraries of KLEE. To replay each consistent CEX in the second step of validating CEXs, we implemented a CEX replayer that runs the source code with concrete values obtained from an SMT solver and keeps track of whether the protocol execution and the CEX agree at each step.

Optimization. For a given protocol stack (*e.g.*, TCP/IP), an application layer protocol depends on the underlying transport layer protocol (*e.g.*, TCP, UDP) to exchange messages between the peers. These protocols are typically developed with the assumption that the transport protocol is operating correctly. Some transport layer protocols can impose restrictions on the feasible ordering of the occurrence of possible network events. For example, a server running on TCP cannot receive any data prior to a connection establishment. Instead of checking the feasibility of an occurred network event, application protocols rely on the transport layer protocols to trigger each network event appropriately.

Recall that our FSM extraction algorithm applies all events to every state (line 12 of Algorithm 1). This can consequently result in many spurious transitions. To reduce spurious transitions, we allow the developer to provide an **event model** that merely dictates the feasible ordering of network events, which can be found in the protocol stack documentation. This *optional* optimization enables CHIRON to reduce the size of the extracted E-FSMs by leveraging developers' domain-knowledge. Applying this optimization can lead to a significant improvement (see §VI-C) for protocols developed for IoT devices as their TCP/IP stacks are often rudimentary compared to traditional TCP/IP stacks (*e.g.*, Linux). Instead of supporting conventional socket-level abstractions, these IoT stacks use several network events[2] to notify the applications. For instance, to analyze a client running on TCP of Contiki, CHIRON needs to explicitly consider 8 types of network events whereas considering only recv event (*i.e.*, arrival of a network packet) will be sufficient for the analysis of a client on TCP of Linux. Note that this optimization is applied to the occurrence of the network events (*e.g.*, recv event), and hence, this does not alter the data the events may carry (*e.g.*, a CCS message).

V. Discussion

We now briefly discuss some aspects of CHIRON.

What if protocol states are not encoded in program variables? CHIRON's FSM extraction requires the protocol states to be encoded in some program variables of the implementation. While a developer is free to implement a protocol's FSM by not encoding protocol states in any program variables, we observe that typically implementations of stateful protocols in the wild encode the states in program variables whereas the opposite is rare.

How about timers? Network protocols invariably use *timers* to conduct appropriate actions upon *timeouts*. There can be two types of timeouts: one triggered as an event by the underlying protocol stack and the other caused by firing off a timer maintained by the implementation. CHIRON handles both cases: the former as a network event and the latter as a library function (a stub of the timer library) that returns a symbolic boolean variable signifying the status of the timer.

How about assuring absence of semantic bugs? CHIRON is geared towards detecting semantic bugs by finding noncompliance instead of assuring compliance with a temporal property. CHIRON uses a model checker to determine whether the extracted E-FSM violates the property. Recall that the E-FSM is essentially an approximation of the protocol's implemented FSM. This is potentially a cause of false positive CEXs, which we rule out using the validation steps. Hence, any realizable CEX reported by CHIRON represents an actual semantic bug. Conversely, in case of no violation, CHIRON does not assure the absence of semantic bugs with respect to the property.

What kind of properties does CHIRON check? To be precise, CHIRON's discovery holds only for temporal safety properties. Checking liveness properties is challenging [29] and outside the scope of this paper. In the context of network protocols, temporal safety properties can be broadly categorized into two groups. (i) G1: Properties whose violations produce discernible external effects. An example from G1 is "*An* HttpResponse *from the server must be preceded by an* HttpRequest". (ii) G2: Properties whose violations produce only silent internal effects (*e.g.*, change in a state variable). An example from G2 is "*Upon receiving a* DHCPOFFER *in the REQUESTING state, the host must silently discard it and change no state*". While prior work [5], [30]–[32] can be tailored to check properties from G1 (but not G2), CHIRON can check properties from both groups.

What about false negatives? In our context, measuring false negatives is non-trivial due to the absence of ground truth about the number of semantic bugs present in an arbitrary protocol implementation.

