DIANE: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices

Nilo Redini*, Andrea Continella†, Dipanjan Das*, Giulio De Pasquale*, Noah Spahn*, Aravind Machiry‡, Antonio Bianchi‡, Christopher Kruegel*, and Giovanni Vigna*
*UC Santa Barbara †University of Twente ‡Purdue University
{nredini, dipanjan, peperunas, ncs, chris, vigna}@cs.ucsb.edu
a.continella@utwente.nl, {amachiry, antoniob}@purdue.edu

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00066

***Abstract*—Internet of Things (IoT) devices have rooted themselves in the everyday life of billions of people. Thus, researchers have applied automated bug finding techniques to improve their overall security. However, due to the difficulties in extracting and emulating custom firmware, black-box fuzzing is often the only viable analysis option. Unfortunately, this solution mostly produces invalid inputs, which are quickly discarded by the targeted IoT device and do not penetrate its code. Another proposed approach is to leverage the companion app (i.e., the mobile app typically used to control an IoT device) to generate well-structured fuzzing inputs. Unfortunately, the existing solutions produce fuzzing inputs that are constrained by app-side validation code, thus significantly limiting the range of discovered vulnerabilities.**

In this paper, we propose a novel approach that overcomes these limitations. Our key observation is that there exist functions inside the companion app that can be used to generate optimal (i.e., valid yet under-constrained) fuzzing inputs. Such functions, which we call *fuzzing triggers*, are executed before any data-transforming functions (e.g., network serialization), but *after* the input validation code. Consequently, they generate inputs that are not constrained by app-side sanitization code, and, at the same time, are not discarded by the analyzed IoT device due to their invalid format. We design and develop DIANE, a tool that combines static and dynamic analysis to find *fuzzing triggers* in Android companion apps, and then uses them to fuzz IoT devices automatically. We use DIANE to analyze 11 popular IoT devices, and identify 11 bugs, 9 of which are *zero* days. Our results also show that without using *fuzzing triggers*, it is not possible to generate bug-triggering inputs for many devices.

I. INTRODUCTION

Internet of Things (IoT) devices have become part of the everyday life of billions of people [53], [64]. Unfortunately, very much like their popularity, the number of vulnerabilities found in these devices has increased as well. In both 2019 and 2018, security researchers published more than 150 vulnerabilities affecting IoT devices [5], [6], [7]. This represented an increment of 15% compared to 2017, and an increase of 115% compared to 2016.

These vulnerabilities reside in the software (or firmware) running on these IoT devices. As several studies have shown, this software often contains implementation flaws, which attackers can exploit to gain control of a device, and cause significant disruption for end-users [17], [18], [27], [48], [59]. One prominent example is the Mirai botnet [51], which infected hundreds of thousands of IoT devices using a collection of vulnerabilities.

In recent years, researchers have developed novel techniques to automatically find vulnerabilities in IoT devices by analyzing their firmware [24], [31], [32], [70], [71], [84]. These approaches, however, present several limitations. First, obtaining the firmware running on an IoT device is difficult: Extracting the firmware from a device typically requires *ad hoc* solutions, and vendors hardly make their software publicly available [70]. Second, unpacking and analyzing a firmware sample is a challenging task: Firmware samples may be available in a variety of formats, and may run on several different architectures, often undocumented. Furthermore, most IoT devices are shipped with disabled hardware debugging capabilities [25], [55], [61], ruling out analyses based on dynamic instrumentation.

For these reasons, security researchers typically have to use black-box approaches when vetting the security of IoT devices. However, the existing black-box approaches [2], [12], [14] require knowledge about the data format accepted by the device under analysis. Consequently, given the heterogeneity and lack of documentation of the protocols adopted by IoT devices, these approaches are not readily applicable.

However, most IoT devices have *companion apps* [82], [88] (i.e., mobile apps used to interact with the device), which contain the necessary mechanism to generate valid inputs for the corresponding device. Based on this observation, Chen et al. [25] proposed a tool, IoTFuzzer, which fuzzes IoT devices by leveraging their companion apps. IoTFuzzer analyzes the companion app and retrieves all the paths connecting the app's User Interface (UI) to either a network-related method or a data-encoding method. Then, IoTFuzzer fuzzes the parameters of the *first* function that handles user input along these paths, thus generating valid fuzzing inputs for the IoT device.

While this approach yields better results than randomly fuzzing the data directly sent to the IoT device over the network, in practice, it consists in mutating variables immediately after they are fetched from the UI, before the app performs any input validation or data processing. Consequently, the effectiveness of IoTFuzzer suffers substantially when the app sanitizes the provided input—our experiments (Section IV-E) demonstrate that 51% of IoT companion apps perform app-side input validation. Indeed, recent research showed that mobile apps often perform input validation to trigger different behaviors [86]. For these reasons, IoTFuzzer's approach cannot produce *under-constrained* (i.e., not affected by app-side sanitization) yet *well-structured* (i.e., accepted by the IoT device) fuzzing inputs, which can reach deeper code locations, uncovering more vulnerabilities.

DOI 10.1109/SP40001.2021.00066

Our Approach. In this paper, we propose and implement an approach that leverages the companion app to generate inputs for the analyzed device. To overcome IoTFuzzer's limitations, we precisely determine (and fuzz) *optimal code locations* within the companion app, which produce valid yet under-constrained inputs for the IoT device.

Our approach considers the app's execution as a sequence of functions that transform the data introduced by the user (e.g., through the app's UI) into network data. Our intuition is that the first functions within this sequence typically convert the user inputs into internal data structures, generating data that is constrained by app-side validation. In contrast, the last functions in this sequence adequately encode the user data, serializing it on the network.

The novelty of our approach is to fuzz an IoT device by invoking specific functions within its companion app. We call these functions *fuzzing triggers*. When invoked, *fuzzing triggers* generate inputs that are not constrained by app-side validation, and, at the same time, are well-structured, so that they are not immediately discarded by the fuzzed IoT device.

Our approach uses a novel combination of static and dynamic analysis and performs two main steps: i) *fuzzing triggers* identification, and ii) fuzzing. To do this, first, we automatically retrieve those functions within an app that send data to the IoT device. Then, for each of these functions, we build an inter-procedural backward slice, which we dynamically analyze to ultimately identify *fuzzing triggers*. Finally, we use dynamic instrumentation to repeatedly invoke these *fuzzing triggers* using different arguments. This generates a stream of network data that fuzzes the functionality of the IoT device, to ultimately spot vulnerabilities.

We implemented our approach in a tool, called DIANE, and ran it against a representative set of 11 popular IoT devices of different types and from different manufacturers. DIANE correctly identified *fuzzing triggers*, and successfully identified 11 bugs, 9 of which are previously unknown vulnerabilities. Additionally, we compared DIANE with IoTFuzzer, showing that the identification of *fuzzing triggers* is essential to generate under-constrained, crash-inducing inputs.

In summary, we make the following contributions:

- We propose an approach to identify *fuzzing triggers*, which are functions that, in the app's control flow, are located between the app-side validation logic and the data-encoding functions. When executed, the identified *fuzzing triggers* produce valid yet under-constrained inputs, enabling effective fuzzing of IoT devices.
- We leverage our approach to implement DIANE, an automated black-box fuzzer for IoT devices.
- We evaluate our tool against 11 popular, real-world IoT devices. In our experiments, we show that by identifying *fuzzing triggers* and using them to generate inputs for the analyzed devices, we can effectively discover vulnerabilities. Specifically, we found 11 vulnerabilities in 5 different devices, 9 of which were previously unknown.
- We show that, for a majority of IoT devices and companion apps, identifying and leveraging *fuzzing triggers* is essential to generate bug-triggering inputs.

```
1  // Android Java Code
2  public
   int PTZ(String adminPwd, int x, int y, int z){
3    //..
4      byte data[] = MsgPtzControlReq(x, y, z);
5      if (!adminPwd.contains("&") && // Input
6          !adminPwd.contains("'")){ // validation
7          SendMsg(adminPwd, camId, data);
8      }
9  }
10
11 public static native int SendMsg(String
   adminPwd, String camId, byte[] data);
12
13 // Java Native Interface
14 int
   Java_SendMsg(char* pwd, char* cam_id, Msg* msg){
15     prepare_msg(pwd, cam_id, msg);
16     notify_msg(msg);
17 }
18 // JNI - Different thread
19 void sender() {
20     Msg* msg = get_message()
21     send_to_device(msg);
22 }
```

Fig. 1. Snippet of code that implements a sanity check on the admin password, and uses the Java Native Interface to send messages to the device. The example is based on the Wansview app in our dataset.

In the spirit of open science, we make the datasets and the code developed for this work publicly available: https://github.com/ucsb-seclab/diane.

II. MOTIVATION

To motivate our approach and exemplify the challenges that it addresses, consider the snippet of code in Figure 1. The app utilizes the method `PTZ` (Line 2) to send position commands (i.e., spatial coordinates) to an IoT camera. To do this, `PTZ` invokes the native function `SendMsg` (Line 7), which prepares the data to be sent (Line 15), and stores it into a shared buffer (Line 16). In parallel, another thread reads the data from the same buffer (Line 20), and sends commands to the device (Line 21). Notice that the IoT camera requires a password to authenticate commands, and the app performs a sanity check on the password string (Lines 5 and 6). This example shows two crucial challenges that have to be faced when generating IoT inputs from the companion apps.

First, apps communicate with IoT devices using *structured data*, encoded in either known protocols (e.g., HTTP), or custom protocols defined by the vendor. Messages that do not respect the expected format are immediately discarded by the device, and, consequently, cannot trigger deep bugs in its code. In the example, the app uses the function `prepare_msg` (Line 15) to create a correctly structured message.

Second, while it is crucial to generate correctly structured inputs, an effective approach has to avoid generating inputs that are constrained by app-side validation code. In the example, the function `PTZ` (Line 2) forbids the password to contain the characters `&` and `'`. However, the presence of these characters may be crucial in generating crash-triggering fuzzing inputs.

The insight from the authors of IoTFuzzer is to leverage the companion app to generate fuzzing inputs in a format that the device can process. This means that the input values need to be mutated before the app "packages" and sends them to the device. While this is true, our crucial insight is that the mutation indeed

has to occur *before* the app packages the inputs, but also *after* the app performs any input validation. Note that, with the expression app-side validation we refer to all types of constraints that the app imposes on the data sent to an IoT device. These constraints might be imposed by typical sanitization checks (e.g., limiting the length of a string) or by parameters hard-coded in the generated request (e.g., hard-coded attributes in a JSON object).

Our work fills this gap: We identify strategic execution points that produce inputs that are not affected by the constraints that the app logic imposes. To achieve this goal, we analyze an IoT device companion app, and focus on identifying effective *fuzzing triggers*: Functions that, when used as entry points for fuzzing, maximize the amount of unique code exercised on the device's firmware, thus potentially triggering security-relevant bugs. Consider, as an example, the app's execution as a sequence of functions that receive data from the UI and send it over the network. On the one hand, if the fuzzed function is too close to the UI, the fuzzing is ineffective due to app-side validation that might be present later in the execution. On the other hand, picking a function too close to the point where data is put onto the network might be ineffective. In fact, some protocol-specific data transformations applied early in the execution would be skipped, causing the generated inputs to be dropped by the IoT device. In Figure 1, the function `sendMsg` represents a *fuzzing trigger*.

