Scrutinizing Implementations of Smart Home Integrations

Kulani Mahadewa, Kailong Wang, Guangdong Bai, Ling Shi, Yan Liu, Jin Song Dong and Zhenkai Liang

Abstract—A key feature of the booming smart home is the integration of a wide assortment of technologies, including various standards, proprietary communication protocols and heterogeneous platforms. Due to customization, unsatisfied assumptions and incompatibility in the integration, critical security vulnerabilities are likely to be introduced by the integration. Hence, this work addresses the security problems in smart home systems from an *integration* perspective, as a complement to numerous studies that focus on the analysis of individual techniques. We propose HOMESCAN, an approach that examines the security of the implementations of smart home systems. It extracts the abstract specification of application-layer protocols and internal behaviors of entities, so that it is able to conduct an end-to-end security analysis against various attack models. Applying HOMESCAN on three extensively-used smart home systems, we have found twelve non-trivial security issues, which may lead to unauthorized remote control and credential leakage.

1 INTRODUCTION

Enabled by various intelligent Internet of Things (IoT) techniques, the smart home paradigm has been significantly changing the lifestyle of its users. New convenient facilities, such as smart TVs, smart lighting and security alarm systems, are becoming ubiquitous. Along with its booming growth, security incidents have been continually observed [2, 3]. Researchers have made efforts to address security issues in smart home systems [4–10], with focus on several aspects ranging from radio communications, networking, operating systems, middleware, and protocols, to backend cloud.

In this work, we investigate security of smart home systems from an *integration* perspective. Our motivation is out of such a key observation—to realize a "smart" automated home, it is essential that multiple subsystems are integrated. The controls are typically initiated from the handheld devices such as smart phones, transmitted over wireless channels such as Bluetooth, ZigBee and Wi-Fi, forwarded by intermediate relays such as gateways, and web-based service portals, and finally executed by the end devices such as bulbs and locks. Due to the involvement of such a wide assortment of technologies and devices (usually from diverse manufacturers), to coordinate them in a secure way is challenging. The challenge may be attributed to the following two factors.

- **Incompatibility.** Since diverse standards are enforced, there may be incompatibilities among the subsystems. For example, in the Philips Hue system that we have analyzed, the authentication between the bulb and the hub is through the Touchlink Commissioning (TLC) over ZigBee, while that between the hub and the control app is through a customized authentication over Wi-Fi. Once these three are integrated, due to the incompatibility between the two mechanisms, there is no way for the bulb to authenticate the control app. This allows a malicious app which has infected the mobile phone that the control app is installed on to acquire control over the bulb.
- **Invalidated Assumptions.** A developer or manufacturer may make assumptions (e.g., trust relation, message format and correct sequence of API calls) when using the interfaces provided by other parties. If any assumption is invalid, the way to use the interfaces may be insecure. For example, in the same system above, the manufacturer of the hub actually assumes the LAN is secure, whereas this assumption may not be true if a malicious app has been installed on the user's mobile phone.

We present an approach named HOMESCAN, which scrutinizes security of the implementations of smart home systems. It extracts the application-layer protocols and security-relevant internal behaviors of each subsystem (or protocol) from the implementations. Through this, it can derive a unified abstraction of the end-to-end system to flatten the difference of the protocols employed by each entity. The challenges yet stem from the partial availability of the implementations. First, the source code is seldom visible, although the executable of the control app (from the app market), the firmware extracted from devices, and SDKs provided by vendors, are available

- *K. Mahadewa, K. Wang, L. Shi, J.S. Dong and Z. Liang are with National University of Singapore, Singapore. J.S. Dong is also with Griffith University, Australia.*
- *G. Bai, the corresponding author, is with the University of Queensland, Australia. E-mail: g.bai@uq.edu.au.*
- *Y. Liu is with Ant Financial.*

This article extends the preliminary results presented in [1]. It includes a more detailed description on the protocol extraction algorithms, a detailed description and additional data on the experiment and evaluation.

for analysis. Second, the cryptographic protocols are used among the entities, so that the communication is blurred to us, even though we are able to capture the exchanged traffic. To alleviate these challenges, HOMESCAN uses a hybrid analysis including *dynamic testing*, *whitebox analysis* and *trace analysis*. The dynamic testing executes test cases, and captures communication traffic and execution traces; the whitebox analysis identifies semantics by analyzing the program that is available; the trace analysis infers the association relation between a value of unknown semantics and an entity, a session or a value whose semantics has been identified.

HOMESCAN uses labeled transition systems (LTSs) [11], which have been extensively used to model and reason various systems, to represent the extracted specification. An LTS describes the execution of a particular entity, including its internal behaviors (e.g., generating a nonce and validating a digital signature) and communication behaviors (e.g., sending and receiving a message). At this abstract level, the security reasoning can ignore the heterogeneity of underlying protocols, but focus on the logic that is implemented by the system. Using this abstraction, reasoning security properties of the whole integration becomes effective, and we show that most of the properties specific to the smart home can be analyzed via *reachability* checking.

It is obvious that obtaining the complete or sound specification is almost infeasible. HOMESCAN focuses on extracting as precise specification as possible, whereby it can identify security issues. We prototype HOMESCAN and apply it to three extensively-used smart home systems, including Philips Hue, LIFX, and Chromecast. It manages to identify twelve security vulnerabilities.

This work makes the following main contributions.

- **Specification Extraction Techniques.** We propose hybrid techniques to extract specifications from the implementations of the smart home systems. Our evaluation of real-world systems demonstrates that the extracted specification is precise enough to identify significant security issues.
- **Vulnerability Identification Techniques.** We have modeled a set of practical attacks to facilitate the vulnerability identification based on LTS representations. We reduce the vulnerability identification to traditional reachability analysis on LTS.
- **Practical Results.** We apply HOMESCAN to real-world systems and successfully identify twelve non-trivial security vulnerabilities from them. The supporting materials are published online for future research [12].

2 PRELIMINARIES

In this section, we present our running example, and define a generic specification model of smart home systems from the *integration* perspective. We also provide an overview on the security properties and attack models in the vulnerability identification of smart home systems.

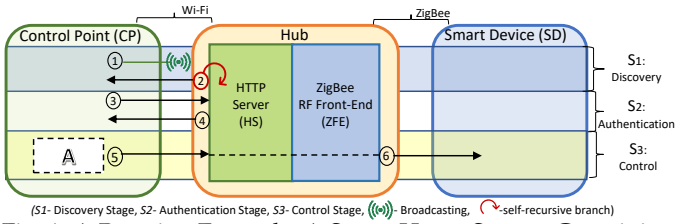

Fig. 1: A Running Example: A Smart Home System Containing a CP, Hub and a SD (Note that the discovery and authentication between Hub and SD are omitted for simplification.)

```
1  public class A {
2      public static void main(String[] args) {
3          String ec = a( "light_ON", this.s);
4      }
5      public static String a(String a, String b) {
6          try {
7              byte[] k = b.getBytes("UTF-8");
8              byte[] bk = Arrays.copyOf(k, 16);
9              SecretKeySpec kspec = new SecretKeySpec(bk, "AES");
10             Cipher c = Cipher.getInstance("AES/ECB/PKCS5Padding");
11             c.init(Cipher.ENCRYPT_MODE, kspec);
12             byte[] m = c.doFinal(a.getBytes("UTF-8"));
13             return Base64.getEncoder().encodeToString(m);
14         } catch (Exception e) {}
15         return null;
16     }
17 }
```

(a)

Wireshark Trace

	"No.", "Time", "Source", "Destination", "Protocol", "Length", "Info"	
1	"61","23.418749","192.168.1.236","239.255.255.250","SSDP","132","M-SEARCH * HTTP/1.1 "	S1
2	"62","23.959659","192.168.1.182","192.168.1.236","SSDP","339","HTTP/1.1 200 OK " >> HTTP/1.1 200 OK\r\n LOCATION: http://192.168.1.182:80/description.xml\r\n SERVER: Linux/3.14.0 UPnP/1.0 IpBridge/1.32.0\r\n hue-bridgeid: 001788FFFE2D5D98\r\n "63","24.008536","192.168.1.182","192.168.1.236","SSDP"," 339 ","HTTP/1.1 200 OK " "67","25.062956","192.168.1.182","192.168.1.236","SSDP"," 339 ","HTTP/1.1 200 OK "	S1 (No.62-67 repeat)
3	"78","31.451885","192.168.1.236","192.168.1.182","HTTP","154","POST /api HTTP/1.1 " >> deviceID= 001788FFFE2D5D98, password=pass123	S2
4	"85","31.912202","192.168.1.182","192.168.1.236","HTTP","60","HTTP/1.1 200 OK >> {auth_token="7B8249C219669C7946D5FBD8C5B178B6FE3299CC" secret_key="hue-secret-key-meethue345"}	S2
5	"105","34.415012"," 192.168.1.236 "," 192.168.1.182 ","HTTP","58","HTTP/1.1 200 OK >> {auth_token="7B8249C219669C7946D5FBD8C5B178B6FE3299CC" command="x95b9ZMtRmDWZ8uRizm4iKsq-/oZkfzDPZXQWZgZ7Fzw="}	S3

ZigBee Trace

	MsgNo, StartTime, Channel, Layer, DataSize, SourceAdd, DestAddress	
	1353, 21/1/19 0:50:23.307, 25, NWL, 39, 001788FFFE2D5D98, 00178801102AA2FB	S3 (heartbeat)
	1354, 21/1/19 0:50:23.337, 25, NWL, 44, 001788FFFE2D5D98, 00178801102AA2FB	
	1355, 21/1/19 0:50:23.427, 25, NWL, 42, 00178801102AA2FB, 001788FFFE2D5D98	
6	1356, 21/1/19 0:50:23.858, 25, NWL, 43, 001788FFFE2D5D98, 00178801102AA2FB	S3

(b)

Fig. 2: (a) Part of CP Source Code (Code Snippet "A" in Fig.1); (b) Part of CP and HS Communication Trace Captured using Wireshark, and Part of ZFE and SD Communication Trace Captured using Perytons. (Values highlighted in blue are extracted from the traces. The three lines covered by the blue bracket are the heartbeat packets over ZigBee channel. The transactions repeated are shown with the red bracket. They are identified as a sequence-recursion and a self-recursion respectively.)

2.1 A Generic Model of Smart Home and the Running Example

In order to facilitate the model extraction, we resort to a manual study to abstract a generic system architecture from several smart home systems popular on the market, such as SmartThings [13] and HomeGenie [14]. In our abstraction, a smart home system consists of three subsystems, i.e., a control point (denoted by *CP*) which interacts with the end users and issues controls, several smart devices (denoted by *SD*) which are operable electronic devices, and several relays (denoted by *hub*) which bridge the communications. Covering from configuration to control, the end-to-end working procedure of smart home systems is divided into three stages, i.e., *discovery*, *authentication* and *control*, which are introduced shortly.

In the remaining of this paper, we use a running

TABLE 1: Intermediate Outcomes and Corresponding HOMESCAN Approach for the Running Example

Column 2: the *id* represents the identity of a transaction. Each *id* corresponds to the circled index in Fig. 1.
Column 3: represents broadcast. Column 4: The *Value*s are extracted from the traces shown in Fig. 2-b.
Column 5: The *msg* includes the inferred message components; if more than one communication paths available they are specified by the *id* in *branch* set; if there is communication between sub components of single device (e.g. *hub* has *HS* and *ZFE*), then specify the communication partner by *local_communication*; if there are local actions done by a entity, they are specified in *local_action* set. Further, the extracted values which has the same identity are inferred as the same message component.
Column 6: The techniques used to infer each message component in column 5.

	id	Sender, Receiver, Channel	Extracted Values (Value, Primary Type, Value ID)	Inferred Specification	Approach Used
S1	1	CP, *, wifi	(M-SEARCH * HTTP/1.1, String, v1)	msg=(UpnpMsearchRequest)	v1-Protocol_Knowledge
	2	HS, CP, wifi	(192.168.1.182, String, v2), (001788...2D5D98, String, v3)	msg=(HubIP, HubID), branch={2}	v2,v3-Protocol_Knowledge
S2	3	CP, HS, wifi	(001788FFFE2D5D98, String, v3), (pass123, String, v4)	msg=(HubID, Password)	v4-Initial_Knowledge
	4	HS, CP, wifi	(7B8249C219669C7946D5FBD8C5B178B6FE3299CC, String, v5), (hue-secret-key-meethue345, String, v6)	msg=(hash(Password,HubID),SecretKey)	v5-Exhaustive_Search, v6-Whitbox_Analysis
S3	5	CP, HS, wifi	(7B8249C219669C7946D5FBD8C5B178B6FE3299CC, String, v5),(x95b9ZMtRmDWZ8uRizm4iKsq/oZkfzDP-ZXQWZgZ7Fzw=, String, v7)	msg=(hash(Password,HubID), senc(SecretCommand,SecretKey)), local_communication={ZFE}	v7-Whitbox_Analysis
	6	ZFE, SD, zigbee	(*Encrypted Zigbee data* (43 bytes), String, v8)	msg=(assoc(SecretCommand)), local_action ={(SD, executeCommand,{msg})}, branch={5}	v8-Differential_Analysis

example demonstrated in Fig. 1 and Fig. 2 to explain our work.This example is designed to include the typical features of on-stock smart home systems. The CP in it is an Android app which supports HTTP protocol over Wi-Fi. To be representative, the SD only supports a near field communication protocol, the ZigBee. Therefore, the *hub* has to include an HTTP server (denoted by *HS*) and a ZigBee front end (denoted by *ZFE*) to bridge the communication between the HTTP-based CP and ZigBee-based SD. In a nutshell, the system works as follows.