[2]For instance, to provide TCP reliability Contiki requires the application to store unacknowledged outgoing packets since Contiki's TCP does not store them due to memory limitations. To retransmit a packet, Contiki notifies the application by using an additional network event (*i.e.*, reXmit). Application layer protocols are expected to handle all such events.

TABLE I: Protocol implementations tested

Protocol	Mode	Implementation	Protocol Notation
TLS 1.0	Client	OpenSSL 1.0.1g	TLS_OP
Telnet	Server	Contiki 2.4	Telnet_C24
		Contiki 2.7	Telnet_C27
		FNET 2.7.2	Telnet_F
DHCP	Client	Contiki 2.7	DHCP_C
		FNET 2.7.2	DHCP_F

VI. Evaluation

In this section, we demonstrate the effectiveness and the practicality of CHIRON by applying it to various protocol implementations. We seek to answer the following research questions: (a) Is CHIRON effective in detecting semantic bugs? (b) How much improvement can we gain by applying the optimization on the event model? (c) How much time does CHIRON require to analyze an implementation?

A. Setup

We applied CHIRON to several implementations of three application layer protocols: one secure protocol (TLS) and two non-secure protocols (Telnet and DHCP). TLS is widely used to secure network connections in various scenarios, including HTTPS. Telnet is a byte-oriented bidirectional communication protocol and often used as means to provide a command line interface for interacting with a (possibly remote) device. Telnet is still being used in the wild by Android and embedded systems' developers and also by Cisco network administrators. DHCP is a binary protocol that assigns IP addresses to devices on a network.

We obtained a total of 6 implementations of these protocols shown in Table I. We focus on the client implementation of TLSv1.0 and used one of the mainstream implementations: OpenSSL. We chose to use OpenSSL 1.0.1g as it was identified to contain the "CCS Injection" vulnerability (CVE-2014-0224) [5], [6]. To establish a secure channel, each TLS connection starts with either a full handshake or an abbreviated handshake. For our demonstration, we only consider a full handshake to be performed between the client and the server. Thus, we analyzed the portion of TLS_OP (see Table I) that implemented the finite state machine of TLS, which spanned across multiple source files (*e.g.*, `ssl/s3_clnt.c`, `ssl/s3_pkt.c`, `ssl/s3_both.c`, `ssl/s3_lib.c`).

To demonstrate the general applicability of CHIRON, we used various Telnet and DHCP implementations from different TCP/IP protocol stacks developed for IoT devices such as uIP (part of the Contiki OS) and FNET. In particular, we focus on the Telnet server and the DHCP client implementations developed for Contiki 2.4, Contiki 2.7, and FNET 2.7.2. We used Contiki 2.7 and FNET 2.7.2 because these were the latest releases at the time of evaluation. Contiki 2.4 came to our attention because of a bug reported in its Telnet server implementation [4]. In the remainder of the section we will use the notation defined in Table I to refer to an implementation.

B. Detected Semantic Bugs

To demonstrate the effectiveness of CHIRON in detecting semantic bugs, we used 6 properties for TLS, 11 for Telnet, and 7 for DHCP. Table II – IV show the properties along with the reports on violation. Given the lack of formalized and complete specification for these protocol implementations, we selected these sets of properties to cover diverse, but essential protocol functionalities.

The properties (OP1 – OP6) we selected for the TLS client (see Table II) are based on the correct chronological sequence of the protocol messages exchanged during a full handshake as dictated by the RFC [1]. A full handshake involves four flights of messages exchanged between the client and the server. The client first sends a `ClientHello`. The server responds with a series of messages starting with `ServerHello` and ending with `ServerHelloDone`. Between these messages, the server can send some optional messages (`ServerCertificate`, `ServerKeyExchange`, and `CertificateRequest`) depending on the parameters being negotiated. Next the client sends a series of messages ending with `ClientFinished`, and the server replies with `ServerFinished` to complete the handshake. However, both the client and the server must send a change cipher spec (`CCS`) message before their respective `Finished` message. Failing to comply with these properties can have critical security implications such as broken TLS guarantees and impersonation attacks.