Our approach identifies these *fuzzing triggers* automatically, relying on a combination of dynamic and static analyses, without the need for any *a priori* knowledge about neither the firmware nor the network protocol used by the analyzed IoT device. Additionally, previous work [25] relies on specific sources of inputs (e.g., text boxes in the app's UI) to bootstrap its analysis, and does not mutate data generated from unspecified sources (e.g., firmware updates through the companion app triggered by a timer). Our bottom-up approach (explained in Section III) does not make any assumptions on input sources and is, therefore, more generic.

The example we discussed in this section is the simplified version of the code implemented in the Wansview app. We also note that app-side validation is prevalent in real-world apps, and that the challenges we described do not only apply to this example.

III. Approach

While our goal is to find bugs in IoT devices, given the general unavailability of their firmware, we focus our analysis on their companion apps. Our key intuition is to identify and use, within these companion apps, those functions that *optimally* produce inputs for the analyzed IoT devices. These optimal functions effectively produce inputs that are valid yet under-constrained.

Automatically identifying these functions is a challenging task because the complexity of the companion apps, the usage of native code, and the presence of multiple threads rule out approaches based entirely on static analysis. Thus, we propose a *novel analysis pipeline* built on top of four different analyses: i) static call-graph analysis, ii) network traffic analysis, iii) static data-flow analysis, and iv) dynamic analysis of the function arguments.

Our approach does not make any assumption on how the app's user interface influences the data sent to the controlled IoT device, and it avoids app-side sanitization on the generated data. Our analysis does not start by considering UI-processing functions, but, on the contrary, uses a "bottom-up" approach. Specifically, we start from identifying low-level functions that potentially generate network traffic, and then we progressively move "upward" in the app call-graph (i.e., from low-level networking functions to high-level UI-processing ones). This approach allows us to identify functions that produce valid yet under-constrained inputs, skipping all the sanitization checks performed by data-processing functions. We then use these functions, which we call *fuzzing triggers*, to efficiently fuzz the analyzed IoT device, while monitoring it for anomalous behaviors, which indicate when a bug is triggered.

We implement our approach in a tool named DIANE, depicted in Figure 2. DIANE works in two main phases: *Fuzzing Trigger Identification*, and *Fuzzing*. In the *Fuzzing Trigger Identification* phase, DIANE identifies optimal functions within the companion app, that, when invoked, generate under-constrained well-structured inputs for the analyzed device. Then, during the *Fuzzing* phase, these functions are used to generate data that is sent to the analyzed device using a local network connection.

Our approach is independent of the network medium used by the analyzed app. We apply it to apps communicating with their related IoT device both over WiFi and Bluetooth (Appendix B). DIANE fuzzes IoT devices that receive commands through a *local* connection between the device and the companion app. Though some devices might receive commands from cloud-based endpoints, research showed that the vast majority of them (95.56%) also allow some form of local communication (e.g., during the device setup phase) [17].

A. Fuzzing Trigger Identification

Intuitively, *fuzzing triggers* are functions that, in the app's control flow, are located in between the app-side validation logic and any data-transforming (e.g., message serialization) function occurring before sending data over the network. Precisely, given an execution trace from a source of input (e.g., data received from the UI) to the function sending data over the network, a *fuzzing trigger* is defined as a function that dominates[1] *all* data-transforming functions and post-dominates *all* input-validating functions. We consider the first data-transforming function in the trace a valid *fuzzing trigger*, as it dominates every other data-transforming function (itself included).

Our bottom-up Fuzzing Trigger Identification algorithm is composed of four steps: i) *sendMessage* Candidates Identification, ii) *sendMessage* Validation, iii) Data-Transforming Function Identification, and iv) Top-Chain Functions Collection. Algorithm 1 lists the pseudo-code of our approach.

Step 1: *sendMessage* Candidates Identification. We begin by identifying the functions, in the companion app, that implement the necessary logic to send messages to the IoT device. We call these functions *sendMessage* functions.

Identifying these functions in an automated and scalable way is difficult. Companion apps might rely on ad-hoc native functions directly invoking system calls to implement *sendMessage* functions. Furthermore, we found that these functions might be

[1] We refer to the dominance concept of the call graph theory, where a node d dominates a node n if every path from the entry node to n must go through d. Also, we say that a node p post-dominates n if every path from n to an exit node passes through p.

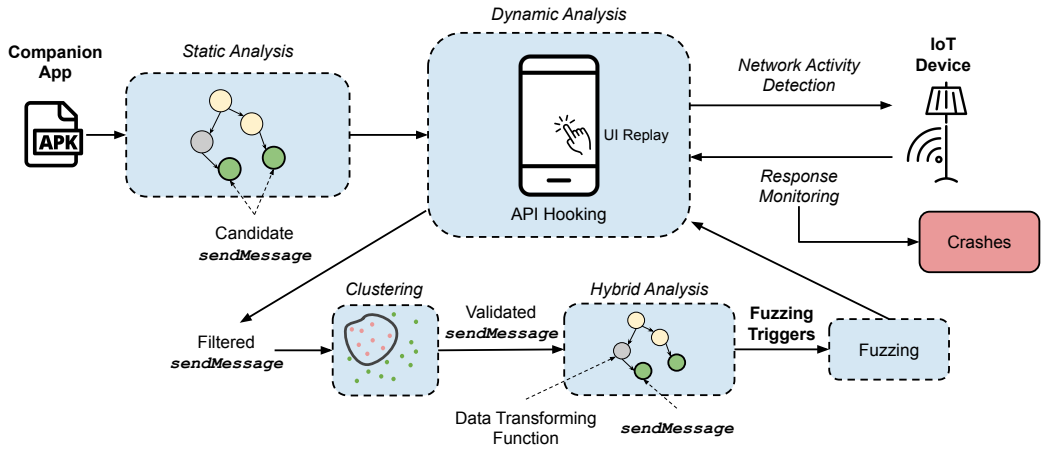

Fig. 2. Using static analysis, DIANE first identifies candidate *sendMessage* functions. Then, it runs the companion app, replaying a recorded user interaction, to validate the candidate *sendMessage* functions. Next, DIANE uses a hybrid analysis to identify data-transforming functions and, therefore, *fuzzing triggers*. Finally, DIANE fuzzes the validated triggers and identifies crashes by monitoring the device responses.

executed within separate threads, which makes it harder for any analyses (both static or dynamic) to precisely track data flows between the app's UI and *sendMessage* functions. However, our key insight is that the companion app must contain "border" functions, situated between the app core functionality and external components (i.e., the Android framework or native libraries), which, when executed, eventually trigger a message to be sent to the IoT device. Throughout the rest of the paper, we consider these border functions our *sendMessage* functions.

In our approach, we first identify *candidate sendMessage* functions by statically analyzing the companion app. We aim at finding all the border methods that might implement the network interactions with the analyzed IoT device (function getBorderMethods in Algorithm 1). Specifically, we collect all the methods that perform (at least) a call to native functions or a call to methods in the Android framework that implement network I/O functionality (see Appendix A for more details).

Step 2: *sendMessage* Validation. We dynamically execute the app and leverage API hooking to validate the candidate *sendMessage* functions. In order to establish whether a border function is a valid *sendMessage* function we could, in theory, i) dynamically execute the function multiple times and check whether it generates network traffic each time, and ii) prevent the app from executing the function and check whether or not network traffic is still generated. Unfortunately, we found that preventing a function to be executed, as well as forcing the app to execute the same function multiple times, usually causes the app itself to crash. To solve these issues, we adopt a different approach, based on timestamps and machine learning.

First, we dynamically hook all the candidate functions and run the app. When we observe network activity, we register the last executed candidate *sendMessage* function. In particular, each time a candidate *sendMessage* function is executed, we collect the elapsed time between its execution and the observed network activity. Then, we leverage the K-mean algorithm to cluster the observed elapsed time measures. Specifically, we group our candidates into two clusters (i.e., $k=2$). To do so, we compute each feature vector as the mean, standard deviation, and mode of the elapsed times of each candidate. The rationale is that functions that cause network activity have a smaller mean and standard deviation, as they are less affected by noise. Finally, among the *sendMessage* candidates, we select those belonging to the cluster having the smallest mean of the elapsed times. Only the *sendMessage* functions within this cluster will be considered in the subsequent steps of our analysis. This approach is represented by the function dynamicFilter in Algorithm 1.

Step 3: Data-Transforming Function Identification. While *sendMessage* functions are intuitively good triggers for performing fuzzing, apps may apply data-transformations in functions executed before a *sendMessage* function. A typical example of a data-transforming function is represented by an encoding method that takes as input a list of integers and serializes it to a sequence of bytes.

As previously explained, *fuzzing triggers* are functions that, in the app's control flow, are located before any data-transforming function. Fuzzing a function located in between a data-transforming function and a *sendMessage* function would likely produce invalid inputs that are discarded by the IoT device. Thus, to find *fuzzing triggers*, we first need to identify the data-transforming functions applied to the data being sent.

This task presents different challenges. First, the data being sent might be contained in a class field, which is referenced by the *sendMessage* function. This field might be theoretically set anywhere in the app code, including within other threads. Furthermore, for each field, we need to consider its parent classes, as the variable holding the message to be sent might be inherited by a different class.

In our approach, we take into account these issues. We first statically identify the possible variables that hold the data being sent by the considered *sendMessage* function, and the code locations where these variables might be set in the app (function getArgAndObjLocs in Algorithm 1). To achieve this, we create a set S_v containing tuples (v,cl), where v is a variable used by the *sendMessage* (i.e., *sendMessage* arguments or objects referenced within the *sendMessage* body), and cl is the code location where v is set.

Then, we identify data-transforming functions. For each tuple $(v,cl) \in S_v$, we perform a static inter-procedural backward slicing (Line 6 in Algorithm 1) from cl up to any function retrieving values from any UI objects. Then, we divide the computed program slices in function scopes (getFunctionScopes at Line 7). Given a program slice, a function scope is defined as a subsequence $inst_f$ of sequential instructions in the slice that belong to the same function f.

For each collected function scope, we perform a liveness analysis [63]: We consider the variables (i.e., local variables and class fields) referenced within the function scope, and we compute the set Li_f of variables that are live at the beginning of the scope, and the set Lo_f of variables that are live at the end of the scope (Line 8). For example, if a function f is traversed by the slice, the variables that are live at the beginning of the function scope $inst_f$ are f's arguments and the class fields that are read before being written in f. The variables that are live at the end of f's scope are the returned variable and the class fields that f creates or modifies.

To identify data-transforming functions, we leverage the observation that these functions increase the entropy of the data they consume, as explored by related work [25]. Therefore, we hook the functions we identified in a slice, we dynamically run the app, and we calculate the Shannon entropy [49] of the data assigned at runtime to each variable v contained in Li_f and Lo_f (more details about how we calculate the entropy are provided in Appendix C). If v is a primitive variable (e.g., `int`), or a known type (i.e., `String`, `Integer`, `Float`, and `Double`), we convert the data it contains in its byte representation and calculate the Shannon entropy of this list of bytes. Conversely, if v is a class field, we retrieve its class definition and consider each field variable v_c of v whose type is either primitive or known. Then, we compute the entropy of each one of these v_c variables, and add them to either the Li_f set or to the Lo_f set, depending on which live set v belongs.

