

Article

HSAS-MD Analyzer: A Hybrid Security Analysis System Using Model-Checking Technique and Deep Learning for Malware Detection in IoT Apps

Alyaa A. Hamza ^{1,2,*} , Islam Tharwat Abdel Halim ^{3,4} , Mohamed A. Sobh ² and Ayman M. Bahaa-Eldin ^{5,†}

- ¹ Computer & Systems Engineering Department, School of Engineering & Technology, Badr University in Cairo, Entertainment Area, Badr City 11829, Egypt
- ² Computer & Systems Engineering Department, Faculty of Engineering, Ain Shams University, 1 Elsarayat St., Cairo 11517, Egypt; mohamed.sobh@eng.asu.edu.eg
- ³ School of Information Technology and Computer Science (ITCS), Nile University, 26th of July Corridor, Sheikh Zayed City 12677, Egypt; ihalim@nu.edu.eg
- ⁴ Center for Informatics Science, Nile University, 26th of July Corridor, Sheikh Zayed City 12677, Egypt
- ⁵ Faculty of Computer Science, Misr International University, KM 28 Cairo—Ismailia Road Ahmed Orabi District, Cairo 11828, Egypt; ayman.bahaa@eng.asu.edu.eg
- * Correspondence: Alyaa.abdou@buc.edu.eg
- † On Leave from Computer & Systems Engineering Department, Faculty of Engineering, Ain Shams University, 1 Elsarayat St., Cairo 11517, Egypt.

Abstract: Established Internet of Things (IoT) platforms suffer from their inability to determine whether an IoT app is secure or not. A security analysis system (SAS) is a protective shield against any attack that breaks down data privacy and security. Its main task focuses on detecting malware and verifying app behavior. There are many SASs implemented in various IoT applications. Most of them build on utilizing static or dynamic analysis separately. However, the hybrid analysis is the best for obtaining accurate results. The SAS provides an effective outcome according to many criteria related to the analysis process, such as analysis type, characteristics, sensitivity, and analysis techniques. This paper proposes a new hybrid (static and dynamic) SAS based on the model-checking technique and deep learning, called an HSAS-MD analyzer, which focuses on the holistic analysis perspective of IoT apps. It aims to analyze the data of IoT apps by (1) converting the source code of the target applications to the format of a model checker that can deal with it; (2) detecting any abnormal behavior in the IoT application; (3) extracting the main static features from it to be tested and classified using a deep-learning CNN algorithm; (4) verifying app behavior by using the model-checking technique. HSAS-MD gives the best results in detecting malware from malicious smart Things applications compared to other SASs. The experimental results of HSAS-MD show that it provides 95%, 94%, 91%, and 93% for accuracy, precision, recall, and F-measure, respectively. It also gives the best results compared with other analyzers from various criteria.

Keywords: data security; triggers/actions; smart homes; software verification



Citation: Hamza, A.A.; Abdel Halim, I.T.; Sobh, M.A.; Bahaa-Eldin, A.M. HSAS-MD Analyzer: A Hybrid Security Analysis System Using Model-Checking Technique and Deep Learning for Malware Detection in IoT Apps. *Sensors* **2022**, *22*, 1079. <https://doi.org/10.3390/s22031079>

Academic Editors: Hany Atlam and Nawfal Fadhel

Received: 28 December 2021

Accepted: 24 January 2022

Published: 29 January 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The Internet of Things (IoT) connects multiple smart objects that work together automatically with limited human activity. With expectations of spreading to 75 billion devices by the end of 2025, IoT is one of the fastest-growing areas in the internet world [1]. As a result, IoT security and privacy have become one of the hottest topics. Because of the growing number of IoT applications, such as in healthcare, industrial, agriculture, and smart home applications, etc., there is increasing vulnerability to attack and violations of privacy due to the different types of devices and sensors. There are several attacks such as (a) physical attacks, (b) software attacks, (c) network attacks, (d) encryption, or data attacks that may occur in IoT [2,3].

The IoT application consists of several IoT systems that connect physical processes with digital communication, and these depend on edge-based (sometimes also known as event-driven) application. Event-driven application is responsible for taking the data from sensors such as motion sensors, contact sensors, and smoke sensors, and command from actuators such as smart power outlets, door controls, and smart locks towards providing different forms of automation [4]. IoT systems also build their infrastructure with (i) devices from the bottom up (ii) networking protocols, and (iii) IoT programming platforms [5].

One problem of IoT is the unequal status of existing platforms. As a result, the software stack designed for the applications responsible for monitoring and controlling all objects. Most of them do not have good essential tools and services to analyze security and privacy. In particular, third-party apps installed by end users to automate IoT devices' tasks may include data leakage and abuse of permissions. Untrusted third-party apps lead to serious errors and misconfigurations which affect the behavior of IoT applications [6,7].

Security analysis systems (SAS) play an essential role in providing security and privacy, as they can analyze the behavior of IoT applications by detecting any malware, violations, or vulnerabilities. SASs use different program analysis techniques and methods of analysis (static, dynamic, or hybrid). The use of an SAS does not present any problem in discovering traditional malware, but what is dangerous is the emergence of new types of these malicious codes. According to science and business studies, around one million malware files are generated daily, and cyberattacks would harm the world economy by around 6 trillion USD yearly by 2021 [8].

Some SASs use static analysis such as [9–11], which analyzes the program without executing; this method has drawbacks, as some data may be lost during the analysis phase. Other SASs use dynamic analysis such as [12–14]. This analytical method discovers the code skipped by the static analytical methods, either because it is not created until rendering or cannot reach statically analyzable control flow [15]. Therefore, the hybrid analysis provides the best result to detect any malware, by avoiding static or dynamic usage drawbacks.

Static analysis extracts static features from source codes, including strings, opcode, N-grams, bytes, etc. On the other hand, dynamic analysis extracts dynamic features, including API traces, instructions traces, network traffic, etc. Therefore, the usage of hybrid analysis is the best tool for analysis to cover all features.

The main contributions of this paper are to introduce:

- HSAS-MD, a new SAS among the effective SASs, which is continuously being developed as a security compiler tool to act as a protective shield to detect malware. It depends on the PA of the IoT APP, which may automatically verify app behavior.
- HSAS-MD, a hybrid analysis which analyzes IoT Apps by extracting static and dynamic features based on model-checking techniques. It also utilizes deep learning to discover new malware and classify it to provide security, safety, and privacy by detecting any abnormal behavior.
- The evaluation of the HSAS-MD performance, which is measured by applying the proposed analyzer after enhancing CNN. The results of the proposed HSAS-MD were compared with similar SASs. HSAS-MD gave the best accuracy, precision, recall, and F-measure results.

Most of this paper's sections are as follows: Section 2 presents an overview of the background and some basic concepts related to IoT security analysis. Section 3 illustrates the recent related works of the security analysis system for IoT and the research goals. Section 4 presents a proposed hybrid security analysis system (HSAS-MD Analyzer) with more details. The implementation and evaluation of HASA-MD are present in Section 5. Finally, Section 6 presents conclusions, limitations, and future research trends in the security analysis system for IoT. The relevant abbreviations used in this paper are shown in Abbreviations.

2. Background and Basic Concepts

This section presents an overview of the security analysis system (SAS) for IoT (Section 2.1), which includes some information about the Model-Checking Technique (MCT), Path Sensitivity, and Control Flow Graph (CFG). The basic concepts of the smart things platform are then presented in (Section 2.2), while (Section 2.3) shows the role of deep learning (DL) for IoT security.

2.1. Security Analysis Systems (SAS) for IoT

Most research has proven that IoT architecture is composed of three layers, which are: (i) the precision layer (or physical layer), which contains actuators, sensors, and intelligent objects; (ii) the network layer, responsible for linking, transmitting, and processing intelligent devices and objects; (iii) the application layer, which handles all applications that users interact with, such as healthcare and smart home applications. In addition, there are more famous control platforms which developers can use to construct smart apps that interact with sensors and actuators wirelessly, such as Samsung's SmartThings platform [16], Apple's HomeKit platform [17], and Amazon's Alexa platform [18].

