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 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, HOME-
SCANS uses a hybrid analysis including **dynamic testing**, **whitebox analysis** and **trace analysis**. The dynamic testing executes test cases, and captures communication trace and execution traces; the whitebox analysis identifies semantics by analyzing the program that is available; trace analysis infers the association relation between value of unknown semantics and an entity, a session or a command whose semantics has been identified.

**HOME SCAN** uses labeled transition systems (LTSs) [11], which have been extensively used to model and reason various systems, to represent | 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 a receiving a message). At this abstract level, the secured reasoning can ignore the heterogeneity of underlying protocols, but focus on the logic that is implemented 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. HOME SCAN focuses on extracting as precise specification as possible, whereby we can identify security issues. We prototype HOME SCAN 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 HOME SCAN 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.

### 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...
TABLE 1: Intermediate Outcomes and Corresponding HOME SCAN Approach for the Running Example

<table>
<thead>
<tr>
<th>id</th>
<th>Sender, Receiver, Channel</th>
<th>Extracted Values (Value, Primary Type, Value ID)</th>
<th>Inferred Specification</th>
<th>Approach Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CP, HS, with</td>
<td>(M-SEARCH * HTTP/1.1, String, v1)</td>
<td>msg=(UpnpMsearchRequest)</td>
<td>v1-Protocol, Knowledge</td>
</tr>
<tr>
<td>2</td>
<td>CP, HS, with</td>
<td>(192.168.1.128, String, v2), (192.168.2.234, String, v3)</td>
<td>msg=(HubID, Password)</td>
<td>v2-Protocol, Knowledge</td>
</tr>
<tr>
<td>3</td>
<td>CP, HS, with</td>
<td>(010/100FFFE2D35DF8, String, v3, (pass123, String, v4)</td>
<td>msg=(HubID, Password)</td>
<td>v3-Initial Knowledge</td>
</tr>
<tr>
<td>4</td>
<td>HS, CP, with</td>
<td>(788294F19969C7946DF0C85B1788FE299C, String, v3)</td>
<td>msg=(hash(hash,Password,HubID),SecretKey)</td>
<td>v5-Exhaustive Search, v6-Whitesbox Analysis</td>
</tr>
<tr>
<td>5</td>
<td>HS, CP, with</td>
<td>(192.168.1.128, String, v3, (pass123, String, v2)</td>
<td>msg=(associative,command), local_action = (SD, executeCommand, {msg}), branch=(5)</td>
<td>v7-Eavesdropping Analysis</td>
</tr>
<tr>
<td>6</td>
<td>ZFE, SD, zigbee</td>
<td>(Encrypted Zigbee data (43 bytes), Strings, v6)</td>
<td>msg=(associative,command), local_action = (SD, executeCommand, {msg}), branch=(5)</td>
<td>v7-Virtual-Analysis</td>
</tr>
</tbody>
</table>

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** (S1 in Fig. 1 and in Fig. 2-b). The CP searches for the hub and pairs with the HS (steps 1 & 2).
- **Authentication Stage** (S2). The CP authenticates itself with the HS at the hub (steps 3 & 4).
- **Control Stage** (S3). 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), HOME SCAN 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.

**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.**

<table>
<thead>
<tr>
<th>Attack Model</th>
<th>Attack Capability Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Malicious Entities</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Network Attacker</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

**Table 2: Attack Models and Capabilities**

Transactions on Software Engineering
3 HOME SCAN OVERVIEW AND PREREQUISITES

3.1 HOME SCAN Overview

HOME SCAN 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 HOME SCAN.

- **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 HOME SCAN 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 $(P,C H)$, where $P$ is the set of entities of the input system , and $C H$ is the set of channels used for communication among entities.

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

**Trace Capturing.** The first step of HOME SCAN 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. HOME SCAN 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 $E L$). In addition, HOME SCAN 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. HOME SCAN 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) rounds of information exchange. We represent the abstraction of a single round as a transaction (TR). We define it as a 5-tuple $(i d, s e, R, E V S e t, B R )$, where $i d$ is the transaction ID, $s e \in P$ is the sender, $R \subset P \setminus s e$ is the set of receivers (In multicast communication, there can be multiple receivers), and $E V S e t = \{ E V _ 1, E V _ 2, ..., E V _ { \nu _ i } \}$ is the set of values (total number $\nu _ { i d }$) extracted from the message exchanged in the TR. Each $E V _ i$ is a 3-tuple $( v, t, i d )$ where $v$ is the value, $t$ is its type, and $i d$ 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 $T R S e t = \{ T R _ 1, T R _ 2, ..., T R _ T \}$ where $T$ is the total number of transactions (rounds).