Finally, we inspect every collected function scope and calculate the quotient d_e between the maximum entropy registered among all the variables in Lo_f and the minimum value of entropy registered among all the variables in Li_f (Line 11). If d_e is greater than a certain threshold T_f (set to 2.0 in our experiments, as previous work suggested [80]), we consider the function f to be a data-transforming function (Line 12).

Step 4: Top-Chain Functions Collection. Data-transforming functions are usually executed in precise sequences to adequately prepare the data to be sent to an IoT device. For instance, a companion app may encode some user data in base64, and then encapsulate it in an HTTP request.

We call a sequence of data-transforming functions a *transformation data chain*, and we refer to the first function in the sequence with the term top-chain function. We say that a top-chain function f *affects* a variable v if modifying the content of f's variables eventually affects v's value.

Of particular interest for us are the top-chain functions that affect *sendMessage* variables. In fact, if we control the variables of these top-chains, we can control the data sent to the analyzed IoT device. In particular, this data is both valid (i.e., accepted by the IoT device) and not affected by unnecessary app-side input validation. As such, top-chain functions that affect *sendMessage* variables represent the optimal *fuzzing triggers* to stimulate the IoT device functionality.

Algorithm 1 Fuzzing Trigger Identification.

```
1:  procedure GETTOPCHAIN(sendMessage)
2:      topChain ← {}
3:      for (v,cl) ∈ getArgAndObjLocs(sendMessage) do
4:          to_hook ← {}
5:          dtf ← []
6:          bsl ← getBackwardSlice(v,cl)
7:          for (f,inst_f) ∈ getFunctionScopes(bsl) do
8:              (Li_f,Lo_f) ← livenessAnalysis(inst_f)
9:              to_hook ← to_hook ∪ {f,(Li_f,Lo_f)}
10:         for {f,(ELi_f,Elo_f)} ∈ getEntropies(to_hook) do
11:             d_e ← maxVarEntropy(Elo_f)/minVarEntropy(Eli_f)
12:             appendIfDataTransforming(dtf,d_e,{f,Li_f})
13:         trees ← getDominatorTrees(dtf)
14:         candidates ← dtf ∪ {sendmessage}
15:         for f_c ∈ candidates do
16:             if not isDominated(f_c,trees) then
17:                 topChain ← topChain ∪ {f_c}
18:     return topChain
19: procedure FUZZINGTRIGGERIDENTIFICATION(CompanionApp)
20:     fuzzingTriggers ← {}
21:     borderMethods ← getBorderMethods(CompanionApp)
22:     for s ∈ dynamicFilter(borderMethods) do
23:         fuzzingTriggers ← fuzzingTriggers ∪ getTopChain(s)
24:     return fuzzingTriggers
```

To identify top-chain functions, we build the dominance tree[2] of each data-transforming function detected at the previous step (Line 13), and select those data-transforming functions that are not dominated by any other data-transforming function (Line 16). Finally, we consider as *fuzzing triggers* the collected top-chain functions.

Note that, if no data-transforming function dominates a *sendMessage* function, we consider the *sendMessage* as a *fuzzing trigger* (Line 14, 15, and 16). This could happen when, for instance, the companion app does not contain data-transforming functions.

Note finally that, in principle, app-side sanitization code might be present in a function within a transformation data chain. We discuss this in Section V.

Example. As a simple example, consider Figure 3, which represents one of the data chains we found on the August Smart Lock device. Assuming that we previously identified `sendToDevice` as being a *sendMessage* function, we set { `c`} as the initial set of variables possibly holding data to be sent, and determine the code locations where `c` is set. As `c` is a function argument, we retrieve the *sendMessage* call site (Line 15), and bootstrap a backward program slicing from the call site, up to the function `unlock` (Line 1). This is achieved by following the data-flow of the variable `e` backward: `sendToDevice` uses the variable `e`, which is the result of a call to the function `encrypt`. Then, we continue the slice backward from the end of the function `encrypt` up to its entry point, and back to the `sendCommand` function. Finally, we reach the entry point of this function, and continue the slice considering its caller (i.e., the function `unlock`).

Following the definition of function scopes above stated, this backward slice contains the following function scopes: i) `sendCommand`: Line 15; ii) `encrypt`: Lines from 6 to 9; iii) `sendCommand`: Lines 12 and 13; iv) `unlock`: Line 3; v) `Command` constructor (code not reported in this example);

[2] A graph where each node's children are those nodes it immediately dominates.

```
1  public boolean unlock() { // unlock request
2    Command cmd = new Command(OP.UNLOCK);
3    return sendCommand(cmd);
4  }
5  /* Encrypts and return its parameters */
6  public byte[] encrypt (Command b){
7    byte[] enc;
8    // ...
9    return enc;
10 }
11
12 public boolean sendCommand (Command cmd){
13   // various checks on the command to send
14   byte[] e = encrypt(cmd);
15   return sendToDevice(e);
16 }
17 /* send a message */
18 public boolean sendToDevice(byte[] c) {/* ... */}
```

Fig. 3. Example of a simple Transformation Data Chains found on the August Smart Lock.

and vi) `unlock`: Lines 1 and 2. For brevity, in the following we only consider the relevant function scopes: ii) `encrypt`, iii) `sendCommand`, and vi) `unlock`. Their sets of live variables are: `encrypt`: $Li_f = \{b\}$, $Lo_f = \{enc\}$; `sendCommand`: $Li_f = \{cmd\}$, $Lo_f = \{cmd\}$; and `unlock`: $Li_f = \{\}$, $Lo_f = \{cmd\}$.

Once we identify the function scopes in the slice, we run the app and compute the entropy of the data assigned to each of their live variables. Then, we calculate the amount of entropy introduced by each function scope and check whether its value exceeds a threshold T_f.

The function `unlock` does not introduce any entropy, as the set Li_f is empty. In the cases where the set Li_f is empty, we do not consider the function f as a candidate data-transforming function, since it does not take any input.

For the function `encrypt`, the entropy of the data stored in `b` is 5.94, whereas the entropy of the data returned in `enc` is 53.16. Since the entropy delta d_e is greater than our threshold ($d_e = 53.16/5.94 > 2.0$), we consider `encrypt` as a data-transforming function. Also, the function `sendCommand` introduces a low amount of entropy ($d_e = 1.03$), and, therefore, it is not considered a data-transforming function. Finally, as the function `encrypt` dominates the function `sendToDevice`, `encrypt` is the only top-chain function, and it is used as the only *fuzzing trigger*.

UI Stimulation. Our approach executes the same app multiple times, being consistent across the different runs. Thus, ideally, we want the app to follow always the same execution paths. To achieve this goal, we require the analyst to run the app once, while DIANE records the generated UI inputs. Then, we automatically replay the same inputs in the subsequent runs, by leveraging RERAN [40]. We do not explicitly handle other sources non-determinism [29], as we found them to not significantly affect our approach.

Fuzzing Intermediate Data-Transforming Functions. In principle, transformation data chains might be arbitrary long. As DIANE's goal is to stimulate the core functionality of IoT devices, our approach ignores intermediate data-transforming functions (i.e., data-transforming functions dominated by a top-chain function) as they generate messages that would likely be discarded by the IoT device. However, as IoT devices might contain bugs also in the procedures used to decode a received message, we provide DIANE with the option to fuzz also all the intermediate data-transforming functions. Likewise, DIANE provides an option to fuzz the *sendMessage* functions directly even when dominated by top-chain functions. In Section IV-C, we empirically show that fuzzing the *sendMessage* functions does not lead to the discovery of new bugs, while it slows down the execution of our tool.

B. Fuzzing

After the first phase of our approach, we obtain a set of *fuzzing triggers*, which are the inputs to our fuzzer.

Test Case Generation. For each *fuzzing trigger*, we generate a set of test cases by mutating the parameters of the identified *fuzzing triggers*, which eventually modify the data sent by a *sendMessage* function. We fuzz the different *fuzzing triggers* one at the time, in a round-robin fashion. To mutate the values of their parameters, we use the following strategies:

- **String lengths:** We change the length of strings in order to trigger buffer overflows and out-of-bound accesses. We generate random strings with different lengths.
- **Numerical values:** We change the values of integer, double or float values to cause integer overflows or out-of-bound accesses. We generate very large values, negative values, and the zero value.
- **Empty values:** We provide empty values, in the attempt to cause misinterpretation, uninitialized variable vulnerabilities, and null pointer dereferences.
- **Array lengths:** We modify the content of arrays by removing or adding elements.

It is important to specify that we do not only fuzz primitive variables (e.g., `int`, `float`), but we also fuzz objects (as explained in Appendix B), by fuzzing their member variables.

Identifying Crashes. As shown by a recent study [61], identifying *all* crashes of network-based services of IoT devices without *invasive* physical access to the devices is challenging. At the same time, getting invasive physical access to IoT devices needs considerable engineering effort [9], since vendors usually prevent this type of access [10], [11].

For these reasons, while fuzzing a device, DIANE automatically analyzes its responses to identify crashes. Specifically, DIANE first performs a normal run of the app and monitor how the device responds during normal activity. Then, while fuzzing, DIANE monitors the network traffic between the app and the device again, and considers an input to be *potentially* crash-inducing, if any one of the following conditions is satisfied.

- **Connection dropped.** If the device abruptly ends an ongoing connection, we consider it as an indication that something wrong happened to the device. Specifically, for TCP connections, we look for cases where the app sent a `FIN` packet and received no response (`FIN + ACK`), and then sent a sequence of two or more `SYN` packets.
- **HTTP Internal Server Error (500).** Instances where the app and the device communicate through HTTP, and the device returns an Internal Server Error [1] (status code `500`), are considered as a signal that the device has entered in a faulty state.
- **Irregular network traffic size.** If the amount of data exchanged between the app and the device overcomes a

threshold S_e, we save the current crash-inducing input. Our intuition is that, when a device enters a faulty state (e.g., due to a crash) it usually becomes temporarily unavailable for the app, thus drastically reducing the amount of data exchanged. In our experiments, we empirically verified that when the amount of exchanged data was less than 50% (compared to a regular run), something unusual happened to the device. For this reason, we set S_e to be 50%.

- **Heartbeat Monitoring.** While fuzzing a given device, we continuously ping it and monitor its response time. We report any crash-inducing inputs causing the response time to be above a certain threshold T_p. In our experiments, we set T_p to 10 seconds, as we empirically verified that the average response time of an IoT device falls within 1 second under normal conditions.

Finally, we use an additional Android smartphone, which we refer to as the *watchdog device*, to monitor the status of the IoT device from a neutral standpoint (i.e., we do not instrument the companion app on this device). We run the companion app on the watchdog device and automatically replay the previously recorded UI inputs to exercise the different IoT device functionality at regular intervals. A human analyst can then observe whether the functionality exercised by the watchdog device (e.g., pressing the light switch UI button) causes the desired effect on the IoT device (e.g., turning the light on) or not. If an undesired effect is detected, it means that Diane was able to bring the analyzed device into an invalid state.

IV. Experimental Evaluation

In this section, we answer two research questions:

1) Is DIANE *able* to find both previously-known and previously-unknown vulnerabilities in IoT devices effectively?
2) Is DIANE *needed* to find vulnerabilities in IoT devices effectively, or can existing (app-based or network-level) fuzzers achieve similar results?

To answer the first research question, we first evaluated DIANE precision in detecting *fuzzing triggers* (Section IV-B) and then we used it to fuzz 11 different IoT devices (Section IV-C). Our system found 11 bugs in 5 devices, including 9 zero-day vulnerabilities, running, in all cases, for less than 10 hours (Section IV-H).