Most IoT security research tends toward the security of platforms, communication protocols, smart applications, and IoT devices. On the other hand, there are few studies on security from a comprehensive IoT perspective, including the interaction between smart objects and their applications and platforms [19]. IoT platforms acquire sensor and device data, then transform it into a lightway that can be easily consumed by other application software. This data must be processed and analyzed to make sure it is not malware. Event-driven technology is one of the technologies responsible for handling the communication between application, platform, and services by events.

Event-driven technology can specify the criteria to be fulfilled; to collect, evaluate, and continue to compare a significant amount of heterogeneous data in order to identify different situations within a given domain [20]. It is also an interactive system of objects managed by an event dispatcher (which acts as an intermediary between objects) through a message-passing mechanism. There are gaps in the event-driven technology; for example, (1) there are no access control mechanisms for object aggregation, event transmission, and event capture; (2) the effect of disfigured events is usually hard to control, and these gaps could easily expose you to various types of malware.

Security analysis systems (SAS) focus on malware detection while transmitting or receiving data to/from IoT devices. IoT apps verify this behavior, as shown in Figure 1. The heterogeneity of IoT applications and platforms makes the malware publisher more accessible. Therefore, SAS has an efficient role in providing security for IoT applications.

An SAS depends on four factors to build an efficient detector: (i) the PA type used (static, dynamic, or hybrid); (ii) the PA technique used in the SAS, such as model-checking, program-slicing, taint-tracking, etc.; (iii) PA sensitivity, describing splitting data into uncertainties in the output such as context, path, API, etc.; (iv) characteristics of analysis, such as a control flow graph (CFG).

2.1.1. Model-Checking Technique (MCT)

It is a significant challenge for IoT systems to prevent any infiltration by various malware types. The model-checking technique (MCT) is used in static and dynamic analysis to detect malware. This technique detecting malware belonging to the same family well and is also successful against malware based on obfuscation and polymorphic techniques. MCT identifies the sequencing of states that occur during the program's execution and determines whether or not the sequences of states meet the safety property [21]. It scans and checks the behavioral property inside the source code to find any mistakes. The code is considered error-free if no errors are detected regarding a given property.

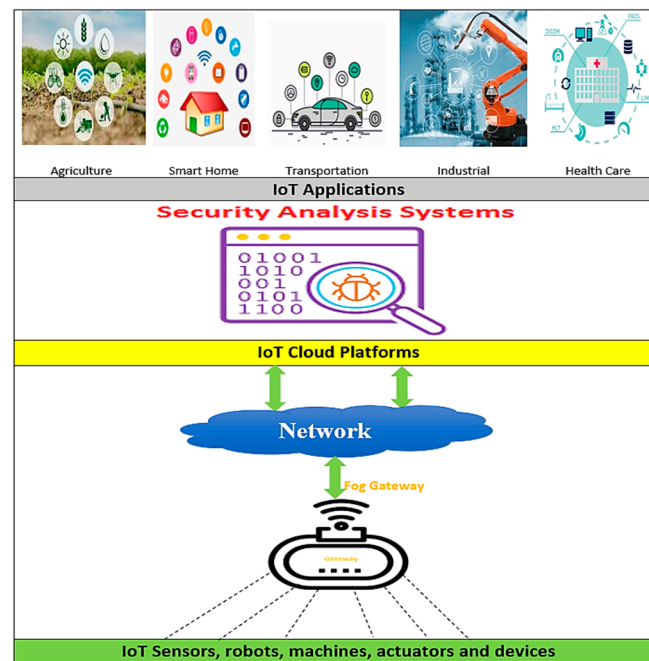


Figure 1. Security analysis systems for IoT applications.

In [22], MCT is used for formal verification in IoT apps, including healthcare, communication protocols, security, and monitoring applications. MCT examines a system's validity against properties and malware. Malware behaviors are manually extracted in this detection method, and behavior groups are structured using linear temporal logic (LTL) to represent a specific feature [23]. This enables flexibility in verifying all desired properties, and offers concrete counterexamples with linear temporal logic (LTL), which is helpful in studying why and how bad states exist. It can catch the holistic essence of the test's multi-application connections and more effective—but not rigorous—checks [4]. MCT is distinguished from other techniques in improving system quality: (1) Testing is the most efficient and straightforward method of detecting problems; (2) Abstract interpretation and other program analyses are similar to model checking, in that they use algorithms to prove program properties automatically; (3) Higher-order proving is a strong and proven method for verifying complex systems [24].

2.1.2. Control Flow Graph (CFG)

CFG is one of the directed graphs in which the nodes represent simple blocks (a linear sequence of program instructions of one entry point and one exit point), and the edges represent control flow paths. CFG has benefits, including: (i) capturing a significant element of code complexity; (ii) having a simpler representation form than source code. (iii) being able to directly list different improvement actions [25]. Each graph is generated using path sensitivity. It is sufficiently reliable for verification. It can justify branch connections, which are typically required to check the number of error reports obtained during the test. In [26], there are two symbolic states at a melting point in the control flow. It has the same ownership status, producing one symbol, the power, by incorporating their implementation cases in the data flow analysis. Symbolic states are otherwise treated separately as in the study of sensitive paths.

2.2. SmartThings Platform

SmartThings is one of the most common open developer platforms for the IoT, with many applications among the platforms available today. Smart devices deal directly with the central hub. An IoT application called Smart Apps, or the third-party Apps, a small Groovy-based system, allows users to optimize their homes using the devices' capabilities. This type of app is the riskiest in IoT as it is untrusted. It may contain misconfiguration—even malicious

intent—and bugs that present a threat. The SmartThings platform has some advantages which make it the best choice from the other platforms: (i) it has a set of 521 apps called Smart Apps. Specific competing platforms such as HomeKit, Weave/Brillo, and AllJoyn have fewer than 50 applications each in formative stages; (ii) it has built-in support from major manufacturers for 132 different devices; (iii) it shares the key principles of security architecture [27–29].

The SmartThings platform comprises three main mechanisms, as shown in Figure 2: (i) a permission system scheme that allows developers to define devices and user inputs needed for a specific device app at the time of launch; (ii) Smart Devices, a software proxy that runs the apps for hardware devices; (iii) SmartApps, a software proxy enabling users to manage their homes using device capabilities. Events are an integral part of the SmartThings platform. They allow SmartApps to respond to physical environment changes and create automated processes around them. SmartApps or Device Handlers do not explicitly create event instances. Instead, they are internally made by the SmartThings platform and passed on to SmartApps event handlers who have subscribed to those events.

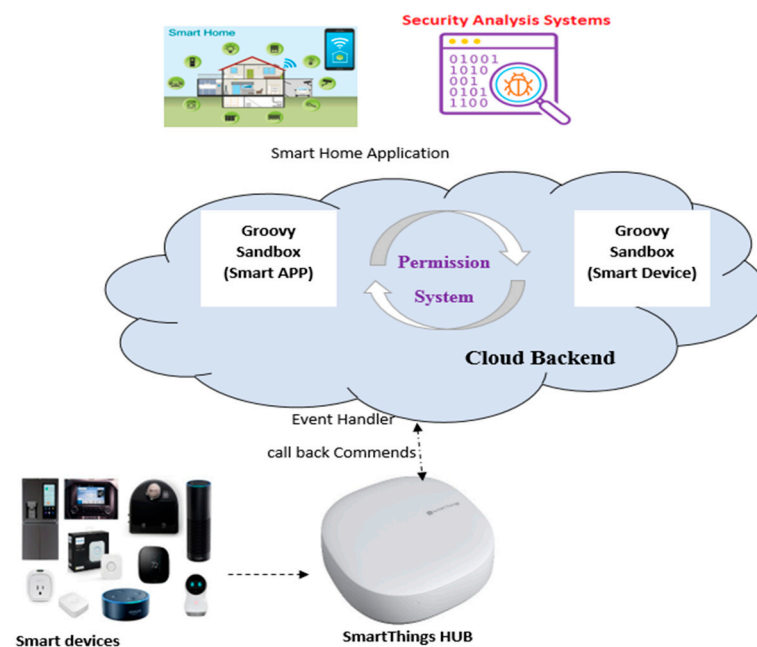


Figure 2. SmartThings Platform Architecture.

