IoTInfer: Automated Blackbox Fuzz Testing of IoT Network Protocols Guided by Finite State Machine Inference

Zhan Shu, Guanhua Yan, *Member, IEEE*

***Abstract*—The popularity of IoT (Internet-of-Things) devices calls for effective yet efficient methods to assess the security and resilience of IoT devices. In this work we explore a new heuristic based on FSM (Finite State Machine) inference to guide generation of test cases for blackbox fuzzing tests of IoT network protocol implementations. Our method, which is called IoTInfer, balances exploration and exploitation by continuously monitoring how likely mutation of an input message leads to counterexamples conflicting with the prediction by the current FSM. IoTInfer also applies clustering techniques to coarsen the FSM inferred when there are limited computational resources provisioned for fuzzing tests. We implement IoTInfer for both Bluetooth and Telnet protocols, which are widely used by existing IoT devices. Our experimental results with a variety of IoT devices reveal that IoTInfer is efficient at generating meaningful test cases, some of which can expose previously unknown vulnerabilities or implementation deviations from protocol specifications. We also compare IoTInfer with two other state-of-the-art blackbox IoT device fuzzing tools and find that IoTInfer is better at eliciting different types of responses from the fuzzing targets.**

***Index Terms*—Device Security, Security and Privacy**

I. Introduction

Recent analysis of the cyber threat landscape has suggested that vulnerable IoT devices have become a primary attack target. Among the 1.2 million IoT devices analyzed by Palo Alto Networks, 57% of them were vulnerable to medium- or or high-severity attacks [37]. These vulnerable IoT devices are low hanging fruits for cyber attackers, as evidenced by F-Secure's recent report that the increase in attack activities captured by its global honeynet in the first half of 2019 from a year ago was largely driven by IoT-related traffic [32].

There is thus an urgent need for techniques that can assess the security and resilience of IoT devices effectively yet efficiently. Since IoT devices are essentially computing devices with networking capabilities, they are not immune to attacks from the Internet as long as they are externally accessible through some network protocols. This motivates us to explore how to perform fuzzing tests for network protocols implemented by IoT devices. As these devices are commonly shipped to the market with few implementation details (e.g., lack of source code or difficulty in extracting firmware), blackbox fuzz testing is often the only practical approach to understanding their security and resilience issues.

However, blackbox fuzzing can be extremely inefficient for vulnerability discovery because the test cases randomly generated from blackbox fuzzing often fail to pass validation tests by the fuzzing targets [31], [19]. Against this backdrop, this work explores a new heuristic based on FSM (Finite State Machine) inference to guide generation of test cases for blackbox fuzzing of IoT network protocols. Our intuition behind this new approach builds upon the observation that implementations of network protocols usually follow FSMs defined in their specifications. In contrast to random mutations of message fields values widely used in network protocol fuzzing, blackbox fuzzing guided by FSM inference aims to prioritize test cases that are more likely to cause state transitions in the fuzzing targets.

There has been a large body of research dedicated to FSM inference of network protocols [34]. Many protocol FSM inference algorithms, such as the L* algorithm [11], gradually approach the correct FSM of a target protocol through a sequence of membership queries and equivalence queries. Our hypothesis is that, when feeding new inputs to the target protocol in hope of eliciting counterexamples for FSM improvement, some of them may be unexpected by a specific implementation of the target protocol, thus exposing its abnormal behaviors with potential security risks. After the fuzzing tests, the FSM inferred can also be compared against the protocol's standard specifications, if they exist, to find questionable deviations.

Straightforward applications of existing FSM inference methods such as the L* algorithm [11] usually require predefined abstractions for input alphabets (*i.e.*, sets of input messages accepted by the target protocols) and output alphabets (*i.e.*, sets of output messages observed from the target protocols). However, ad-hoc workarounds of these issues can lead to undesirable results. For example, using a small set of valid seed input messages may undershoot the input alphabet, while random generation of input messages according to predefined message formats is likely to produce a large fraction of invalid ones immediately rejected by the target. Moreover, if the output alphabet is assumed to include all possible output messages from the target protocol, its size may be too large, making fuzz testing guided by FSM inference intractable.

Inspired by multi-armed bandit reinforcement learning [42], we propose a new method called *IoTInfer* to strike a balance between exploration and exploitation in generation of fuzzing tests. It mutates the fields in an input message randomly chosen from the current input alphabet (*exploration*) and monitors the responses from the target protocol. If the new message variants lead to counterexamples that conflict against the predictions by the current FSM, IoTInfer increases the likelihood of choosing the same message for mutation in the future; otherwise, if mutation of an existing valid message does not aid FSM improvement, IoTInfer lowers its chance of getting selected for future mutation (*exploitation*). Moreover, when there are only limited computational resources provisioned for fuzzing tests, IoTInfer can cluster both the input

Z. Shu and G. Yan are with the Department of Computer Science, Binghamton University, State University of New York, Binghamton, NY, 13902 USA email {zshu1, ghyan}binghamton.edu.

and output alphabets to coarsen the FSM inferred.

In a nutshell, our key contributions include the following:

- We develop a new heuristic based on FSM inference to improve efficiency of blackbox fuzzing tests of network protocol implementations by COTS IoT devices.
- We propose to dynamically adjust both the input and output alphabets with clustering methods in FSM inference to address lack of prior knowledge about acceptable input messages by the target IoT devices and also reduce the complexity of FSMs inferred due to various output messages observed from them.
- We implement IoTInfer as a generic blackbox fuzzing tool and evaluate its performance on the implementations of Bluetooth and Telnet protocols in various IoT devices. Our experiments show that IoTInfer is not only efficient at generating meaningful fuzzing test cases but also able to find previously unknown security issues in some of them.
- We compare the performance of IoTInfer against those of two other state-of-the-art blackbox IoT device fuzzing tools and find that IoTInfer is better at eliciting different types of responses from the fuzzing targets.

The remaining of this paper is organized as follows. Section II discusses related work. Section III presents the problem formulation and explains the key methodology of this work. Section IV introduces the general methodology applied by this work. Section V presents the key data structures and workflow of IoTInfer and Section VI discusses the algorithm details of its key components. Section VII elaborates on the implementation details. We show experimental results in Section VIII. We discuss the limitations of our work in Section IX and draw concluding remarks in Section X.

II. Related Work

In this section we review related works in the literature.

A. Generic IoT security

Many techniques have been developed to enhance IoT security due to its rising importance. They include, among many others, SDN (Software Defined Networking)-assisted hardening of IoT architecture [29], IoT security enforcement through wireless context analysis [28], secure configurations for IoT [41], secure thing-centered IoT communications [22], conflict detection in IoT systems based on formal methods [8], behavioral fingerprinting of IoT devices [13], graph-based IoT malware detection [10], record and replay that facilitate security testing of IoT devices [20], sensitive information tracking for commodity IoT devices [15], and context-aware security hardening for smart home systems [40]. These techniques have been summarized within multiple comprehensive surveys on the topic of IoT security, including potential solution to IoT security [39], IoT security from CISCO's seven-level reference model [33], legal aspects of IoT security [43], access control for IoT [36], IoT security taxonomy in the contexts of application, architecture, and communication [9], IoT vulnerabilities [35], and IoT fuzzing [19]. IoTInfer adopts a proactive approach to IoT security, which aims to discover and patch vulnerabilities in IoT devices before they are shipped to the market.

B. Fuzzing methodology

Existing fuzzing test methods can be classified into three types: blackbox fuzzing, whitebox fuzzing, and graybox fuzzing. A blackbox fuzzer like zzuf [1] does not need to access the source code or know the internal implementation details of the fuzzing target. Although it is easy to perform blackbox fuzzing through random generation of test cases, it can be extremely inefficient if most of these test cases cover only a small fraction of code branches in the target (e.g., rejection due to wrong formats). Whitebox fuzzing techniques [26], [24], [25] take advantage of the source code of the fuzzing target and use dynamic symbolic execution techniques to optimize test case generation. Graybox fuzzing does not require the source code, which makes it more practical than whitebox fuzzing in many real-world scenarios; balancing practicality and efficiency, graybox fuzzing tools such as AFL [4] often instrument the fuzzing targets and collect useful information such as code coverage to guide generation of new test cases dynamically.

For many COTS IoT devices whose internal implementation details are unavailable, it is difficult to perform whitebox or even graybox fuzzing tests on them. However blackbox fuzzing through random generation of test cases can be inefficient at discovering their vulnerabilities. This work explores a new heuristic based on FSM inference to guide generation of test cases for blackbox fuzzing of IoT devices. As the nature of IoT devices suggests that they must communicate with the external world through some network protocols, IoTInfer aims to find the sequences of network packets that allow the fuzzing target to enter previously unexplored states.

C. IoT device fuzzing tools

More related to IoTInfer are those tools that also apply fuzz testing to improve IoT device security. IoTFuzzer finds memory corruption vulnerabilities in an IoT device by modifying the program logic in its companion mobile application to mutate messages sent to the device [16]. Snipuzz is a recent blackbox fuzzing technique that infers the message snippets accepted by the target IoT devices and mutates message fields based on these snippets to generate test messages for vulnerability discovery [21]. Firm-AFL uses augmented process emulation, which leverages the high-fidelity of full-system emulation and high-performance of user-mode emulation, to achieve high-throughput greybox fuzzing of IoT firmware [45]. SweynTooth is a fuzzing platform for testing BLE (Bluetooth Low Energy) device security [23]; it adopts a mutation strategy based on particle swarming to generate test cases and depends upon manually constructed protocol state machines to capture invalid responses. IoTInfer differs from these previous efforts as it is applicable to IoT devices without companion mobile apps and guides generation of fuzzing tests through FSM inference. As IoTFuzzer and Snipuzz are two other state-of-the-art generic blackbox fuzzing tools for IoT devices, we shall compare the performances of their fuzzing strategies against that of IoTInfer later in Section VIII.