To answer the second research question, we first compared our tool with IoTFuzzer [25] by running it on the 11 analyzed devices (Section IV-D). Our experiment shows that DIANE outperformed IoTFuzzer in 9 devices, and performs as well as IoTFuzzer for the remaining 2 devices. Then, we performed a larger-scale automated study (Section IV-E) to measure how often companion apps perform app-side validation, which would limit the efficiency of approaches like IoTFuzzer. Our experiment revealed that 51% of the analyzed apps contain, indeed, app-side sanitization. Finally, we compared DIANE with existing network-level fuzzers (Section IV-F), and showed that network-level fuzzers are unable to find bugs in the analyzed devices.

We conclude this section by presenting a detailed case study about two zero-day bugs DIANE found in one of the analyzed devices (Section IV-G).

A. Dataset & Environment Setup

To evaluate DIANE, we used popular real-world IoT devices of different types and from different brands. Specifically, in October 2018 we searched for "smart home devices" on Amazon and obtained the list of the top 30 devices. Among these, we excluded 5 expensive devices (price higher than 200 USD), 1 device that does not communicate directly with the companion app (the communication passes through a Cloud service), and other 13 devices because they require other appliances (e.g., a smart ceiling fan controller).

Our dataset contains the remaining 11 devices, which are listed in Table I. This dataset encompasses devices of different types (cameras, smart sockets, bulbs, smart locks). Note that the respective companion apps of these devices are quite complex as they contain, on average, over 9 thousand classes, 56 thousand functions, and 766 thousand statements. The complexity of these apps is in line with the complexity of the apps used by the related work [79], which contains the largest dataset of validated IoT apps.

We installed the IoT devices in our laboratory, we deployed DIANE on an 8-core, 128GB RAM machine running Ubuntu 16.04, and we ran the Android companion apps on a Google Pixel and a Google Nexus 5X running Android 8.0. The smartphones, the IoT devices, and the machine running DIANE were connected to the same subnet, allowing DIANE to capture the generated network traffic. To configure each device, we manually performed its initial setup phase, such as registering an account on the device and on the Android companion app.

B. Fuzzing Trigger Identification

Table II shows the results of each step of DIANE's *fuzzing triggers* identification phase: For each IoT device, we report the protocols in use to communicate with the companion app, whether or not the app contains native code, if it sanitizes user inputs, the number of candidate *sendMessage* functions found by DIANE, the number of validated *sendMessage* functions, and the number of *fuzzing triggers*. For each intermediate result, we calculated the number of true positives and false positives, and investigated false negatives.

Since there is no available ground truth, we validated our ability to identify *sendMessage* functions and *fuzzing triggers* by manually reversing (both statically and dynamically) the Android companion apps in our dataset. Specifically, an expert analyzed each app for an average of five hours.

Reverse engineering of real-world apps is known to be difficult. Therefore, while we did our best to fully comprehend the dynamics of these apps, in a few cases we could not verify our results completely, as indicated in the following sections. We also acknowledge that this manual evaluation cannot completely exclude the presence of false negatives.

To measured DIANE's ability to find *sendMessage* functions precisely, we manually analyzed the *sendMessage* functions returned by the first two steps of our analysis. Specifically, we classified each function returned by the *sendMessage* candidates identification step (Step 1 in Section III-A) and by the *sendMessage* function validation step (Step 2 in Section III-A) as either true positive or false positive (TP and FP in the fifth and sixth columns of Table II). To perform this classification, we hooked each of these functions and manually exercised the IoT device's functionality

TABLE I
SUMMARY OF OUR DATASET OF IOT DEVICES (* ACCOUNT REQUIRED TO OPERATE THE DEVICE).

Device ID	Type	Vendor	Model	Firmware Vers.	Android App Package Name	App Vers.	Online Account*	Setup Time [Seconds]
1	Camera	Wansview	720P X Series WiFi	00.20.01	wansview.p2pwificam.client	1.0.10	✗	219
2	Camera	Insteon	HD Wifi Camera	2.2.200	com.insteon.insteon3	1.9.8	✓	427
3	Smart Socket	TP-Link	HS110	1.2.5	com.tplink.kasa_android	2.2.0.784	✗	311
4	Camera	FOSCAM	FI9821P	1.5.3.16	com.foscam.foscam	2.1.8	✓	406
5	Camera	FOSCAM	FI9831P	1.5.3.19	com.foscam.foscam	2.1.8	✓	403
6	Smart Socket	Belkin	Wemo Smart Socket	2.0.0	com.belkin.wemoandroid	1.20	✗	211
7	Bulb	iDevices	IDEV0002	1.9.4	com.idevicesllc.connected	1.6.95	✗	274
8	Smart Socket	iDevices	IDEV0001	1.9.4	com.idevicesllc.connected	1.6.95	✗	276
9	Camera	Belkin	NetCam	Unknown	com.belkin.android.androidbelkinnetcam	2.0.4	✓	1,040
10	Bulb	LIFX	Z	2.76	com.lifx.lifx	3.9.0	✓	313
11	Smart Lock	August	August Smart Lock	1.12.6	com.august.luna	8.3.13	✓	213

TABLE II
SUMMARY AND FEATURES OF OUR DATASET OF IOT COMPANION APPS. TP INDICATES A TRUE POSITIVE RESULT, FP A FALSE POSITIVE RESULT, AND NC A RESULT WE WERE NOT ABLE TO CLASSIFY EITHER AS TRUE POSITIVE NOR FALSE POSITIVE. **?** INDICATES THAT WE COULD NOT VERIFY WHETHER AN APP APPLIED DATA SANITIZATION. THE LAST THREE COLUMNS INDICATE THE COMPLEXITY OF THE APPS IN TERMS OF NUMBER OF CLASSES, FUNCTIONS, AND STATEMENTS RESPECTIVELY.

Device ID	Network Protocol	Native Code	Sanity Checks	No. Candidate sendMessage	No. sendMessage	No. Fuzzing Triggers	No. Classes	No. Functions	No. Statements
1	UDP	✓	✓	4 (1 TP, 3 FP)	1 (1 TP)	7 (6 TP, 1 FP)	4,341	31,847	409,760
2	HTTP	✓	✓	12 (8TP, 4FP)	9 (6 TP, 3 FP) ⋆	6 (6 TP)	11,870	76,558	1,180,817
3	TCP + JSON	✗	?	6 (2 TP, 4 FP)	6 (2 TP, 4 FP)	3 (2 TP, 1 FP)	16,461	107,935	1,267,785
4	UDP	✓	✓	10 (2 TP, 7 FP, 1 NC)	2 (2 TP)	2 (2 TP) •	6,859	41,256	615,410
5	TCP	✓	✓	10 (2 TP, 7 FP, 1 NC)	2 (2 TP)	2 (2 TP) •	6,859	41,256	615,410
6	HTTP + SOAP	✓	✗	15 (3 TP, 12 FP)	6 (2 TP, 4 FP)⋆	9 (8 TP, 1 FP)	4,169	30,462	378,733
7	TCP	✓	✓	8 (2 TP, 6 FP)	3 (2 TP, 1 FP)	4 (3 TP, 1 NC)	8,418	52,013	813,444
8	TCP	✓	✓	8 (2 TP, 6 FP)	3 (2 TP, 1 FP)	4(3 TP, 1 NC)	8,418	52,013	813,444
9	TCP	✓	?	6 (3 TP, 3 FP)	1 (1 TP)⋆	1 (1 TP)•	6,010	42,358	467,670
10	UDP	✓	?	9 (1 TP, 8 FP)	3 (1 TP, 2 FP)	0	5,646	33,267	457,719
11	Bluetooth	✓	✓	9 (4 TP, 5 FP)	9 (4 TP, 5 FP)	16 (14 TP, 2 FP)	22,406	108,507	1,411,798
Total		10/11	7/11	97 (30 TP, 65 FP, 2 NC)	45 (25 TP, 20 FP)	54 (47 TP, 5 FP, 2 NC)	101,457	617,472	8,431,990

• *fuzzing triggers* coincide with *sendMessage* functions.

TABLE III
SUMMARY OF THE BUGS DETECTED BY DIANE, IOTFUZZER, AND BY EXISTING NETWORK FUZZERS (BED, SULLEY, UFUZZ, AND BSS). NO. GENERATED ALERTS INDICATES THE NUMBER OF UNIQUE *fuzzing triggers* FOR WHICH DIANE AUTOMATICALLY GENERATED AT LEAST ONE ALERT. TIME INDICATES THE TIME REQUIRED TO FIND ALL THE REPORTED BUGS (AND THE NUMBER OF FUZZING INPUT GENERATED TO FIND THE BUGS). NO. FUZZED FUNCTIONS INDICATES THE NUMBER OF FUNCTIONS IDENTIFIED BY IOTFUZZER FOR FUZZING.

	DIANE					IoTFuzzer			Other Fuzzers			
Device ID	No. Generated Alerts	No. Bugs	Zero-day	Vuln. Type	Time [hours] (No. Generated Inputs)	No. Fuzzed Functions	No. Bugs	Time [hours]	No. Bugs BED	Sulley	uFuzz	bss
1	1	1	✓	Unknown	$\leq$ 0.5 (60,750)	• 1	0	N/A	N/A	0	N/A	N/A
2	3	7	✓	Buff overflow	$\leq$ 0.5 (322)	5	2	0.98	0	0	N/A	N/A
3	1	1		Unknown	$\leq$ 1.2 (7,344)	1	1	4	0	0	N/A	N/A
4	1	0		N/A	N/A	• 1	0	N/A	N/A	0	N/A	N/A
5	1	0		N/A	N/A	• 1	0	N/A	0	0	N/A	N/A
6	4	1		Unknown	$\leq$ 10 (34,680)	1	1	$\leq$ 10	0	0	0	N/A
7	3	0		N/A	N/A	N/A	N/A	N/A	0	0	N/A	N/A
8	3	0		N/A	N/A	N/A	N/A	N/A	0	0	N/A	N/A
9	0	0		N/A	N/A	3	0	N/A	0	0	0	N/A
10	1	0		N/A	N/A	N/A	N/A	N/A	N/A	0	N/A	N/A
11	0	† 1	✓	Unknown	2.2 (3,960)	N/A	N/A	N/A	N/A	N/A	N/A	0

• We manually instrumented IoTFuzzer to identify a valid send function.

† Vulnerability discovered through the watchdog device.

through its companion app, while monitoring the network traffic. We considered a candidate *sendMessage* function a true positive if: i) We registered network traffic when the companion app invoked the *sendMessage* function, and ii) the code and semantic of the function indicated network functionality. If either of these two conditions were false, we considered the *sendMessage* function a false positive. There were cases where the app was heavily obfuscated, and we could not establish if the considered *sendMessage* function was indeed sending data (NC in Table II).

As shown in Table II, DIANE was able to remove 45 false positive results during its *sendMessage* function validation step, . For Device IDs 2, 6, and 9 (indicated with ⋆), one might think that we lost some true positives during the validation step. However, this was not the case. After manual verification (using both static and dynamic analyses), we discovered that the missing true positives were just wrappers of other validated true positives. We also looked for false negatives, that is, *sendMessage* functions that were not identified as such. To the best of our ability, we found no such false negatives.