2.3. Deep Learning (DL) for IoT Security

The SAS systems that use machine-learning or deep-learning methods are more productive than those that depend on program analysis techniques. They are significant in converting IoT systems' security from simply supporting secure connectivity between devices to security-based intelligence systems. In [30], deep learning (DL) is a branch of machine learning that has advantages in previous studies at detecting security breaches. It has a high effect on malware detection due to learning more conceptual functions, reduced training complexity, increasing accuracy, managing massive datasets, and supporting feature extraction. In [31] a DL approach is proposed based on Stochastic Gradient Descent (SGD), which identifies attacks in the social IoT. The results show that DL methods outperformed more profoundly in every assessment aspect. A multitask deep-learning model for malware detection and classification was proposed in [32]. A combined feature set consisting of null-terminated tokens, API event plus parameter value, and API trigrams, were obtained from static and dynamic analysis. A deep feed-forward neural network was trained using a planned feature vector. In [33], CNN with a one-dimensional kernel (CNN-1D) is used to detect network intrusions. The detection of network intrusions aims

to identify network traffic as either normal or an attack carrier. The information taken from network flow packets is used to classify the data [34].

The SAS's main objective is to detect any malware present in the data sent or received between cloud backend platforms and smart mobile applications.

3. Related Works and Research Goals

This section presents an overview of various related SASs recently applied to IoT applications (Section 3.1). Section 3.2 gives the proposed analyzer's research scope and objectives, and Section 3.3 presents the triggers/actions concept in the IoT app.

3.1. Related Work

IoT apps are convenient, but they expose us to new security risks. These risks may come from the collaborative behavior of several apps compared to traditional software applications. Early work used customized safety and security principles to detect these risks; however, these principles may not predict all device and application usages in a smart home, ending in false alarms. In [35], extraction sandboxes are used as a security measure in IoT. After analyzing a set of behaviors from various apps and devices, a sandbox is installed, reinforcing the idea that previously undiscovered behaviors are not permitted. As a result, the execution of malicious behavior introduced by software upgrades or concealed by strategies to resist program analysis is prevented. Therefore, there are two main critical issues in IoT applications. These issues are as follows: (i) verifying the codes by detecting any abnormal behavior in the IoT applications; (ii) detecting if there is malware or not. Therefore, the analyzers are built to solve these issues.

In addition, the analyzers have important metrics that play an essential role in improving their accuracy, such as analysis type and analysis technique. In [36], the author argues that static analysis continues to improve by using various methods to deal with the most advanced malware obfuscation techniques. However, there is a fundamental limit to what can be consistently detected to detect opaque constants; an obfuscation transformation has been created that obscures program control flow, hides access to local and global variables, and disrupts tracking values held in processor registers, using the proposed obfuscation method. Moreover, the opaque constant primitive can be implemented in a manner that may be difficult for any static-code analyzer to analyze. This therefore shows that static analysis alone may no longer be sufficient to identify malware. Instead, the hybrid analysis is the best for obtaining accurate results.

In [37], a framework was built for malware detection, focused on correlating and configuring static and dynamic API call sequences. Throughout their study, they examined the distinction and relationship between static and dynamic API call sequences by identifying various forms of malicious behaviors. After correlation and fusion, a hybrid vector space is developed for detection and classification to test their method's efficacy. Finally, four classifiers were trained to detect/classify malware, including Random Forest, K-Nearest Neighbor, Decision Tree, and Extreme Gradient Boosting. In addition, [38] presented a framework that combines static and dynamic features into a single classification scheme. Function length frequency and printable string information are extracted and translated to vector representations for each executable file. First, all vector features are merged into a single vector for each executable. The resulting vector is then used as input to four simple classifiers.

Several surveys discussed malware detection using different techniques; for example, articles [39–41] offer a systematic and comprehensive description of machine-learning malware-detection strategies and DL techniques. It offers DL mechanisms categorized by the type of network input, according to the methods used to extract a feature vector representing the executable. The methods use the grayscale description of the executable as input, as well as use it to fed with the series of API function infection. these methods depend on describing programs according to a series of instructions, a byte stream, and data traffic.

In [5,41], security and privacy issues are presented in IoT platforms such as Samsung's SmartThings, Apple's HomeKit, Open-HAB, Amazon AWS IoT, and Android Things, that motivate program analysis techniques such as model checking, taint analysis, code instrumentation, and symbolic execution).

The code of an IoT application can be transformed into a sensor-device state-actuator platform structure with three types of common building blocks: permissions, events, and call graphs. It focuses on program analysis goals, i.e., sensitive data leaks and permission misuse. It also discusses the issues and challenges in IoT program analysis according to three points: firstly, according to IoT-specific issues such as physical channels, simulation and modeling of IoT programs, automated tests-case generation, multi-app analysis, interaction between IoT devices, and trigger-action platform services; secondly, according to IoT application idiosyncrasies such as RESTful APIs, and language-inherited operations; thirdly, according to analysis sensitivities, such as flow sensitivity, context sensitivity, field sensitivity, path sensitivity, and provenance tracking). It includes FlowFence, Saint, Context-IoT, SmartAuth, ProvThings, and Soteria. Many researchers are still conducting security studies for IoT applications to provide a practical overview of the security solutions for supporting security by applying many SASs, as shown in Table 1.

Analysis types (static, dynamic, or hybrid) and analysis techniques are essential for building security analyzers to detect malware [41]. Many security analysis systems (SASs) are based on static analysis. Soteria [10] is a static analyzer using the model-checking technique (MCT), that extracts a state model from the code of an IoT application to verify if an application or multi-app system respects security, safety, and functional properties. Another version is Soteria2 [42], a static analyzer using the convolutional neural network (CNN). It is a random walk-based traversal method for feature extraction that employs both density-based and level-based CFG labels to achieve consistent representation.

IoTSan [4] employs the MCT and state model to check the approach to identify the causes of cyber vulnerabilities, and provides actual precedents to clarify such triggers. ForeSee [43] creates a multi-layer IoT hypothesis graph by simultaneously representing all of the critical components of IoT systems, such as the physical surroundings, devices, protocols, and apps. If there are any violations, the model checker can evaluate the created hypothesis graph to validate system security or generate attack paths.

There are other static analysis SASs that use MCT. However, they depend on extracting rule models, rather than state models, from the code of IoT applications such as IoTCOM [6], SIFT [44], iRuler [45], and TAPInspector [46]. They all utilize static analysis algorithms to automatically identify IoT app behavior from rule models.

The SASs based on dynamic analysis are IoTGuard [12] and IOTFUZZER [14]. IoTGuard works using the code instrumentation technique to analyze code. It stores data of the apps in a dynamic model, then checks security issues on the dynamic model. However, IOTFUZZER identifies vulnerabilities to memory corruption in IoT devices without accessing their firmware images using dynamic analysis.

The use of static or dynamic analysis is not enough to detect all malware. Every type of analysis cannot cover all malware. PATCHEC-KO [47] is a hybrid analyzer that depends on static and dynamic analysis. It optimizes deep learning and hybrid binary analysis to execute multi-platform binary code similarity analysis to identify vulnerabilities without high-precision source code access. It is noted that hybrid analysis is more effective in detecting malware after studying different SASs. Moreover, MCT is more effective at detecting malware in IoT apps.

In this paper, the proposed HSAS-MD based on MCT uses hybrid analysis types (static and dynamic) and hybrid models (state and rule) to avoid the disadvantages of using one analysis type or analysis model. These hybrid analysis and hybrid models will increase the accuracy malware detection.

Table 1. An Overview of Security Analysis Systems (SAS) in IoT applications.