III. Problem Formulation and Assumptions

This work considers the following problem: *given the target protocol of an IoT device, how can we identify its security vulnerabilities or questionable implementation deviations from protocol specifications?* We assume that the inputs include authentication credentials to interact with the target protocol, request message formats, and a set of seed request messages. Each message format is abstracted as a list of tuples; each element in the list can be represented as a tuple (*field-name*, *field-len*, *field-type*), meaning the name, length, and type of the field, respectively. All the message formats from the input are abstracted as an ordered list $F = \{f_i\}$, where f_i is a particular message format represented as a list of tuples described above. The inputs also include a set of seed request messages, M, each of which instantiates one of the message formats in F.

We assume that the network protocol in the target IoT device can be modeled as a *reactive* system which processes incoming requests from an external party and then sends back its responses, if there are any. Like other protocol fuzzers [12], [27], the IoT protocol fuzzer continuously performs the following steps: generate a new request message, send the request message to the IoT device, and wait for the response message from the IoT device (or time out if there is no response from the IoT deivce after a certain period of time).

Sometimes the network protocol in the IoT device needs to authenticate the requesting party in order to continue the protocol. The availability of such an authentication credential to the fuzzer affects the exploitability of the vulnerabilities exposed by fuzzing tests. When configuring the fuzzer, the authentication credential provided to the fuzzer should mirror the threat model taken to exploit the vulnerabilities exposed from the fuzzing tests.

We assume that some prior knowledge is available to speed up fuzzing tests. First, the request message formats should be known to the fuzzer. Such formats usually can be found from the protocol specifications if the IoT device implements a standard protocol, or through reverse engineering efforts (e.g., Discoverer [18] and Polyglot [14]) when proprietary protocols are used. Second, some seed request messages should also be available to bootstrap the fuzzing tests. The seed request messages can be obtained by passively monitoring the traffic destined to an operational IoT device. Third, sometimes knowing the meaning of a specific field in the request message can be instrumental in speeding up fuzzing tests.

Regarding the response messages from the IoT device, our work does *not* assume any prior knowledge because different IoT devices may implement the same network protocol in different manners, including what responses should be sent back to the fuzzer.

IV. Methodology

We propose a new heuristic to guide blackbox fuzzing tests of IoT network protocols based on FSM inference. In our method, fuzzing test cases are continuously generated to explore the unknown behaviors of an IoT device through its network protocols, in hope of finding abnormal ones. As FSMs are widely used to design and implement network protocols, our method uses FSM inference to assist with test case generation. Moreover, the FSM inferred can be compared against the standard protocol specifications, if they exist, to expose questionable deviations implemented by the device.

For reactive systems such as network protocols, which do not have accepting or rejecting states, it is desirable to represent their FSMs as Mealy machines [17]. A Mealy machine is a sextuple $(Q, q_0, \Sigma, \Lambda, \delta, \lambda)$, where Q is a finite set of states, q_0 represents the initial state of the system, Σ is the input alphabet, Λ is the output alphabet, the transition relation $\delta : Q \times \Sigma \to Q$ maps the pair of a current state and an input symbol to the corresponding next state, and the output relationship $\lambda : Q \times \Sigma \to \Lambda$ maps the pair of a current state and an input symbol to the corresponding output symbol.

A. L* algorithm

The classical L* algorithm proposed by Angluin [11], [38] can be used to learn the Mealy machine of a reactive system. The L* algorithm operates on a data structure called an *observation table*. Assuming set concatenation $A \cdot B = \{ab | a \in A, b \in B\}$, the observation table is a tuple (S, E, T) with function $T : ((S \cup S \cdot \Sigma) \cdot E) \to \Lambda^+$, where S denotes a prefix-closed set of strings from Σ^* and E a non-empty suffix-closed set of strings from Σ^+. Note that Σ^* is the set of all possible strings over the input alphabet Σ while $\Sigma^+ = \Sigma^* - \{\epsilon\}$ where ϵ is the empty string.

The FSM can be constructed from the observation table (S, E, T) as follows. Each *unique* row in S represents a state. A transition from state q_1 to q_2 due to input symbol a exists if and only if for the row representing q_1 in S, denoted by s_1, we have a row in the bottom part of the table, $s_1 \cdot a$, that equals the row representing state q_2 in S.

The L* algorithm relies on two operations offered by an imaginary teacher to infer the FSM of a target protocol P: *(1) Membership query*: Given a sequence of requests, denoted by R (i.e., each $r \in R$ is a request message in Σ), a membership query returns a sequence of outputs from protocol P, denoted by O, where each $o \in O$ is an output symbol in Λ. *(2) Equivalence query*: Given a hypothesis model H, an equivalence query either acknowledges that H and P are equivalent or returns a counterexample C. Due to limited space we refer the readers to [11] for the details of the L* algorithm.

B. IoTInfer methodology

Before we can apply the L* algorithm to infer the Mealy machine of an IoT network protocol, we need to define the input alphabet Σ and the output alphabet Λ. Based on the algorithm inputs discussed in Section III, there are two ways of initializing Σ: we can treat M, the list of seed input messages, as Σ, or randomly generate a list of input messages based on message formats in F. If M is small with respect to the entire set of valid input messages accepted by the target protocol, the first method explores only a small portion of the FSM. On the other hand, random generation of input messages based on message formats in F produces many invalid input messages that are immediately rejected by the target IoT device, making

them useless to explore their protocol behaviors from deep program branches.

To overcome these challenges, IoTInfer strikes a balance between exploitation and exploration. It mutates fields in an existing message, monitors the responses from the target during a fuzzing campaign, evaluates the reward based on counterexamples conflicting with the current FSM inferred, and uses it to further improve the FSM. Applying a similar idea as multi-armed bandit reinforcement learning [42], IoTInfer uses these reward signals to adjust the likelihood of choosing the same message for mutation in the future.

As we do not assume any prior knowledge about output messages from the IoT device, we can let the output alphabet Λ of the L* algorithm include all possible message values. This naive approach, however, may incur high computational burden when the output message space is large. To make the problem tractable, IoTInfer uses clustering schemes to coarsen the FSM inferred. It groups similar output messages from the IoT device into the same output symbol; when necessary, it also merges multiple input messages into the same input symbol if they cause identical state transitions in the FSM.

V. Data Structures and Workflow

IoTInfer operates on two key data structures, *STree* (Structure Tree) and *VTree* (Value Tree), which are defined below. An illustrating example will be given in Section VI.

STree: The STree is derived from user inputs. Each node in the STree is called an *SNode*. In addition to the structural information (*e.g.*, node ID, parent node, and children nodes), each SNode also stores the following: (1) *length*: the length of the field represented by this node. (2) *type*: the type of the node, which is used to decide how the field should be mutated.

VTree: The VTree stores intermediate state information during the fuzzing process. Each node in the VTree is called a *VNode*. In addition to the structural information (*e.g.*, node ID, parent node, and children nodes), each VNode stores the following information: (1) *snode*: the pointer to the corresponding SNode in the STree. Hence each VNode also corresponds to a field in the request message. (2) *ranges*: the value ranges represented by this VNode. Each range should be continuous but there may be multiple ranges stored in the same VNode. (3) *futility:* a performance counter characterizing how futile fuzzing this node is for improving the FSM.

Each leaf node of the VTree also stores the following: (4) *sample*: a representative request message. (5) *cleader*: the cluster leader. If the *cleader* field points to this VNode itself, it means that the node represents a list of other leaf VNodes; otherwise, it points to another VNode, which is a cluster leader representing this node. (6) *(init_pos, init_neg, freshness)*: these performance counters, along with *futility*, are used by a Beta distribution to determine the likelihood of picking this leaf node for mutation.

For ease of presentation we define the following functions related to the VTree: $PATH(VTree, u)$, which returns the list of nodes from the root to node u in the VTree, and $LEAVES(VTree)$, which returns the entire set of leaf nodes whose *cleader* fields point to themselves in the VTree.

Algorithm 1: Fuzzing test guided by FSM inference

Input: F: request message formats, M: seed messages, P: target protocol
Output: FSM

function IoTInfer(F, M)
  Initialize $STree$, $VTree$, and FSM based on inputs F and M
  $X \leftarrow \emptyset$ ▷ X: masking rule set
  $tries \leftarrow 0$
  while $tries \leq \theta$ **do**
    $failures \leftarrow 0$
    $f \leftarrow$ a leaf node randomly chosen from $LEAVES(VTree)$ based on a Beta sampling scheme using leaf nodes' $init\_pos$, $init\_neg$, $freshness$, and $futility$ fields
    $f.init\_neg$++
    $f.freshness \leftarrow 0$
    for *each node* $u \in LEAVES(VTree)$ **do**
      if $u \neq f$ **then**
        $u.freshness$++
    $L \leftarrow PATH(VTree, f)$
    while L *is not empty* **do**
      $v \leftarrow$ head node extracted from list L
      if v *is mutable* **then**
        $R \leftarrow$ set of request messages generated by mutating v in message *sample* stored in leaf node f
        $Z \leftarrow \emptyset$
        while R *is not empty* **do**
          $r \leftarrow$ a message extracted from R
          for *each state* q *in* FSM **do**
            Send message r to target protocol P to obtain $o(q, r)$
            Use FSM to predict output $o'(q, r)$
            if $o(q, r) \neq o'(q, r)$ **then**
              $Z \leftarrow Z \cup \{(q, r)\}$
        if $Z \neq \emptyset$ **then**
          $failures$ ++
          for *each tuple* $(q, r) \in Z$ **do**
            Cluster output $o(q, r)$
          Update output masking rule set X
          Update $VTree$
    if $failures > 0$ **then**
      Rebuild FSM with the L* algorithm
      $tries \leftarrow 0$
      Perform input clustering if necessary
    else
      $tries$++
      for *each node* $u \in PATH(VTree, f)$ **do**
        $u.futility$++
  return FSM

Workflow. The workflow of IoTInfer is illustrated in Algorithm 1. From inputs F (request message formats) and M (seed messages), IoTInfer first initializes *STree*, *VTree*, and *FSM* (Line 2). The termination criterion of IoTInfer is decided by global variable *tries*. A fuzzing experiment is finished if there are at least θ rounds of fuzzing tests without any counterexample found against the current FSM inferred. IoTInfer performs fuzzing tests in an iterative fashion. In each iteration, it randomly picks *one leaf node* from the VTree using a Beta sample scheme and gets a representative request message from its *sample* field (Line 7). Let list L denote the sequence of nodes on the path from the root to the leaf node picked. Note that each node in the VTree (except the root node) represents a specific field in a request message. For every node v on list L, if its type suggests that the corresponding field is *mutable*, a set of request messages, denoted by R, is randomly generated by mutating the message field represented by v while keeping the other fields in the sample request message intact (Line 17).