- **Discovery Stage** (*S*1 in Fig. 1 and in Fig. 2-b). The CP searches for the *hub* and pairs with the HS (steps ① & ②).
- **Authentication Stage** (*S*2). The CP authenticates itself with the *HS* at the *hub* (steps ③ & ④)
- **Control Stage** (*S*3). The CP controls the SD which has been connected to the *hub* by sending control commands to the HS (step ⑤). Once receiving a command, the *hub* converts it to a ZigBee packet and sends it to the SD (step ⑥).

By analyzing the communication traces in these stages and the available code (Fig. 2), HOMESCAN aims to extract the specification listed in Table 1.

2.2 Security Properties and Attack Models

Security Properties. Our approach analyzes the security properties including data security (i.e., data confidentiality and integrity) and access security (i.e., authentication and authorization), given that various works have shown the importance of these security properties to IoT [15–17].

- **Data Security.** The property ensures that the data transmitted in a smart home system should be delivered to the intended entities without being revealed or altered by the attacker. More specifically, we consider the confidentiality of the security analysts annotated credentials such as passwords and access token, and the integrity of control commands from the CP to the SD via *hub*.
- **Access Security.** The property ensures that all entities in a smart home system can verify the identities of their communicating entities, and only the authenticated and authorized entities are granted access to services and information. In particular, this security property guarantees that the SD is only under control of the intended CP and *hub*, i.e., the SD only executes commands from the intended CP and *hub*.

TABLE 2: Attack Models and Capabilities

Attack Model	Attack Capability Description
Malicious Entities	**Malicious CPs** aim to send unauthorized commands to manipulate victim SDs over the same local network or Internet, compromising access security of the SDs.
	Malicious *hubs* aim to send unauthorized commands to manipulate the victim SDs in the vicinity, compromising access security of the SDs.
	Malicious SDs aim to capture the sensitive information (e.g., identity, address and credentials of the *hub*), which could compromise the victim SDs in the vicinity. This attack model violates the access security of the SDs.
Network Attacker	**Eavesdropping.** The attacker aims to obtain crucial information (e.g., session keys and the identity of the *hub*) by eavesdropping, compromising data confidentiality.
	Intercepting and Modifying Commands. The attacker aims to manipulate the system behavior by replaying/-modifying control commands (such as ON/OFF of SDs, casting a video and changing light color) and administrative commands (such as device authentication/removal/reset, possibly causing functionality disruption like Denial of Service). This attack model violates the data integrity of the command messages sent from the user and the access security of the SDs.

Attack Models. The common threats to a smart home system are unauthorized access and manipulation by malicious entities [18, 19], and vulnerable settings of wireless communications [20]. Hence, we consider two types of attackers in this work, i.e., malicious entities and network attackers, whose capabilities are described in Table 2 in a nutshell.

- A malicious entity refers to any device/subsystem that is under attacker's control. We conservatively assume that the attacker is able to control extra devices and establish extra connections with the protocol entities (e.g., in multicast scenarios). The security of a system is trivial if all entities are under attacker's control. Therefore, we remark that ***when analyzing the extracted protocol, only one single entity is considered compromised each time.***
- A network attacker is able to eavesdrop, intercept and modify messages within the local network (e.g., Wi-Fi and ZigBee) in which the attacker

resides or over the Internet. We assume the system entities, including the *hub*, CP and SD, are honest while analyzing system security properties against the network attacker.

3 HomeScan Overview and Prerequisites

3.1 HomeScan Overview

HomeScan uses a set of techniques for specification extraction and vulnerability identification. It takes the following inputs.

- **Implementation of the System under Analysis.** A runnable setup of the smart home system and a set of programs (*PS*), including available source code, libraries, and binaries of entities are input to HomeScan.
- **Test Cases.** A set of test cases (*TC*) is required to trigger the functionality of the system under analysis. There must be at least one test case (which we call initial test case) which can drive the system to walk through all its three stages (i.e., discovery, authentication and control). which allows HomeScan to generate a base for mutation. Each test case corresponds to a configuration of the system. Configurations refer to the entities (e.g., CP, SD and *hub*) of the system and the different users (e.g., admin, general user and guest).
- **Initial Knowledge.** Initial knowledge (*IK*) is represented as a pair (P, CH), where P is the set of entities of the input system , and CH is the set of channels used for communication among entities.

As shown in Fig. 3, HomeScan includes three major components including *trace capturing & pre-processing*, *specification extraction* and *flaw identification*.

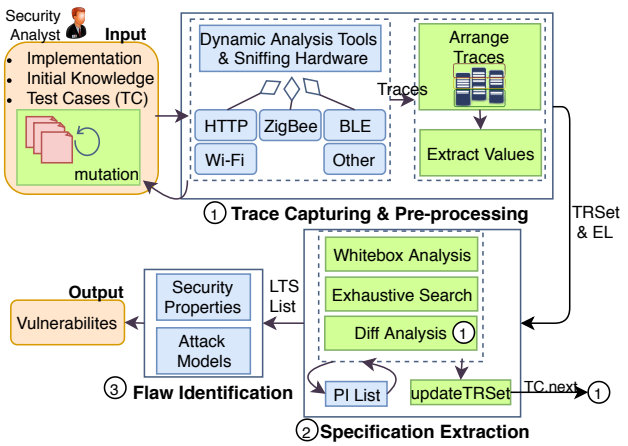

Fig. 3: Overview of HomeScan

Trace Capturing. The first step of HomeScan is to capture the trace of the system under analysis by executing the initial test case. It captures two types of traces, i.e., traffic traces and execution logs. HomeScan uses existing sniffers to capture the traffic traces, and records the execution of the entities whenever instrumentation can be done (The execution logs generated from executing the initial test case is referred as $\widehat{EL}$.). In addition, HomeScan generates new traces by mutating the values (e.g., HTTP header values or HTTP parameters) from the captured traces, after executing each test case.

Pre-Processing. Pre-processing takes the set of captured traces as input and aims to generate a set of transactions (defined soon). A captured trace is a sequence of messages, containing the exchanged data between two or more entities. HomeScan first merges the traces in chronological order and then extracts the values from the traces. For traces whose underlying protocols can be recognized, it extracts data referring to their standard message formats. For other traces, the extraction is done using keyword (e.g.,"host" in an HTTP request) searching, pattern matching (e.g., IP addresses) and string splitting with delimiters (e.g.,"&").

Specification Extraction. The objective of this step is to generate local LTS representation of the system, given the transactions generated from the pre-processing component. We propose a hybrid extraction technique including *whitebox analysis* and *trace analysis* for the specification extraction. The extracted specification is represented by LTS. In Section 4, we detail the specification extraction component.

Flaw Identification. In this step, we propose a verification algorithm to check IoT-specific security properties of the LTS representation against predefined attack models. Essentially, the verification algorithm is a reachability analysis. It can apply any of classic searching algorithms (e.g., DFS and BFS) on the generated LTS to search the reachability of a bad state wherein the security property is violated. In Section 5, we detail our verification algorithm.

3.2 Prerequisites

In order to bridge the semantic gap between the low-level traces and the high-level LTS, we introduce several intermediate data structures to maintain the information required to generate an LTS. In this section, we present their definitions.

Transactions. A protocol consists of several (typically sequential) *round*s of information exchange. We represent the abstraction of a single round as a *transaction* (*TR*). We define it as a 5-tuple $(id, se, R, EVSet, BR)$, where *id* is the transaction ID, $se \in P$ is the sender, $R \subset P \setminus se$ is the set of receivers (In multicast communication, there can be multiple receivers), and $EVSet = \{EV_1, EV_2, ..., EV_{\mathcal{V}_{id}}\}$ is the set of values (total number $\mathcal{V}_{id}$) extracted from the message exchanged in the *TR*. Each EV_i is a 3-tuple (v, t, id) where v is the value, t is its type, and *id* is the value ID. The transaction also includes branch information (*BR*), which is defined soon.

To represent the output of the pre-processing component, we propose a transaction set denoted by $TRSet = \{TR_1, TR_2, ..., TR_{\mathcal{T}}\}$ where $\mathcal{T}$ is the total number of transactions (rounds).

Branch Information. Each transaction *TR* includes a branch set (denoted by *BR*), which is a set of transaction IDs that represent the transactions branching from the

current transaction. There are three types of branches, i.e., options, self-recursions and sequence-recursions. An option branch is either labeled as an option in the test case, resulted from test case mutation or configuration changes. HOMESCAN identifies self-recursions or sequence-recursions when data of a single transaction or data of a sequence of transactions are repeated in the trace respectively. Self-recursion is a repetition of the same action (defined soon) which is represented as a self-loop, and sequence-recursion is a repetition of a sequence of actions.

Types. For each extracted value $EV \in \bigcup EVSet_i$ $(1 \leq i \leq \mathcal{T})$, HOMESCAN attempts to identify a type (t) during the specification extraction. HOMESCAN defines two categories of types, i.e., *primitive* and *domain-specific*. The primitive type can be an integer, boolean, or string. The domain-specific type can be any of network address (used in ZigBee-like protocols), IP address, MAC address, username, password, encryption key, hash value, ciphertext, etc. During pre-processing, HOMESCAN assigns a primitive type to each value and updates it to a domain-specific type (which is more precise) when more information is inferred.

The domain-specific types are formalized as *term*s (denoted by T). Terms are categorized into three subsets, i.e., *constants* (denoted by C), *functions* (denoted by F), and *variables* (denoted by V), such that $T = C \cup F \cup V$. Ground terms are terms that only contain constants and functions. Variables are terms that are not ground. Table 3 lists the function terms used by HOMESCAN.

TABLE 3: Function Terms

Function Term (F)	Definitions	Meaning
senc$(message, k)$	message $message \in T$; symmetric key $k \in T$	ciphertext created by symmetric encryption
sdec$(encmsg, k)$	ciphertext $encmsg \in T$; symmetric key $k \in T$	extracted message by symmetric decryption
aenc$(message, pk)$	message $message \in T$; public key $pk \in T$	ciphertext created by asymmetric encryption
adec$(encmsg, sk)$	ciphertext $encmsg \in T$; private key $sk \in T$	extracted message by asymmetric decryption
hash$(message)$	message $message \in T$	hash value generated by hash function
sign$(message, sk)$	message $message \in T$; private key $sk \in T$	signature generated by by signature function
checksign$(sign, pk)$	signature $sign \in T$; public key $pk \in T$	result of signature verification
assoc(t)	existing term $t \in T$	new term generated by association
$(a, \cdots, b)$	$a, \cdots, b \in T$	concatination of terms
$\{m, \cdots, n\}$	$m, \cdots, n \in T$	set construction of terms

Actions. A label of an LTS is an *action* which can be either a *communication action* or a *local action*. The actions which exchange (send and receive) messages with other entities are communication actions, and the actions that execute local behaviors of each entity are local actions. Table 4 lists the action labels used by HOMESCAN.