The properties (DP1 – DP7) of a DHCP client (see Table IV) are all extracted from the RFC [13]. They govern how a DHCP client implementation must react to various received messages. In contrast, for the Telnet server, we selected the properties from various sources (see Table III). The TP1 property is specific to implementations that support only one active client session at a time. TP2 – TP4 are obtained from the Telnet RFC [12] and describe how an implementation must interpret and react to incoming data. Properties like TP5 – TP7, though not extracted from RFCs, are used to demonstrate how a developer can use CHIRON to reason about whether their implementation moves correctly between states as desired. Moreover, any telnet implementation must be able to operate as a Network Virtual Terminal (NVT), which is a bare bone implementation of the Telnet protocol where all options are disabled. Therefore, we derived four additional properties (TP8 – TP11) from the Telnet RFC specifically targeting NVTs.

We discovered a total of 11 semantic bugs: 1 in TLS_OP, 5 in Telnet_C24, 4 in Telnet_C27, and 1 in DHCP_C. Each bug signifies that CHIRON found a realizable `CEX` against the property in question. We now describe the discovered semantic bugs in detail. For brevity, we group the similar bugs together.

Bug 1 (Accepting early CCS during TLS handshake). According the RFC [1], during a handshake, the `CCS` message from the server (*i.e.*, `ServerCCS`) is expected to arrive at the client right before the `ServerFinished` message. Therefore, the client must not accept any `ServerCCS` received in out of order (denoted as property OP1). In our tests, CHIRON detects that TLS_OP violates OP1. The realizable `CEX` reported by CHIRON demonstrates that there exists an execution path where the client accepts and processes `ServerCCS` received (in out of order) right after receiving `ServerHello`. Along this execution path, the client accepts another `ServerCCS` received (in the correct order) right before receiving `ServerFinished` and eventually completes the handshake successfully.

A close inspection of the source of TLS_OP reveals that the client must receive `ServerCCS` right before `ServerFinished` to complete the handshake. However, TLS_OP does not restrict the client from accepting and processing `ServerCCS` received in out of order anytime after `ServerHello`. The implication of this semantic bug exacerbates because TLS_OP calculates the new keys upon receiving the first `ServerCCS` and does not recalculate the keys for any `ServerCCS` received later. A man-in-middle attacker can easily exploit this vulnerability to trigger the client calculate the new keys based on an empty master secret. As a result, the attacker can successfully break the guarantees of TLS (*e.g.*, confidentiality). This was first reported by Masashi Kikuchi as CVE-2014-0224 and later fixed in the following release of OpenSSL.

Bug 2 (Accepting multiple Telnet client connections simultaneously). According to the Telnet server documentation in Contiki, the server must not accept any new connection from a (possibly new) Telnet client during an ongoing session, which we denote as property TP1. In our tests, CHIRON generates a realizable `CEX` for Telnet_C24 demonstrating that the Telnet server accepts a new connection from a client even if there is an ongoing session. In fact, this semantic bug can manifest upon receiving any additional connection. This bug was, however, already reported [4] and later fixed in the following release of Contiki.

After a close inspection, we discovered that this semantic bug can have critical implications: (a) incorrect protocol behavior as the server re-initializes variables and (b) security issues as the server leaks data to the unauthorized client(s).

Bug 3 (No reply with appropriate Telnet command). Both Telnetd implementations from Contiki (Telnet_C24 and Telnet_C27) violate properties TP2 and TP3, which require that the Telnet server must reply the appropriate Telnet command if it receives `WILL` (for TP2) or `DO` (for TP3) from the connected Telnet client. The realizable `CEX` generated by CHIRON demonstrates that there exists an execution path in the corre-

TABLE II: Properties for TLS client and the report on violations (✗-mark signifies violation)