**Branch Information.** Each transaction $T R$ includes a branch set (denoted by $B R$), 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. HOME SCAN 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 $EV \in \bigcup EVSet_i \ (1 \leq i \leq T)$, 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 terms (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**

<table>
<thead>
<tr>
<th>Function Term ($f$)</th>
<th>Definitions</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(message, $k$)</td>
<td>message $\in T$; symmetric key $k \in T$</td>
<td>ciphertext created by symmetric encryption</td>
</tr>
<tr>
<td>aenc(message, $k$)</td>
<td>message $\in T$; symmetric key $k \in T$</td>
<td>ciphertext created by asymmetric encryption</td>
</tr>
<tr>
<td>aenc(message, $sk$)</td>
<td>message $\in T$; private key $sk \in T$</td>
<td>ciphertext created by asymmetric encryption</td>
</tr>
<tr>
<td>hash(message)</td>
<td>message $\in T$</td>
<td>hash value generated by hash function</td>
</tr>
<tr>
<td>sign(message, $sk$)</td>
<td>message $\in T$; private key $sk \in T$</td>
<td>signature generated by signature function</td>
</tr>
<tr>
<td>checksign(sign, $pk$)</td>
<td>signature $\in T$; public key $pk \in T$</td>
<td>result of signature verification</td>
</tr>
<tr>
<td>assoc($t$)</td>
<td>existing term $t \in T$</td>
<td>new term generated by association</td>
</tr>
<tr>
<td>(a, $\ldots$, $b$)</td>
<td>$a, \ldots, b \in T$</td>
<td>concatenation of terms</td>
</tr>
<tr>
<td>[m, $\ldots$, n]</td>
<td>$m, \ldots, n \in T$</td>
<td>set construction of terms</td>
</tr>
</tbody>
</table>

**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.

**Table 4: Communication and Local Actions**

<table>
<thead>
<tr>
<th>Type</th>
<th>Action</th>
<th>Definitions</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm</td>
<td>send(ch, message)</td>
<td>$ch \in C$, $message \in T$</td>
<td>sending a message message via channel $ch$</td>
</tr>
<tr>
<td></td>
<td>receive(ch, $x$)</td>
<td>$x \in V$</td>
<td>receiving a message via channel $ch$ and storing in $x$</td>
</tr>
<tr>
<td></td>
<td>newnonce($x$)</td>
<td>$x \in V$</td>
<td>generating a new nonce and storing it in $x$</td>
</tr>
<tr>
<td></td>
<td>newskey($x$)</td>
<td>$x \in V$</td>
<td>generating and storing a pair of public-private keys</td>
</tr>
<tr>
<td></td>
<td>executeCommand($c$)</td>
<td>constant $c \in C$</td>
<td>executing the command $c$</td>
</tr>
</tbody>
</table>

**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 PLs from the transactions, and then transforms the PLs into LTS representations.

**4.1 Inference of Protocol Information**

Given the transactions generated from trace processing, HOMESCAN uses several analysis techniques to infer the PLs. This is outlined in Algorithm 1. It takes a 5-tuple ($\text{TRSet}, \text{PS}, \text{EL}, \text{IK}, \text{TC}$) as input, where $\text{TRSet}$ is the set of transactions; $\text{PS}$ is the set of programs; $\text{EL}$ is the sequence of execution logs; $\text{IK}$ is the set of initial knowledge; $\text{TC}$ is the set of test cases. The output of the algorithm is a list of inferred PL ($\text{PIL}$), each of which correlates with one transaction. The algorithm executes the next test case ($\text{TC}.\text{next}$ at line 13) and iteratively identifies new semantics until no new information can be found from the input. In each iteration, the $\text{TRSet}_{\text{next}}$ includes new values and new branch information (BR) corresponding to the new configuration specified in the $\text{TC}.\text{next}$. 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.
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2960690, IEEE Transactions on Software Engineering