For every request messages r in set R, IoTInfer tests what is the output from protocol P if r is sent to the protocol at every state q in the FSM learned so far (Lines 19-25). Towards this goal, for state q, the FSM is first used to derive a sequence of request messages that can trigger the target protocol to reach state q from its initial state q_0. During the test, all these request messages are first sent to protocol P sequentially, followed by message r. IoTInfer further checks whether the output, $o(q, r)$, received from protocol P at state q, is the same as what is predicted by the FSM, $o'(q, r)$. If the output differs, meaning that this input is a counterexample, the correct output $o(q, r)$ is added to list Z (Line 25), which is initialized to be empty after the mutation step (Line 18).

If for all the request messages created in set R and all the states in the FSM, the prediction is always correct by the FSM (*i.e.*, $Z = \emptyset$), then another set of request messages is generated by mutating the next node on list L (Line 15). Otherwise, the $failures$ variable is increased by one (Line 27). Next, given the new outputs from the tests, IoTInfer performs output clustering to simplify the output alphabet and then updates the VTree accordingly (Lines 28-31).

After all the nodes on List L have been considered to generate mutated request messages, IoTInfer checks whether there have been failures in predicting the output by the current FSM (Line 32). If there are, the FSM is retrained with the new observations, using the standard L* algorithm (Line 33), and the $tries$ variable is reset to be 0 (Line 34); if the new FSM is too large, the input symbols are clustered to reduce the complexity (Line 35). Otherwise, the $tries$ variable is increased by one (Line 37). If the number of tries without any failures exceeds a given threshold θ, the current FSM is returned by IoTInfer (Line 40). Otherwise, IoTInfer chooses another leaf node and the whole process stated above repeats.

VI. Algorithm details

In this section we explain some key steps of the algorithm shown in Algorithm 1 through a walking example.

A. Initialization (Line 2)

STree. The conversion from input F (the list of message formats) to an STree is straightforward. We first create a root node for the STree, whose *length* and *type* fields are set to be 0 and ROOT, respectively. Next for each format $f \in F$, we add a new path from the root node. The i-th node on the path is created based on the i-th tuple in f: its *length* and *type* fields are set to be the same as *field-len* and *field-type* in the tuple, respectively. Figure 1(1) shows a simplified STree derived from three message formats of the Bluetooth L2CAP protocol (more details will be given in Section VII.A).

VTree. The VTree is initialized to have the same structure as the STree, except that each VNode in it has different fields. In a VNode, its *snode* field points to the corresponding SNode in the STree. The *ranges* field of a VNode is initialized to cover the complete range for a given length. For each leaf VNode, its *cleader* field is initialized to be the node itself and its *sample* message is randomly chosen from the seed request messages of the same format (if there is no such seed message, the *sample* message is created by setting each of its fields with a random value uniformly chosen from the corresponding range). The *init_pos*, *init_neg*, *freshness*, and *futility* fields of each VNode are initialized to be 1, 1, 0, and 0, respectively. Following the same example, Figure 1(2) presents the VTree after initialization, whose VNodes have one-to-one correspondences with those in the STree.

FSM. The initial FSM is learned using the standard L* algorithm [11], assuming that the input alphabet is M which contains all the input seed request messages and the output alphabet includes all the raw outputs from the target protocol.

B. Leaf node selection (Line 7)

A leaf node is randomly chosen from the VTree at Line 7 based on Beta sampling scheme. The representative request message given by its *sample* field is mutated into a number of variants to test the prediction accuracy of the current FSM in each of its states. As such tests may be time consuming, it is desirable to select a leaf node that feeding its mutated messages to the target protocol is likely to produce counterexamples for further FSM refinement. Motivated by this intuition, we sample each leaf node in the VTree with a Beta distribution and the one with the highest *Beta score* is picked for mutation.

The Beta score of each leaf node is derived by sampling the Beta distribution whose PDF (Probability Density Function) is:

$$f(x; \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}, \qquad (1)$$

where $\alpha =$ *init_pos* + *freshness*, $\beta =$ *init_neg* + *futility*, and function $\Gamma(n) = (n-1)!$.

The four performance counters used to compute the Beta score are updated as follows: (1) *init_pos*: it is initialized to be 1 and does not change. (2) *init_neg*: it is initialized to be 1 and if the leaf node has been selected at Line 7, it increases by one (Line 8). (3) *freshness*: it is initialized to be 0. If the leaf node has been selected at Line 7, it is reset to 0 (Line 9); otherwise, it increases by one (Lines 10-12). (4) *futility*: it is

initialized to be 0. If the leaf node has been selected at Line 7 *and* the check result at Line 32 shows that variable *failures* is 0, the *futility* field of every VNode on the path from the root to the leaf node increases by one (Lines 34-35).

The intuition of the Beta score for leaf node selection can be explained as follows. If a leaf node has not been selected for a long time, it should be given a higher priority to be selected (increasing *freshness* and thus α). On the other hand, if the leaf node has been selected but mutation based on this leaf node does not improve the FSM through counterexamples, it should be given a lower priority to be selected (increasing both *init_neg* and *futility* and thus also β). For the Beta distribution, when $\alpha > \beta$, the sampled value tends to have a higher value, thus increasing the chance of selecting the leaf node; otherwise, the sampled value tends to be smaller, thus lowering the chance of choosing the leaf node for future mutation.

Following the previous example, the three leaf nodes shown in Figure 1(2) (i.e., v_3, v_6, and v_{10}) have the same α and β parameter values in Eq. (1), both of which are 1. Hence, they have an equal chance to be selected for mutation. Supposing that v_{10} is chosen, its *init_neg* field is updated to be 2 and its *freshness* field is reset to be 0. For both v_3 and v_6, their *freshness* fields increase by 1.

C. Mutation (Line 17)

After a leaf node has been chosen at Line 7 in Algorithm 1, the representative message stored in its *sample* field is mutated to create variants for fuzzing tests. For each field in this message, if it is mutable, it is randomly mutated by $\Delta_{mut}/(1+futility)$ times to generate that many variants, where Δ_{mut} is a configurable parameter for the maximum number of mutations per field, while keeping the other fields intact. The *futility* value used here is the one stored in the *futility* field of the corresponding VNode. Hence, if selection of a leaf node at Line 7 does not help improve the FSM, the number of mutations is reduced for all the nodes on its path. The *ranges* field stores the current value ranges for the VNode. When this node is under mutation, a variant is created by uniformly choosing a value from all the ranges stored at the VNode.

The *type* stored in the SNode determines whether the corresponding field is mutable. In the current implementation we rule out the following types for mutation: IDENTIFIER, ROOT, PORT, and ECHO. The reason for the first two is straightforward. Fuzzing the destination port number loses the network connection, and fuzzing an ECHO field (the same contents are returned from the target protocol) expands the FSM rapidly without new useful information.

Following the previous example, only nodes v_8 and v_9 on the path from the root to the chosen leaf node v_{10} are mutable.

D. Prediction (Lines 28-29)

The message variants created from mutation (stored in set R in Algorithm 1) are used to find counterexamples with outputs contradicting with the predictions made by the current FSM. For each message $r \in R$, it is used to test every state in the FSM. That is to say, for every pair (s, r) where s is a state in the FSM and $r \in R$, if the output observed from the target protocol $o(s, r)$ differs from that predicted by the FSM, $o'(s, r)$, we add (s, r) to list Z. If list Z is empty, meaning that message variants created from mutation of the current VNode do not yield any counterexample, the next VNode on the path is considered to generate new fuzzing test cases.