Property	Property Description	TLS_OP
OP1	The client must not accept any change cipher spec message from the server (ServerCCS) received in out of order during a handshake	✗
OP2	The client must not complete a handshake without receiving a ServerCCS from the server	
OP3	The client must not complete a handshake if the server skips the ServerHelloDone message	
OP4	The client must not complete a handshake when the ServerFinished message is received early (even before ServerHelloDone)	
OP5	The client must not accept a ServerCertificate message after accepting a ServerKeyExchange message	
OP6	The client must not accept a ServerKeyExchange message after accepting a CertificateRequest message	
	Total:	1

TABLE III: Properties for Telnet server and the report on violations (✗-mark signifies violation)

Property	Property Description	Telnet_C24	Telnet_C27	Telnet_F
TP1	The server must not accept any new connections during an on-going session	✗		
TP2	If receive WILL after IAC, must send DO or DONT	✗	✗	
TP3	If receive DO after IAC, must send back WILL or WONT	✗	✗	
TP4	If receive IAC IAC, must consume the 2nd IAC as regular data			
TP5	If receive IAC in NORMAL state, must go to IAC state and eventually go back to NORMAL state			
TP6	If receive DO after IAC, must go to DO state			
TP7	If receive WILL after IAC, must go to WILL state			
TP8	For NVT, if receive DONT after IAC, must NOT send WONT	✗	✗	
TP9	For NVT, if receive WONT after IAC, must NOT send DONT	✗	✗	
TP10	For NVT, never send DONT request			
TP11	For NVT, never send WONT request			
	Total:	5	4	0

TABLE IV: Properties for DHCP client and the report on violations (✗-mark signifies violation)

Property	Property Description	DHCP_C	DHCP_F
DP1	If receive DHCPNAK in REQUESTING state, must immediately start over DHCP negotiation	✗	
DP2	If receive DHCPOFFER in SELECTING state, must immediately send out DHCPREQ and move to REQUESTING state		
DP3	If receive no DHCPOFFER in SELECTING state and response timer expired, must resend DHCPDISCOVER		
DP4	If receive DHCPOFFER in REQUESTING state, must discard, change no state, take no actions		
DP5	If receive DHCPACK in REQUESTING state, must immediately move to BOUND state		
DP6	If receive no DHCPACK in REQUESTING state and response timer expired, resend DHCPREQUEST		
DP7	If receive no DHCPACK in REQUESTING state and state timer expired, start over DHCP negotiation		
	Total:	1	0

sponding implementation where the Telnet server fails to send back its response while the buffer (named telnetd_buf in the source) is full. In both implementations, the Telnet server uses this buffer to temporarily store all outgoing data including the Telnet command responses and sends the data over the network from time to time.

A careful inspection of the source reveals that the sendopt function of the Telnetd implementation does not check if it has failed to append the response command to the buffer; as a result, the server never sends back the response to the client. This semantic bug can cause an interoperability issue since the client would keep waiting for the reply from the server.

Bug 4 (Potential endless acknowledgment loops). Both Telnetd implementations from Contiki (Telnet_C24 and Telnet_C27) violate properties TP8 and TP9. According to the Telnet RFC [12], the protocol must acknowledge a DONT (resp., WONT) command by sending out a WONT (resp., DONT) only if the received DONT (resp., WONT) command causes a change in the current enabled options; otherwise, it must not acknowledge. This is needed to prevent potential endless acknowledgment loops where each party considers the incoming commands as new commands rather than acknowledgments. Since both Telnet_C24 and Telnet_C27 implement the Telnet server as NVT, they must not acknowledge any DONT/WONT command requests. For both implementations, CHIRON generates a realizable CEX, which demonstrates that the Telnet server actually replies back WONT (resp., DONT) when it receives a DONT (resp., WONT) command request from the client.

There are two possible scenarios where such endless acknowledgment loops can occur: (a) when the client allows multiple new requests about an option that is currently under negotiation and (b) if the server connects with a (possibly

TABLE V: Extracted E-FSMs by CHIRON. (**EM1** corresponds to the restricted event model described in § IV, and **EM2** considers all possible events with an arbitrary order.)