Protocol Information. We use *Protocol Information* (denoted by *PI*) to indicate the information obtained during the specification extraction. A *PI* is a 5-tuple $(msg, \widehat{ACSeq}, ch, lc, BR)$, where msg is a concatenation of terms representing the messages transmitted by the corresponding TR, and $\widehat{ACSeq} = \langle AC_1, AC_2, ..., AC_{\mathcal{A}} \rangle$ is a sequence of action information where $\mathcal{A}$ is the total number of actions. An action information AC_i is a 3-tuple (u, a, X) where $u(\in P)$ is the entity which performed the action, a is the name of action and X is a set of terms taken as parameters to a. *PI.ch* is the communication channel. Further, if the message *PI.msg* needs to be transmitted between two sub-components within a device, which acts on different protocols, the algorithm introduces local communication actions (e.g., between HS and ZFE of *hub* shown by the broken lines in Fig.1). $PI.lc \in P$ is the receiver ($lc \notin TR.R$) when local communication between two sub-components exists. *PI.BR* is the branch information.

TABLE 4: Communication and Local Actions

Type	Action	Definitions	Meaning
Comm.	$send(ch, message)$	$ch \in C$; $message \in T$	sending a message *message* via channel *ch*
	$receive(ch, x)$	$ch \in C$; $x \in V$	receiving a message via channel *ch* and storing in x
Local	$newnonce(x)$	variable $x \in V$	generating a new nonce and storing it in x
	$newskey(x)$	variable $x \in V$	generating a new symmetry key and storing it in x
	$newkeypair(pk, sk)$	variables $pk, sk \in V$	generating and storing a pair of public-private keys
	$executeCommand(c)$	constant $c \in C$	executing the command c

Parameterized Labeled Transition System. A traditional labeled transition system (LTS) is a 4-tuple $\mathcal{L} = (S, s_0, A, \rightarrow)$ where S is a set of states (locations); $s_0 \in S$ is the initial state; A is a set of actions; $\rightarrow \subseteq S \times A \times S$ is a labeled transition relation. We extend the LTS with parameters to differentiate the instances of the same behavior pattern to facilitate the attacker modeling. For example, we use the parameter *HubID'* to represent the identity of the malicious hub compared with the *HubID* for the benign hub.

4 SPECIFICATION EXTRACTION

The goal of specification extraction is to generate a representation of system integration. One challenge that can be foreseen is the gap between the execution traces (to be precise, the transactions after pre-processing) and the target LTS. To bridge the gap, we design a two-step extraction approach, which first extracts *PI*s from the transactions, and then transforms the PIs into LTS representations.

4.1 Inference of Protocol Information

Given the transactions generated from trace processing, HOMESCAN uses several analysis techniques to infer the PIs. This is outlined in Algorithm 1. It takes a 5-tuple $(TRSet, PS, \widehat{EL}, IK, TC)$ as input, where *TRSet* is the set of transactions; *PS* is the set of programs; $\widehat{EL}$ is the sequence of execution logs; *IK* is the set of initial knowledge; *TC* is the set of test cases. The output of the algorithm is a list of inferred *PI* (*PIL*), each of which correlates with one transaction. The algorithm executes the next test case (*TC.next* at line 13) and iteratively identifies new semantics until no new information can be found from the input. In each iteration, the $TRSet_{new}$ includes new values and new branch information (*BR*) corresponding to the new

```
input : (TRSet, PS, ÊL, IK, TC)
output: A List PIL = [PI^1, PI^2, ..., PI^δ] where δ = T; Each
        transaction in TRSet is mapped to a PI.
1  F = {f_1, f_2, ..., f_η} where η is the number of selected hash,
     cryptography and encoding/decoding functions.;
2  g : GEVSet × ℙ(TRSet) is relation indicating the transactions
     which a value appears.;
3  TRSet_new ← TRSet, TRSet_old ← TRSet;
4  do
5      TRSet ← TRSet_new;
6      GEVSet = ⋃ EVSet_i (1 ≤ i ≤ T) // global set of EVSet.;
7      g ← Grouping(TRSet) ;
8      GEVSet ← WB(GEVSet, PS, ÊL, IK);
9      GEVSet ← ES(GEVSet, F, IK);
10     GEVSet ← DA(GEVSet, PIL, TRSet, TRSet_old, IK);
11     PIL ← updatePIL(GEVSet, g);
12     TRSet_old ← TRSet;
13     TRSet_new ← updateTRSet(TC.next) ;
14 while TRSet_new ≠ TRSet;
15 return PIL;
```

Algorithm 1: PI Inference Algorithm

configuration specified in the *TC.next*. The remaining of this section details the Algorithm 1 by elaborating with a few examples on the techniques HOMESCAN uses to infer the types of new values.

4.1.1 Whitebox Analysis

HOMESCAN uses $WB(GEVSet, PS, \widehat{EL}, IK)$ (line 8 in Algorithm 1) to infer the type of values that are produced or consumed by the given program. This is conducted in Algorithm 2. It begins by initializing the global variables *cgraph* (call graph), *la* (local actions), and *br* (branch information) (line 1). For each *program* in the input program set (PS), it performs a code analysis (lines 2-8). This analysis identifies the code (*clsCode*) which produces or consumes the extracted values, parses it into an Abstract Syntax Tree (AST) and resolves the symbols in it using a symbol solver (lines 4-5). The parsed AST with symbols resolved (*parsedSmblAST*) is then input to the *AnalyzeClass* function (line 7) which recursively analyzes all related classes to identify the dependencies among variables. During the class analysis, each method which belongs to the class is analyzed by the *AnalyzeMethods* function, to find the domain-specific types, local actions and branch information (lines 26-44). If a value *Ev.v* is equivalent to a variable with a known domain type (dT), HOMESCAN assigns the *dT* to the *Ev.t* (lines 16-24). The call graph $(cgraph)$ of the *program* is used to retrieve the control information during the analysis (line 6).

Below we brief some key techniques used in this algorithm. To ease the understanding, we use the AST shown in Fig. 4 as an example. It includes the AST of the method `a` in Fig. 2-a (i.e., the code snippet **A** in Fig. 1).

Code Snippet Identification (lines 3-4). Since generating the AST and solving the symbols of the whole *program* (e.g., a java jar file) is expensive, HOMESCAN first identifies part of the *program* (e.g., a class) that is likely to produce or consume the extracted values. To this end, *findCodeSnippet* conducts a string matching to search the call sites of security APIs such as `"javax.crypto.Cipher"` in the *program*. This results in a list of classes which have called those APIs. It

```
input : GEVSet, PS, ÊL, IK
output: GEVSet
1  cgraph ← null, la ← null, br ← List();
2  for program ∈ PS do
3      initC ← findCodeSnippet(program);
4      clsCode ← ReverseEngineerClass(initC, program);
5      parsedSmblAST ← JavaSymbolSolver(clsCode);
6      cgraph ← GenerateCallGraph(program);
7      GEVSet ← AnalyzeClass(parsedSmblAST, initC, null, IK);
8  end
9  Function AnalyzeClass(ast, class, dTmap, IK):
10     (dTmap, m) ← UpdateDT(AnalyzeMethods(ast, dTmap, IK));
11     dTmap ← PropDomainTwithinClass(class, dTmap);
12     (nxtM, nxtC) ← GetCallerOCallee(cgraph, m, class);
13     while nxtM ≠ null do
14         AnalyzeNextClass(nxtM, nxtC, dTmap);
15     end
16     for (vnode, dT ∈ T, mname) ∈ dTmap do
17         for EV ∈ GEVSet do
18             if IsEVMapVarNode(vnode, mname, ÊL, Ev.v) then
19                 EV.t ← dT;
20                 GEVSet ← UpdateGEVSet(EV)
21             end
22         end
23     end
24     return GEVSet;
25 end
26 Function AnalyzeMethods(ast, dTmap, IK):
27     fieldsPT ← GenerateFieldPrimaryTypeMap(ast.fields);
28     for methodNode ∈ ast do
29         varPT ← null, dT ∈ T ← null, sig ← null;
30         for n ∈ methodNode.childNodes do
31             if n ∈ dTmap then dT ← GetDT(n, dTmap) ;
32             if (n.expr ∈ VariableDeclaration) then
33             varPT ← UpdateVarPT(n, n.pT) ;
34             else if (n.expr ∈ MethodCall) then
35                 sig ← GenSignature(n.expr, fieldsPT, varPT);
36                 if sig ∈ securityAPICallList then
37                 (dT, lcA) ← GenTerm(sig, n.expr, IK);
38                 la ← UpdateLA(n, lcA);
39                 dTmap ← UpdateDT(n, dT, methodNode.name) ;
40             else if (n ∈ BlockStatement) then
41                 br ← UpdateBranchInfo(BranchAnalysis(n));
42         end
43         return dTmap;
44     end
45 end
```

Algorithm 2: Whitebox Analysis Automation Algorithm

then reverse engineers them using off-the-shelf tools (line 4), and uses a symbol solver to parse the decompiled source code *clsCode* into an AST with resolved symbols (*parsedSmblAST*) (line 5).

AST with Symbol Solving (line 5). In our parsed AST, each node has at least one child except the *leaf* nodes, and all the nodes except the *root* has exactly one *parent*. A node can be an expression, a statement, a name (i.e. fields, variables, parameters or types), a parameter, or a return type. The *root* is a java class file and its *child* nodes are import statements and the class declarations. At the next level, each class declaration node has *fields* and *methods* as its children. Similarly, each node is divided into *child* nodes until the *leaf* node is a name expression.

However, the AST is an abstract representation which does not have enough information to identify the types of the variables used in the program. Therefore, we use a symbol solver to calculate additional information such

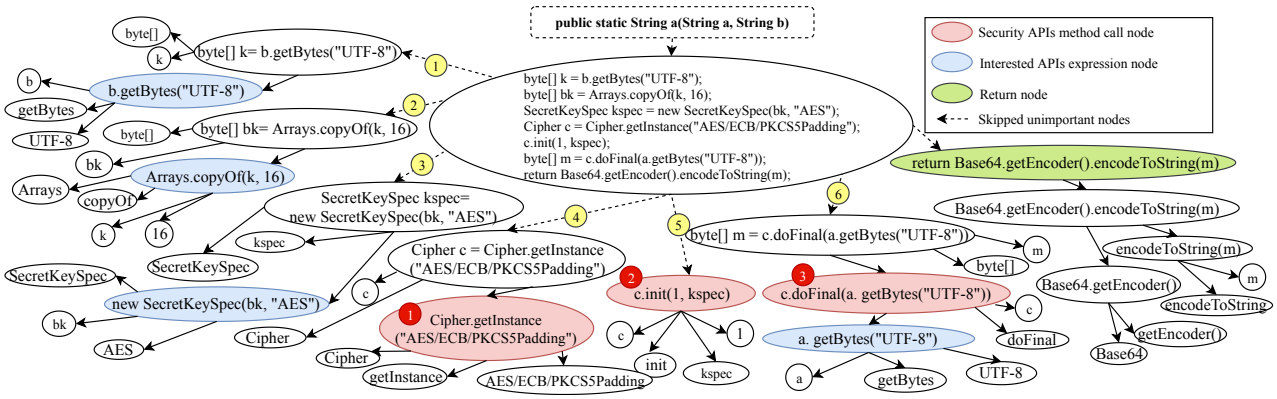

Fig. 4: AST for the method `a` in Fig. 2-a

as resolving references and finding dependencies among nodes. For example, this helps to find out whether an expression is a mathematical operation or a method-call, and then further to identify the semantics of its children such as method-name, arguments and so on. The type information included in a declaration statement can be propagated to child nodes or dependent nodes to find out the type of variables. With this, we are able to obtain the primary type (*pT*) of a variable representing an *EV*.*v*, as shown in Fig. 4. After the symbols in the AST are resolved, HOMESCAN further analyzes the nodes based on its expression, i.e., a variable declaration, an assignment, a method-call or an object creation.

Domain Type Annotation and Local Action Identification (lines 26-45). After deriving the AST of the identified *program*, the next step is to infer the domain type of the program variables (which could potentially mapped to an *EV* $\in$ *GEVSet*). The basic idea is to annotate parameters and return values with types obtained from the knowledge of the cryptographic APIs. To this end, HOMESCAN maintains a set of rules for each specified security API[1]. As an instance, Fig. 5 shows such rules for the symmetric encryption.