input : (TRSet, PS, ĖL, IK, TC) 
output: A List PIL = [P1, P2, ..., Pδ] where δ = T; Each transaction in TRSet is mapped to a PI. 
1 T = {f1, f2, ..., fn} where n is the number of selected hash, cryptography and encoding/decoding functions.; 
2 g : GEVSet × F(TRSet) is relation indicating the transactions which a value appears.; 
3 TRSetnew ← TRSet, TRSetsigd ← TRSet; 
4 do T = TRSetnew; 
5 GEVSet = \{EVSet(1 ≤ i ≤ T)\} / global set of EVSet.; 
6 g ← Grouping(TRSet); 
7 GEVSet ← WB(GEVSet, PS, ĖL, IK); 
8 GEVSet ← ES(GEVSet, F, IK); 
9 GEVSet ← DA(GEVSet, PIL, TRSet, TRSetsigd, IK); 
10 PIL ← updatePIL(GEVSet, g); 
11 TRSetnew ← TRSet; 
12 return PIL;

Algorithm 1: PT Inference Algorithm

4.1.1 Whitebox Analysis

HOMESCAN uses WB(GEVSet, PS, ĖL, 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 (lines 16-24). The call graph (cgraph) of the program is used to retrieve the control information during the analysis (line 6).

Below we briefly 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 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).

Algorithm 2: Whitebox Analysis Automation Algorithm

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 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 \((\piT)\) of a variable representing an \(EV\_v\), as shown in Fig. 4. After the symbols in the AST are resolved, HOME SCAN 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 \(EV\_v\)). The basic idea is to annotate parameters and return values with types obtained from the knowledge of the cryptographic APIs. To this end, HOME SCAN 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, HOME SCAN 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 \((node, domainType, relatedMethod)\); \(IK\) 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, HOME SCAN 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, HOME SCAN 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 \((sig\) 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, HOME SCAN also records local actions (\(lCA\)) of the entities related to the generation of a term \((dT)\) (line 37). For example, in the running example, when HOME SCAN 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

1. Currently, HOME SCAN supports Java cryptographic library javax.crypto.
k. Once the \( dT \) and \( leA \) are determined, they are added to the \( dTmap \) and the \( la \) (lines 38-39). Consequently, the \( \text{AnalyzeMethods} \) function recursively analyzes all method nodes in the input \( \text{parsedSmblAST} \), and returns the \( dTmap \) to the \( \text{AnalyzeClass} \) 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 \( \text{AnalyzeClass} \) function. It takes a 4-tuple \( (ast, \text{class}, \text{dTmap}, \text{IK}) \) as input, where \text{class} is the current code snippet (that calls security APIs) in analysis, and outputs the updated \( \text{GEVSet} \) including the domain types inferred (line 24). The \( \text{AnalyzeClass} \) function first calls the \( \text{AnalyzeMethods} \), 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 \text{class}) which have dependency with the nodes with known \( dTs \) (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 \( \text{initC} \), 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 \( \text{kspec} \) at branch 5 (marked in yellow circle) is \( k \in T \) (from Table. 3) representing a symmetric key. Using the four rules, \( \text{HOME SCAN} \) 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 \( \text{kspec} \) leaf node at branch 3. Second, using rule 1, \( k \) is propagated to the node new \( \text{SecretKeySpec} \) \((bk, "AES") \) 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 \text{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, \( \text{HOME SCAN} \) finds the next method (\( \text{nxtM} \)) and its class (\( \text{nxC} \)) which requires analysis to find all relevant \( dTs \) (line 12). The \( \text{nxtM} \) is either the caller or a callee of the current method. The \( \text{AnalyzeNextClass} \) function calls the \( \text{AnalyzeClass} \) 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 (\( \text{GEVSet} \)). \( \text{HOME SCAN} \) uses the \( \text{IsEVMapVarNode} \) function (line 18) to do this. The \( \text{EL} \) 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 \( \text{vnode} \) (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 \( \text{EL} \). If the value is equal to \( Ev.v \), then the corresponding \( dT \) is assigned to the \( EV.t \) (line 19). For example, \( \text{HOME SCAN} \) 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 \text{SecretKey} at row 4 of Table 1).

### Branch Information Inference (line 41)
\( \text{HOME SCAN} \)
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 BlockStatements (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 (∈ 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 $\mathbb{C} = \{C^1, C^2, \ldots, C^\lambda\}$ where $\lambda$ is the number of configurations that can be changed (e.g., $\mathbb{C} = \{\text{user}, C^{CP}\}$ where $C^{user} = \{\text{general, guest}\}$ and $C^{CP} = \{\text{mobile, desktop}\}$).