Protocol Notation	Event Model 1 (EM1)			Event Model 2 (EM2)		
	States	Transitions	Propositions	States	Transitions	Propositions
Telnet_C24	6	84	19	6	114	19
Telnet_C27	12	162	21	12	306	21
Telnet_F	7	18	11	7	34	11
DHCP_C	4	46	17	4	47	17
DHCP_F	8	80	45	8	140	45
TLS_OP	35	669	58	35	669	58

TABLE VI: Execution time (in **Seconds**) required by each component of CHIRON. ('–' means CHIRON found no CEX to replay.)

Protocol Notation	FSM Extraction	Property Checking	CEX Replay	Experiment Time
Telnet_C24	0.98	0.26	0.21	4.85
Telnet_C27	6.29	0.57	0.28	13.65
Telnet_F	0.16	0.15	–	1.91
DHCP_C	7.01	0.17	0.24	8.41
DHCP_F	15.09	0.55	–	18.96
TLS_OP	103.15	2.05	6.81	122.26

faulty) client that initiates a DONT/WONT request and also acknowledges the received DONT and WONT commands. Such loops can impair the performance of the IoT devices running either of these implementations.

Bug 5 (No immediate start over of DHCP configuration). According to the RFC [13], a DHCP client receiving a DHCPNAK message from the DHCP server as a response to its previously sent DHCPREQUEST message must immediately restart the DHCP configuration process by sending a new DHCPDISCOVER message (property DP1). In our analysis of the DHCP client implementation for Contiki (DHCP_C), CHIRON generates a realizable CEX demonstrating an execution path of the implementation that violates this property.

A close inspection of the source reveals that DHCP_C does not handle the reception of DHCPNAK messages. Instead, it keeps on retransmitting its DHCPREQUEST upon timeout for multiple times before giving up and then starts over the configuration process. Though this does not lead to any inconsistencies, it hinders the performance by continuing ineffective retransmissions, which can drain power constrained IoT devices as switching the radio on is a power-hungry operation.

C. Performance

Our experiments were run on a commodity machine equipped with an Intel Core i7-2620M CPU and 8GB of RAM, running Ubuntu 14.04 with Linux kernel 3.13.

Size of E-FSM. In Table V, we demonstrate the advantage of the optimization about enforcing the feasible order of network events (see §IV) by comparing the E-FSMs extracted for two event models: (a) Event Model 1 (EM1) corresponds to the user-provided restricted event model that considers only the feasible order of the occurrence of the network events in an actual execution of the protocol, and (b) Event Model 2 (EM2) represents the less restrictive event model where an event from the set of all possible network events can occur in any arbitrary order. For both models, the E-FSMs contain the same number of FSM states and propositions. However, the E-FSMs for EM2 have more transitions as expected. Most of them are spurious since they can never actually occur. In case of TLS_OP, the E-FSMs for both the event models have the same number of transitions since there is only one relevant network event (*i.e.*, recv meaning arrival of a packet) for TLS_OP.

Detection time. We evaluated the feasibility of CHIRON as a practical tool for detecting semantic bug through measuring the execution time incurred by its major components (see Table VI). In this experiment, we considered only the restricted event model (EM1). Each reported execution time is an average of 10 independent runs. Once the E-FSM is extracted, we can use it for detecting semantic bugs against an arbitrary number of properties. For property checking, we measured the total required time to model check all the properties (6 for TLS, 11 for Telnet and 7 for DHCP) until either a consistent CEX or no CEX is found. However, for comparison, we report only the average required time to model check per property (see Table VI). CEX replay time is measured only if CHIRON found a consistent CEX to replay. We report the time to replay per consistent CEX. Experiment time signifies the total time required to finish the entire analysis of each implementation.

Among the three Telnet server implementations, CHIRON requires the longest time (6 sec) to extract the E-FSM from Telnet_C27, which has a relatively larger E-FSM size (see Table V). The same trend is observed for the two DHCP client implementations. Note that CHIRON requires longer time for both the DHCP implementations compared to the Telnet implementations. The reason is that to analyze a DHCP client CHIRON processes a symbolic packet of size at most 552 bytes for each receive event as opposed to a Telnet server where it processes 1 byte at a time. Similarly, CHIRON requires about 100 sec to extract the E-FSM from TLS_OP, which is expected due to its large E-FSM size (see Table V).