Overall, though we registered some false positives (20 in total), we always identified correctly *sendMessage* functions (i.e., no false negatives). We investigated the false positives and we found

that they were due to border functions containing calls to native methods, which were called within (or right before) the correct *sendMessage* functions. As such, their execution times were close to the actual *sendMessage* functions, causing our *sendMessage* validation step to label them as valid *sendMessage* functions. Also, it is important to say that false positive results do not affect the effectiveness of DIANE (i.e., the number of bugs found), rather its efficiency (i.e., the time spent to find those bugs). In fact, considering a non-*sendMessage* function as a *sendMessage* would only result in identifying additional, wrong *fuzzing triggers* that would not generate any network traffic when fuzzed, thus not affecting the IoT device.

For each true positive *sendMessage* function, we verified that DIANE correctly identified the top-chain functions (i.e., *fuzzing triggers*). *Fuzzing triggers* for Device IDs 4, 5 and 9 (marked with •) coincided with the *sendMessage* functions. This happens in apps that either do not have data-transforming functions, or where the functions that transform the data also embed the send functionality. Consequently, these functions are both *sendMessage* and top-chain functions.

For three apps (Device IDs 3, 9 and 10), we could not trace the data-flow from the identified *sendMessage* functions back up to the UI elements. This was due to imprecisions of the employed reverse engineering tools. Therefore, we could not establish whether they performed app-side data sanitization.

We also investigated false positives and negatives in the identified *fuzzing triggers*. Overall, our transformation data chain identification algorithm generated 5 false positives. In 2 cases, our backward slicer could not find any callers of a given function, and, therefore, our algorithm ended and considered the last detected data-transforming function f a *fuzzing trigger*. After manual verification, we found that the correct *fuzzing trigger*, in both cases, was a caller of function f. Although f is a valid data-transforming function, DIANE cannot assure that it is a top-chain function, as there might be another data-transforming function calling f that dominates f. The remaining 3 false positives were due to the fact that these functions introduced an entropy higher than our threshold, though they were not data-transforming functions. However, we maintained our threshold to 2 as this value is indicated as optimal by related work [80]. As we explained before, these false positives do not influence the effectiveness of DIANE, but only its efficiency.

Finally, we evaluated the false negatives generated by DIANE. To the best of our ability, we did not encounter any false negative while manually reversing the apps.

C. Vulnerability Finding

Finally, we fuzzed the obtained *fuzzing triggers*, and verified the alerts produced by our tool. Table III shows the results of our fuzzing. Note that, while DIANE can also use *sendMessage* functions as entry points for fuzzing, it identified all the detected bugs only when leveraging *fuzzing triggers*. We discuss the human effort required to verify the alerts produced by DIANE in Section IV-I.

We validated our findings as follows. The seven bugs for Device ID 2 were confirmed by analyzing both the network traffic and the camera firmware. Through the analysis of the firmware, we were able to verify our findings and craft a proof-of-work exploit that stalls the device for an arbitrary amount of time. We reported these bugs to the manufacturer, who confirmed our findings.

As for Device ID 1, after finding the candidate crash input, we verified it, through the app, by observing how the device behaved. We noticed that, after sending the crafted input, the device did not respond anymore, unless it was rebooted. Also, after fuzzing it for 24 hours the device entered a malfunctioning state, and we were unable to correctly restore it, even after multiple factory resets. We then purchased another camera of the same model, and the same result was obtained after 24 hours. We are still investigating to find whether some crash-inducing inputs we provided also cause irreparable damage to the device.

When validating the crash reports for the Device ID 3, we noticed that, after sending the crash-inducing input, the TCP connection was dropped, and the device response time significantly increased. We found that this bug, as well as the bug affecting the Device ID 6, were known vulnerabilities [25].

For Device ID 11 (a popular smart door lock), we noticed that after around two hours of fuzzing the device became unreachable for the watchdog device. Even more interestingly, the device then started to make an intermittent noise, which we realized being "SOS" encoded in morse code[3]. We then reset the door lock, and we observed that it started to show erratic behavior. For example, we noticed that it was not possible to control it through two different Android phones anymore: If the lock status was shown as "online" on one companion app, it would be "unreachable" on the same companion app on another phone. We are still working with the vendor to find the root cause of the problem.

We reported our findings to the appropriate manufacturers and, to the best of our knowledge, all bugs have been fixed.

D. DIANE *vs. IoTFuzzer*

To compare our approach to IoTFuzzer [25], we contacted the authors and obtained their tool. We also attempted to purchase the same devices used to evaluate IoTFuzzer, but we could only obtain Device 3 and 6, as the remaining ones were only available in China.

IoTFuzzer required manual intervention to be adapted to different devices and companion apps. In particular, we had to i) limit the scope of the analysis (i.e., number of hooked functions) to a subset of Java packages present in the Android apps—to keep the analysis tractable and avoid crashes—and ii) manually specify any encryption functions present in the app. After this manual configuration step, we were able to replicate the results presented in the original paper for the devices we were able to obtain (Device 3 and Device 6). Additionally, IoTFuzzer is based on TaintDroid, whose latest release supports up to Android 4.3 (2012). For this reason, we were not able to analyze Device 10 and Device 11, as their companion apps require newer Android SDK versions.

Our results are reported in Table III. IoTFuzzer crashed Device 3 and 6 (the two devices used in the original paper) and Device 2, but failed to find any bugs for the other 8 devices.

For Device 2, IoTFuzzer identified 5 functions to fuzz. We manually analyzed these functions and found that three of them were false positives, as they were used to save user information

[3]Audio recording: http://bit.ly/3oWcjgD

on the Android phone. To confirm our findings, we fuzzed these functions and observed that none of them generated network traffic.

Then, we proceeded to fuzz the two remaining functions, named `HouseExtProperty` and `changeCameraUsernamePassword`. While fuzzing the `HouseExtProperty` function for an hour, we discovered that the generated messages were directed to the vendor's cloud, rather than the actual device, therefore not producing any meaningful fuzzing input for the IoT device.

The `changeCameraUsernamePassword` function is, instead, used to change the credentials on the IoT device. We fuzzed this function for 24 hours, and IoTFuzzer rediscovered 2 of the 7 bugs that DIANE found on this device.

To understand better why IoTFuzzer missed some of the bugs we found, we examined `changeCameraUsernamePassword` (shown in Figure 4). This function calls the functions `cam.changeUsername` and `cam.changePassword` to generate the requests to change the username and password, respectively (the first argument of these functions represents the current username of the camera). Also, the variable `cam` is an internal structure that the app uses to store the details of the camera (e.g., the camera model), and its content is not directly influenced by the data received from the app's UI. On the other hand, both `newUsr` and `newPwd` contain user data, which is passed through the app's UI. As IoTFuzzer fuzzes only the function arguments that contain user data (when a function is invoked), it fuzzes the second and third function arguments, but it does not fuzz the first.

Unfortunately, as we explain in detail in Section IV-G, this camera contains a bug that can be exploited if the request generated by the companion app contains a username whose length is larger than a particular buffer size. However, by fuzzing the second two arguments of `changeCameraUsernamePassword` IoTFuzzer only mutate the second parameter of `cam.changeUsername` and `cam.changePassword`—`newUsr` and `newPwd` respectively—and it does not mutate their first parameter (`cam.user`), which would lead to the discovery of an additional bug. This case highlights a limitation of IoTFuzzer's approach, as it shows that assuming that all the data being sent to the device comes directly from the app's UI is ineffective to find bugs in an IoT device. On the other hand, our bottom-up approach, which bootstraps its analysis from *sendMessage* functions (see Section III), is agnostic with respect to the sources of input, and, therefore, is more generic.

In addition, `changeCameraUsernamePassword` allows one to modify the credentials only for specific camera models (Line 2, `cam.checkCameraModel`). This means that IoTFuzzer cannot effectively fuzz other camera models. By identifying a *fuzzing trigger* deeper in the control flow, DIANE, instead, bypasses this check and is effective independently from the device version.

For Device IDs 7 and 8, IoTFuzzer caused the app to crash immediately due to the number of hooked functions. We narrowed the analysis to only the package containing the code to interact with the device, but the app would crash regardless. Thus, we could not run IoTFuzzer on these devices.

```
1 boolean changeCameraUsernamePassword(Camera
  cam, String newUsr, String newPwd) {
2   if(cam.checkCameraModel()) {
3     if(cam.user.compareTo(newUsr) != 0)
4       cam.changeUsername(cam.user, newUsr);
5     if(cam.pwd.compareTo(newPwd) != 0)
6       cam.changePassword(cam.user, newPwd);
7   }
8   //...
9 }
```

Fig. 4. Fuzzing function found by IoTFuzzer for the Insteon camera (Device ID 2). We report only the relevant code for space reasons.

```
1 void
  changeCredentials(String newUsr, String newPwd) {
2   if(this.confirm_credentials()) {
3     if(!this.get_user().equals(newUsr)
  && !this.get_pwd().equals(newPwd))
4       this.changeUserAndPwd(newUser, newPwd);
5     //...
6   }
7 }
```

Fig. 5. Fuzzing function found by IoTFuzzer for the Foscam cameras companion app (Device IDs 4 and 5).

For Device ID 9, IoTFuzzer identified 3 functions to fuzz. However, we found these functions to be false positives, as they were used to log user data on the smartphone.

For Devices IDs 1, 4, and 5 (marked with • in Table III) IoTFuzzer failed to identify any functions to fuzz. The reason is that to find a function to fuzz, IoTFuzzer has to first find a data flow between a UI element of the app and the Android's socket send function. However, in these devices the "send" functionality is implemented in native code (i.e., these devices do not rely on the Android's send function). As IoTFuzzer cannot identify send functions in native code, it failed to identify what UI events would eventually generate network traffic, and, therefore, it did not generate any valid fuzzing inputs. DIANE overcomes this limitation by using dynamic analysis, and find the border functions that generate network traffic, as explained in Section III-A.

To help IoTFuzzer and have a direct comparison with our tool, we hard-coded the send functions found by DIANE in IoTFuzzer, and re-ran the analysis for these devices. For Device IDs 4 and 5, IoTFuzzer identified one candidate function to fuzz, which, similarly to Device ID 2, is used by the app to change the device's credentials. This function is depicted in Figure 5, and it implements a check (through `confirm_credentials`) that asks the user to provide their credentials in order to proceed. As a result, fuzzing `changeCredentials` did not produce any meaningful input to the camera, as the check would constantly fail. Instead, DIANE identified as a fuzzing trigger the function `changeUserAndPwd`, which is not affected by any checks, and effectively sends commands to the camera when fuzzed. These cases highlight another limitation of IoTFuzzer's approach, as they show that fuzzing the first function in the app's control flow that handles user-provided data is ineffective.

For Device ID 1, IoTFuzzer identified a function called `setUser`, which sends the user's login information to the device. In this case, this function is guarded by a check that forbids the user's password to contain some special characters (e.g., "&"). We fuzzed this function for 24 hours and we did not register any

anomaly in the device. Also in this case, DIANE selected a function deeper in the control flow of the app, after any client-side checks. This was necessary to successfully discover a (zero-day) bug.

Overall, DIANE performed as well as IoTFuzzer only in two cases (Device IDs 3 and 6), and it outperformed IoTFuzzer in all the other cases—either because IoTFuzzer was unable to identify any meaningful send functions, or because it did not produce any crash-inducting input.