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 $v=7\text{B}8\text{2}4\ldots2\text{9}9\text{C}$ 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_id, 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_id and TRSet for the value with identity EV.id ∈ 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 ∈ 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., $C^{hub} = \{\text{hub}_1, \text{hub}_2, \ldots, \text{hub}_n\}$

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 (i) 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 heartbeat (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 inaudited by the heartbeat packets. To remove the heartbeat packets 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 ((connection, command, disconnection)). To achieve this, HomeSCAN re-runs 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 approach. 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 $(s_0)$,
input : PIL
output : A List LTSL = [LTS\textsubscript{i}, LTS\textsubscript{2}, \ldots, LTS\textsubscript{p}] where \(\sigma = |P|\).

Each entity in \(P\) is mapped to an LTS

1. for \(p \in P\) do
   
   \(srcp \leftarrow s_p, dstp \leftarrow null, LTSP = (srcp, \{srcp\}, \varnothing, \varnothing)\)

2. foreach \(PIL \in PIL\) do \(s_p^0 \leftarrow null\);

3. end

4. for \(PIL \in PIL\) do
   
   5. PIL.ACSeq\textsubscript{k} = CreateLCActions(PIL.msg, PIL.lc);

   6. uch \leftarrow UniqeCH(PIL.ch);

   7. for ac \(\in PIL.ACSeq\textsubscript{k}\) do

   8. \(p = ac.a, l \leftarrow CreateLabel(ac, uch)\);

   9. dstp \leftarrow GenState(ac);

   10. if \((s_p^0 \neq null)\) then \(srcp \leftarrow s_p^0\);

   11. \(LTSP.A \leftarrow LTSP.A \cup \{l\}, LTSP.S \leftarrow LTSP.S \cup \{dstp\}\);

   12. \(LTSP.Tr \leftarrow LTSP.Tr \cup \{l, p, dstp\}\);

   13. \(srcp \leftarrow dstp, s_p^0 \leftarrow null\);

   14. \(LTSP \leftarrow LTSP \cup \{l, exdstp\}\);

   15. \(LTSP \leftarrow LTSP\);

   16. end

17. end

18. return LTSL;

Algorithm 3: LTS Representation Algorithm

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 PIL.ACSeq\textsubscript{k} 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, HOME\textsc{SCAN} uses state \(s_p^0\) 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 PIL is used to create labels (actions). The PIL.ACSeq\textsubscript{k} states the actions information with their sequence. The algorithm creates labels for actions in the stated order (e.g., \((AC_1, AC_2, AC_3, AC_4)\) 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 \(AC_4 = (r, receive, msg)\) is an action of message receiving). Further, HOME\textsc{SCAN} uses the CreateLCActions function to add information of the local sending and local receiving actions to PIL.ACSeq (e.g., PIL.ACSeq = \(((r, R, send, msg), (l, receive, msg))\) (line 6).

Each label is created using the function \(CreateLabel\) (line 9). To the function, i.e., \(ac\), has information about action (\(a\) and \(X\)). If \(ac\) is a local action, then \(a \in \{\text{newmessage, newkey, newkeypair, executeCommand}\}\) and \(X \in T\). If \(ac\) is a communication action, then \(a \in \{\text{send, receive}\}\) 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, \(LTSP\) is updated such that \(src_p \rightarrow dst_p\) is added (lines 12-13).

Branches. If the \(PIL\) 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 \(s_p^0.TR.id\) (line 16). After it is set, \(s_p^0.TR.id\) 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 \(PIL.TR.id\) is already processed. Hence, the \(dst_p\) of the first action (as stated in sequence PIL.TR.id.ACSeq\textsubscript{k}) of the branch exists. The \(GenState\) function returns that existing state as \(exdstp\).

HOME\textsc{SCAN} adds a transition from the current state \(dst_p\) to \(exdstp\) (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, HOME\textsc{SCAN} merges the current and existing \(src_p\) states. After all actions are processed, the LTS representation is generated.