Ref.	SAS. Name	Analysis Type	Analysis Technique	The Main Work	Advantages	Limitations
[17]	Soteria	Static	Model-Checking	Extracts a state model from the code of an IoT application to verify if an application or multi-app system respects security, safety, and functional properties.	SOTERIA analysis on MALIOT indicated it was accurate at recognizing 17 of 20 separate property violations in those 17 apps.	1: The use of call diagnosis by reflection. 2: Dynamic device permissions and app configurations.
[34]	IoTCOM	Static	Model-Checking	Explains how to construct a flexible model extractor that utilizes static analysis algorithms to identify the behavior of IoT apps automatically.	IoTCOM reduces the violation detection time by 92.1%	Depends on all static and dynamic features, which may include unnecessary features.
[5]	IotSan	Static	Model-Checking	Employs the model-checking approach to identify the causes of cyber vulnerabilities and provides actual precedents to clarify such triggers.	Recognizes 147 vulnerabilities. Can identify possible security violations	1: The checker of the Spin model cannot manage a file size more significant than that of the Promela code. 2: The G2J translator does not recognize heterogeneous sets.
[19]	IoTGuard	Dynamic	Code-Instrumentation	Works in three phases: (a) execution of a code instrumentor; (b) storing data of the apps in a dynamic model; (c) detection of IoT security on the dynamic model	IoTGuard introduces 11 single measures, then blocks 16 in 6 (17.1%) Smart-Things, then 5 (16.6%) IFTTT apps.	Enables a user to define policies through IOTGUARD's GPL.
[30]	PATCH EC-KO	Hybrid	Deep-Learning	Optimizes deep learning and hybrid static-dynamic binary analysis to execute multi-platform binary code similarity analysis to identify vulnerabilities without high-precision source code access.	The PATCHECKO differential engine identifies between still-vulnerable functions and those set with an accuracy of 96%.	The accuracy is not a high ratio.
[21]	IOTFUZZER	Dynamic	Taint-Tracking and Machine-learning	IOTFUZZER, which aims at identifying vulnerabilities to memory corruption in IoT devices without accessing their firmware images.	IOTFUZZER successfully found 15 vulnerabilities of memory corruption (which include eight newly discovered vulnerabilities).	1: Scope of testing. 2: Connection mode 3: Cloud relay 4: Result judgments 5: Result accuracy
[7]	Soteria2	Static	Convolutional Neural network	A random walk-based traversal method for feature extraction employs both density-based and level-based CFG labels to achieve consistent representation.	Soteria achieves a 97.79% accuracy rate for detecting AEs and 99.91% accuracy of malware groups	1: CFG does not necessarily reflect the actual code. 2: Obtaining a CFG representation cannot be performed under obfuscation

3.2. Research Scope and Objectives of the Proposed Analyzer

The research scope depends on converting the source code of the third-party new IoT apps to analyzable formats, as shown in Figure 3, then selecting the static features' "actions". The proposed HSAS-MD analyzer works on these features using the CNN model to detect malware in these features, then verifies if there is any abnormal behavior using MCT.

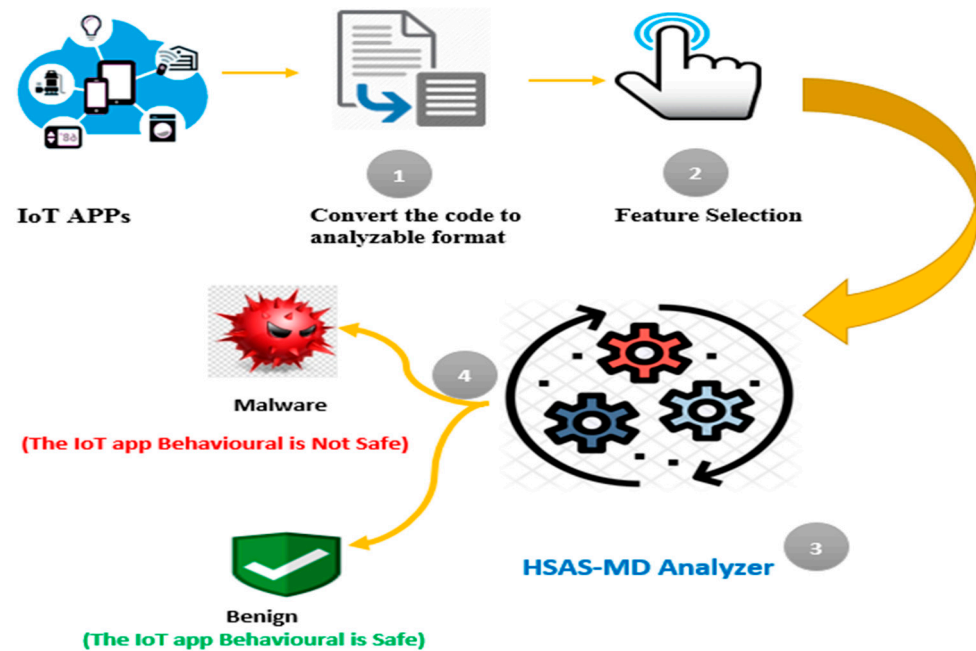


Figure 3. The research scope of the proposed analyzer.

The proposed analyzer's main objectives are to detect vulnerabilities in supporting IoT apps' security through detecting malware and classifying them, as well as, using CNN, to detect any abnormal behavior (conflict actions) in the IoT application using MCT, such as breaches due to connectivity or device failures, misconfiguration problems, data leakage, and physical security breaches.

3.3. The Trigger/Actions in the Smart Home Apps' "Third-Party App"

IoT environments consist of various applications that deal with others through commands (trigger, conditions, and actions) to operate or control devices and sensors. Every IoT app has rules specified using "an event-condition-action paradigm." These rules represent the behavior of IoT APPs for the proposed HSAS-MD as groups of triggers and their corresponding conditions and actions. The rule model represented is $R_h = \langle Th, Ch, Ah \rangle$, explained as follows:

- **Triggers (Th):** Cyber or physical events that devices transmit to smart homes, such as the activation of a motion sensor which triggers the rules;
- **Conditions (Ch):** If a rule may apply, the logical predicates are evaluated on the current status of the devices. For example, a rule runs only if the system is in "home" mode;
- **Actions (Ah):** When the conditions are satisfied, the rule changes the state of one or more devices, leading to a physical change, such as the activation of a light switch.

Each rule attribute (Th, Ch, Ah) can be represented as a triple of a device $D(\alpha)$, an attribute $A(\alpha)$, and a set of values $V(\alpha)$, in which α is a number from 1 to n [6].

The main problem is that end users can use untrusted apps, including malicious commands. This app is called a third-party app in an IoT environment. If this app contains

malware (malicious commands), it will lead to unpredicted behavior of the other apps, devices, and sensors connected with it, as shown in Figure 4.