These rules are derived based on the knowledge of how the APIs are used to implement a symmetric encryption. In brief, first, the `getInstance` method of `java.crypto.Crypto` is called with the transformation (i.e.,symmetric or asymmetric) specified as the first argument (line 10 in Fig. 2-a). Next, the `init` method of `java.crypto.Crypto` instance is called, specifying the operation mode (i.e., encryption or decryption) as the first argument and the key as the second argument (line 11 in Fig. 2-a). Finally, the `doFinal` method of `java.crypto.Crypto` instance is called with the data (i.e., plaintext or ciphertext) as the first argument (line 12 in Fig. 2-a).

With these rules, HOMESCAN first traverses through the AST for the nodes which represent method-call expressions that invoke security APIs, for example, `getInstance(java.lang.String)`, `init(int,javax.crypto.spec.SecretKeySpec)`, and `doFinal(byte[])` in Fig. 4. Whenever a node is found, the rules are applied to annotate its arguments, reference or return value with a domain type.

In particular, this is done by the *AnalyzeMethods* function in Algorithm 2. It takes a 3-tuple (*ast*, *dTmap*, *IK*) as input, where *ast* is the parsed AST with resolved symbols; *dTmap* is a map of (*node*, *domain_type*, *related_method*); *IK* is the initial knowledge and returns an updated *dTmap*. The function analyzes each method node (*methodNode*) in the input *ast* (lines 28-44). During the analysis, HOMESCAN gets the child nodes of *methodNode*, and further analyzes each child node *n* based on its expression (lines 30-42). If *n* has already been analyzed, its domain type is retrieved from the *dTmap* (line 31). Otherwise, if the expression of *n* is a `MethodCall`, the method signature (*sig* in line 35) of the method-call is generated using *GenSignature*. In order to do that, HOMESCAN requires the primary types of the *n*'s arguments. This information can be obtained from the `class-field` or `method-variable` declarations. Therefore, Algorithm 2 records the primary types of the class-field nodes (*fieldsPT* at line 27) and method-variable nodes (*varPT* at line 29). The algorithm then verifies whether the method signature (*sig*) is in a pre-specified list of security APIs *securityAPICallList* (line 36). If yes, the *dT* for *n* is generated by the *GenTerm* function (line 37), and is also assigned to its relevant siblings (i.e., method-reference), updating the *dTmap* (line 39).

We use our running example to illustrate this process. The first rule in Fig. 5 is applied on the node marked as **1** in red circle in Fig. 4. The domain type of `c` is obtained from its parent node, which is an assignment expression. As a result, *dT* of the `c` node is inferred as *s* representing symmetric transformation. The second and the third rules are applied on the node marked as **2** in red circle in Fig. 4. The second input argument `kspec` is inferred as *k* representing symmetric key. As the reference of this node is `c` and the first input argument is `1`, the *dT* of the reference node (`c`) is updated as `senc` representing symmetric encryption. The fourth rule is applied on the node marked as **3** in red circle in Fig. 4. Based on the rule, the first input argument node `a.getBytes("UTF-8")` is inferred as *message* representing a plaintext, since the *dT* of the reference is symmetric encryption (senc).

While traversing the AST, HOMESCAN also records local actions (*lcA*) of the entities related to the generation of

1. Currently, HOMESCAN supports Java cryptographic library `javax.crypto`.

$$\frac{n.method = \text{“}Cipher.getInstance(java.lang.String)\text{”}, n.arg[0] = \text{“}AES/ECB/PKCS5Padding\text{”}}{n.ret.dT \leftarrow scipher} \quad [\ Symmetric\ getInstance\ ]$$

$$\frac{n.method = \text{“}Cipher.init(int, javax.crypto.spec.SecretKeySpec)\text{”}, n.ref.dT = \text{“}scipher\text{”}}{n.arg[1].dT \leftarrow k} \quad [\ Symmetric\ init\ arg1\ ]$$

$$\frac{n.method = \text{“}Cipher.init(int, javax.crypto.spec.SecretKeySpec)\text{”}, n.ref.dT = \text{“}scipher\text{”}, n.arg[0] = 1}{n.ref.dT \leftarrow senc} \quad [\ Symmetric\ init\ reference\ ]$$

$$\frac{n.method = \text{“}Cipher.doFinal(byte[])\text{”}, n.ref.dT = \text{“}senc\text{”}, n.arg[0].dT = \text{“”}}{n.arg[0].dT \leftarrow message} \quad [\ Symmetric\ doFinal\ arg0\ ]$$

Fig. 5: Rules for Symmetric Encryption APIs (*n* stands for the AST node being visited. These rules annotate the node `n` itself or its children/dependants (including its arguments (`n.arg[]`), reference (`n.ref`) and return value(`n.ret`)) with the domain types learnt from the knowledge of the security APIs.

a *term* (*dT*) (line 37). For example, in the running example, when HOMESCAN infers the node `kspec` as symmetric key *k*, it also records a local action *newskey(x)* (listed in Table. 4) for the entity *CP*, to represent the generation of *k*. Once the *dT* and *lcA* are determined, they are added to the *dTmap* and the *la* (lines 38-39). Consequently, the *AnalyzeMethods* function recursively analyzes all method nodes in the input *parsedSmblAST*, and returns the *dTmap* to the *AnalyzeClass* function (line 43).

Domain Type Propagation (line 7 and lines 9-25). After annotating the domain types at the nodes which invoke the security API, the next step is to propagate these types to other variables in the program. This is done by the *AnalyzeClass* function. It takes a 4-tuple $(ast, class, dTmap, IK)$ as input, where *class* is the current code snippet (that calls security APIs) in analysis, and outputs the updated *GEVSet* including the domain types inferred (line 24). The *AnalyzeClass* function first calls the *AnalyzeMethods*, and then combines the returned *dTmap* with type information derived previously. The new *dT*s (in the *dTmap* at line 10) are then propagated to the other nodes (in the *class*) which have dependency with the nodes with known *dT*s (line 11). To this end, the following four propagation rules are applied.

1) In the variable declaration and assignment expressions, if the source has a *dT*, then the *dT* of the target variable is propagated from the source, or vice versa.
2) In the variable declaration and assignment expressions, if the source is a method-call expression and the corresponding method is implemented in *initC*, then the *dT* of the method's return-statement (e.g., marked in green circle in Fig. 4) is propagated to the target.
3) In method-call or object creation expressions, if the expression is a call expression to security APIs or interested APIs (marked in blue in Fig. 4), the *dT* of the expression is propagated to the method-reference or to a method-argument, or vice versa.
4) When propagating the *dT* from one expression to another, if both expressions have a child node (i.e., a variable name or an argument) with the same name expression, then the *dT* for one child is propagated to the other.

We use our running example to illustrate this process. The *dT* of the leaf node `kspec` at branch 5 (marked in yellow circle) is $k \in T$ (from Table. 3) representing a symmetric key. Using the four rules, HOMESCAN infers that the *dT* of the leaf node `b` (an input to the method `a`) at branch 1 is also *k*. First, using `rule 4`, the type *k* is propagated to the `kspec` leaf node at branch 3. Second, using `rule 1`, *k* is propagated to the node `new SecretKeySpec(bk,"AES")` at the same branch. Third, using `rule 3`, *k* is propagated to the leaf node `bk`. Fourth, using `rule 4`, *k* is propagated to the leaf node `bk` at branch 2. Similarly, using the rules `1` and `3`, *k* is propagated to the leaf node `k` in branch 2. Fifth, again using `rule 4`, *k* is propagated to the leaf node `k` in branch 1. Finally, using the rules `1` and `3`, *k* is propagated to the leaf node `b` which is the second input to the method `a`. Hence, the *dT* of the input argument `String b` of method `a` is symmetric key *k*.

The *PropDomainTWithinClass* function iteratively performs the propagation until a fix point is reached where the *dTmap* has no new changes. Afterwards, HOMESCAN finds the next method (*nxtM*) and its class (*nxtC*) which requires analysis to find all relevant *dT*s (line 12). The *nxtM* is either the caller or a callee of the current *method*. The *AnalyzeNextClass* function calls the *AnalyzeClass* and recursively analyzes all related classes within the *program* (lines 13-15).

After obtaining the type information of the values in the program, the next step is to map these values with those extracted from the trace (*GEVSet*). HOMESCAN uses the *IsEVMapVarNode* function (line 18) to do this. The $\widehat{EL}$ has the values of the input arguments of each method which is called during the execution of the control app. This function maps the variable node *vnode* (e.g., node `b` leaf node at branch 1 in Fig. 4) with the `value` of the corresponding input argument (e.g., `b` at line 5 in Fig. 2-a) of the method *mname* (e.g., `a(String,String)`) on the $\widehat{EL}$. If the `value` is equal to $EV.v$, then the corresponding *dT* is assigned to the $EV.t$ (line 19). For example, HOMESCAN identifies that the $EV.v$ `"hue-secret-key-meethue345"` (at row 4 of Table 1) is mapped with the node `b` (leaf node at branch

1 in Fig. 4). Hence, the dT of this $EV.v$ is inferred as $k \in T$ (symmetric key; also named as *SecretKey* at row 4 of Table 1).

Branch Information Inference (line 41). HOMESCAN identifies the branch information resulted from configuration changes in the system. In the *AnalyzeMethods* function, it identifies potential branches by further analyzing nodes which are *BlockStatment*s (lines 40-41). The block statements which are `if-else` or `case-switch` may trigger different transactions based on the input values assigned to the variables in the program. In addition, HOMESCAN utilizes all programs ($\in PS$ at line 2, e.g., mobile and desktop CP source code) to uncover branches introduced during the change of entities.

For example, different privileges may be assigned to different user (e.g., general/guest) or CP (e.g., mobile/desktop) configurations. To formalize the configurations, we assume the finite configuration set $\mathbb{C}=\{C^1, C^2, ..., C^i, ..., C^\lambda\}$ where λ is the number of configurations that can be changed (e.g., $\mathbb{C}=\{C^{user}, C^{CP}\}$ where $C^{user}=\{general, guest\}$ and $C^{CP}=\{mobile, desktop\}$).

As an example, in the LIFX system that we studied, the *desktop* app (CP) is allowed to control the SD over SD's open Wi-Fi hotspot whilst the *mobile* app enforced the setup of SD with the home Wi-Fi before starting the control. Hence, HOMESCAN records the control (over open Wi-Fi) and setup (with home Wi-Fi) actions as two option-branches in the *PI* corresponding to the discovery success transaction.

4.1.2 Exhaustive Search

HomeScan uses exhaustive search to identify the type of a value with respect to a known function applied on a subset of extracted values. Hence, in this search, a finite set of existing functions are executed on all extracted values to check whether the values of unknown types can be generated. As shown in (line 9) Algorithm 1, the *GEVSet* is input into the $ES(GEVSet, F, IK)$ with F a set of existing functions (e.g., MD5, SHA-1 and Base64) and IK. For example, consider v=7B824...299CC in our running example (at row 4 in Table 1). HOMESCAN performs all the existing hash functions on the values it has collected in *GEVSet*. Once it finds that $SHA1$(`Password`,`HubID`) has the same value, it can infer that the type of this value ($EV.v5$ in Table 1) is a hash value over `(Password,HubID)`.

4.1.3 Differential Analysis

HOMESCAN uses $DA(GEVSet, PIL, TRSet, TRSet_{old}, IK)$ (line 10 in Algorithm 1) to infer the types based on the associations from two categories of changes, i.e., configurations and control commands. HOMESCAN identifies the association for the difference of the v in $TRSet_{old}$ and $TRSet$ for the value with identity $EV.id \in TR$. Further, HOMESCAN triggers the trace capturing component to re-execute a particular test case during an analysis to assure the consistency of values $EVSet \in TR$.

Configuration Changes. In our generic architecture, the configuration $\mathbb{C}=\{C^{hub}, C^{SD}, C^{CP}\}$ is a set of entities. Hence, for example, HOMESCAN can substitute the hub with other hubs using the same interface (e.g., the communication protocol), i.e., $C^{hub}=\{hub_1, hub_2, ..., hub_{\mathcal{H}}\}$ where $\mathcal{H}$ indicates the number of the hubs under the control of HOMESCAN, to check the difference of the target $EV.v$ against the change of the hub. For a value $EV.v$ whose domain-specific type is unknown, HOMESCAN infers its type (t) as follows.

- If C^i and $EV.v$ always change together, then they are likely correlated, e.g., `HubID` in the running example.
- If $EV.v$ always changes in every execution, then it is likely a session-specific random nonce, e.g., `nonce`.
- If $EV.v$ keeps constant, then it is likely a protocol-specific value, e.g., `UPnPMsearchRequest`.