It is noted that for the observation table of the L* algorithm, a state is represented as a prefix string in input alphabet S. Each input symbol in a prefix string corresponds to an instantiated request message. Hence, to derive $o(s, r)$, we can append message r to the prefix string, which is sent to the target protocol sequentially to get its output. This output is compared against the FSM's prediction $o'(s, r)$ to see if they

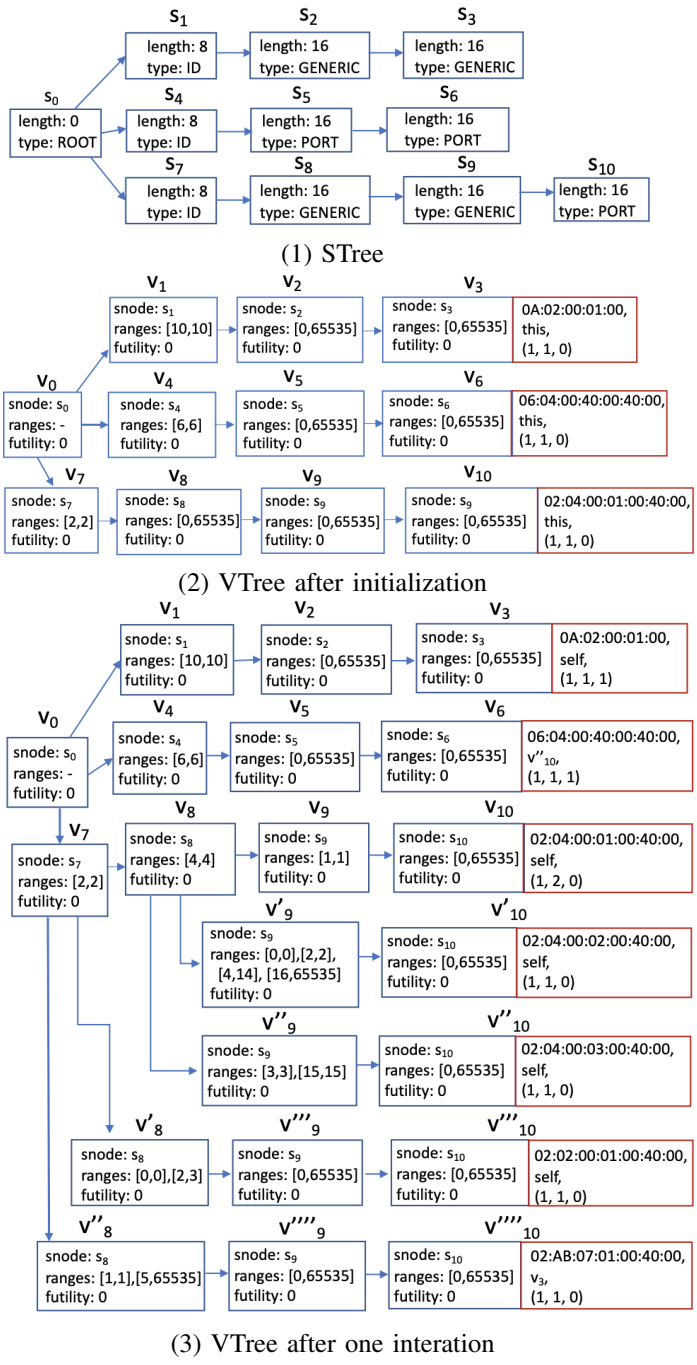


Fig. 1. An illustrating example. The right box of each leaf node includes fields *sample*, *cleader*, and *(init_pos, init_neg, freshness)* in order.

are the same. However, we may not be able to find $o'(s, r)$ directly from the observation table, because it is possible that the new message r, which is a variant of the *sample* message stored at the chosen leaf node at Line 7, may not belong to the input alphabet Σ for the FSM learned. Recall that the FSM is initialized only with input symbols representing seed messages from the input. Therefore, for output prediction, we always use the *sample* message stored at the chosen leaf node, r_0, to look up the observation table and find the corresponding output $o'(s, r_0)$ for comparison.

In our example, if no counterexamples are found after mutating either v_8 or v_9, the *futility* fields of both v_8 and v_9 should increase by 1. Supposing that this is not the case, we do not need to update the *futility* values.

E. Output clustering (Lines 28-29)

A fine-grained FSM trained with all the raw request messages as the input alphabet and all possible outputs from the target protocol as the output alphabet may be large and dense, incurring high computational overhead. Therefore, IoTInfer coarsens the resolution of the FSM with a clustering approach, which groups similar outputs and input messages to reduce the output alphabet Λ and the input alphabet Σ, respectively.

IoTInfer uses a simple method to cluster outputs based on set Z. IoTInfer decides to merge some outputs of *the same length* together when the number of distinct outputs exceeds threshold Δ_{output}, where Δ_{output} is a configurable parameter. It is noted that all the outputs in set Z are observed by mutating a single field in the same sample message stored at the leaf node selected at Line 7. The algorithm for output clustering is given as below. For each distinct length l of outputs in Z, if the number of distinct outputs exceeds threshold Δ_{output}, we search k continuous bytes replacing which with wildcard $*$ can reduce the number of distinct outputs the most. We start with single byte replacement (i.e., $k = 1$). If replacing none of any k continuous bytes with wildcard $*$ makes the number of distinct outputs below threshold Δ_{output}, we consider replacement of $k+1$ continuous bytes with wildcard $*$ that can reduce the number of distinct outputs the most. The process repeats until the number of distinct outputs is reduced to below threshold Δ_{output}.

The result of output clustering is a set of masking rules, each of which is represented as a tuple (l, x) where l is the length of an output message and x a regular expression with k continuous bytes masked with wildcard $*$. IoTInfer keeps a global rule set X, which contains all the masking rules discovered. Set X is accumulative: for each output observed from the target protocol (Line 22 or 33), it is first matched against each masking rule in X. If it matches rule (l, x), the output is replaced with x for comparison at Line 24 or used by the L* algorithm at Line 33.

F. Updating VTree (Line 31)

After output clustering, the VTree is updated as follows. Based on the masked outputs, we can partition the original ranges stored at the VNode under mutation into continuous ranges, each producing the same output regardless of the state. For each input cluster derived, we create a sibling VNode, whose *ranges* field is assigned to the ranges covered by the input cluster. We also remove these ranges from the *ranges* field of the VNode under mutation. The new node's *snode* is the same as that of the current one. Moreover, from this sibling VNode, new descendant VNodes are also created to mirror the path to the leaf node chosen at Line 7; for each of these descendant VNodes, its *ranges* field is set to cover the full range dictated by its length and its *snode* field points to the same one stored in the corresponding VNode. For the newly created leaf node, its *sample* field can store any message variant leading to the input cluster and its *cleader* field points to this new leaf node itself. For each VNode created, its *init_pos*, *init_neg*, *freshness* and *futility* fields are always initialized to be 1, 1, 0, and 0, respectively.

In our example, mutation of node v_9 leads to two counterexamples with conflicting observations from the sample stored in node v_{10}. As seen in Figure 1(3), we create two sibling paths starting at node v'_9 and v''_9, respectively, move value ranges with different observations from node v'_9 to either v'_9 or v''_9, and choose a representative mutated message as the sample stored for each of the new leaf nodes (i.e., v'_{10} and v''_{10}). Similarly, mutation of node v_8 also causes different observations and correspondingly, two respective sibling paths starting at node v'_8 and v''_8 are added to the VTree.

G. Rebuilding the FSM (Line 33)

We rebuild the FSM using the L* algorithm [11]. The input alphabet Σ includes all the sample messages stored at those leaf nodes of the VTree whose *cleader* fields point to themselves. The output alphabet includes all the masked output messages from the target protocol.

H. Input clustering (Line 35)

When the FSM trained at Line 33 is so complex that further fuzzing tests based on it become computationally prohibitive, IoTInfer applies input clustering, which merges multiple input symbols in Σ into a single one to coarsen the FSM. Note that the input alphabet of the FSM consists of those sample messages at the leaf nodes in the VTree whose *cleader* fields point to themselves. Hence, the effect of input clustering is to make the *cleader* fields of some leaf nodes point to others.

The precondition for performing input clustering is that the number of input symbols Σ exceeds a user-configurable threshold, Δ_{input}. Two input symbols a and b are merged together if the following two conditions are satisfied: (1) The two columns corresponding to suffix string a and b are exactly the same in the observation table, which means that at any state, the target protocol returns the same output, no matter whether request message a or b is received. (2) For any row corresponding to prefix s in the upper part of the observation table (i.e., $s \in S$), the two rows corresponding to $s \cdot a$ and $s \cdot b$ in the observation table must be exactly the same; that is to say, the target at any state must go to the same next state if a or b is received. When two input symbols represented by two leaf nodes are merged together in the VTree, the *cleader* field of one leaf node is modified to be the other.

In the same example, suppose that input clustering merges leaf node v_6 with v''_{10} and v''''_{10} with v_3. Their *cleader* fields are updated accordingly as seen in Figure 1(3). Hence, the FSM constructed after this iteration includes five sample messages stored at those leaf nodes with their *cleader* fields being *self*; they are used as the input alphabet in the next round.

VII. Implementations

This section presents implementation details for the Bluetooth and Telnet network protocols widely used in IoT devices.

A. Bluetooth

L2CAP fuzzing. For Bluetooth, IoTInfer currently focuses on fuzzing its L2CAP (Logical Link Control and Adaptation Protocol) on the target device. L2CAP deals with data multiplexing, segmentation and reassembly of packets, and QoS (Quality of Service) control for upper layer protocols. The operation of L2CAP is based on channels, whose endpoints are identified with *Channel IDs* (CIDs). According to the Bluetooth Specification [5], a few fixed CIDs should be supported by each Bluetooth device, including L2CAP Signaling channel (CID = 0x0001) and L2CAP LE Signaling channel (CID = 0x0005). The L2CAP signaling packets transmitted in C-frames (control frames) over these two channels, whose formats are shown in Figure 2. A signaling packet includes a basic L2CAP header and an information payload. The *code* field of the information payload decides the command type and its *id* field is used to match request and response commands transmitted between two L2CAP entities.

IoTInfer performs fuzzing tests on seven types of L2CAP signaling commands shown in Figure 2. Only request commands are considered because they can trigger internal state changes in the other L2CAP entity on the target Bluetooth device. Moreover, the *ECHO* request command is not included because it causes state explosion in FSM inference without producing meaningful results for security analysis. For the *configuration request* command, the Bluetooth Specification Version 4.0 includes seven options, one or more of which can be packed into the same command [5].