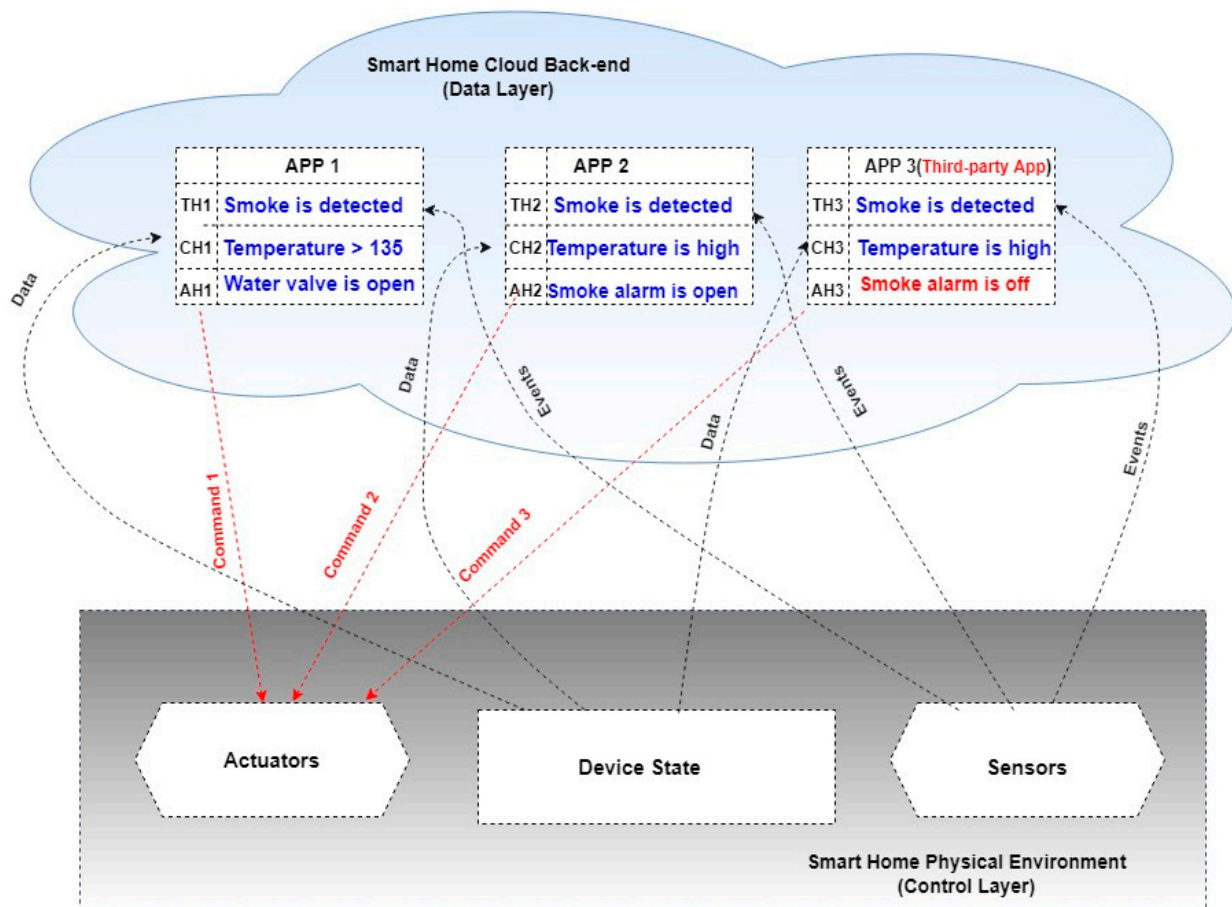


Figure 4. Smart home apps, including malicious third-Party apps.

The smoke alarm application is one of the typical applications used in smart homes that end users may install. The main goal of this application is to detect smoke and act as quickly as possible. This application consists of a smoke-detection-alert system, a water valve (on the lower floor), and a light switch (in the living room). Operating this application issues a smoke alarm, and the lights open when smoke is detected. When the high-temperature level is reached, it opens the water valve to operate the fire sprinklers; finally, in the event of fire control, it turns off both the alarm and the water valve. Furthermore, the application is launched when the smoke detector battery is low. If malware breaches this application, it will make the application work. However, there is no reason to operate it, and nor does it use it in the event of a possible fire that causes great danger. Therefore, the presence of an SAS is more critical for the purpose of supporting security and protecting other IoT applications from receiving any actions from a malicious third-party app.

Events generated from the sensors are forwarded to the backend using software proxies, as shown in Figure 2; these events invoke triggers defined in the rules. If the present state of the cyber-physical system meets the conditions of a given rule, the rule initiates one or more actions, which are transmitted as commands to devices [5]. Therefore, it is important to detect the malware actions from the rules in the early stages to save time in the static analysis process.

Due to the various threats which are generated from the interaction of various apps, it is essential to consider the risks created by event-condition-action automation. Therefore, the security of the IoT platform requires an understanding of the interaction between two rules of APPS [45].

These two rules may include risks as bugs and malware. These risks are classified into various classes. In [6,46], the interaction between the two rules was organized into seven classes: T1 (Action–Trigger); T2 (Action–Condition when matched); T3 (Action–Condition when not matched); T4 (Self-coordination), T5 (Action–Action (Conflict)); T6: Action–Action (repeat); and T7 (Exclusive Event).

This paper focuses on the rule of the Action conflicts using a deep-learning CNN model. This rule means that two rules conflict with each other’s actions ($a_i = -a_j$, in which a_i is the action of Condition 1, and a_j is the action of Condition 2. They operate on the same device and attribute but have different values, such as turning on or turning off at the same switch). This risk can lead to the system entering an unstable or unknown state with the same latency of rules.

In [48], the authors present a conflict-classification survey of the interaction of IoT apps using model checks and formal notations due to the risk of conflicting rule issues. The Action–Action conflict rules can be represented as if there are two rules—R1 and R2—which are triggered by a similar event to access the same device $D(\alpha)$ and an attribute $A(\alpha)$, but with conflict values $V(\alpha)$, as shown in Box 1.

Box 1. The interaction between two rules.

$$R_{AC}(R_i, R_j) \equiv (\exists A_{h1} \in A_{R_i}; A_{h2} \in A_{R_j}; R_1, R_2 \in R_{AC}. (\text{match}(A_{h1}, A_{h2}) \wedge (V(\alpha_1) \neq V(\alpha_2)) \wedge T_1 * (R_1, R_i) \wedge T_1 * (R_2, R_j) \wedge ((R_1 = R_2) \vee \text{sibling}(R_1, R_2))))).$$

Where:

- $R_{AC} \dots$ Rule of action conflicts
- $R_i \dots$ A series of rules, $i = 1$ to N
- $R_j \dots$ A series of rules, $i = 1$ to N
- $R_1 \dots$ Rule 1
- $R_2 \dots$ Rule 2
- $A_{h1} \dots$ Action of rule 1
- $A_{h2} \dots$ Action of rule 2
- $V\alpha_1 \dots$ Value of the device in the rule 1
- $V\alpha_2 \dots$ Value of the device in the rule 2
- $T_1 \dots$ Trigger 1
- $\text{sibling}(R_1, R_2) \dots$ The two rules R_1, R_2 have the same trigger (T_1) and a set of non-exclusive conditions for them.

This paper, HSAS-MD, focuses on using hybrid analysis to convert source code to an analyzable form. The CNN model extracted the malware by checking the actions in the rules before installing the app and verifying the app’s behavior using MCT.

4. The Proposed Hybrid Security Analysis System Based on the Model-Checking Technique and Deep Learning (HSAS-MD Analyzer)

This section presents the proposed hybrid security analysis system (HSAS) using model checking and deep learning (MD). An efficient SAS is one of the most critical issues related to IoT app security, serving as a security compiler. In a successful SAS, three factors must be considered: (i) Detecting malware (action conflict) at an early stage of analysis; (ii) Identifying and classifying new malware using the CNN model; (iii) Verifying the new IoT app’s behavior.

The proposed HSAS-MD consists of three phases to detect malware early, as shown in Figure 5: hybrid analysis, the deep-learning CNN model, and the model checker. Each phase has an essential role in the proposed analyzer. The main goal of the hybrid analysis phase is to convert the source code (groovy code) of IoT apps to an analyzable format before using MCT. The output of the dynamic analysis is the dynamic features and configuration extractor which will be verified directly by the final phase using the model-checking technique. However, the output of the static analysis is the rule model, which consists of triggers, conditions, and actions. These will be formatted in a vector-based format. It will then be used in the following critical phase: the deep-learning phase.