Control Command Changes. During the control stage, the commands sent to the SD may be encrypted. HOMESCAN exploits the association between the control commands and the meta-data of the encrypted messages by using differential analysis, to infer the types (e.g., ON/OFF/color-change command) of the encrypted messages. According to the connection through which a control command can be sent to the SD, HOMESCAN uses the following approaches to infer its type.

- ***Persistent Connection.*** Typically, the heartbeats (e.g., shown in Fig. 2-b) are required in order to maintain a persistent connection. In this scenario, the packets including the commands may be inundated by the heartbeat packets. To remove the packets of the heartbeat from the trace, HOMESCAN captures the packets when *no command* is issued by the CP, and labels it as the heartbeat. This enables HOMESCAN to remove the heartbeat packets from the trace and infers the remaining packets as the control command(s). For example, $EV.v8$ in Table 1 is inferred as an association of the command `SecretCommand` when the heartbeat packets (shown in Fig. 2-b) are removed.
- ***Non-persistent Connection.*** In non-persistent connection, a handshake is often used to establish the connection before a control command is sent. Therefore, given a trace of control command execution, HOMESCAN identifies the packets on the trace corresponding to three different stages in a handshake based protocol ($\langle$*connection, command, disconnection*$\rangle$). To achieve this, HOMESCAN reruns test cases for *different* control commands. The packets common in all runs are considered to be relevant to *connection* and *disconnection* stages. The remaining packets are inferred as the *command* data packets.

4.2 Local LTS Generation

After extracting the PIs, HOMESCAN translates them into the LTS representations. Algorithm 3 shows our ap-

input : *PIL*
output: A List $LTSL = [LTS_1, LTS_2, ..., LTS_\sigma]$ where $\sigma =\mid P \mid$. Each entity in P is mapped to an *LTS*

1 **for** $p \in P$ **do**
2 $\quad src_p \leftarrow s_0$, $dst_p \leftarrow null$, $LTS_p = (src_p, \{src_p\}, \varnothing, \varnothing)$;
3 $\quad$ **foreach** $PI^q \in PIL$ **do** $s_p^q \leftarrow null$;
4 **end**
5 **for** $PI^q \in PIL$ **do**
6 $\quad PI^q.ACSeq \frown CreateLCActions(PI^q.msg, PI^q.lc)$;
7 $\quad uch \leftarrow UniqueCH(PI^q.ch)$;
8 $\quad$ **for** $ac \in PI^q.ACSeq$ **do**
9 $\quad\quad p = ac.u$, $l \leftarrow CreateLabel(ac, uch)$;
10 $\quad\quad dst_p \leftarrow GenState(ac)$;
11 $\quad\quad$ **if** $(s_p^q \neq null)$ **then** $src_p \leftarrow s_p^q$;
12 $\quad\quad LTS_p.A \leftarrow LTS_p.A \vee \{l\}$, $LTS_p.S \leftarrow LTS_p.S \vee \{dst_p\}$;
13 $\quad\quad LTS_p.Tr \leftarrow LTS_p.Tr \vee \{src_p, l, dst_p\}$;
14 $\quad\quad src_p \leftarrow dst_p$, $s_p^q \leftarrow null$;
15 $\quad\quad$ **for** $TR.id \in BR$ **do**
16 $\quad\quad\quad$ **if** $(q < TR.id)$ **then** $s_p^{TR.id} \leftarrow dst_p$;
17 $\quad\quad\quad$ **else if** $(q = TR.id)$ **then**
18 $\quad\quad\quad\quad LTS_p.Tr \leftarrow LTS_p.Tr \vee \{dst_p, l, dst_p\}$;
19 $\quad\quad\quad$ **else if** $(q > TR.id)$ **then**
20 $\quad\quad\quad\quad exdst_p \leftarrow GenState(AC_1 \in PI^{TR.id}.ACSeq)$;
21 $\quad\quad\quad\quad LTS_p.Tr \leftarrow LTS_p.Tr \vee \{dst_p, l, exdst_p\}$;
22 $\quad\quad$ **end**
23 $\quad\quad LTSL \leftarrow LTS_p$
24 $\quad$ **end**
25 **end**
26 **return** *LTSL*;

Algorithm 3: LTS Representation Algorithm

proach. It takes the *PIL* (output of Algorithm 1) as input and generates a list of LTSs. It begins with initializing an LTS_p for each entity $p \in P$ with the initial state (s_0), the set of states (S), the set of actions (A), and the set of transitions (Tr) in a tuple $(s_0, \{s_0\}, \varnothing, \varnothing)$ (lines 1-4). Then it iterates through the *PIL* and transforms each *PI* into *LTS* transitions. First, it extends the *PI.ACSeq*, if a private communication exists (line 6). Next, it creates a unique channel (line 7) before creating an action label (line 9). Once the source and destination states and labels are created (lines 9-11), it updates the *LTS* components of entity p identified at line 9. If the *PI* has branch information, it either records the source state of options (line 16), adds self-recursions (line 18), adds sequence-recursions, or merges branches (lines 20-21). Below, we detail the LTS generation.

States. A transition involves two states. Its source state is denoted by src_p, while the destination state is denoted by dst_p. In addition, HOMESCAN uses state s_p^q to track the src_p of a branch, where q is the transaction ID (*TR.id*). The dst_p is given by the function *GenState* (line 10). If the input *ac* represents a new action, *GenState* outputs a new dst_p. If the action has been mapped to a dst_p by the function before, the function outputs the existing dst_p. Moreover, the src_p of the immediate transition is the dst_p of the current transition, when it is not a branch (line 14).

Actions and Transitions. During the iterations through *PIL*, the information in each PI^q is used to create labels (actions). The $PI^q.ACSeq$ states the actions information with their sequence. The algorithm creates labels for actions in the stated order (e.g., $\langle AC_1, AC_2, AC_3, AC_4 \rangle$ where AC_1 and AC_2 are local actions conducted by the sender, $AC_3 = (se, send, msg)$ is an action of message sending, and $AC_4 = (r_i \in R, receive, msg)$ is an action of message receiving). Further, HOMESCAN uses the *CreateLCActions* function to add information of the local sending and local receiving actions to $PI^q.\widehat{ACSeq}$ (e.g., $PI^q.\widehat{ACSeq} \frown \langle (r_i \in R, send, msg), (lc, receive, msg) \rangle$ (line 6).

Each label is created using the function *CreateLabel* (line 9). The input to the function, i.e., *ac*, has information about action (*a* and *X*). If *ac* is a local action, then $a \in \{newnonce, newkey, newkeypair, executeCommand\}$ and $X \in T$. If *ac* is a communication action, then $a \in \{send, receive\}$ and $X = msg$. The input *uch* generated using the *UniqueCH* function is used to send/receive the *msg* via a unique channel (line 7). If *ac* is a local communication, then the *CreateLabel* function uses a unique private channel to transmit the *msg*. Once the label and the next state are ready, LTS_p is updated such that $src_p \xrightarrow{l} dst_p$ is added (lines 12-13).

Branches. If the PI^q includes information about branches (represented by $TR.id \in BR$), it is analyzed from line 15 to line 22. Fig. 6 shows different types of branches in an LTS. If *TR.id* of the branch is greater than that of the current *PI*, it is an *option*. Hence, current dst_p is tracked using $s_p^{TR.id}$ (line 16). After it is set, $s_p^{TR.id}$ is taken as the src_p (line 11) in the next iteration. If the *TR.id* of the branch is the same as that of *PI*, this branch is a *self-recursion*. It is represented as an edge from dst_p to dst_p (line 18). Otherwise, the $PI^{TR.id}$ is already processed. Hence, the dst_p of the first action (as stated in sequence $PI^{TR.id}.ACSeq$) of the branch exists. The *GenState* function returns that existing state as $exdst_p$. HOMESCAN adds a transition from the current state dst_p to $exdst_p$ (lines 20-21). This is called a branch merge. If the first action of the branch exists in the current path (root to the src_p), this branch is a *sequence-recursion*. Hence, HOMESCAN merges the current and existing src_p states. After all actions are processed, the LTS representation is generated.

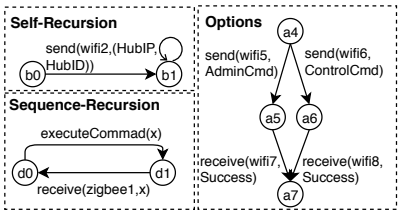

Fig. 6: Types of Branches in an LTS

5 FLAW IDENTIFICATION

After the specification extraction, the local LTS representation is generated to model the behaviors of the entities and their communications. We can further analyze the security properties of the extracted protocol by verifying the generated LTS model against the attack models.

In HOMESCAN, the behavior of an attacker is modeled as an LTS $\mathcal{L}_{att} = (S, s_0, A_{att}, \rightarrow_{att})$, where A_{att} is a set of actions performed by the attacker. In Fig. 8, we illustrate the behaviors of the malicious entities and the network attacker using the examples of the malicious CP and the Wi-Fi network attacker in the running example. The malicious CP pretends to be an honest one in the same network. It sends out its own decided *password'*

$$\frac{s_i \xrightarrow{Send(ch,M)}_i s_i',\ s_j \xrightarrow{Receive(ch,x)}_j s_j',\ a_i = Send(ch,M),\ a_j = Receive(ch,x)}{(s_1,\cdots,s_i,\cdots,s_j,\cdots s_n,(s_{att},NS_{att})) \xrightarrow{(a_i,a_j[M/x])} (s_1,\cdots,s_i',\cdots,s_j',\cdots s_n,(s_{att},NS_{att}))} \quad [\ comm\ ]$$

$$\frac{s_i \xrightarrow{Send(ch,M)}_i s_i',\ s_{att} \xrightarrow{Receive(ch,x)}_{att} s_{att}',\ a_i = Send(ch,M),\ a_{att} = Receive(ch,x)}{(s_1,\cdots,s_i,\cdots,s_n,(s_{att},NS_{att})) \xrightarrow{(a_i,a_{att}[M/x])} (s_1,\cdots,s_i',\cdots,s_n,(s_{att}',Upd(NS_{att}\cup\{M\})))} \quad [\ att\_rec\ ]$$

$$\frac{s_i \xrightarrow{Receive(ch,x)}_i s_i',\ s_{att} \xrightarrow{Send(ch,M)}_{att} s_{att}',\ a_i = Receive(ch,x),\ a_{att} = Send(ch,M)}{(s_1,\cdots,s_i,\cdots,s_n,(s_{att},NS_{att})) \xrightarrow{(a_{att},a_i[M/x])} (s_1,\cdots,s_i',\cdots,s_n,(s_{att}',NS_{att}))} \quad [\ att\_send\ ]$$

$$\frac{s_i \xrightarrow{Receive(ch,x)}_i s_i',\ s_{att} \xrightarrow{Send(ch,\forall)}_{att} s_{att}',\ a_i = Receive(ch,x),\ \ \exists M_i \in NS_{att} \bullet a_{att} = Send(ch,M_i)}{(s_1,\cdots,s_i,\cdots,s_n,(s_{att},NS_{att})) \xrightarrow{(a_{att},a_i[M_i/x])} \rightarrow (s_1,\cdots,s_i',\cdots,s_n,(s_{att}',NS_{att}))} \quad [\ att\_send\_any\ ]$$

Fig. 7: Execution Rules where $x, x^{-1} \in V, M \in T$ and $ch \in C$

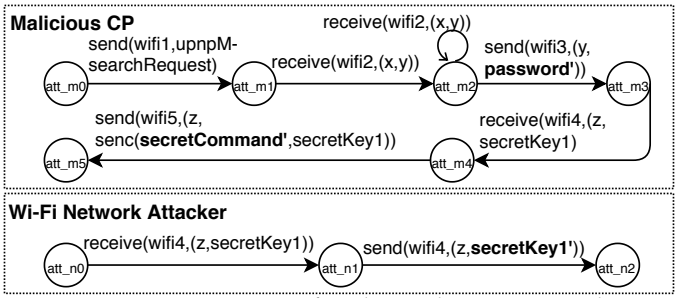

Fig. 8: LTS Representation for the Malicious CP and Wi-Fi Attacker

(state *att_m*2), trying to receive an authenticated token $\mathbf{hash}(HubID, password')$ (this value is stored in a variable z in the LTS in Fig. 8) and the *secretKey*1 from the hub (state *att_m*3). Once successful, the malicious CP is able to control the smart device by sending its own encrypted command ***senc(secretCommand',secretKey1)*** (state *att_m*4). The Wi-Fi network attacker resides between the CP and the HS. It is able to intercept and replace the secretKey1 sent from the honest HS with ***secretKey1'*** (state *att_n*1).