This evaluation highlights the importance of carefully selecting the right function to fuzz within the companion app, and that app-side sanitization checks hinder the efficacy of a fuzzing campaign. This issue is exacerbated by the frequency in which app-side sanitization is present in companion apps. For instance (as shown in Table II), in our dataset we found that at least 7 out of 11 apps contain sanity checks. We further measure this aspect in Section IV-E.

E. App-side Sanitization and Fuzzing Triggers

App-side Sanitization. To evaluate how common app-side sanitization code is in companion apps, we first manually reverse-engineered the 11 companion apps of the IoT devices in our dataset. As shown in the *Sanity Checks* column of Table II, at least 7 out of 11 apps contain sanity checks.

As an additional evaluation of this aspect, we performed a large-scale study on the presence of app-side sanitization code in companion apps. For this experiment, we used 2,081 apps, which we gathered from related work [79]. This dataset is ideal for our evaluation as it specifically contains Android companion apps of IoT and smart home devices, which have been collected from the Google Play Store and manually inspected by the authors of the related work. To the best of our knowledge, this is the largest dataset of validated IoT companion apps. Since we did not have access to all the physical devices that these apps interact with, we could not run DIANE against them, and, therefore, we implemented a fully-static automated approach, suitable for a large-scale study.

Specifically, given a companion app, we identified its *sendMessage* functions by locating functions that contained I/O operations (as detailed in Appendix A). We were able to identify *sendMessage* functions for 1,304 of the apps (~63%). For the remaining apps, we were not able to statically identify any network-related operations, as we could not find, for instance, a socket send operation. Then, we performed an inter-procedural backward slice from every argument of each identified *sendMessage* function, and considered the instructions in each slice. Finally, we counted the comparisons against constant data (e.g., using a string comparison in a `if` statement) in these slices.

In this experiment, we found that 663 (~51%) companion apps implement sanitization of the data being sent, and that, on average, the variables handled by a *sendMessage* function are affected by 7 checks across the companion app. To validate these results, we randomly selected 100 *sendMessage* functions and found 85 to be true positives, 14 to be false positives (these functions were sending messages to another Android thread), and for 1 of them we could not determine its functionality, as it was heavily obfuscated. Also, we randomly sampled 30 functions that we detected were applying input sanitization code, and found 29 to be true positives: the companion app applied checks on the user data.

These results show how app-side sanitization code is common in companion apps. Note that, this experiment is only an approximation of our approach, which requires the physical devices to be fully effective. Therefore, these results do not aim to evaluate our approach, rather they serve as an indication of the presence of input validation code in mobile apps. Our results are in line with a recently published study [86].

Fuzzing Triggers. We also evaluated how prevalent *fuzzing triggers* are in Android companion apps. As DIANE relies on dynamic analysis to find *fuzzing triggers*, we replaced the parts of our approach that leverage dynamic analysis with symbolic execution. We used the Java support provided by the angr [71] tool to symbolically execute the app's functions in a slice (see Algorithm 1), so to calculate the Shannon entropy. In particular, we concretize the input of a function (i.e., its live variables) with known values, symbolically execute the function, and observe the values in the output (i.e., its live variables when the function returns). Then, we replicate our approach explained in Section III-A, and calculate the difference of entropy introduced by each function to identify the data-transforming functions.

We sampled 100 apps from the 2,081 aforementioned apps, ran our analysis, and manually verified the results. For 37 apps, our analysis found *fuzzing triggers*, and for the remaining 63, it did not. We investigated our results and found that our analysis correctly identified a *fuzzing trigger* for 25 of the 37 apps, and it produced false positives in the remaining 12 cases. These false positives were due to imprecisions in our inter-procedural backward slicer (i.e., our static analysis could not find the callers of a given function).

On the other hand, in 63 apps our analysis did not find any *fuzzing trigger* because of imprecisions of the symbolic execution. In fact, to keep the analysis tractable, we symbolically execute every function up to 10 minutes and follow up to 2 consecutive function calls (we drop the collected symbolic constraints when a function call is not followed). As such, when the analysis fails to calculate the added entropy of a given function, we stop the analysis.

Overall, we found *fuzzing triggers* for 25% of the analyzed apps. While this number sufficiently demonstrates that such sweet spots are, indeed, present in many apps, we highlight that, in our analysis, this is a lower bound. In fact, our attempt to emulate our approach using symbolic execution introduces imprecisions that would not occur when using DIANE together with the real devices. Therefore, we expect this number to be even higher in practice. This further emphasizes the need for a system that can identify fuzzing triggers that are located past client-side checks in the companion apps.

F. DIANE *vs. Network-Level Fuzzing*

We also compared DIANE to well-known network fuzzers: BED [2], Sulley [3], uFuzz [13] (UPnP endpoints), and bss [4] (Bluetooth fuzzer). Table III shows the results of the comparison. Note that the labels N/A indicate that the corresponding network fuzzer does not handle the network protocols employed by the corresponding IoT device.

We configured BED and Sulley as indicated by previous work [25], and the remaining tools as suggested by their related web pages. We ran each tool for 24 hours. However, uFuzz finished its fuzzing cycle before the allocated time, and bss was

```
1  public static String httpRequest(String req, ...){
2    // perform the requested HTTP request
3  }
4  /* Camera class */
5  private Result sendCommand(String cmd, TreeMap t){
6
   String fmt = "http://%s/CGIProxy.fcgi?cmd=%s:%s";
7
   toSend = String.format(fmt,CAMERA_ENDPOINT, cmd);
8    Iterator it = t.keySet().iterator();
9    while(it.hasNext()) {
10     String key = (String)it.next();
11     String val = (String)t.get(key);
12     toSend
   += "&" + key + "=" + this.encodeUrlParam(val);
13   }
14   String encUser = this.encodeUrl(this.user);
15   String encPwd = this.encodeUrl(this.passwd);
16   fmt = "&usr=%s&pwd=%s"
17   toSend += String.format(fmt, encUser, encPwd);
18   HttpUtil.httpRequest(toSend,"GET",null,10,10);
19 }
20
21 public boolean
   changePassword(String user, String newPwd) {
22   TreeMap t = new TreeMap();
23   t.put("usrName", user);
24   t.put("newPwd", newPwd);
25   res = this.sendCommand("changePwd", t).resCode;
26   return res != ResCode.SUCCESS ? false : true;
27 }
28
29 boolean changeCameraUsernamePassword(Camera
   cam, String newUsr, String newPwd) { /*...*/ }
```

Fig. 6. Snippet of code for the Insteon Camera app.

not able to generate input for Device ID 11, as the device does not accept connections outside the companion app.

Overall, no bugs were found by any of these network fuzzing tools. The reason why no network fuzzers triggered any crash is that these fuzzers are general-purpose [83], [68], and they fail to trigger deeper code paths in the devices' firmware. For instance, BED only fuzzes HTTP headers without considering the syntax or the semantics of HTTP payloads.

G. Case Study: Insteon HD Wifi Camera

In this section, we present a case study regarding two bugs that DIANE found in the Insteon Camera (Device ID 2). Note that, these bugs have been fixed in the latest version of the firmware running on the Insteon camera.

Among the functionality offered by the app, a user can change their credentials (username and password). Figure 6 depicts a simplified version of the app's code that accomplishes this task (we omit the code of the function `changeCameraUsernamePassword` as it is already shown in Figure 4). In particular, when the user wants to change their password, the companion app invokes the function `changeCameraUsernamePassword` (Line 29). As explained in Section IV-D, this function first checks that the camera belongs to a certain camera family, and if so, the app invokes the function `changePassword` (Line 21). This function creates a TreeMap structure containing couples "key:values," which will be placed in the request generated by the app. Then, `changePassword` invokes the `sendCommand` function (Line 5), which is a helper function used to send commands to the camera. This function prepares the request by using the

```
1 int key_strcpy(char *dst, char *URI, char *key){
2   int len = 0;
3   char* val = get_ptr_val(URI, key, &len);
4   strncpy(dst, val, len);
5   return 0 ;
6 }
```

Fig. 7. Simplified snippet of code from Insteon firmware.

TreeMap, and it eventually calls `httpRequest` (Line 1) to send the request to the device.

For this particular device, we could gather the firmware running on the camera (by sniffing the wireless network during the initial firmware update). Figure 7 shows a simplified version of the firmware function used to copy the values of parameters from a given URI. This function acts as an unsafe `strcpy`: it takes as input a destination buffer (allocated by the caller function) and copies the value of a pair "key:value" present in a given URI. This function is called 789 times within the Insteon firmware, and, to the best of our knowledge, for 9 of them, we can trigger a buffer overflow. In particular, when a user wants to change the camera password, the firmware allocates two buffers on the stack, and it uses this function to copy the username and new password values from the URI into the allocated buffers (of 88 and 64 bytes respectively). As a result, if we provide two values for username and password large enough, we can trigger two buffer overflows.

By looking at the code in Figure 6 and Figure 4, we can see that fuzzing the function `encodeUrl` (Lines 12,14, and 15 in Figure 6) allows us to i) skip any app-side validation (Line 2 in Figure 4), and ii) trigger both bugs discovered by IoTFuzzer (as shown in Section IV-D) and two additional bugs due to a long username and password.

DIANE identified 9 different *sendMessage* functions (6 true positives), and 6 *fuzzing triggers* (6 true positives) for the Insteon Camera companion app. Among these, DIANE automatically identified the function `httpRequest` as a *sendMessage* function, and the function `encodeUrl` as a *fuzzing trigger*. When DIANE fuzzed `encodeUrl`, DIANE immediately generated an alert.

Finally, note that the `sendCommand` represents another valid *fuzzing trigger* for `httpRequest`, as it modifies the command being sent. Indeed, DIANE correctly identified `sendCommand` as a further *fuzzing trigger*.

H. Runtime Performance

We assessed the runtime performance of our tool by measuring the execution time required by the *fuzzing triggers* identification phase. In our experiments, we setup DIANE to run the fuzzing phase for 24 hours. First, we measured the entire execution time required, on average, for DIANE to analyze an app and identify *fuzzing triggers*. DIANE analyzes a given app in slightly less than 150 minutes on average. Figure 8 shows the average and standard deviation of the execution time required for each phase of our analysis process. As shown in Figure 8, the execution time of DIANE has a high standard deviation. This is due to the following implementation detail: Frida, which we leverage to hook Android APIs and methods at runtime, sometimes fails, causing the running app to crash. This requires automatically restarting the hooking procedure, randomly slowing down DIANE.

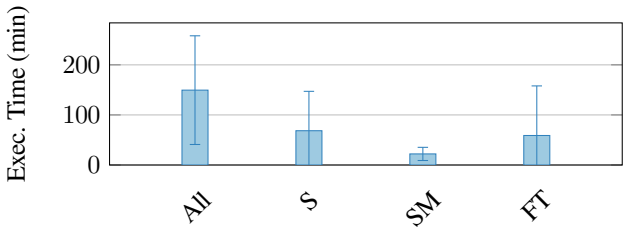

Fig. 8. Average and standard deviation of the execution time of the phases that DIANE performs (**S** = Setup, **SM** = *sendMessage* functions Identification, **FT** = Fuzzing Trigger Identification).