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 HOME\textsc{SCAN}, the behavior of an attacker is modeled as an LTS \(L_{att} = (S, s_0, A_{att}, \rightarrow_{att})\), 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\textsuperscript{′} (state att\_m2), trying to receive an authenticated token hash(HubID, password\textsuperscript{′}) (this value is stored in a variable z in the LTS in Fig. 8) and the secretKey\textsubscript{1} from the
B. Execution Rules

\[ \text{executeCommand} (c) \rightarrow i \rightarrow s' \]

\[ (s_1, \ldots, s_i, \ldots, s_n, (s_{\text{att}}, \text{NS}_{\text{att}})) \rightarrow (s_1, \ldots, s_i', \ldots, s_n, (s_{\text{att}}, \text{NS}_{\text{att}})) \]

Fig. 10: Execution Rules where \( x, x^{-1} \in V, M \in T \) and \( c \in C \)

---

Malicious CP

- **send(wifi1,upnpM-searchRequest)**
- **receive(wifi2,(x,y))**
- **send(wifi3,(y,rec(m)))**
- **send(wifi4,(z,publickey1))**

Wi-Fi Network Attacker

- **receive(wifi4,(z,publickey1))**
- **send(wifi4,(z,publickey1))**

Fig. 8: LTS Representation for the Malicious CP and Wi-Fi Attacker hub (state \( att\_m3 \)). Once successful, the malicious CP is able to control the smart device by sending its own encrypted command \( \text{send(secretCommand'},\text{secretKey1}) \) (state \( att\_m4 \)). 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 \( att\_m1 \)).

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 \( L_i = (S_i, s_0, A_i, \rightarrow_i) \) be the model of entity \( i \), \( L_{\text{att}} = (S_{\text{att}}, s_{0_{\text{att}}, \text{A}_{\text{att}}, \rightarrow_{\text{att}}}) \) be the attack model, \( S_{\text{att}} \) be the attacker’s knowledge set, and \( A_{\text{att}} \) be the sending action and \( A_r \) be the receiving action (\( A_{\text{att}}, A_r \subseteq A \)). The model of the whole system is an LTS \( (S, s_0, A', \rightarrow) \), where \( S \subseteq S_1 \times \cdots \times S_n \times (S_{\text{att}} \times P T) \), initial state \( s_0 = (s_0_1, \ldots, s_0_n, (s_{0_{\text{att}}, \emptyset})) \), \( A' = A \cup A_{\text{att}} \cup A_r \), \( A_r = (A_2 \times A_r) \) 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 \( NS_{\text{att}} \). **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 \( NS_{\text{att}} \) to the interpolated honest receiver.

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

**Definition 2 (Attacker Knowledge Set Update)**

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

\[
\begin{align*}
\{ \text{pak}(m,n) \}, & \quad m,n \in NS_{\text{att}} \\
\{ m \}, & \quad \text{senc}(m,n), n \in NS_{\text{att}} \\
\{ \text{aenc}(m, pk) \}, & \quad m, pk \in NS_{\text{att}} \\
\{ m \}, & \quad \text{aenc}(m, pk), sk \in NS_{\text{att}} \\
\{ \text{sign}(m, sk) \}, & \quad m, sk \in NS_{\text{att}} \\
\{ m \}, & \quad \text{sign}(m, sk), pk \in NS_{\text{att}} \\
\{ \text{hash}(m) \}, & \quad m \in NS_{\text{att}}
\end{align*}
\]

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 \( att\_m5 \) from state \( att\_m4 \) in Fig. 8. Alternatively, we can also query the existence of a particular set of terms in the attacker’s knowledge set \( NS_{\text{att}} \) to determine if the attacker has enough information to launch an attack. For example, we can query if the set \( \{ \text{senc(secretCommand'},\text{secretKey1}) \} \),
## 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 with 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.

**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.

**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. Consequently, 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. HOME SCAN 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. HOME SCAN 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 through 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. HOME SCAN 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. HOME SCAN 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. HOME SCAN 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 HOME SCAN identifies from the Philips Hue to further demonstrate how HOME SCAN works on real-world systems.

**Input.** The JK 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.** HOME SCAN is given 9 test cases. It generates 7 extra test cases and 38 transactions.

**PI Inference.** HOME SCAN generates 38 PIs, and four LTSs.

**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 HOME SCAN 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.** HOME SCAN 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, HOME SCAN 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 (PanID 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 victim 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 an approach to address how the sensitive data processed by third party apps after obtaining the access. 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.