Given the extracted LTS models of both entities and attackers, HOMESCAN generates the execution of the whole smart home system defined in Definition 1.

Definition 1 (Global LTS Generation) *Let* $\mathcal{L}_i = (S_i, s_{0_i}, A, \rightarrow_i)$ *be the model of entity* i, $\mathcal{L}_{att} = (S_{att}, s_{0_{att}}, A_{att}, \rightarrow_{att})$ *be the attack model,* NS_{att} *be the attacker's knowledge set, and* A_s *be the sending action and* A_r *be the receiving action* $(A_s, A_r \subseteq A)$*. The model of the whole system is an LTS* $(S, s_0, A', \rightarrow)$*, where* $S \subseteq S_1 \times \cdots S_n \times (S_{att} \times \mathbb{P}\, T)$*, initial state* $s_0 = (s_{0_1}, \cdots, s_{0_n}, (s_{0_{att}}, \varnothing))$*,* $A' = A \cup A_{att} \cup A_{sr}$*,* $A_{sr} = (A_s \times A_r)$ *is a set of sending and receiving action pairs denoting synchronization, and* $\rightarrow \subseteq S \times A' \times S$ *is the transition relation.*

Due to the page limitation, we list part of our LTS generation rules in Fig. 7, and the full list can be found in our technical report [12]. Here we intuitively introduce it. Rule *comm* denotes a communication action between two honest entities. Rules *att_rec* and *att_send* represent the attacker's capabilities. *att_rec* captures the message sent from an honest entity and those generated by the attacker (attacker can apply a cryptographic function to the captured message and generate new terms using function *Upd*). These new terms are added to the set NS_{att}. *att_send* sends out a fake simulated message to pretend as an honest entity. Rule *att_send_all* represents the network attacker's capability that it can intercept the communication between honest entities and thereafter randomly send a message from its knowledge set NS_{att} to the intercepted honest receiver.

Notice that we define an additional sending action $send(ch, \forall)$ to represent the network attacker's capability of sending any message from the attacker's knowledge set $NS_{att} \subset K$ where the knowledge set K is a set of terms. According to Definition 2, an attacker has the capability of updating his knowledge set NS_{att} by applying the attacker knowledge's set update function *Upd* defined as follows.

Definition 2 (Attacker Knowledge Set Update) *Let* NS_{att} *and* NS_{att}' *be the input and output of the attacker's knowledge update function Upd such that* $NS_{att}' \leftarrow Upd(NS_{att})$*. Let* $m, n,\ pk, sk \in T$ *where pk and sk represent a public-private key pair such that:*

$$NS_{att}' \leftarrow NS_{att} \cup \begin{cases} \{\mathbf{senc}(m,n)\}, & m, n \in NS_{att} \\ \{m\}, & \mathbf{senc}(m,n), n \in NS_{att} \\ \{\mathbf{aenc}(m,pk)\}, & m, pk \in NS_{att} \\ \{m\}, & \mathbf{aenc}(m,pk), sk \in NS_{att} \\ \{\mathbf{sign}(m,sk)\}, & m, sk \in NS_{att} \\ \{m\}, & \mathbf{sign}(m,sk), pk \in NS_{att} \\ \{\mathbf{hash}(m)\}, & m \in NS_{att} \end{cases}$$

In order to verify the security properties, HOMESCAN applies the reachability analysis to the generated execution of the smart home systems, using the classical algorithms such as BFS and DFS. It determines whether a vulnerability exists by searching whether a particular state (referred to *bad state* hereinafter) can be reached in the whole system. For example, in order to determine if the CP can have unauthorized control of the hub and the SD, we can query if the system execution in the running example can reach state *att_m*5 from state *att_m*4 in Fig. 8. Alternatively, we can also query the existence of a particular set of terms in the attacker's

TABLE 5: Summary of Trace Capturing and Pre-processing

Column 2: The no. of generated test cases (all test cases are listed online [12]). Column 3: The no. of captured traces (each test case is executed for three times for differential analysis). Column 4: The no. of identified transactions. Column 5: The no. of extracted unique values.

Case Study	Test Cases	Traces	Transactions	GEVSet
Philips Hue	17	51	41	43
LIFX	11	33	21	17
Chromecast	22	66	30	79

knowledge set NS_{att} to determine if the attacker has enough information to launch an attack. For example, we can query if the set $\{senc(secretCommand', secretKey1), \mathbf{hash}(HubID, password')\}$ exists in the attacker's knowledge set in Fig. 8 to determine if the malicious CP can have unauthorized control of the hub and the SD.

6 Case Studies

To evaluate HomeScan, we conduct case studies on three popular real-world smart home systems from leading smart home brands. In this section, we present our experiment setup and overall results. Afterwards, we focus on one of our findings to demonstrate the stepwise experiment. The recorded demonstration of the security issues and other supporting materials are published online [12].

6.1 Subjects of Our Evaluation

Philips Hue System. Philips Hue is a smart lighting system produced by Philips, and it is claimed to be the world's most popular smart home lighting system (31% market share) [21]. The components and the working process of this system are similar to the running example discussed in Section 2.1. We have analyzed its hub of API version "1.19.0" and bulb with model id "LCT007". This system is comprised of three basic components including a smart bulb (SD), a hub (consisting in HS and ZFE), and a mobile application (CP). The hub is connected to a Wi-Fi router, enabling communication between the CP and the HS over Wi-Fi. The SD and ZFE communicate over ZigBee channel. In each of the three stages, the following system configuration and control are completed.

The CP sends a UPnP M-SEARCH request to discover the HS, while the SD broadcasts a ZigBee beacon request to discover the ZFE on the hub. The CP sends an HTTP POST request with a random string to the HS. After the owner clicks the button, the boolean value in the Philips Hue protocol called "linkbutton" becomes `true`. This enables the hub to respond to the authentication requests from the CPs. However, the "linkbutton" value can also be set by the command `LinkButtonTrue` which can be sent by any authenticated CP. This property results in a vulnerability with several consequences which is discussed soon. The HS authenticates the CP by replying a unique token that represents the CP's identity. The HS also adds this token to the list of whitelisted CP users. Next, the CP sends a `SearchLight` request using the received token to the HS. It initiates TLC between ZFE and SD. After being authenticated by the HS, the CP can send control commands (e.g., turning on/off and changing color/brightness) to the HS. Furthermore, the CP is capable of sending administrative commands, e.g., `LinkButtonTrue`.

TABLE 6: Statistics of Whitebox Analysis

Column 2: The no. of code snippets in the input program that use security APIs. Column 3: The no. of classes recursively analyzed by HomeScan in each code snippet, and their sizes in terms of nodes. Column 4: The no. of nodes labelled with a domain type. Column 5: Total analysis time (in minutes).

Case Study	Code Snippets	Classes (AST Sizes)	Labelled Nodes	Time (min)
Philips Hue	1	3 (448, 472, 2200)	54	0.06
LIFX	3	1 (508)	34	0.04
		1 (344)	6	0.03
		1 (359)	18	0.04
Chromecast	2	2 (6175, 1107)	65	1.50
		7 (1549, 339, 318, 7455, 324, 305, 107)	39	6.19

LIFX Lighting System. LIFX is another smart lighting system which comprises a CP and a SD (i.e., the smart bulb). The SD is Wi-Fi enabled and initially provides an open Wi-Fi hotspot. The CP first joins this hotspot and then broadcasts a `GetService` UDP packet to discover the SD. After the SD is discovered, the CP sends credentials (`SSID` and `Password`) of the home Wi-Fi to the SD over its joined open Wi-Fi. Once the SD joins the home Wi-Fi, its open Wi-Fi is disabled, and the CP broadcasts a `GetService` packet again to discover the SD in the home Wi-Fi. Now, the SD can be controlled by any CP which joins the same wireless LAN as the SD. The CP then can send commands, e.g., `SetColorRequest`, to control SD.

Chromecast System. Google's Chromecast is a streaming media player, which allows streaming a video to a TV. It comprises a CP, a Chromecast receiver, i.e., the SD, and a Google's server (denoted by GS). The Chromecast SD also provides an open Wi-Fi hotspot. The CP joins this hotspot and requests for the device information (e.g., `PublicKey`) of the SD. Next, the CP sends the credentials (`SSID` and password encrypted with the `PublicKey`) of the home Wi-Fi to the SD. Once the SD is connected to the home Wi-Fi, the CP uses Multicast DNS (MDNS) to discover the services provided by the SD. Further, to pair the CP and the GS, the CP sends the `ScreenID` of the SD to the GS. The CP obtains this `ScreenID` by sending `GetMdxSessionStatus` request to the SD. The GS responds to the CP with a `token`, which is later used as an authentication token by the CP at the control stage. After being authenticated by the GS, the CP sends the `PostBindRequest` request with a `VideoID` and the `token` to the GS for casting a YouTube video. The same request without the `VideoID` can be sent to the GS to receive the current status (e.g., current/last `VideoID`) of the SD.

6.2 Setup and Summary

Trace Capturing and Pre-Processing. We use 2.4 GHz deRFusb23-E00 USB sniffing radio stick and Perytons Analyzer to capture ZigBee traces, and Wireshark tool to capture the Wi-Fi traffic. We use Xposed framework [22] to obtain the execution log of the Android app (i.e., the CP). A summary of the statistics related to this component is listed in the Table 5.

TABLE 7: Summary of Flaw Identification

Types of True Positives (**TP**s): **TP#1**: mis-response to discovery request, **TP#2**: flawed authentication protocol, **TP#3**: lack of authorization, **TP#4**: misuse of insecure underlying protocols, **TP#5**: unprotected SD's Wi-Fi hotspot, **TP#6**: lack of device/user authentication protocol, **TP#7**: vulnerable to network traffic replay
Causes of False Positives (**FP**s): **FP#1**: incomplete model extracted, **FP#2**: unrealistic assumption, **FP#3**: infeasible attacker model

Case Study	Violations Reported by HOMESCAN	TP	FP
Philips Hue	The *HS* accepts the discovery request (`UPnPMsearchRequest`) from a malicious *CP*, and replies with `HubIP`, `HubID` and `AssoPermit`.	#1	
	The SD accepts the discovery request (`BeaconRequest`) from a malicious *hub*, and replies with `DeviceID` and `PanID`.	#1	
	The *HS* accepts the authentication request (including a `nonce`) from a malicious *CP*, and replies with a `hash(nonce)`.	#2	
	A malicious *CP* gets authenticated from *hub* and sends the `LinkButtonTrue` admin command to *HS* to enable the functionality of auth-token generation in the *hub*.	#3	
	The SD accepts `LinkNetworkJoinRequest` (of the flawed ZLL protocol) from a malicious ZFE, and replies with a `LinkNetworkJoinResponse`.	#4	
	The *CP* sends a `Controlcmd` to the malicious *hub* which sends the `Encryptedcmd` to its connected *SD*. (During manual confirmation, the malicious *hub* fails to generate the `Encryptedcmd` due to the algorithm for encryption being unspecified in the specification.).		#1
	The *CP* requests an authentication token from a malicious *HS* by sending a `nonce`. The *CP* accepts the token `hash(nonce)` from the malicious *HS*. (During confirmation, we find this attack requires that the malicious *HS* has been authenticated with the *SD*.)		#2
LIFX	The *SD* incorrectly allows a malicious *CP* to connect with its hotspot. Then *SD* authenticates and connects with the attacker's Wi-Fi when the malicious *CP* sends `AttWifi` and `AttPasswrd`.	#5	
	The *CP* connects to a malicious *SD*'s hotspot and sends the `HomeWifiPassword` to the malicious *SD*.	#5	
	The *SD* connects to a malicious *CP* which sends request `SetColorRequest`. The *SD* accepts the request and changes its color.	#6	
	The *SD* accepts a replayed message (`SetPowerRequest`) by a network attacker and changes its on/off status.	#7	
Chromecast	The *SD* accepts the discovery request (`MDNSDiscoveryRequest`) from a malicious *CP*, and replies with `MDNSDiscoveryResponse`.	#1	
	A malicious *CP* connects to the *SD*'s hotspot. Then the malicious *CP* sends `AttWifi` and `AttPasswrd` to authenticate and connect the *SD* to the attacker's Wi-Fi.	#5	
	The *GS* authenticates a malicious *CP* and replies with the `CurrentVideoID` (video ID cast by the victim user) upon receiving `PostBindRequest` from the malicious *CP*.	#6	
	The *CP* connects to the malicious *SD*'s hotspot and sends `aenc(Password,PublicKey)` to the malicious *SD*. The malicious *SD* replies with `adec(aenc(Password,PublicKey), PrivateKey)`. (During confirmation, we find this attack requires all *SD*s share the same key pair, which is unrealistic.)		#2
	The *CP* connects to a malicious *SD* and requests `GetMdxSessionStatus`. The *SD* replies the `ScreenID`. (During manual confirmation, we find even though the `ScreenID` is received, no insecure consequence is caused.)		#2
	The *SD* pairs with a malicious *GS* and replies with `ScreenID` upon the `ScreenIDRequest` from the malicious *GS*. (During manual confirmation, we find that a malicious *GS* is infeasible.)		#3
	The *CP* pairs with a malicious GS and requests an authentication token (`GetLoungeToken`) from the malicious *GS*. The malicious *GS* replies with a `ScreenIDAssociation`. (During manual confirmation, we find that a malicious *GS* is infeasible.)		#3

PI Inference and LTS Representation. In Table 6, we summarize the statistics of the whitebox analysis. The extracted specifications and the detailed LTSs for the three systems are available online [12].