Implementation details. IoTInfer, which is implemented as a C library with Python binding, builds upon the HCI (Host-Controller Interface) and L2CAP sockets provided by BlueZ, the official Linux Bluetooth protocol stack [6]. To initiate fuzzing tests on a Bluetooth device, IoTInfer first establishes a Bluetooth L2CAP socket provided by BlueZ with the target. This is done by providing *PF_BLUETOOTH* and *BTPROTO_L2CAP* as the domain and protocol parameters, respectively, to the standard C *socket* function. Using this L2CAP socket, a sequence of request messages chosen from the seven types of signaling commands shown in Figure 2 is sent to the target device. Between any two consecutive request messages, there is a delay of 0.5 seconds for receiving the response messages, which is implemented by calling the *setsocketopt* function to set a delay value for option *SO_RCVTIMEO*. If the corresponding response does not arrive within 0.5 seconds, it is assumed that there is no response from the target protocol. For each response message received,

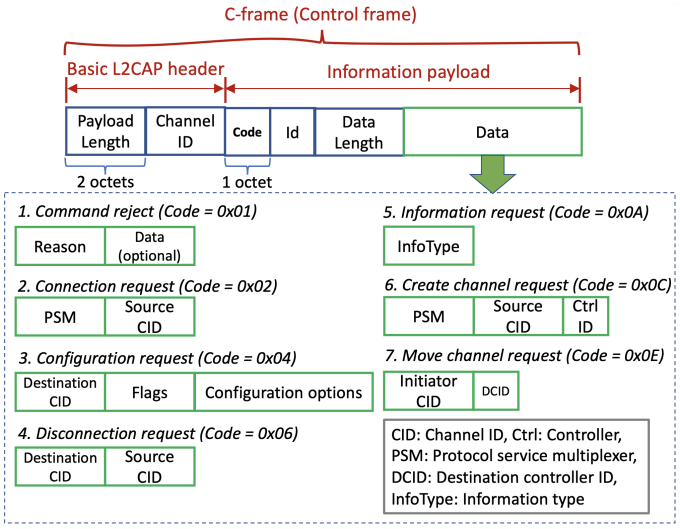


Fig. 2. L2CAP signaling commands

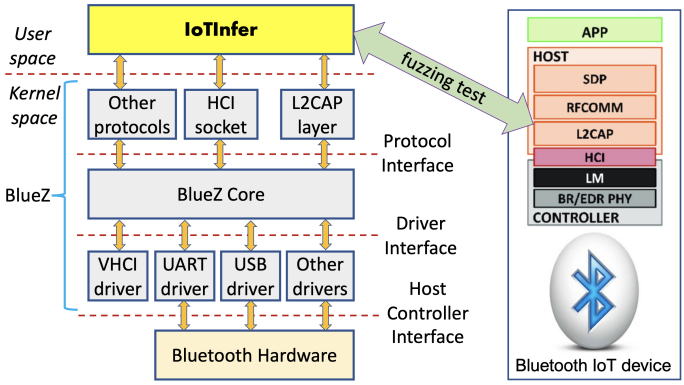


Fig. 3. IoTInfer implementation for Bluetooth L2CAP fuzzing

IoTInfer uses its *Id* field (see Figure 2) to match the request message; everything after *Id* is deemed as the output of the target protocol.

After finishing each test case, IoTInfer resets the L2CAP socket with the target device. This should not be done by using the *disconnect request* command because this command, which shuts down only the channel instead of the L2CAP connection, can be part of a test case (see Figure 2). Therefore, IoTInfer resets the HCI connection as follows. First, IoTInfer calls *hci_open_dev*(HCI0) to establish an HCI socket connection with the microcontroller on the only local Bluetooth adapter, which is identified as *HCI0*, of the machine where IoTInfer runs. Next, using function *ioctl*, IoTInfer extracts low-level connection information associated with the adapter. Finally, IoTInfer calls function *hci_disconnect* to terminate the low-level connection with the target device.

When malformed input messages freeze or crash the target device, we need to reboot the device so fuzzing tests can continue. Towards this end, we use a smart plug to control the power of the target device. In our current implementation, if IoTInfer fails to establish an HCI connection with the target three times in a row, it sends a control message to the smart plug to reboot the device.

TABLE I
LIST OF TELNET COMMANDS FUZZED BY IOTINFER WHERE FIELD ⋆ IS MARKED AS MUTABLE.

Command Type	Command Format: (field name, field length (octet))	RFC
WILL	[(IAC, 1), (WILL, 1), **(⋆, 1)**]	854
WON'T	[(IAC, 1), (WON'T, 1), **(⋆, 1)**]	854
DO	[(IAC, 1), (DO, 1), **(⋆, 1)**]	854
DON'T	[(IAC, 1), (DON'T, 1), **(⋆, 1)**]	854
Data entry terminal	[(IAC, 1), (SB, 1), (DET, 1), **(⋆, 1)**, (IAC, 1), (SE ,2)]	732
Negotiate window size	[(IAC, 1), (SB, 1), (NEG, 1), **(⋆, 1)**, **(⋆, 1)**, (IAC, 1), (SE,1)]	1073
Terminal speed	[(IAC, 1), (SB, 1), (TERMINAL-SPEED, 1), **(⋆, 1)**, (IAC, 1), (SE,1)]	1079
X display location	[(IAC, 1), (SB, 1), (X-DISPLAY-LOCATION, 1), **(⋆, 1)**, (IAC, 1), (SE, 1)]	1096
New Environment	[(IAC, 1), (SB, 1), (NEW-ENVIRON, 1), **(⋆, 1)**, (IAC, 1), (SE, 1)]	1572
Terminal type	[(IAC, 1), (SB, 1), (TERMINAL-TYPE, 1), **(⋆, 1)**, (IAC, 1), (SE, 1)]	884

B. Telnet

Telnet, which is widely used for remote administration of IoT devices, has caused severe security damages in the past [2], [3]. The implementation of IoTInfer for Telnet builds upon a TCP socket connection to port 23 on the target. We consider only Telnet commands, each of which starts with an IAC (Interpret As Command) character, 0xFF. Table I summarizes the list of Telnet commands fuzzed by IoTInfer.

The meaning of each command can be found in the corresponding RFC (Request for Comments), which is given in the last column of Table I. The first four command types are used to negotiate options: the sender indicates its intention about the option given in the ensuing field as WILL (willing to do), WON'T (unwilling to do), DO (agree to accept), and DON'T (refuse to accept). The final six command types are used for particular subnegotiations, each of which is enclosed between IAC+SB (Subnegotiation Begin) and IAC+SE (Subnegotiation End). For each Telnet command, IoTInfer only mutates the option/suboption value fields, which are marked as ⋆ in Table I.

VIII. EXPERIMENTS

This section presents our experimental results. Table II summarizes the devices used. In all our experiments, we have $\theta = 5$, $\Delta_{mut} = 100$, $\Delta_{output} = 5$, and $\Delta_{input} = 20$. When testing each Bluetooth device, for each L2CAP signaling command shown in Figure 2, the inputs fed to IoTInfer include its format as well as one example as the seed. When testing the Telnet protocol, we perform fuzzing experiments on the Telnet commands listed in Table I with a seed example for each command. For both Bluetooth and Telnet, IoTInfer does not know any credentials for remote connections with the target devices. We run IoTInfer on a Lenovo ThinkPad P50 laptop with 2.6GHz Intel i7-6700HQ CPU and 32GB RAM.

A. Input alphabet

In Section IV-B we argued that when applying the L* algorithm, neither the seed messages nor randomly generated input messages based on the message formats should be used as the input alphabet. To support this argument, we perform two experiments on the Bluetooth protocol of the Dell laptop.

TABLE II
LIST OF DEVICES TESTED

Protocol	Device Type	Vendor	Device Model
Bluetooth	Laptop	Dell	Inspiron 15 7000
	Raspberry Pi	CanaKit	Raspberry Pi 3b+
	Keyboard	ARTECK	HB030B
	Android phone	Samsung	Galaxy 10
	Speaker	Sony	SRS-XB12
Telnet	Laptop	Dell	Inspiron 15 7000
	Router	NETGEAR	R6230 (flashed with a PandoraBox firmware)

In the first experiment, we use the seed messages provided from the input as the input alphabet and execute the L* algorithm implemented by LearnLib, a comprehensive tool for automata learning [7]. The FSM inferred includes only two states, corresponding to the CLOSE and CONFIG states in the Bluetooth specification [5], eight outputs, and 10 edges. By contrast, IoTInfer obtains a much larger FSM, which includes five states, 121 edges, 25 input symbols, and 27 output symbols after running the algorithm for 25 iterations. The difference is because although LearnLib can try different combinations of input symbols (i.e., seed messages) from the input alphabet to infer the FSM, it does not mutate the fields of these seed messages to generate new input symbols.