The amount of time spent to model check each property is influenced by the E-FSM size, the length of the property, and the number of propositions. This trend can be noticed among the implementations we tested (see Table VI), with TLS_OP taking the longest time (2 sec) to model check a single property compared to the other implementations. Replaying a CEX takes a small fraction of time compared to the extraction of the respective E-FSM because CHIRON's CEX replayer drives the actual execution of the protocol along only one execution path. Finally, CHIRON finishes the complete analysis for each implementation within a few seconds (with a maximum of 2 minutes for TLS_OP).

VII. Related Work

We outline the prior work closely related to CHIRON.

Software model checking. CHIRON tries to automatically check whether a protocol implementation violates a given temporal property and thus detect the underlying semantic bug.

Software model checking [33]–[39] generalizes this for any program and safety properties. A software model checking approach can either be geared towards *finding violations* or *proving properties*. Based on the used underlying technique, software model checking approaches can be broadly categorized into two classes: *execution-based approaches* [33]–[35] and *abstraction-based approaches* [36], [37]. CHIRON follows the abstraction-based approach by abstracting the protocol implementation with an E-FSM. Typically, execution-based approaches cannot explore the entire state-space completely due to the state-space explosion problem whereas abstraction-based approaches suffer from spurious CEXs.

There is another class of software model checking approaches, namely *counterexample-guided abstraction refinement* (*CEGAR*) [38]–[40] which enjoys the advantages of both the above approaches by automatically generating abstractions of the program under analysis and refining the program (or, the model) when a spurious CEX is encountered. Although CHIRON, in principal, does not exactly follow the CEGAR approach, adding invariants based on spurious CEXs during model checking can be viewed as model refinement.

Protocol analysis. Prior work checks correctness of protocols by analyzing either manually formalized specifications [41], [42] or implementations in domain-specific languages [43]–[45]. However, these approaches cannot find bugs in the actual implementations. Holzmann *et al.* [46] requires a heavily annotated source and user provided rules (map the property in question to relevant statements in the source) to extract the abstract event-driven program model from the source. The model is then verified by utilizing a given non-deterministic test driver to simulate the necessary behavior of the external system. Contrarily, CHIRON uses symbolic execution to automatically extract the E-FSM of the protocol with little input from the user and does not require any test driver.

Several explicit-state model checkers (CMC [7], [8], NICE [9]) are used to verify protocol implementations against user provided state invariants, not temporal properties. While directly model checking the code can help them detect low-level programming errors, this can quickly lead to the state-space explosion problem. Contrarily, CHIRON focuses on temporal properties expressed in pLTL and check them against the extracted E-FSM using a symbolic model checker.

Several tools [31], [32], [47] have been developed by extending dynamic symbolic execution [27], [48] to analyze network protocol implementations; however, they cannot detect semantic bugs due to violations of temporal properties. While SymbexNet [31] can be tailored to test temporal behavior limited to only discernible effects (*e.g.*, exchanged messages), CHIRON can check properties even with silent internal effects. PIC [32] identifies non-interoperable implementations by finding any discrepancy between what the sender can send and what the receiver can receive. In contrast, CHIRON detects semantic bugs in implementations with respect to user provided temporal properties, often derived from RFCs.

Fuzzing has been another predominant approach to hunt for bugs in protocol implementations [30], [49], [50], including secure protocols like TLS [5], [6]. In essence, they rely on black-box testing and thus find bugs causing discernible external effects only (*e.g.*, crash, exchange of incorrect messages). In contrast to CHIRON, their capabilities are fundamentally different and complementary. While some fuzzing tools can be tailored to check temporal behavior with discernible effects, they cannot find semantic bugs causing silent incorrect behavior, nor can they identify the buggy execution paths.