Flaw Identification. HOMESCAN uses a model checker called PAT [23] as the inference engine in our experiments. By analyzing the LTS representations of the systems against the attack models defined in Section 2.2, HOMESCAN reports twelve security flaws. We have reported our findings to the affected parties. Philips Hue confirmed them and proposed fixes, Chromecast has accepted our report, and LIFX confirmed that they are investigating our findings. In Table 7, we summarize our confirmation and analysis on the violations reported by HOMESCAN.

6.3 Details of Findings

As shown in Table 7, vulnerabilities discovered by HOMESCAN can be further categorized into the following seven categories.

Mis-response to Discovery Request (TP#1). During the discovery stage, entities send or reply to discovery requests to identify other possible entities of the system. However, if an entity fails to validate the source of the discovery requests, it may incorrectly respond to the attacker. HOMESCAN identifies three vulnerabilities which belong to this category. First, Philips Hue HS replies to discovery requests, from any UPnP (a known flawed protocol [24]) enabled devices. Second, Philips Hue ZFE always replies to the discovery requests from ZigBee enabled devices. Third, the Chromecast SD replies to MDNS discovery requests from any device in the home Wi-Fi. As a consequence, the attacker can initiate a connection with the victim device and keep them under their control.

Flawed Authentication Protocol (TP#2). Due to the resource limitations, smart home systems may adopt customized authentication protocols. This may result in flawed protocols. HOMESCAN identifies one vulnerability from Philip Hue which can be exploited by a malicious CP. In the authentication stage, the Philips Hue HS relies on the user to press the button on the *hub* to enable the authentication token generation. However, after the pressing, this protocol does not guarantee that the HS only generates the token to the benign CP requests. Con-

sequently, the token can be received by a CP controlled by the attacker.

Lack of Authorization (TP#3). In the control stage, the CP is allowed to send administration commands, such as adding/removing SDs. However, this permission should be limited to authorized parties. HOMESCAN identifies one vulnerability from Philips Hue—any CP authenticated by the HS, instead of only the admin user, can re-configure Philips Hue. This may lead to severe consequences, including uncontrolled authentication and denial-of-service against both the *hub* and the SD.

Misuse of Insecure Underlying Protocols (TP#4). Smart home systems typically rely on existing protocols, but some of them may select an insecure one. HOMESCAN identifies such a vulnerability from Philips Hue, which uses ZLL for authentication. However, ZLL is designed to allow an entity to reset the established connection. In particular, after the SD and the hub have established a connection though ZLL, the attacker can send a `LinkNetworkJoinRequest` to the SD to trigger it to re-execute the protocol. After that, the attacker can impersonate as a hub to establish another connection with the SD.

Unprotected SD's Wi-Fi Hotspot (TP#5). SDs may come with on-board open Wi-Fi hotspots. These unprotected Wi-Fi hotspots can be exploited by malicious entities at all stages of the system. HOMESCAN identifies three vulnerabilities which belong to this category. First, in the discovery stage of LIFX, any CP which joins the SD's hotspot can obtain the SD's configurations and forcefully connect the SD to an attacker's Wi-Fi. Another vulnerability of this category is found in the CPs of the LIFX and Chromecast, which causes them to be deceitfully connected to a fake SD's hotspot. This vulnerability leads to a severe consequence in LIFX's authentication stage, where the CP sends the credentials of the home Wi-Fi in plain text so that the attacker can exploit this vulnerability to steal these credentials.

Lack of Device or User Authentication Protocol (TP#6). Due to the resource limitations, smart home systems may be developed without any authentication protocol. These systems can be exploited by malicious entities to take over control or obtain sensitive information. HOMESCAN identifies two vulnerabilities of this category. In the LIFX system, any CP which joins the home Wi-Fi can control the SD. Similarly, but with a serious consequence, a malicious CP in the Chromecast system which joins the home Wi-Fi can obtain the `VideoID` of a private YouTube video and cast it to the TV screen.

Vulnerable to Network Traffic Replay (TP#7). The network packets exchanged among entities over channels may not include any session related data (e.g., timestamp and nonce). These packets can be intercepted and later replayed by a network attacker who taps on the communication channel. HOMESCAN identifies one vulnerability which belongs to this category. The UDP packets sent by LIFX CP can be intercepted and replayed by a network attacker to manipulate the victim SD.

6.4 Analysis of a Vulnerability

In this section, we use one of the vulnerabilities HOMESCAN identifies from the Philips Hue to further demonstrate how HOMESCAN works on real-world systems.

Input. The *IK* includes that the CP and the HS use Wi-Fi channel, the ZFE and the SD use ZigBee channel, and the 6-digit serial number of the SD. The detailed test cases for the Philips Hue system is included in the technical report [12].

Trace Capturing and Pre-Processing. HOMESCAN is given 9 test cases. It generates 7 extra test cases and 38 transactions.

PI Inference. HOMESCAN generates 38 *PI*s, and four LTSs.

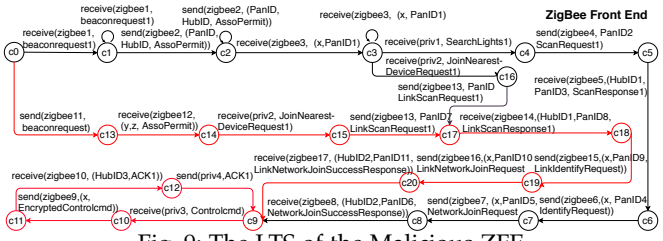

Fig. 9: The LTS of the Malicious ZFE

Flaw Identification. We use the vulnerability "Use of insecure underlying protocols" of Philips Hue to explain this step. The four LTSs and the attacker models are used by HOMESCAN to generate the execution of the whole system. In the following, we explain the attack model, security property, algorithm, counter example and our investigations about the vulnerability.

Attack Model. We consider a malicious *hub* as the attacker. Here, we explain the capabilities reasoned for the vulnerability using the LTS shown in Fig. 9. First, the ZFE of the malicious *hub* discovers the victim SDs by sending `beaconrequest` (from the state c0 to c13). Then, the ZFE is capable of sending a sequence of unauthorized commands including `LinkScanRequest1`, `LinkIdentifyRequest` and `LinkNetworkJoinRequest` to the victim SD.

Security Property Checking. HOMESCAN finds whether the malicious *hub* violates the authorization property. If this property is violated, then the malicious *hub* becomes capable of sending unauthorized commands to the benign SD. To check this property, HOMESCAN finds whether the execution of the whole system reaches the *bad state* `c9` in the LTS (shown in Fig.9) as described in the Section 5. The *bad state* for the property is identified by the fact that, ZFE receives an `ACK` for the `EncryptedControlcmd` it sends and reaches the state `c9`. The malicious ZFE reaches the *bad state* in three traces. In the following, we explain one trace marked in red in Fig.9.

Counter Example. First, the ZFE of the malicious *hub* sends the `beaconrerequest` and receives `y` (`PandID` of the network to which the victim SD is being joined), `z` (`DeviceID`) and `AssoPermit` (from the state c0 to c14) from the victim SD. Next, the malicious ZFE sends the unauthorized `LinkScanRequest1` to the

victim SD. After receiving the `LinkScanResponse1` from the victim SD, the malicious ZFE sends the unauthorized `LinkIdentifyRequest` and `LinkNetworkJoinRequest` to the victim SD. After receiving the `LinkNetworkSuccessResponse`, malicious ZFE sends `EncryptedControlcmd` to the victim SD and receives `ACK`.

Our Investigations. These sequence of messages trigger the TLC of ZLL protocol between the malicious *hub* and the victim SD, forcing the SD to disconnect from the benign ZFE and join the ZFE of the malicious *hub*.

7 LIMITATIONS

HOMESCAN aims to detect as many security vulnerabilities as possible from the partially available implementation of smart home integrations. To this end, it extracts a unified specification of the entire integration. Since our extraction approach is mainly based on the execution and communication traces, capturing a complete specification is infeasible. As a result, false positives may be reported by the flaw identification. In order to remove these, we take as future work to automatically construct attack test cases from the output of the model checker, and execute them against the system under analysis. This serves as a flaw confirmation, and the triggered actions and traces are further given as feedback to HOMESCAN to optimize the extracted specification.

We demonstrate the use of static analysis and testing for specification extraction and security issue detection in smart home integration. Our current approach still requires interaction from the security analyst during the specification extraction process. Although the whitebox analysis and trace analysis can be automated, during the testing, HOMESCAN requires the security analyst to interact with the UI of the control app and to perform actions on physical devices (e.g., press the button on the hub during pairing process), to trigger the functionalities of the system. Translating the generated LTS into the input of the model checker and interpreting the traces given by the model checker also require manual effort from the analyst.

8 RELATED WORK

HOMESCAN targets security of the smart home integration, and thus is related to the research work on specification extraction and IoT security.

8.1 Specification Extraction

Extracting models from the implementation/traces is not a new topic. In the literature, there exist different extraction approaches and algorithms, such as L* and Adaptive Discrimination Tree. In particular to security protocols, Prospex [25] automatically infers protocol specification from the logs of network traces. Discoverer [26] reverse engineers the protocol messages from the network traces. AuthScan [27] extracts the specifications of the authentication protocols and Ye et al [28] extracts models from the payment protocol implementations. Aizatulin et al [29] extract verifiable models from the code of SSL/TLS libraries using symbolic execution. Lo et al [30–32] propose to mine automata models of software from execution traces.

8.2 IoT Security

The research of IoT security mainly focuses on three domains, i.e., IoT devices, protocols and platforms.

Security of IoT Devices. Recently, IoTFuzzer [33] was proposed to find memory corruptions in IoT devices. To overcome the unavailability of firmware for analysis, IoTFuzzer uses the control app to manipulate the input values send to the smart devices, while HOMESCAN performs dynamic analysis on traffic traces to extract protocol information in communication with the smart device. Ho et al. [34] present flaws in the design of smart locks and show how they lead to unauthorized home access. Fawaz et al. [35] propose a system that protects BLE equipped devices from privacy leakages during the device discovery. Das et al. [36] have discovered privacy leakage in BLE network traffic of wearable fitness trackers.

Security of IoT Protocols. Ronen et al. [37] discover a worm attack against Philips Hue lamps by exploiting the ZigBee protocol. Zilliner et al. [38] show that the actual implementations of ZigBee certified smart devices have insufficient security controls. Santos et al. [39] reveal the information leakage on ZigBee network and propose countermeasures. Fouladi et al. [40] demonstrate that proprietary Z-Wave protocol vulnerabilities could lead to remote unlocking of locks. Siby et al. [41] propose IoTScanner which provides an overview of operations in all observed wireless networks. Choi et al. [42] develop an automatic spoofer tool which reconstructs protocols over IEEE 802.15.4. Compared with these studies, our work focuses more on the application layer of the integration of such protocols which may introduce novel attacks.