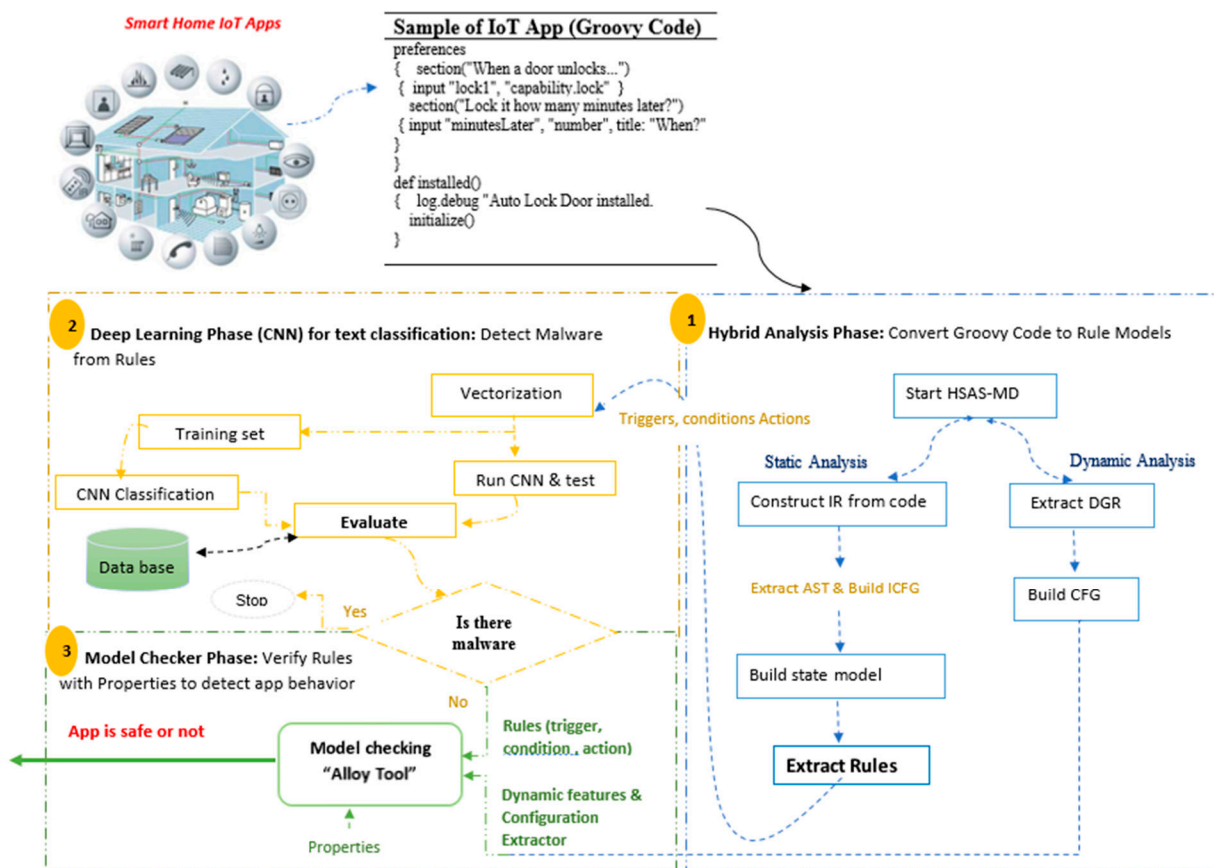


Figure 5. HSAS-MD analyzer phases.

The main architecture of the proposed HSAS-MD is composed of six phases, as shown in Figure 6. (1) Static analysis phase is responsible for extracting rules from the source code of the IoT APP; (2) Dynamic analysis phase is responsible for extracting dynamic CFG from IoT app source code; (3) Rule model phase is responsible for verifying the rules; (4) Deep-learning (DL) phase is responsible for detecting and classifying features; (5) Filtration phase is responsible for deciding the final decision of whether the feature is malware or benign; (6) Finally, performing model-checking technique for verifying whether app behavior is safe or not in each analysis phase.

This second phase uses CNN for text classification. It depends on checking the actions from the rule model extracted from the first phase before sending them to IoT devices. The CNN for text classification detects malware early, and it stops app analysis if there is malware (malicious actions). This early-stage (CNN) saves time for the SAS to analyze another IoT app. If there is no malware, the actions from the rules will transmit to the model checker's final phase.

The third phase is the model checker, which verifies the rules after also verifying dynamic features the configuration extractor with the properties of the apps from the CNN phase. The output of this phase is the decision of whether the behavior of the app is safe or not. All steps of the proposed analyzer phases can be illustrated as follows:

Step 1: Convert the app's source code to analyzable form using phase 1 of the proposed analyzer.

- Static analysis: Start converting code to IR, extract AST, and build ICFG to make the state model, then convert the state model to the rule model form.
- Dynamic analysis: Extract DGR to build CFG.

Step 2: Start phase 2 with the extracted actions from the rule model.

Step 3: Convert the rules that contain actions to be vector-based.

Step 4: Run deep-learning “CNN for text classification” with the vector-based and test actions.

Step 5: Evaluate the output of the CNN. If there is malware (malicious actions), stop the analysis process; however, if there is no malware, phase 3 will operate with these static features.

Step 6: Start phase 3 using the model-checking technique (MCT) to verify the app’s behavior. It will check the static and dynamic features with the properties that describe all specifications of this app. Therefore, the proposed HSAS-MD analyzer is concerned with detecting security problems in IoT apps and detecting and classifying malware.

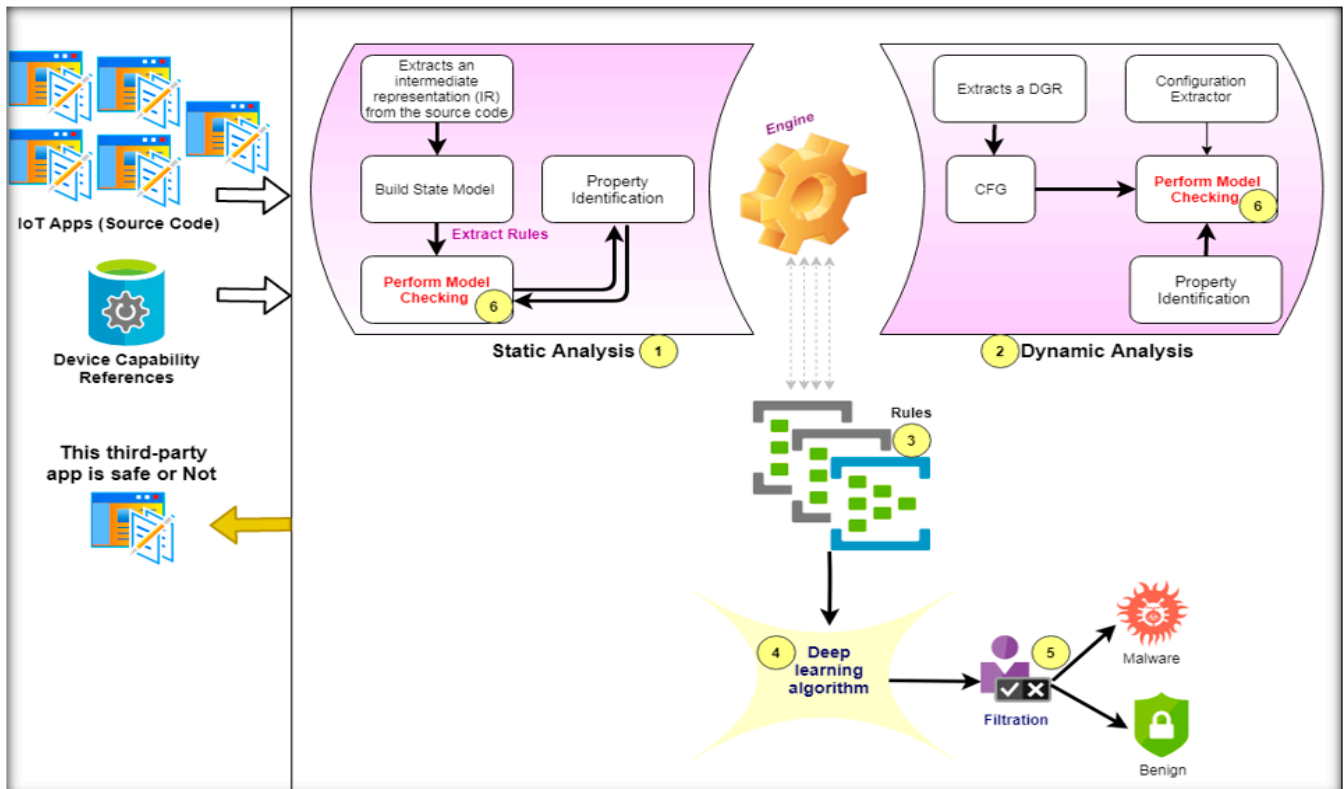


Figure 6. HSAS-MD Analyzer Architecture.

4.1. Static Analysis Phase in the Proposed HSAS-MD

The static analysis phase depends on converting the IoT apps’ codes to a rule model that includes triggers, conditions, and actions before using the model checker, as shown in Figure 7. The static analysis phase depends on enhancing deep learning to check actions before using a model checker. It is composed of four steps: Step (A) is to extract an IR from the source code of an IoT application, including the event handler methods, entry points, and call graphs; Step (B) builds a state application model that includes its states and transitions from the IR, then converts the states to the rule model; in Step (C), model checking used to check that the app’s behavior is in line with those properties developed in step (D) when running independently, or interacting with other applications. Each step are described in more detail below:

Step (A) is the intermediate representation (IR). The IR has several advantages: it allows precise modeling of the application's life cycle, removes parts of the code that are not relevant to property analysis, and finally enables the efficient extraction of states and state transitions from implementation. It consists of three mechanisms to convert the source code successfully to IR. The IR needs to access permissions, obtain events/actions, and build CFG. (i) Permissions are responsible for obtaining all information from DCR about the IoT application devices; (ii) Events/Actions represent the relation between events and actions (when an event occurs (is triggered), a related action is taken); (iii) CFC represents the relationship between entry points and their corresponding functions in the target application and facilitates analysis using path sensitivity. IR constructed via AST represents the Groovy source code of the IoT app. Analyzing the AST will build a call graph and API methods as a CFG for each local method using path-sensitive analysis [49]. Additionally, the AST visitors analyze the expressions from AST nodes that are built during the compilation process to create ICFG in order to extract the state model.