I. Quantifying Required Human Effort

We evaluated the human effort required to use DIANE. In general, DIANE scales linearly with the number of analyzed devices, as it requires the analyst to perform the same steps for each new analyzed device. DIANE requires human intervention to setup a new IoT device. During this phase, an analyst has to install the IoT device, which involves installing the companion app, configuring the device, and, in some cases, registering an online account. In addition, during this phase, DIANE requires the analyst to run the app and test the basic functionality of the IoT device, so that our tool can record the generated UI interactions (see Section III). We measured the time we spent to setup each device, as reported in Table I. On average, we spent 6 minutes and 12 seconds to setup a new IoT device, of which 41 seconds were spent to interact with the device. Note that, an analyst has to take these steps only once per device.

Human effort is also required if the analyst desires to monitor the state of the watchdog device during fuzzing (recall Section III-B). The watchdog device is optional and useful if the analyst wants to detect semantic issues. In this case, the analyst has to check whether the functionality automatically exercised by the watchdog device results in an undesired effect in the IoT device due to our fuzzer triggering a vulnerability (e.g., unauthenticated requests suddenly open a door lock). The frequency of these manual checks depends on the analyst, as they might want to monitor the watchdog device at regular intervals for the whole duration of the fuzzing campaign, or only at the end of it. In our fuzzing campaign we checked the watchdog device approximately every two hours. In our experiments, we needed the watchdog device only to detect the issue for Device ID 11, as explained in Section IV-C. The other 10 bugs were automatically detected by DIANE by monitoring the network traffic, as explained in Section III-B.

When a bug is detected, DIANE generates an alert. In this case, an analyst may want to manually reproduce and verify the bug triggering the alert. DIANE allows this manual verification, since it produces as output the input triggering the detected bug. With this information in hand, the analyst can use DIANE to send the crashing input to the analyzed IoT device. Then, the analyst can manually check the device functionality to assess if it misbehaves after receiving the crashing input (e.g., the device reboots or does not reply to further requests). In our evaluation, we needed about 6 minutes, on average, to follow this procedure and verify each alert produced by DIANE.

V. Limitations and Future Work

While we addressed the major challenges for performing black-box fuzzing of IoT devices, our overall approach and the implementation of DIANE still have some limitations.

We currently cannot bypass app-side sanity checks when they are implemented in native code, in a data-transforming function or directly in a *sendMessage* function. Though we acknowledge that such checks could be present in any of these classes of code, we manually verified that none of the apps in our dataset contain sanity checks in any of these categories. In fact, as shown by previous work [16], native code is typically not used to implement the main application's logic, but it is used, instead, in library helper functions Also, note that, differently from previous work, this does not mean that DIANE cannot handle native code at all. In fact, even if the *sendMessage* function is implemented natively, DIANE can identify it and fuzz its *fuzzing triggers*. However, if sanity checks are present in any of the aforementioned classes of code, the fuzzing is less effective.

As any approach based on dynamic analysis, DIANE suffers from limited code coverage, i.e., it cannot identify *fuzzing triggers* that are not executed by the app. To mitigate this limitation, we manually stimulate the apps to trigger most of the available functionality, and we perform our analysis on real smartphones.

The current implementation of DIANE cannot fuzz nested Java objects. We plan to address this in future work.

DIANE could be enhanced to automatically discover semantic vulnerabilities (e.g., a smart lock unlocks a door instead of locking it). Currently, this feature is semi-automatic as it requires the analyst to check and interact with the watchdog device.

VI. Related Work

IoT devices are plagued with vulnerabilities [59], [48], [27], [18]. Consequently, different automated vulnerability detection tools [31], [32], [70], [71] have been proposed, relying on static analysis techniques. Unfortunately, the applicability of these tools is limited since they require the device firmware.

Dynamic analysis, and in particular fuzzing [60], is a popular alternative technique that mitigates the problems of static analysis at the cost of missing potential bugs. American Fuzzy Lop (AFL) [85], along with its several improved versions [19], [20], [56], [35], [67], is the most popular coverage-guided fuzzing tool.

However, coverage-guided fuzzing is hard to perform on real devices, as it requires to keep track of the code locations reached while fuzzing the analyzed firmware. It is possible to perform this tracking using hardware debugging capabilities [72], but, unfortunately, they are usually disabled in consumer devices [55], [61], [25]. Other approaches have proposed to fuzz IoT devices by emulating the corresponding firmware [87], [73], [43], [24], [42]. Unfortunately, a faithful emulation of a firmware image is a hard problem, and these approaches have scalability issues.

Input generation is one fundamental capability underpinning any fuzzing technique. Taint tracking [44], [36], [26], symbolic execution [22], [38], [39], [23], [74], [62] and static analysis [57], [69] have been commonly used to handle the problem of input generation. Unfortunately, all of these techniques require to instrument the analyzed firmware code.

Grammar-based fuzzing techniques [66], [77] side-step the problem of input generation by requiring a model of the inputs [41], [46], [37], [54], [33] accepted by the program under test. Consequently, techniques based on static analysis [30] and dynamic analysis [58], [28], [45] have been developed to create models of valid inputs accepted by the analyzed software. However, these approaches are not suitable to fuzz IoT devices, as user requests are usually encrypted by the companion app, and devices might rely on ad-hoc grammars.

DIANE solves the problem of input generation by using network traffic and hybrid analysis of the app controlling the target IoT device. Recently, Chen et al. proposed IoTFuzzer [25]. IoTFuzzer analyzes the IoT device's APK, and automatically finds paths where user-generated input is handled. Then, for each of them, it fuzzes the arguments of the first function handling such data. While IoTFuzzer shares some similarities with our approach, it suffers from a number of drawbacks, which we discussed in detail in Section II. Wang et al. [79] leverage an IoT device's companion app to detect the characteristics of the device (e.g., a smart plug), and infer its vulnerabilities by relying on a set of known vulnerabilities affecting similar devices. However, their approach does not find unknown vulnerabilities.

RPFuzzer [81] finds vulnerabilities in router protocols. However, its techniques are customized to find routing-related issues, and are not applicable to generic IoT devices.

Peng et al. [67] recently proposed T-Fuzz, an approach for input generation that first transforms the target program to skip checks on input values, and then fuzzes the transformed program. T-Fuzz goal is to increase code coverage by bypassing input validation – T-Fuzz flips the if-conditions guarding those branches where the fuzzer gets stuck. However, T-Fuzz is not suitable for our scenario, since it requires access to the hardly available firmware of the IoT device.

T-Fuzz could be used to fuzz the companion app. However, our goal is not to increase our coverage of the companion app, but to identify *fuzzing triggers* and use them to fuzz the code running in the analyzed IoT device. In fact, T-Fuzz flips if-conditions without considering whether they are part of sanitization functions or data-transforming functions. As such, T-Fuzz would inevitably modify parts of the companion app's code that are essential in producing valid inputs for the IoT device. Therefore, applying the T-Fuzz approach on a companion app will not increase the effectiveness of the fuzzing of the controlled IoT device. For this reason, we need an approach like DIANE, which is able to selectively determine the checks to be bypassed.

Similarly, Wang et al. [78] proposed TaintScope, a *directed checksum-aware* fuzzing approach that bypasses sanity checks to penetrate deeper in the analyzed program. As such, TaintScope prevents generated test cases from being prematurely dropped by integrity checks on the targeted program. However, this tool is applicable only when the firmware of the device is available. When the firmware is available, other dynamic analysis tools like AVATAR [84], SURROGATES [52], FIRM-AFL [87], and FirmaDyne [24] could be used to improve the effectiveness of vulnerability identification. On the other hand, we performed our evaluation on real devices, without the need for firmware.

Data-flow tracking tools could be helpful to identify our *fuzzing triggers*. Dynamic analysis tools track interactions within the Dalvik [34] or the ART [75] runtime. However, it is not trivial to extend these techniques to apps containing native code.

Ispoglou et al.[47] proposed FuzzGen, a tool to automatically generate fuzzing harnesses for libraries. Unfortunately, this approach is not directly applicable to our usage scenario. In particular, FuzzGen requires the code of the library it generates the harness for. Therefore, it cannot be applied to IoT devices, since their code is typically unavailable. FuzzGen's approach could be used to generate fuzzing harnesses for the libraries used by the companion apps. However, this approach would not be effective in generating inputs targeting the analyzed IoT device. In fact, in companion apps, the logic to prepare and send well-structured messages to the controlled IoT device is usually implemented in the core of the companion app itself, rather than in its libraries.

For known protocols, network-based approaches like BED [2], Katyusha [12], WSFuzzer [14] could be used to perform fuzzing. However, extending them to custom data formats used by IoT devices is non-trivial given the heterogeneity of these devices. Furthermore, these tools cannot handle challenge-response sequences [21], heavily limiting their fuzzing effectiveness.

Finally, Junior et al. [50] assessed the security of the communication channels between IoT devices and their companion apps, identifying several flaws (e.g., lack of authentication).

VII. CONCLUSIONS

In this paper, we studied the effectiveness of IoT device fuzzers. On the one hand, randomly fuzzing network packets sent to the devices requires knowledge about the data format accepted by a device, which is seldom available when devices use custom firmware. On the other hand, approaches that leverage the UI of the companion mobile app to produce syntactically correct messages are ineffective because of the constraints that the app-side code imposes.

Conversely, we proposed a novel approach that sits in the sweet spot between network-level fuzzing and UI-level fuzzing. Our approach aims at identifying *fuzzing triggers*, which are portions of code in the IoT companion apps that are executed after input validation and right before any data-transforming function, and that maximize the fuzzing outcome. We implemented our approach in a tool, called DIANE, and evaluated it on 11 real-world IoT devices of different brands. DIANE outperforms the current state-of-the-art approach, and it can successfully detect critical bugs (9 zero-days) that cannot be triggered by existing fuzzers.

ACKNOWLEDGEMENTS

We would like to thank our reviewers for their valuable comments and inputs to improve our paper. This material is based upon work supported by AFRL under Award No. FA8750-19-C-0003, and by ONR under Awards No. N00014-17-1-2011 and N00014-17-1-2897. Research was also sponsored by DARPA under agreement number HR001118C0060. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the U.S. Government, or the other sponsors.

REFERENCES

[1] "500 internal server error," https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/500.

[2] "bed - A network protocol fuzzer," https://tools.kali.org/vulnerability-analysis/bed.

[3] "boofuzz: Network Protocol Fuzzing for Humans, successor to the venerable Sulley fuzzing framework," https://github.com/jtpereyda/boofuzz.

[4] "BSS (Bluetooth Stack Smasher) Fuzzer," https://securiteam.com/tools/5NP0220HPE/.

[5] "CVE Database for Iot Vulns1," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=router.

[6] "CVE Database for Iot Vulns2," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=belkin.

[7] "CVE Database for Iot Vulns3," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=smart-home.

[8] "Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers." https://frida.re/docs/android/.

[9] "Jtag debugging," https://blog.attify.com/hack-iot-device/.

[10] "Jtag fuse and protection using a trusted execution environment," http://www.freepatentsonline.com/9021585.html.

[11] "Jtag fuse flow," https://e2e.ti.com/support/microcontrollers/msp430/f/166/t/18936?JTAG-FUSE-BLOW.

[12] "Katyusha rest and soap fuzzer," https://github.com/lpredova/Katyusha.

[13] "UFuzz, or Universal Plug and Fuzz, is an automatic UPnP fuzzing tool." https://github.com/phikshun/ufuzz.

[14] "Web services fuzzing tool for http and soap," https://sourceforge.net/projects/wsfuzzer/files/.