Security of IoT Platforms. Safechain [43] detects hidden attack chains by exploiting combinations of rules in trigger-action platforms. Although Safechain model the IoT environment, their abstraction is in terms of the status of the devices and automation rules, while HOMESCAN models the communication protocol. Bu et al. [44] also propose an approach to find problems when executing automation rules in an IoT system using model checking and verification. However, to generate the model the authors assume the availability of device specification in a given format, while in HOMESCAN specification extraction is done from a given implementation. Jia et al. [5] propose a context-based permission system for applied IoT platforms. Fernandes et al. [45] propose Fernandes et al. [17] demonstrate that CP applications could be exploited by evaluating the security design of Samsung SmartThings framework. AutoTap [46] provides a platform to ease property specification. The existing studies mainly focus on the application frameworks, which is part of our consideration in our work.

9 Conclusion

We present HomeScan, a semi-automatic approach to extract the abstract specification of the application-layer protocol and internal behaviors of smart home systems from their implementations, whereby it is possible to conduct an end-to-end security analysis against various practical attack models. Using HomeScan, we have found twelve security vulnerabilities from three real-world smart home systems. Our work has demonstrated the necessity of considering the security issues in IoT systems from the perspective of integration.

Acknowledgment. This work is supported by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No.NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate, and the Corporate Laboratory@University Scheme, National University of Singapore, and Singapore Telecommunications Ltd.

References

[1] K. Mahadewa, K. Wang, G. Bai, L. Shi, J. S. Dong, and Z. Liang, "Homescan: Scrutinizing implementations of smart home integrations," in *ICECCS*, 2018, pp. 21–30.

[2] Y. Oren and A. D. Keromytis, "From the Aether to the Ethernet-Attacking the Internet using Broadcast Digital Television," in *USENIX Security*, 2014, pp. 353–368.

[3] K. Townsend, "Attacking smart TVs ," http://itsecurity.co.uk/2014/06/attacking-smart-tvs/, 2017.

[4] Y. Michalevsky, S. Nath, and J. Liu, "Mashable: mobile applications of secret handshakes over bluetooth le," in *MobiCom*, 2016, pp. 387–400.

[5] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash, "Contexiot: Towards providing contextual integrity to appified iot platforms," in *NDSS*, 2017.

[6] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in iot apps," in *CCS*, 2018, pp. 1102–1119.

[7] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity iot," in *USENIX Security*, 2018, pp. 1687–1704.

[8] Z. B. Celik, G. Tan, and P. McDaniel, "IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT," in *NDSS*, 2019.

[9] W. Ding and H. Hu, "On the safety of iot device physical interaction control," in *CCS*, 2018, pp. 832–846.

[10] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized Action Integrity for Trigger-Action IoT Platforms," in *NDSS*, 2018.

[11] R. M. Keller, "Formal verification of parallel programs," *Communications of the ACM*, vol. 19, pp. 371–384, 1976.

[12] HomeScan. https://sites.google.com/view/homescandemo/home.

[13] Samsung SmartThings. http://www.samsung.com/us/smart-home/.

[14] HomeGenie. https://genielabs.github.io/HomeGenie/.

[15] M. M. Hossain, M. Fotouhi, and R. Hasan, "Towards an analysis of security issues, challenges, and open problems in the internet of things," in *IEEE SERVICES*, 2015, pp. 21–28.

[16] T. Denning, T. Kohno, and H. M. Levy, "Computer security and the modern home," *Communications of the ACM*, vol. 56, pp. 94–103, 2013.

[17] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *IEEE S&P*, 2016, pp. 636–654.

[18] H. Ryu and J. Kwak, "Secure data access control scheme for smart home," in *Ubicomp*, 2015, pp. 483–488.

[19] S. Sicari, A. Rizzardi, L. Grieco, and A. Coen-Porisini, "Security, privacy and trust in internet of things: The road ahead," *Computer Networks*, pp. 146 – 164, 2015.

[20] O. Mouaatamid, M. Lahmer, and M. Belkasmi, "Internet of things security: Layered classification of attacks and possible countermeasures," *Electronic Journal of Information Technology*, 2016.

[21] P. den Dunnen. Philips. http://www.newsroom.lighting.philips.com/news/2017/20170831-philips-hue-marks-5th-birthday-with-new-products-and-entertainment-capability.

[22] Xposed. http://repo.xposed.info/.

[23] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "Pat: Towards flexible verification under fairness," in *CAV*, 2009, pp. 709–714.

[24] H. Moore, "Security flaws in universal plug and play: Unplug. don't play," https://hdm.io/writing/SecurityFlawsUPnP.pdf.

[25] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *IEEE S&P*, 2009, pp. 110–125.

[26] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *USENIX Security*, 2007, pp. 14:1–14:14.

[27] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "Authscan: Automatic extraction of web authentication protocols from implementations." in *NDSS*, 2013.

[28] Q. Ye, G. Bai, K. Wang, and J. S. Dong, "Formal analysis of a single sign-on protocol implementation for android," in *ICECCS*, 2015, pp. 90–99.

[29] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and verifying cryptographic models from c protocol code by symbolic execution," in *CCS*, 2011, pp. 331–340.

[30] D. Lo and S.-C. Khoo, "Smartic: Towards building an accurate, robust and scalable specification miner," in *FSE*, 2006, pp. 265–275.

[31] T. D. B. Le and D. Lo, "Deep specification mining," in *ISSTA*, 2018, pp. 106–117.

[32] T.-D. B. Le, X.-B. D. Le, D. Lo, and I. Beschastnikh, "Synergizing specification miners through model fissions and fusions (t)," in *IEEE ASE*, 2015, pp. 115–125.

[33] 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 *NDSS*, 2018.

[34] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, "Smart locks: Lessons for securing commodity internet of things devices," in *ASIACCS*, 2016, pp. 461–472.

[35] K. Fawaz, K.-H. Kim, and K. G. Shin, "Protecting privacy of ble device users," in *USENIX Security*, 2016, pp. 1205–1221.

[36] A. K. Das, P. H. Pathak, C.-N. Chuah, and P. Mohapatra, "Uncovering privacy leakage in ble network traffic of wearable fitness trackers," in *HotMobile*, 2016, pp. 99–104.

[37] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "Iot goes nuclear: Creating a zigbee chain reaction," in *IEEE S&P*, 2017, pp. 195–212.

[38] T. Zillner and S. Strobl, "Zigbee exploited: The good the bad and the ugly," in *Black Hat*, 2015.

[39] J. Dos Santos, C. Hennebert, and C. Lauradoux, "Preserving privacy in secured zigbee wireless sensor networks," in *WF-IoT*, 2015, pp. 715–720.

[40] B. Fouladi and S. Ghanoun, "Honey, i'm home !!-hacking z-wave home automation systems," in *Black Hat*, 2013.

[41] S. Siby, R. R. Maiti, and N. O. Tippenhauer, "Iotscanner: Detecting privacy threats in iot neighborhoods," in *IoTPTS*, 2017, pp. 23–30.

[42] K. Choi, Y. Son, J. Noh, H. Shin, J. Choi, and Y. Kim, "Dissecting customized protocols: Automatic analysis for customized protocols based on ieee 802.15.4," in *ACM WiSec*, 2016, pp. 183–193.

[43] K.-H. Hsu, Y.-H. Chiang, and H.-C. Hsiao, "Safechain: Securing trigger-action programming from attack chains," *IEEE Transactions on Information Forensics and Security*, 2019.

[44] L. Bu, W. Xiong, C.-J. M. Liang, S. Han, D. Zhang, S. Lin, and X. Li, "Systematically ensuring the confidence of real-time home automation iot systems," *ACM Transactions on Cyber-Physical Systems*, vol. 2, no. 3, p. 22, 2018.

[45] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "Flowfence: Practical data protection for emerging iot application frameworks," in *USENIX Security*, 2016, pp. 531–548.

[46] L. Zhang, W. He, J. Martinez, N. Brackenbury, S. Lu, and B. Ur, "AutoTap: synthesizing and repairing trigger-action programs using LTL properties," in *ICSE*, 2019, pp. 281–291.

GLOSSARY

TABLE 8: The Glossary of Terms and Abbreviations

	Term/ Abbreviation	Description
A	a	Name of action in AC
	A	A set of Actions in LTS
	AC	An Action Information (u, a, X)
	$ACSeq$	A sequence of Action Inofrmation
	API	Application Programming Interface
	AST	Abstract Syntax Tree
B	BR	A Set of Branch
C	C	Constant Terms
	$\mathbb{C}$	A set of Configurations
	ch	A channel in CH
	CH	A set of Channels
	CP	Control Point
D	DFS	Depth-First Search
	dT	Domain Type
E	EL	A set of Execution Logs
	EV	Extracted Value
	$EVSet$	A set of Extracted Values in TR
F	F	Function Terms
G	$GEVSet$	Global set $EVSet$
	GS	Google Server
H	HS	HTTP Server
	HTTP	HyperText Transfer Protocol
I	id/ID	The Identity
	IK	A set of Initial Knowledge
	IoT	Internet of Things
	IP	Internet Protocol
K	k	Symmetric Key Term
L	lc	Local Communication
	LAN	Local Area Network
	LTS	Labelled Transition System $\mathcal{L} = (S, s_0, A, \rightarrow)$
M	MDNS	Multicast DNS
	msg	A concatenation of Terms (A message)
	$message$	PlainText (Term)
N	NS	Knowledge Set of Attacker
P	P	A set of Entities
	PI	Protocol Information
	PIL	A list of Protocol Information
	PS	A set of Programs
R	R	A set of Receivers of TR
S	S	A set of States in LTS
	S_1	Discovery Stage
	S_2	Authentication Stage
	S_3	Control Stage
	SD	Smart Device
	SDK	System Development Kit
	se	Sender of TR
T	t	Type of EV
	T	Terms
	TC	A set of Test Cases
	TLC	Touch Link Commissioning
	Tr	A set of Transitions in LTS
	TR	A Transaction
	$TRSet$	A set of Transactions
U	u	Entity which perform the action in AC
	UDP	User Datagram Protocol
	UPnP	Universal Plug and Play
V	v	Value in EV
	V	Variable Terms
X	X	A set of Terms in AC
Z	ZFE	ZigBee Front End
	ZLL	ZigBee Light Link

Kulani Mahadewa received the bachelor's degree in Information Technology from University of Moratuwa, Sri Lanka, in 2013. She is currently a Ph.D. candidate with the Department of Computer Science, National University of Singapore. Her research interests include IoT security and privacy, program analysis, and protocol verification.

Kailong Wang received the bachelor's degree in Electrical and Electronics Engineering from Nanyang Technological University, in 2015. He is currently a Ph.D. candidate and a Research Assistant with the Department of Computer Science, National University of Singapore. His research interests include IoT and web security and privacy analysis.

Guangdong Bai received the bachelor's and master's degrees in computer science from Peking University, China, in 2008 and 2011, respectively, and the Ph.D. degree in computer science from the National University of Singapore in 2015. He is now a Senior Lecturer with the University of Queensland. His research interests include cyber security, protocol verification, and software engineering.

Ling Shi received the bachelor's degree from Institute of Software Engineering, East China Normal University, China and the PhD degree from School of Computing, National University of Singapore. She is a research scientist in School of Information System, Singapore Management University. Her research interests include formal semantics, software/system modeling and verification, and IoT security.

Yan Liu received the bachelor's degree in computer science from Southeast University, China, in 2009, and the Ph.D. degree from National University of Singapore in 2014. She is now a Senior Engineer with Ant Financial-Blockchain Platform. Her research interests include model checking, programming language, IoT and cyber security.

Jin Song Dong received the bachelor's (First Class Hons.) and Ph.D. degrees in computing from the University of Queensland in 1992 and 1996, respectively. From 1995 to 1998, he was a Research Scientist with CSIRO Australia. Since 1998, he has been with the School of Computing, National University of Singapore, where he received full professorship in 2016. He is on the Editorial Board of the ACM Transactions on Software Engineering and Methodology and the Formal of Computing.

Zhenkai Liang received the B.S. degree from Peking University in 1999 and the Ph.D. degree from Stony Brook University in 2006. He is currently an Associate Professor with the Department of Computer Science, National University of Singapore. His research interests include software security, web security, and mobile security.