Step (B) is the state model representing the codes to a set of states from IR to detect violation from the interactions of state models between apps. Every state model will then be converted to a rule model: $R_h = \langle Th, Ch, Ah \rangle$. The conversion from a state model to a rule model is used for the first time in an SAS. The rule model is better than the state model method at detecting violations via testing actions. The state model is formulated as (Q, S, δ) , where Q is a series of states. Q represents the device's attributes that change due to the event handlers; S represents a series of transition labels representing the events; δ is a function of the state transition using the Cartesian product of the attributes of its device. The state model (Q, S, δ) is created from the physical IoT device's manufacturer's specifications $\langle \text{device, attribute, value} \rangle$. Each device has its attributes. Each device's attributes (conditions in the rule model) have specific values (actions in the rule model). Therefore, the rule model $R_h = \langle Th, Ch, Ah \rangle$ is matched with $\langle \text{device, attribute, value} \rangle$.

The advantage of the state model is the filtration of the deterministic and non-deterministic states using an SMT solver. This is essential to verify the path conditions [50]. The deterministic states represent the conditions for the safe operation of an IoT device. However, the non-deterministic states represent the safety violations. Therefore, it is an important step for using the state model. However, the state model is not effective with an increasing number of IoT devices, and malware actions were difficult to detect from the state model. In [10], two challenges to extracting states are listed: the first is state explodes, and the second is the precision of the state model is low as it depends on approximation. Therefore, the rule model is extracted from rewriting the logical equation: rl for rules, and crl for conditional rules [45].

For example, from app 1, as shown in Figure 4, when smoke is detected and the temperature is >135 , the handler method is responsible for opening the water valve. From analyzing the handler method, there is a transition with the smoke detection and temperature >135 event labels from the "smoke undetected and valve closed" state to the "smoke detected and valve opened" state. The values (actions) in this app are "valve opened" and "valve closed".

The property description needed in the static phase is in Step (D), the property identification. These properties are configured in temporal logic to be checked with the generated model of the target app using the model checker [51]. After discovering malware in actions (rule model), a group of security properties was generated to be validated using a model checker [46]. Some of these properties can be configured by the user. These properties were classified into three types: safety (S), liveness (L), and general (G), written using LTL or CTL. Examples for safety properties are as follows: S1, the water valve must be closed if a leak is detected; S2, the door must always be locked when the user is not home; S3, the alarm must always go off when there is smoke. Examples for liveness properties are as follows: L1, after 9 pm the light will eventually be turned off; L2, after 1 h the door will be locked. Examples for general properties are as follows: G1, do not turn on the living room light when no-one is home.

Finally, model checking (C) is responsible for verifying the code from the rule model with the corresponding properties, which is discussed in the next Section 6. More details for the workflow of the static phase are shown in Algorithm 1.

Algorithm 1: Static analysis Algorithm

➤ **Inputs:**

1. Source Groovy code of new IoT app.
2. **Device Capability References** that includes Set of attributes & actions for the devices used in IoT applications.

➤ **Steps:**

- 1: **For each IoT APP, Do**
- 2: HSAS-MD Converts Source code to IR.
- 3: **For each IR block, Do**
- 4: HSAS-MD extracts devices and their user inputs from the Permissions block
- 5: Defines all the devices included in the application
- 6: Defines all the user inputs
- 7: **input** (*device name, variable name, type: device*)
- 8: HSAS-MD describes how the application subscribes to events from the events/actions block
- 9: Define all events and actions for devices
- 10: **Subscribe** (*device handle, a device event, an event handler method*)
- 11: Define all entry points that describe an event handler method
- 12: HSAS-MD builds CFG for each entry point to create ICFG
- 13: **Next**
- 14: HSAS-MD builds state model
- 15: HSAS-MD converts state model to rule model
- 16: HSAS-MD verifies actions from rule model using CNN model
- 17: HSAS-MD checks the property description
- 18: HSAS-MD uses the Model checking technique
- 19: **Next**

➤ **Output:** New IoT app behavioural is safe or not

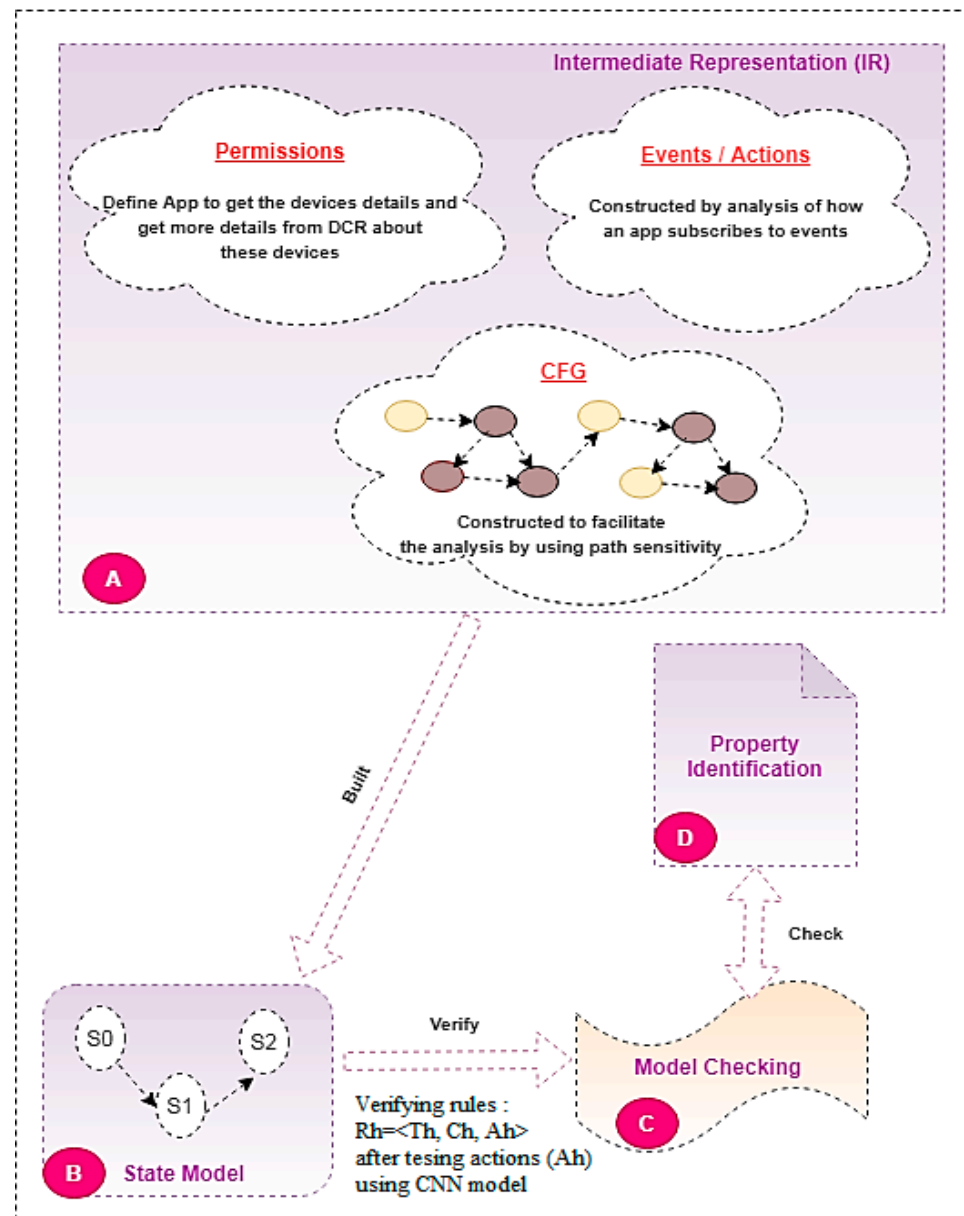


Figure 7. The static analysis phase in the HSAS-MD.