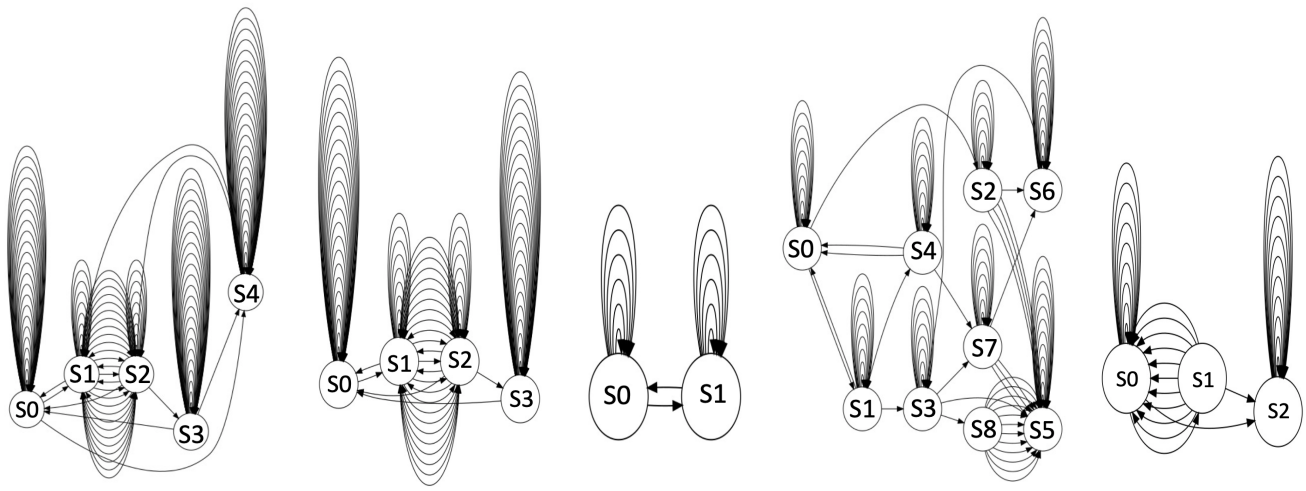
In the second experiment, we randomly generate 6000 L2CAP signaling commands, each using a message format uniformly chosen from the seven ones shown in Figure 2 and with values at mutable message fields uniformly chosen from their corresponding ranges. We monitor the response from the Dell laptop for each of these 6000 messages and find that only three of them have triggered observable output messages from the target. If we use these 6000 input messages as the input alphabet and apply the L* algorithm, a great amount of computation would be wasted on membership queries that do not generate any responses from the target. If the W-method or the partial W-method is used for equivalence testing, the number of tests needed can also be affected because it includes a factor of $|\Sigma|^{d(d+1)/2}$ where d is the difference in the number of states between the FSM inferred and the target [30].

B. Effectiveness results

1) Bluetooth: The Mealy machines inferred from the five Bluetooth devices are shown in Figure 4.

DoS (Denial of Service) attacks. We have found three types of DoS attacks against the Bluetooth devices, which are summarized in Table III. The details are explained below.

(1) *Temporary disconnection*: Bluetooth is temporarily unavailable on the target system through command *hciconfig HCI0*, which prints the basic information about the given Bluetooth adapter (HCI0). We have found an L2CAP *move channel request* message (e.g., "0e:19:01:00:40:00:01") that can disable Bluetooth for about half a second on both the Dell laptop and the Raspberry Pi. As we cannot run the *hciconfig* command on the other three Bluetooth devices, it is unclear whether the same attack holds against them.



(1) Dell laptop (2) Raspberry Pi (3) Keyboard (4) Android phone (5) Sony Speaker

Fig. 4. Mealy machines inferred from the five Bluetooth devices. We have removed the input/output label on each edge for readability.

TABLE III
POSSIBLE DOS ATTACKS AGAINST THE FIVE BLUETOOTH DEVICES FOUND FROM THE FUZZING EXPERIMENTS

Device	Temporary Disconnection	Permanent Disconnection	System Freezing
Dell Laptop	✓	✓	✓
Raspberry Pi	✓	✗	✓
Keyboard	?	✗	✗
Android phone	?	✗	✗
Sony Speaker	?	✓	✗

(2) *Permanent disconnection*: It is possible to disable Bluetooth permanently on the target device through some L2CAP command sequences unless the device is rebooted. For the Dell laptop with kernel 4.14.97, if the fuzzer first connects with and then disconnects from it, the target machine allocates a new USB number for the Bluetooth adapter; this operation is repeated for multiple times to make the new number allocated exceed a maximum value so Bluetooth is disabled unless the target machine is rebooted. When a newer kernel version is used on the Dell laptop, this problem does not exist.

For the Sony speaker, the attack involves a sequence of operations: (a) set up a Bluetooth socket connection to the speaker; (b) send L2CAP command "02:06:04:00:01:00:40:00" to the speaker, which means an L2CAP connection request with the identifier to be 0x06 and the PSM (Protocol and Service Multiplexer) field to be 0x0001 (Service Discovery Protocol); (c) send L2CAP command "02:07:04:00:03:00:40:00" to the speaker, which means a L2CAP connection request with the identifier to be 0x07 and the PSM field to be 0x0003 (Radio Frequency Communication); (d) call *hci_disconnect* to disconnect from the speaker. If this sequence is repeated by at least 16 times within a short period of time (around 40 seconds in our experiments), it exhausts all the 16 channel IDs available to the speaker and therefore no other device can connect to it (although the speaker can still be discovered).

(3) *System freezing:* We have observed two types of system freezing scenarios for both the Dell laptop and the Raspberry Pi. (a) CPU fully loaded: The CPU usage becomes 100%, and all USB devices are frozen. On the Dell laptop, the touch screen does not work but the keyboard is still functional, which allows us to use the *top* utility to check the system usage; for the Raspberry Pi, the system's CPU is fully loaded for about 10 minutes before it becomes totally unresponsive. Tracing the root cause of this issue, we have found that the error occurs inside function *bt-sock-poll* of file *af-bluetooth.c*, which is called repeatedly on the target machine. To establish a normal L2CAP socket connection, both entities expect to get the configuration response from the other side. In one test case, the fuzzer does not send any configuration response to the target machine so the socket is never established completely. Hence, the L2CAP entity on the target is always in a listening state when calling function *bt-sock-poll*; when a certain condition is satisfied, the target machine gets into a freezing situation as mentioned above. (b) CPU is not fully loaded but the system is freezing: This happens only to the Dell laptop. The system becomes totally frozen unless it is rebooted. From the system logs, we have found that the CPU was not fully loaded before it became frozen. Before the system became unresponsive, it called *printk* to print out a large buffer of unprintable characters to the log file.

Deviation from Bluetooth specification. (1) The Bluetooth specification states that a device receiving an L2CAP_ConfigReq message in a CLOSE state should go to the CLOSE state again [5]. For the Dell laptop, however, if the message's *length* field is 0 and *continuation* flag is 1, the target goes to an unspecified state where it rejects any new packets. We have to disconnect the device from the HCI layer to reset the L2CAP connection. (2) In the Bluetooth specification, there are many substates for a device in a CONFIG state. When state transitions are only driven by input messages from the fuzzer, the device can only be in a substate of either WAIT_CONFIG_REQ_RSP or WAIT_CONFIG_RSP. Regardless of which substate the device is in, it should respond to an L2CAP_ConfigReq message with an L2CAP_ConfigRsp message of either success or rejection. However, for both the Dell laptop and the Raspberry Pi, if the target receives

an L2CAP_ConfigReq message with a large non-matching length in its configuration options, it can enter a weird state where it responds to some L2CAP_ConfigReq messages but not others. (3) The Bluetooth specification states that a device receiving an L2CAP_ConnectReq message in a CLOSE state should reply with an L2CAP_ConnectRsp message indicating success, pending, or refused. For the Android phone, however, if the connection request is for BNEP (Bluetooth Network Encapsulation Protocol) where the PSM value is 0x0F, the target device does not send back any response; at this time, the target cannot be in a CONFIG state because it does not respond to any further L2CAP_ConfigReq messages or in a CLOSE state because the channel ID is still occupied by the current L2CAP connection.

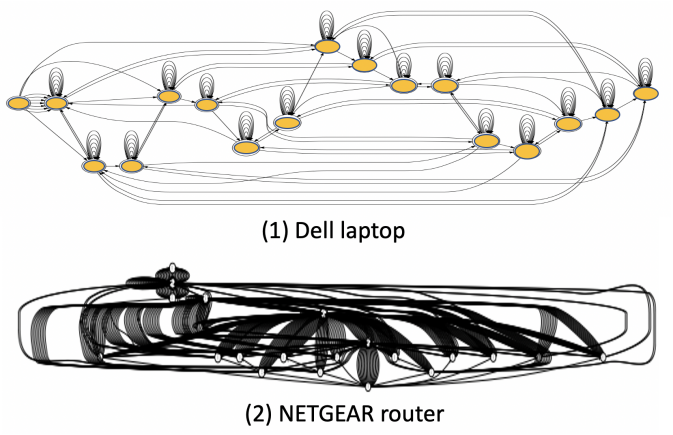


Fig. 5. Mealy machines inferred from the two devices running Telnet. Labels are removed for readability.

2) Telnet: The Mealy machines inferred from the two devices running Telnet are shown in Figure 5. Although our fuzzing experiments did not reveal any externally observable failures (e.g., crashes) of the two devices, we observe some implementation deviations from the Telnet specifications.

The Telnet server on the Dell laptop performs as expected according to the specifications. When an IAC command is received, it either replies with another IAC command or does not respond. For example, the target responds to command "IAC DO 0x00"/"IAC WILL 0x00" with "IAC WILL 0x00"/"IAC DO 0x00" because the target can support binary transmission (option 0x00), but for command "IAC WILL 0x2e"/"IAC DO 0x2e", it replies with "IAC DON'T 0x2e"/"IAC WON'T 0x2e" because it does not support TLS (option 0x2e).

In contrast, the FSM inferred from the NETGEAR router is much denser than that of the Dell laptop. For each received command, the router's reply, if there is any, may *not* be another valid IAC command. The FSM includes four types of states: starting state, negotiation state, user name state where the router is expecting a user name for login, and password state where the router is expecting a password. When the first command is received, the router always sends back command "IAC DO ECHO IAC DO WINDOWSIZE IAC WILL ECHO IAC WILL SGA", which may or may not be followed by a prompt for user name. From the starting state, the target may transition to a user name state with a prompt for user name, or a negotiation state where the target's behaviors vary with the commands further received. Based on the FSM inferred, it is clear that the implementation of the Telnet server on the NETGEAR router does *not* follow strictly the Telnet specifications listed in Table I.