[15] "Debugging Bluetooth With An Android App," https://blog.bluetooth.com/debugging-bluetooth-with-an-android-app, 2016.

[16] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *The Network and Distributed System Security Symposium*, 2016, pp. 1–15.

[17] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based iot deployments," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[18] Ben Herzberg, Dima Bekerman, Igal Zeifman, "Breaking Down Mirai: An IoT DDoS Botnet Analysis," https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html.

[19] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.

[20] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1032–1043. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978428

[21] R. M. Bolle, J. H. Connell, and N. K. Ratha, "System and method for liveness authentication using an augmented challenge/response scheme," Feb. 1 2005, uS Patent 6,851,051.

[22] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *Proceedings of the 2006 ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 322–335. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180445

[23] G. Campana, "Fuzzgrind: un outil de fuzzing automatique," *Actes du*, pp. 213–229, 2009.

[24] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, 2016.

[25] 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 *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2018.

[26] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *arXiv preprint arXiv:1803.01307*, 2018.

[27] Chris Brook, "TRAVEL ROUTERS, NAS DEVICES AMONG EASILY HACKED IOT DEVICES," https://threatpost.com/travel-routers-nas-devices-among-easily-hacked-iot-devices/124877/.

[28] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, ser. SP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 110–125. [Online]. Available: http://dx.doi.org/10.1109/SP.2009.14

[29] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[30] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2123–2138.

[31] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares." in *USENIX Security Symposium*, 2014, pp. 95–110.

[32] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *USENIX Security Symposium*, 2013, pp. 463–478.

[33] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using clp (t)," in *Proceedings of the 2015 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 482–493. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.65

[34] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[35] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[36] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 2009 International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 474–484. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070546

[37] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 206–215. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375607

[38] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[39] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *Proceedings of the 2008 Symposium on Network and Distributed System Security*, ser. NDSS '08, San Diego, CA, USA, 2008.

[40] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 72–81.

[41] G. Grieco, M. Ceresa, and P. Buiras, "Quickfuzz: An automatic random fuzzer for common file formats," in *Proceedings of the 2016 International Symposium on Haskell*, ser. Haskell '16. New York, NY, USA: ACM, 2016, pp. 13–20. [Online]. Available: http://doi.acm.org/10.1145/2976002.2976017

[42] Z. Gui, H. Shu, and J. Yang, "Firmnano: Toward iot firmware fuzzing through augmented virtual execution," in *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, 2020, pp. 290–294.

[43] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 135–150. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/gustafson

[44] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowser: a guided fuzzer to find buffer overflow vulnerabilities," in *Proceedings of the 2013 USENIX Security Symposium*, ser. SEC '13, Washington, DC, USA, 2013, pp. 49–64.

[45] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2345–2358.

[46] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments." in *Proceedings of the 2012 USENIX Security Symposium*, ser. SEC '12, Bellevue, WA, USA, 2012, pp. 445–458.

[47] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

[48] James Lyne, "Uncovering IoT Vulnerabilities in a CCTV Camera," https://www.rsaconference.com/videos/demo-uncovering-iot-vulnerabilities-in-a-cctv-camera.
[49] L. Jost, "Entropy and diversity," *Oikos*, vol. 113, no. 2, pp. 363–375, 2006.
[50] D. M. Junior, L. Melo, H. Lu, M. d'Amorim, and A. Prakash, "Beware of the app! on the vulnerability surface of smart devices through their companion apps," *arXiv preprint arXiv:1901.10062*, 2019.
[51] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "Ddos in the iot: Mirai and other botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.
[52] K. Koscher, T. Kohno, and D. Molnar, "Surrogates: Enabling near-real-time dynamic analyses of embedded systems." in *WOOT*, 2015.
[53] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric, "All things considered: An analysis of iot devices on home networks," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1169–1185.
[54] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '15. New York, NY, USA: ACM, 2015, pp. 386–399. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814319
[55] K. Lee, Y. Lee, H. Lee, and K. Yim, "A brief review on jtag security," in *2016 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, July 2016, pp. 486–490.
[56] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," *CoRR*, vol. abs/1709.07101, 2017. [Online]. Available: http://arxiv.org/abs/1709.07101
[57] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 627–637. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106295
[58] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 2010 Annual Information Security Symposium*, ser. CERIAS '10. West Lafayette, IN: CERIAS - Purdue University, 2010, pp. 5:1–5:1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2788959.2788964
[59] Lucian Constantin, "Hackers found 47 new vulnerabilities in 23 IoT devices at DEFCON," https://www.csoonline.com/article/3119765/security/hackers-found-47-new-vulnerabilities-in-23-iot-devices-at-def-con.html.
[60] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279
[61] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, San Diego, UNITED STATES, 02 2018. [Online]. Available: http://www.eurecom.fr/publication/5417
[62] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The borg: Nanoprobing binaries for buffer overreads," in *Proceedings of the 2015 ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '15. New York, NY, USA: ACM, 2015, pp. 87–97. [Online]. Available: http://doi.acm.org/10.1145/2699026.2699098
[63] F. Nielson, H. Riis Nielson, and C. Hankin, *Principles of Program Analysis*, 01 1999.
[64] A. Nordrum, "The internet of fewer things [news]," *IEEE Spectrum*, vol. 53, no. 10, pp. 12–13, 2016.
[65] J. Palsberg and M. I. Schwartzbach, *Object-oriented type inference*. ACM, 1991, vol. 26, no. 11.
[66] Peach, "The peach fuzzer," 2017, http://www.peachfuzzer.com/.
[67] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.
[68] P. Pokorny and M. Royal, "Dumb fuzzing in practice," 2012.
[69] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.
[70] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2020.
[71] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, 2015.
[72] M. Smith, M. Helmi, and J. Miller, "Comparison of approaches to use existing architectural features in embedded processors to achieve hardware-assisted test insertion," *Proceedings Work-in-Progress Session*, 2010.
[73] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "FirmFuzz: Automated IoT Firmware Introspection and Analysis," in *Proc. ACM CCS Workshop on IoT Security and Privacy (IoT S&P)*, 2019.
[74] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings of the 2016 Network and Distributed System Security Symposium*, ser. NDSS '16, San Diego, CA, USA, 2016.
[75] M. Sun, T. Wei, and J. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 331–342.
[76] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing java bytecode using the soot framework: Is it feasible?" in *International conference on compiler construction*. Springer, 2000, pp. 18–34.
[77] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 579–594.
[78] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. San Jose, CA, USA: IEEE, 2010, pp. 497–512.
[79] X. Wang, Y. Sun, S. Nanda, and X. Wang, "Looking from the mirror: Evaluating iot device security through mobile companion apps," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1151–1167. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/wang-xueqiang
[80] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," in *Computer Security – ESORICS 2009*, M. Backes and P. Ning, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 200–215.
[81] Z. Wang, Y. Zhang, and Q. Liu, "Rpfuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing." *KSII Transactions on Internet & Information Systems*, vol. 7, no. 8, 2013.
[82] H. Wen, Q. Zhao, Q. A. Chen, and Z. Lin, "Automated Cross-Platform Reverse Engineering of CAN Bus Commands From Mobile Apps," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2020.
[83] T. Wilson, "Evaluation of fuzzing as a test method for an embedded system," 2018.
[84] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *NDSS*, 2014.
[85] M. Zalewski., "American fuzzy lop," 2017, http://lcamtuf.coredump.cx/afl/technical_details.txt.
[86] Q. Zhao, C. Zuo, D.-G. Brendan, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," 2020.
[87] 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 *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1099–1114. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng
[88] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang, "Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1133–1150. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zhou

APPENDIX

In this appendix, we provide technical details about DIANE's different components. We implemented DIANE in about 4,500 lines of Python code, following the high-level architecture depicted in Figure 2. DIANE is implemented on top of pysoot [4], which leverages Soot [76] to translate the companion app's bytecode into an intermediate representation. DIANE currently only handles Android applications.

[4]https://github.com/angr/pysoot/

A. Static Analysis

To find the initial set of *sendMessage* candidates within a companion app, we analyze its internal representation. In particular, we select all those functions that either contain calls (Soot intermediate-representation `invoke` instructions) to native methods (having the `native` attribute) or calls to methods in the Android framework known to implement network I/O operations (e.g., `java.net.*`, `javax.net.*`, or `android.net.*`). By applying these rules, we obtain a list of functions that, when invoked, potentially send network messages to the IoT device.

B. Dynamic Analysis

APK Instrumentation. To hook methods of the APK under analysis and to fuzz them, we use Frida [8]. More precisely, each method is hooked and dynamically modified to include additional code. This injected additional code is used to enable fuzzing of the method arguments and of the used class fields and to extract information necessary for our analysis, such as the timestamp when the method is invoked and the contents of its parameters.

Network Interception. DIANE intercepts the network traffic generated by the companion app at runtime. DIANE supports the interception of traffic sent using both the WiFi and Bluetooth interfaces. Note that our approach is independent of the specific network medium and only requires to passively observe the communication channel without accessing the content of the exchanged data. For traffic transmitted over WiFi, DIANE leverages a router and the tool `tcpdump` to capture the packets sent from the smartphone to the IoT device, filtering the IP addresses. Traffic transmitted using the Bluetooth interface is instead captured using the *Bluetooth HCI snoop* Android debugging functionality [15]. Unless otherwise specified, we use the term network activity to refer both to WiFi and Bluetooth network traffic.

Fuzzing Objects. DIANE fuzzes both primitive variables (e.g., `int`, `float`) and class instances. To do this we use pysoot to retrieve the class definition of the considered class instances, and we fuzz each field whose type is either primitive or known (e.g., `java.lang.String`).

C. Hybrid Analysis

Fuzzing Trigger Identification Details. To implement the *fuzzing triggers* algorithm described in Section III-A, we implemented a static inter-function backward slicer on top of pysoot. Theoretically, the backward slice of a given variable might traverse an arbitrary number of functions. Therefore, to keep our analysis tractable, our backward slicer algorithm adopts a conservative approach.

Specifically, when calculating the backward slice of a variable v, our backward slicer traverses up to N consecutive function calls (we set N to five in our experiments), and it over-approximates data dependencies when a function call is not followed. For instance, if a function call takes v as one of its arguments, and the function call is not followed, we assume that v is data-dependent on all the other arguments. Although this approach might lead our static analysis phase to produce false positives, it does not affect the performance of our tool, since, as explained in Section III-A, we use dynamic analysis to validate the results produced by static analysis.

To build the data-transforming function dominator trees, as explained in Section III-A, we first need to build the companion app call graph. To achieve this, we perform intra-procedural type inference [65] to determine the possible dynamic types of the object on which a method is called. When this fails, we over-approximate the possible targets as all the subclasses of its static type.

Entropy Calculation Details. To find data-transforming functions, DIANE needs to calculate the entropy of each variable v within the live sets Li_f and Lo_f of a function scope f. To achieve this, if v is a primitive variable (e.g., `int`), or a known type (i.e., `String`, `Integer`, `Float`, and `Double`), we convert the data it contains in its byte representation and calculate the Shannon entropy of this sequence of bytes. Note that the entropy is computed on the entire sequence of bytes, rather than the single bytes considered separately.

Conversely, if v is a class object, we use pysoot to retrieve its class definition, and we consider each field variable v_c whose type is either primitive or known. For all these field variables, we compute their entropy as specified above, and we add them to the Li_f set or to the Lo_f set, based on to which live set v belongs.