In addition to the CCS injection bug, SmackTLS [5] finds some additional bugs in OpenSSL, which CHIRON cannot detect as they are not directly realizable through the TLS client's E-FSM. To detect those bugs, SmackTLS relies on two *manually* derived components: a state machine of TLS for generating test cases and a verified reference implementation of TLS to decide if the outcome of a test case is correct. Contrarily, CHIRON requires neither a state machine nor a reference implementation; instead, it relies on the desired temporal properties derived from RFCs and the developer provided meta-information of the protocol implementation.

Inferring protocol specification. Prior work aims to infer the protocol specification (FSM)—using network traces [51]–[54], using active queries [6], [55], using program analysis [24], [56], [57], or through model checking [36], [58]. These extracted FSMs represent either discernible external interactions of the protocol (*e.g.*, the sequences of exchanged messages) or the low-level program state machines (not E-FSMs). Contrarily, CHIRON extracts the E-FSM from the implementation by primarily capturing precise internal interactions of the protocol.

VIII. Conclusion

We presented an automated tool, CHIRON, to help a developer detect semantic bugs in an event-driven network protocol implementation by checking if the implementation violates given temporal properties. CHIRON first automatically extracts the E-FSM from the implementation by utilizing our FSM extraction technique based on symbolic execution and then uses a symbolic model checker to detect whether the E-FSM violates the properties. We demonstrated CHIRON's efficacy by applying it on 6 mature implementations of 3 protocols. CHIRON detected 11 semantic bugs violating the properties derived from the documentation and RFCs of the protocols. Our results demonstrate that CHIRON can be useful for developers to discover semantic bugs.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by the grant CNS-1600266 and CNS-1314688 from the SaTC program of the National Science Foundation. Its contents are solely the responsibility of the authors and do not represent the official view of the National Science Foundation.

References

[1] T. Dierks and C. Allen, "The tls protocol version 1.0," Internet Requests for Comments, RFC 2246, 1999.