REFERENCES


### GLOSSARY

#### TABLE 8: The Glossary of Terms and Abbreviations

<table>
<thead>
<tr>
<th>Term/Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>Name of action in AC</td>
</tr>
<tr>
<td>$A$</td>
<td>A set of Actions in LTS</td>
</tr>
<tr>
<td>$AC$</td>
<td>An Action Information $(a, b, X)$</td>
</tr>
<tr>
<td>$AC$</td>
<td>A sequence of Action Information</td>
</tr>
<tr>
<td>$API$</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>$ASIS$</td>
<td>Malicious SNS tree</td>
</tr>
<tr>
<td>$BR$</td>
<td>A Set of Branches</td>
</tr>
<tr>
<td>$C$</td>
<td>Constant Terms</td>
</tr>
<tr>
<td>$CH$</td>
<td>A set of Channels</td>
</tr>
<tr>
<td>$CP$</td>
<td>Control Point</td>
</tr>
<tr>
<td>$DFS$</td>
<td>Depth-First Search</td>
</tr>
<tr>
<td>$EL$</td>
<td>A set of Execution Logs</td>
</tr>
<tr>
<td>$EV$</td>
<td>Extracted Value</td>
</tr>
<tr>
<td>$EV$</td>
<td>A set of Extracted Values in TR</td>
</tr>
<tr>
<td>$F$</td>
<td>Function Terms</td>
</tr>
<tr>
<td>$GEVSet$</td>
<td>Global set EVSet</td>
</tr>
<tr>
<td>$GS$</td>
<td>Google Server</td>
</tr>
<tr>
<td>$HTTP$</td>
<td>HTTP Server</td>
</tr>
<tr>
<td>$HTTP$</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>$s//ID$</td>
<td>The Identity</td>
</tr>
<tr>
<td>$IK$</td>
<td>A set of Initial Knowledge</td>
</tr>
<tr>
<td>$IoT$</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>$IP$</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>$K$</td>
<td>Symmetric Key Term</td>
</tr>
<tr>
<td>$lc$</td>
<td>Local Communication</td>
</tr>
<tr>
<td>$LNS$</td>
<td>Labeled Transition System $L = (S, s_0, A, \rightarrow)$</td>
</tr>
<tr>
<td>$msg$</td>
<td>A concatenation of Terms (A message)</td>
</tr>
<tr>
<td>$message$</td>
<td>Plaintext (Term)</td>
</tr>
<tr>
<td>$NS$</td>
<td>Knowledge Set of Attacker</td>
</tr>
<tr>
<td>$P$</td>
<td>A set of Entities</td>
</tr>
<tr>
<td>$PI$</td>
<td>Protocol Information</td>
</tr>
<tr>
<td>$PIL$</td>
<td>A list of Protocol Information</td>
</tr>
<tr>
<td>$PS$</td>
<td>A set of Programs</td>
</tr>
<tr>
<td>$R$</td>
<td>A set of Receivers of TR</td>
</tr>
<tr>
<td>$S$</td>
<td>A set of States in LTS</td>
</tr>
<tr>
<td>$S1$</td>
<td>Discovery Stage</td>
</tr>
<tr>
<td>$S2$</td>
<td>Authentication Stage</td>
</tr>
<tr>
<td>$SDK$</td>
<td>System Development Kit</td>
</tr>
<tr>
<td>$se$</td>
<td>Sender of TR</td>
</tr>
<tr>
<td>$T$</td>
<td>Type of EV</td>
</tr>
<tr>
<td>$T$</td>
<td>Set of test cases</td>
</tr>
<tr>
<td>$TLC$</td>
<td>Touch Link Commissioning</td>
</tr>
<tr>
<td>$TR$</td>
<td>A Transaction</td>
</tr>
<tr>
<td>$TRSet$</td>
<td>A set of Transactions</td>
</tr>
<tr>
<td>$u$</td>
<td>Entity which performs the action in AC</td>
</tr>
<tr>
<td>$UDP$</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>$UPnP$</td>
<td>Universal Plug and Play</td>
</tr>
<tr>
<td>$v$</td>
<td>Value in EV</td>
</tr>
<tr>
<td>$v$</td>
<td>Variable terms</td>
</tr>
<tr>
<td>$X$</td>
<td>A set of terms in AC</td>
</tr>
<tr>
<td>$ZFE$</td>
<td>Zigbee Front End</td>
</tr>
<tr>
<td>$ZLL$</td>
<td>Zigbee Light Link</td>
</tr>
</tbody>
</table>