C. Efficiency results

1) Benefits of input/output clustering: We measure the execution time in seconds per iteration with and without input/output clustering when IoTInfer is used to perform fuzzing tests on the Bluetooth L2CAP protocol of the Dell laptop and the Samsung Android phone. For the laptop, we run each experiment for ten iterations and for the Android phone, we stop each experiment after five iterations.

From Table IV we observe the following. (1) The execution time per iteration shows high variation across different runs. This results from the randomness in selecting a command stored in a leaf node of the VTree at Line 7 in Algorithm 1: it takes much longer time to fuzz test some commands (e.g., configuration request) than the others. (2) On average an iteration of fuzzing tests on the Android phone takes longer time to finish than the Dell laptop, suggesting that the execution time of fuzzing tests varies with the IoT device under evaluation. (3) Most importantly, input/ouput clustering significantly accelerates the fuzzing tests. When clustering is disabled, none of the five runs has finished ten iterations for the Dell laptop and only one run has finished all five iterations for the Samsung Android phone, because an experiment stops when an iteration takes more than 10 hours to finish. By contrast, when input/output clustering is enabled, all the five runs in our experiments have successfully finished ten (five) iterations for the Dell laptop (the Samsung Android phone).

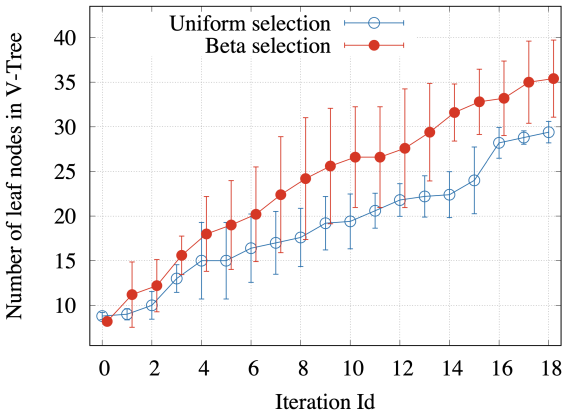


Fig. 6. Performance comparison between uniform selection and Beta selection

2) Benefits of Beta sampling for leaf node selection: IoTInfer uses a Beta sampling scheme to select a leaf node. For brevity, we call it a *Beta selection* scheme. To understand its benefits, we compare the numbers of leaf nodes in the VTree at the end of each iteration against those when a *uniform selection* scheme is used. As a new leaf node is created only if there are mispredictions by the current FSM (i.e., Lines 26-31 in Algorithm 1), having more leaf nodes in the VTree implies better effectiveness in generating meaningful test cases.

TABLE IV
COMPARISON OF EXECUTION TIME IN SECONDS WITH AND WITHOUT INPUT/OUTPUT CLUSTERING. ITERATION -1 MEANS THE INITIALIZATION STEP. AN ENTRY MARKED WITH '−' MEANS THAT THE TIME IS NOT MEASURED ANY MORE BECAUSE THIS ITERATION OR ANY OF ITS PREVIOUS ONES TAKES MORE THAN 10 HOURS TO FINISH, AND AN ENTRY MARKED WITH '×' MEANS THAT THE TIME IS NOT MEASURED AFTER FIVE ITERATIONS. THE TWO COLUMNS WITH CLUSTERING ENABLED SHOW THE MEAN EXECUTION TIME AND THE STANDARD DEVIATION OVER FIVE RUNS.

Iteration Id	Dell laptop (Bluetooth)						Samsung Android Phone (Bluetooth)					
	With clustering (mean/std)	Without clustering					With clustering (mean/std)	Without clustering				
		Run 1	Run 2	Run 3	Run 4	Run 5		Run 1	Run 2	Run 3	Run 4	Run 5
-1 (init)	221.4 / 106.2	170.7	427.2	171.7	448.1	172.1	1182.4 / 58.4	1080.3	1160.9	1328.3	1305.1	1221.4
0	222.3 / 80.5	–	182.2	202.7	–	254.7	878.2 / 110.9	757.6	–	834.5	944.9	1164.7
1	127.1 / 122.8	–	–	355.4	–	307.8	1839.6 / 1692.0	12909.7	–	–	–	48.6
2	230.4 / 130.5	–	–	43.7	–	–	4792.5 / 5888.9	–	–	–	–	1055.0
3	186.3 / 90.8	–	–	184.7	–	–	2280.7 / 2006.8	–	–	–	–	39.5
4	3565.3 / 2943.6	–	–	91.1	–	–	3039.0 / 4192.7	–	–	–	–	21605.8
5	2465.9 / 2303.5	–	–	262.7	–	–	×	×	×	×	×	×
6	1360.6 / 1356.1	–	–	–	–	–	×	×	×	×	×	×
7	2509.5 / 3858.2	–	–	–	–	–	×	×	×	×	×	×
8	881.9 / 752.9	–	–	–	–	–	×	×	×	×	×	×
9	1030.0 / 1238.1	–	–	–	–	–	×	×	×	×	×	×

The performance comparisons are observed from fuzzing tests on the Bluetooth L2CAP protocol of the Dell laptop. Figure 6 depicts the mean and standard deviation of the number of leaf nodes over five runs at the end of each iteration for both uniform selection and Beta selection schemes. Clearly, the Beta selection scheme enhances the chance of finding conflicts between predictions by the FSM inferred and the observations from the target protocol. After 19 iterations of the algorithm, due to the Beta selection scheme, the average number of leaf nodes in the VTree has increased from 29.4 to 35.4, an improvement of 20.4%.

D. Comparison results

In another set of experiments we compare the fuzzing performance of IoTInfer with those of two state-of-the-art fuzzers for IoT devices, IoTFuzzer [16] and Snipuzz [21], using the five Bluetooth devices shown in Table II. In our experiments, we use our own implementations of IoTFuzzer and Snipuzz in Python based on BlueZ [6], the official Bluetooth implementation in Linux:

- **IoTFuzzer**: As its code is not publicly available, we derive our best knowledge about its implementation details from the paper [16]. IoTFuzzer requires static analysis of companion mobile apps to infer the formats of packets sent to the target IoT device. In our experiments we assume that such packet formats are known to the mutation algorithm of IoTFuzzer a priori.
- **Snipuzz**: The code of Snipuzz released by its authors was written in C# for Windows machines. As BlueZ has not been ported to run on Windows systems yet, we re-implement Snipuzz based on our understanding of its mutation algorithm presented in the paper [21] as well as its C# implementation code. For Snipuzz, the packet formats are inferred from the communication packets and are thus not given as its input in a fuzzing experiment.

We run each fuzzing experiment for an hour, using IoTFuzzer, Snipuzz, or IoTInfer. Based on the Bluetooth protocol specification [5], we classify each Bluetooth device's responses into the following categories:

- **R1:** The target device becomes frozen during the fuzzing tests.
- **R2:** The target device sends back a command reject packet with the reason to be "Command not understood."
- **R3:** The target device sends back a command reject packet with the reason to be "Signaling MTU exceeded."
- **R4:** The target device sends back a command reject packet with the reason to be "Invalid CID in request."
- **R5:** The target device sends back a connection response packet indicating "Connection successful."
- **R6:** The target device sends back a connection response packet indicating "Connection pending."
- **R7:** The target device sends back a connection response packet indicating "Connection refused – PSM not supported."
- **R8:** The target device sends back a connection response packet indicating "Connection refused – security block."
- **R9:** The target device sends back a connection response packet indicating "Connection refused – no resources available."
- **R10:** The target device sends back a configuration response packet that indicates "Success" and includes an *empty* list of configuration parameters.
- **R11:** The target device sends back a configuration response packet that indicates "Success" and includes a *nonempty* list of configuration parameters.
- **R12:** The target device sends back a configuration response packet that indicates "Failure – unacceptable parameters."
- **R13:** The target device sends back a configuration response packet that indicates "Failure – unknown options."
- **R14:** The target device sends back a disconnection response packet.
- **R15:** The target device sends back an information response packet indicating "Success."
- **R16:** The target device sends back an echo response packet.

It is noted that the 16 categories are not exhaustive, but they cover the different types of responses we have seen from the five Bluetooth devices. Intuitively speaking, a good fuzzing strategy should elicit as diverse responses from the target IoT device as possible. More importantly, these responses may include unexpected ones with security consequences,

such as device crashes. Therefore, we use a *diversity* metric, which is the total number of response types seen in a fuzzing experiment, to compare fuzzing performances.

Table V summarizes the observations about the different types of responses seen from the target Bluetooth devices with the three fuzzers used. We observe that for all the five Bluetooth devices, IoTInfer leads to the highest diversity measurements among the three fuzzers used. Moreover, target devices are seen frozen (i.e., response type R1) only in those fuzzing tests where IoTInfer is used. These observations result from IoTInfer's fuzzing strategy guided by FSM inference, which enables it to explore deep state transitions of the target IoT devices with different sequences of input packets. In contrast, both IoTFuzzer and Snipuzz consider only stateless fuzzing, which does not optimize generation of test packets to trigger new state changes in the target devices.

From Table V it is also observed that none of the response packets include echo responses (i.e., R16) when IoTInfer is used. This is because the input packet formats provided to IoTInfer do not include echo requests. L2CAP echo packets are used to test the link between two Bluetooth devices without any mutual authentication. As the echo response packet contains the same data from the echo request packet, mutating these data in IoTInfer creates a large number of different output observations from the target device, thus causing a state explosion problem for the FSM inferred. It is noted that even with echo request packets excluded from the input formats fed to IoTInfer, it still evokes more response types than IoTFuzzer and Snipuzz in all the fuzzing experiments.