[2] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Softw. Engg.*, vol. 19, no. 6, 2014.
[3] "OpenSSL toolkit," http://www.openssl.org/.
[4] "Contiki bug report," http://github.com/contiki-os/contiki/commit/d862e.
[5] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss *et al.*, "A messy state of the union: Taming the composite state machines of TLS," in *S&P*. IEEE, 2015.
[6] J. de Ruiter and E. Poll, "Protocol state fuzzing of tls implementations," in *USENIX Security*, 2015.
[7] M. Musuvathi and D. Engler, "Model checking large network protocol implementations," in *NSDI*, 2004.
[8] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill, "CMC: Pragmatic approach to model checking real code," in *OSDI*, 2002.
[9] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *NSDI*, 2012.
[10] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[11] B. Alpern and F. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, no. 3, pp. 117–126, 1987.
[12] J. Postel and J. Reynolds, "Telnet protocol specification," RFC 854, 1983.
[13] R. Droms, "Dynamic host configuration protocol," RFC 2131, 1997.
[14] A. Dunkels, "Full tcp/ip for 8-bit architectures," in *MobiSys*, 2003.
[15] "Fnet embedded tcp/ip stack," http://fnet.sourceforge.net/.
[16] S. Chong, J. Guttman, A. Datta, A. Myers, B. Pierce *et al.*, "Report on the NSF Workshop on Formal Methods for Security," *eprint arXiv:1608.00678*, 2016.
[17] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, "Delayed internet routing convergence," in *SIGCOMM*, 2000.
[18] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 21, 2009.
[19] G. Nelson and D. Oppen, "Fast decision procedures based on congruence closure," *J. ACM*, vol. 27, no. 2, 1980.
[20] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
[21] K. L. McMillan, *Symbolic model checking*. Springer, 1993.
[22] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," in *SIGCOMM*, 2011.
[23] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing api usages through semantic cross-checking," in *USENIX Security*, 2016.
[24] N. Kothari, T. Millstein, and R. Govindan, "Deriving state machines from tinyos programs using symbolic execution," in *IPSN*. IEEE, 2008.
[25] A. Bauer and M. Leucker, "The theory and practice of SALT," in *NASA Formal Methods*, ser. LNCS, 2011.
[26] "Contiki OS," http://www.contiki-os.org.
[27] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008, pp. 209–224.
[28] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore *et al.*, "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking," in *CAV*, 2002.
[29] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno *et al.*, "Ironfleet: Proving practical distributed systems correct," in *SOSP*, 2015.
[30] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: toward a stateful network protocol fuzzer," in *Information Security*. Springer, 2006.
[31] J. Song, C. Cadar, and P. Pietzuch, "Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications," *TSE*, vol. 40, no. 7, 2014.
[32] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein, "Analyzing protocol implementations for interoperability," in *NSDI*, 2015.
[33] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, "Zing: A model checker for concurrent software," in *CAV*, 2004.
[34] P. Godefroid, "Model checking for programming languages using verisoft," in *POPL*. ACM, 1997.
[35] G. Holzmann, "The model checker SPIN," *TSE*, vol. 23, no. 5, 1997.
[36] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu *et al.*, "Bandera: Extracting finite-state models from java source code," in *ICSE*, 2000.
[37] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," in *PLDI*, 2002.
[38] T. Ball and S. Rajamani, "The slam project: Debugging system software via static analysis," *SIGPLAN Not.*, vol. 37, no. 1, 2002.
[39] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast: Applications to software engineering," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5, 2007.
[40] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, 2000, pp. 154–169.
[41] K. Bhargavan, D. Obradovic, and C. Gunter, "Formal verification of standards for distance vector routing protocols," *JACM*, vol. 49, no. 4, pp. 538–576, 2002.
[42] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets," in *SIGCOMM*, 2005.
[43] E. M. Clarke, S. Jha, and W. Marrero, "Verifying security protocols with brutus," *TOSEM*, vol. 9, no. 4, 2000.
[44] S. Chaki and A. Datta, "Aspier: An automated framework for verifying security protocol implementations," in *IEEE CSF*, 2009, pp. 172–185.
[45] K. Bhargavan, C. Fournet, A. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *TOPLAS*, vol. 31, no. 1, 2008.
[46] G. Holzmann and M. Smith, "A practical method for verifying event-driven software," in *ICSE*. ACM, 1999.
[47] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment," in *IPSN*. ACM, 2010.
[48] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*. ACM, 2005.
[49] H. J. Abdelnur, R. State, and O. Festor, "Kif: a stateful sip fuzzer," in *IPTComm*. ACM, 2007.
[50] S. Jero, H. Lee, and C. Nita-Rotaru, "Leveraging state information for automated attack discovery in transport protocol implementations," in *DSN*, 2015.
[51] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *S&P*. IEEE, 2009.
[52] Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo, "Inferring protocol state machine from network traces: a probabilistic approach," in *ACNS*. Springer, 2011.
[53] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *ACM CCS*, 2009.
[54] C. Cho, D. Babić, E. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *CCS*, 2010.
[55] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining model learning and model checking to analyze tcp implementations," in *CAV*, 2016.
[56] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery." in *USENIX Security*, 2011.
[57] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *ACM CCS*, 2007.
[58] D. Lie, A. Chou, D. Engler, and D. L. Dill, "A simple method for extracting models from protocol code," in *ISCA*. IEEE, 2001.

Appendix A. Symbolic Execution

Symbolic execution is a program analysis technique that executes the code using *symbolic* values instead of concrete values (say, α instead of 2) for program inputs. After executing each program statement, the executor updates the symbolic store maintaining information about program variables (*e.g.*, $x = 5\alpha$). Special attention is given to handling branches (*i.e.*, if-else, loops). At each branch, the executor consults with a constraint solver (*e.g.*, an SMT solver) to determine the feasibility of the branch condition given the information in the symbolic store so that the executor can continue exploring only feasible branches. When both branches are feasible, the executor explores both of them, creating two different execution paths. Upon termination of the execution, the executor constructs a tree of all possible execution paths of the program. Each execution path is represented by a unique *path constraint*, which is the conjunction of branch choices that need to be made to follow the path. If necessary, each path constraint can be solved using a constraint solver to obtain concrete inputs for that path.