Our implementations of IoTFuzzer, Snipuzz, and IoTInfer have comparable memory usages in the experiments. For instance, when used to fuzz-test the Galaxy Android phone for an hour, their peak memory usages are 26.6MB, 38.2MB, and 30.0MB, respectively. Snipuzz uses the most memory because our implementation uses the *scipy.spatial.distance* module to calculate cluster distances, which requires approximately 10.0MB memory. Compared with IoTFuzzer, extra memory is needed by IoTInfer to maintain the FSMs inferred.

IX. Limitations

In this work we have only considered two network protocols, Bluetooth and Telnet. However, IoT devices are usually immersed within an IoT ecosystem with various other protocols deployed at different layers, such as point-to-point communication layer (e.g., NFC and NB-IoT), multi-hop routing layer (e.g., ZigBee and 6LowPAN), and application layer (e.g., MQTT and XMPP). Although in principle these protocols can also be tested with input messages generated with guidance of FSM inference, we have to extend the current implementation of IoTInfer to further improve fuzzing efficiency, such as dealing with messages sent in the flexible JSON format and inferring input message formats when protocol specifications are unavailable.

With the protocol FSMs inferred from the target device, manual analysis is required by IoTInfer currently to assess whether they comply with the corresponding protocol specifications. Our own experiences with both the Bluetooth L2CAP and Telnet protocols suggest that manual verification can be ad-hoc and error-prone. It remains an interesting research direction to automate verification of protocol FSMs inferred from IoT devices against their specifications – if they are available – or compare FSMs of the same protocol inferred from different IoT devices.

In Section VIII, we have empirically evaluated the effectiveness of IoTInfer in finding potential implementation-level vulnerabilities in different IoT devices and also compared its performance against those of IoTFuzzer and Snipuzz. However, for blackbox fuzz testing which does not have access to the source code or firmware of the target IoT device, it is difficult to theoretically quantify the effectiveness of any fuzzing strategy. The heuristic adopted by IoTInfer, which uses FSM inference to guide generation of test messages in blackbox fuzzing, may also suffer the limitations implied by the no free lunch theorem for optimization [44].

X. Conclusions

The growing popularity of IoT devices has raised concerns about their security and resilience in adversarial environments. In this work we have developed a new method called IoTInfer, which leverages FSM inference to guide fuzzing tests of IoT network protocols. We have applied IoTInfer to evaluate the Bluetooth and Telnet protocols implemented by various IoT devices. Our experimental results have demonstrated both its efficiency in generating meaningful test cases for IoT devices and its effectiveness in finding previously unknown security issues and implementation deviations from protocol specifications. The comparison results with two other state-of-the-art blackbox IoT device fuzzing tools have also shown that IoTInfer is better at eliciting diverse responses from the fuzzing targets.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This work is partially supported by the US National Science Foundation under award CNS-1943079.

References

[1] *https://github.com/samhocevar/zzuf*.
[2] *https://www.zdnet.com/article/critical-vulnerabilities-impact-over-a-million-iot-radio-devices/*.
[3] *https://www.zdnet.com/article/hacker-leaks-passwords-for-more-than-500000-servers-routers-and-iot-devices/*.
[4] American Fuzzy Lop. *https://lcamtuf.coredump.cx/afl/*.
[5] Bluetooth Specification Version 4.0. *https://www.bluetooth.com/specifications/protocol-specifications/*.
[6] BlueZ: Official Linux Bluetooth protocol stack. *http://www.bluez.org/*.
[7] LearnLib. *https://learnlib.de/*.
[8] A. Al Farooq, E. Al-Shaer, T. Moyer, and K. Kant. IoTC^2: A formal method approach for detecting conflicts in large scale IoT systems. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 442–447. IEEE, 2019.
[9] F. A. Alaba, M. Othman, I. A. T. Hashem, and F. Alotaibi. Internet of Things security: A survey. *Journal of Network and Computer Applications*, 88:10–28, 2017.
[10] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi, A. Abusnaina, A. Awad, D. Nyang, and A. Mohaisen. Analyzing and detecting emerging Internet of Things malware: A graph-based approach. *IEEE Internet of Things Journal*, 6(5):8977–8988, 2019.
[11] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

TABLE V
COMPARISON OF THREE FUZZERS FOR IOT DEVICES, IOTFUZZER, SNIPUZZ, AND IOTINFER. DIVERSITY MEASURES THE TOTAL NUMBER OF DIFFERENT RESPONSE TYPES SEEN FROM THE TARGET IOT DEVICE IN A FUZZING EXPERIMENT.

Device	**Fuzzer**	**Response Type**																**Diversity**
		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	
Dell Laptop	IoTFuzzer	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	10
	Snipuzz	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	4
	IoTInfer	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	**13**
Raspberry Pi	IoTFuzzer	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	10
	Snipuzz	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	4
	IoTInfer	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	**13**
Keyboard	IoTFuzzer	✗	✓	✗	✓	✓	✗	✓	✗	✓	✓	✓	✗	✗	✓	✓	✗	9
	Snipuzz	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	5
	IoTInfer	✗	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	**10**
Android Phone	IoTFuzzer	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✓	✓	✓	✓	8
	Snipuzz	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	5
	IoTInfer	✗	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✗	**11**
Sony Speaker	IoTFuzzer	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	5
	Snipuzz	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	5
	IoTInfer	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓	✗	**9**

[12] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. Snooze: toward a stateful network protocol fuzzer. In *International Conference on Information Security*. Springer, 2006.

[13] B. Bezawada, M. Bachani, J. Peterson, H. Shirazi, I. Ray, and I. Ray. IoTSense: Behavioral fingerprinting of IoT devices. *arXiv preprint arXiv:1804.03852*, 2018.

[14] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM conference on Computer and Communications Security*, 2007.

[15] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac. Sensitive information tracking in commodity IoT. In *USENIX Security Symposium*, 2018.

[16] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *Network and Distributed System Security Symposium*, 2018.

[17] C. Y. Cho, D. Babi ć, E. C. R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *ACM Conference on Computer and Communications Security*, 2010.

[18] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the USENIX Security Symposium*, 2007.

[19] M. Eceiza, J. L. Flores, and M. Iturbe. Fuzzing the Internet of Things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems. *IEEE Internet of Things Journal*, 2021.

[20] K. Fang and G. Yan. IoTReplay: Troubleshooting COTS IoT devices with record and replay. In *IEEE/ACM Symposium on Edge Computing*, 2020.

[21] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang. Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 337–350, 2021.

[22] C. Gao, Z. Ling, B. Chen, X. Fu, and W. Zhao. SecT: A lightweight secure thing-centered IoT communication system. In *International Conference on Mobile Ad Hoc and Sensor Systems*. IEEE, 2018.

[23] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurniawan. SweynTooth: Unleashing mayhem over Bluetooth Low Energy. In *USENIX Annual Technical Conference*, 2020.

[24] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[25] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[26] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, 2008.

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

[28] T. Gu, Z. Fang, A. Abhishek, H. Fu, P. Hu, and P. Mohapatra. IoTGAZE: IoT security enforcement via wireless context analysis. In *IEEE Conference on Computer Communications*, 2020.

[29] K. K. Karmakar, V. Varadharajan, S. Nepal, and U. Tupakula. SDN enabled secure IoT architecture. *IEEE Internet of Things Journal*, 2020.

[30] F. B. Khendek. Test selection based on finite state models. *IEEE Transactions on software engineering*, 17(591-603):10–1109, 1991.

[31] J. Li, B. Zhao, and C. Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[32] M. Michael. Attack Landscape H1 2019: IoT, SMB traffic abound. *https://blog.f-secure.com/attack-landscape-h1-2019-iot-smb-traffic-abound/*, 2019.

[33] A. Mosenia and N. K. Jha. A comprehensive study of security of Internet-of-Things. *IEEE Transactions on emerging topics in computing*, 5(4):586–602, 2016.

[34] J. Narayan, S. K. Shukla, and T. C. Clancy. A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys (CSUR)*, 48(3):1–26, 2015.

[35] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani. Demystifying IoT security: An exhaustive survey on IoT vulnerabilities and a first empirical look on Internet-scale IoT exploitations. *IEEE Communications Surveys & Tutorials*, 21(3):2702–2733, 2019.

[36] A. Ouaddah, H. Mousannif, A. Abou Elkalam, and A. A. Ouahman. Access control in the Internet of Things: Big challenges and new opportunities. *Computer Networks*, 112:237–262, 2017.

[37] Palo Alto Networks. 2020 Unit 42 IoT Threat Report. *https://unit42.paloaltonetworks.com/iot-threat-report-2020/*, 2020.

[38] M. Shahbaz and R. Groz. Inferring mealy machines. In *International Symposium on Formal Methods*, pages 207–222. Springer, 2009.

[39] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini. Security, privacy and trust in Internet of Things: The road ahead. *Computer Networks*, 76:146–164, 2015.

[40] A. K. Sikder, L. Babun, H. Aksu, and A. S. Uluagac. Aegis: a context-aware security framework for smart home systems. In *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019.

[41] H. Soroush, M. Albanese, M. A. Mehrabadi, I. Iganibo, M. Mosko, J. H. Gao, D. J. Fritz, S. Rane, and E. Bier. SCIBORG: Secure configurations for the IoT based on optimization and reasoning on graphs. In *IEEE Conference on Communications and Network Security*, 2020.

[42] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[43] R. H. Weber and E. Studer. Cybersecurity in the Internet of Things: Legal aspects. *Computer Law & Security Review*, 32(5):715–728, 2016.

[44] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[45] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *USENIX Security Symposium*, 2019.