4.2. Dynamic Analysis Phase in the Proposed HSAS-MD

The dynamic analysis phase focuses on dynamic features. It differs from the static phase, as dynamic analysis depends on understanding how the code performs during the execution process. This phase contains four steps, as shown in Figure 8.

More details for the workflow of the static phase are shown in Algorithm 2.

Step (A) is DGR. It collects all device attributes and actions from DCR and extracts the dynamic features. It then converts the dynamic features to a set of dependency graphs representing the interactions between event handlers of the app and reducing all unnecessary event handlers. Step (B) is the state model, an intermediate module that converts the output of dependency graphs to the representation of CFG. Step (C) is the configuration extractor, responsible for extracting all configurations related to the application and the connected devices. A set of property descriptions is stored in step (D), referred to as property identification. These properties are the same as in the static phase. In step (E), the generator is responsible for matching the configuration and its related properties, as in the static phase. Finally, Step (F), model checking, is responsible for verifying the code to ensure it does not contain any abnormal behavior.

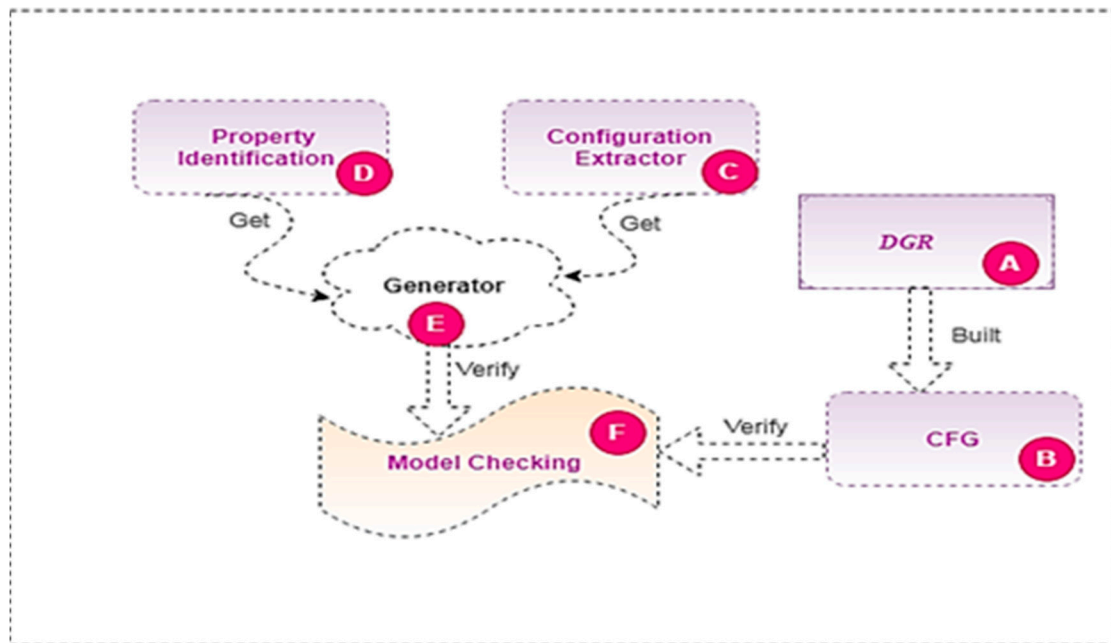


Figure 8. The dynamic analysis phase in the HSAS-MD.

Algorithm 2: Dynamic analysis Algorithm

➤ **Inputs:**

1. Source Groovy code of new IoT app.
2. *Device Capability References* that include Set of attributes & actions for the devices used in IoT applications.

➤ **Steps:**

- 1: **For each IoT application, Do**
- 2: HSAS-MD Converts Source code to DGR.
- 3: DGR gets the Device's information from DCR.
- 4: Define all events and actions for devices
- 5: **Subscribe** (*device handle, a device event, an event handler method*)
- 6: HSAS-MD creates CFG for each entry point from the dependency graph
- 7: HSAS-MD gets the configuration of the interactions between devices and IoT application
- 8: HSAS-MD checks the property description
- 9: HSAS-MD uses the Model checking technique
- 10:
- 11: **Next**

➤ **Output:** New IoT app behavioural is safe or not

4.3. Rule-Model Phase in the Proposed HSAS-MD

The third phase in the HSAS-MD is the rule-model phase. The smart home model includes a set of IoT apps. Each consists of a group of rules; these rules describe the

function of the IoT app and the related devices if they communicate physically with each other. These devices include sensors, switches, thermostats, etc. However, if the IoT app communicates virtually through the cloud, the direct mapping used to trigger the virtual device or scheduling method is used by utilizing rule calls to invoke the second rule after a delay run the API from SmartThings platform apps.

The HSAS-MD extracts the rule model using the model builder from the static phase. It generates a formal rule-model representation, as shown in Table 2, including six slots which describe each rule. The trigger action rules play an essential role in discovering the bugs through the interactions between the rules.

Table 2. Rule representation model.

No.	Slot Names	Description
1	Rule (Rh)	(trigger) (condition) (action)
2	Trigger (Th)	(capabilities) (attribute) (value)
3	Condition (Ch)	(capabilities) (attribute) (value)
4	Action (Ah)	(capabilities) (attribute) (value)
5	Event	(subject) (attribute)
6	Constraint	logical expression null

Each rule includes three main elements: the trigger, condition, and action. All of them are described as follows: capabilities, attributes, and value, as shown in Table 2. The trigger starts when an event happens. The event represents the temperature, motion, state change, etc. The value of the trigger is a Boolean constraint, e.g., active, true. Some rules do not include a value in the trigger, such as AutolockafterXminutes. As shown in Table 3, the condition is essential in the rule.

The condition is represented as a Boolean such as true, value on. The condition is a Boolean constraint representing the states. When all states in the conditions are satisfied, the rule can be executed. Regarding actions, there are three types. The first type of action is one where the attributes do not change, such as sending a mobile notification. The second type of action depends on time. The third type of action is affected by actuators. This type changes the value of the attribute, such as turning on or locking something.

The HSAS-MD focuses on the third type of action; the value of attributes may include a conflicting value. This value may be changed when software is updated or by the end user to give a contradictory value of the desired actuators. These action-conflicts lead to serious situations in IoT apps.

To further illustrate the action conflicts, the AutolockafterXminutes app includes two rules (R0 and R1).

The first rule (R0), assigned to lock1.capabilities and attributes are found with no value.

The condition value in this app is to cap_lock_attr_lock_val_locked. If the condition of lock1 is already completed, the action value of the state will be cap_runIn_attr_runIn_val_on.

The second rule (R1) is assigned to the state, starting with the condition of the cap_runIn_attr_runIn_val_on. This condition depends on the same action of R0. The action will then be cap_lock_attr_lock_val_lock. The value of the action plays an essential role in making the app do its function or make unwanted behavioral. This phase therefore has a vital role in representing the rules $R_h = \langle Th, Ch, Ah \rangle$ as features using vector-based.

As shown in Table 3, this table is a dataset of rule-based IoT apps that includes ten applications. Each application includes its capabilities and rules. The capabilities contain the sensors and actuators.

Each application may include one rule or more rules depending on the task of the application. Each rule consists of a vector of triggers, conditions, and actions. The trigger has three elements: the capability object's name, attribute, and value. Additionally, conditions and actions have the same elements.

Table 3. Rules description of the IoT apps dataset in a smart home.

Application Name	Capabilities	Rule No.	Triggers	Conditions	Actions
AutoLockafterXminutes	lock1 + state	R0	capabilities = app_AutoLockafterXminutes.lock1 attribute = cap_lock_attr_lock no value	capabilities = app_AutoLockafterXminutes.lock1 attribute = cap_lock_attr_lock value = cap_lock_attr_lock_val_locked	attribute = app_AutoLockafterXminutes.state attribute = cap_runIn_attr_runIn value = cap_runIn_attr_runIn_val_on
		R1	no triggers	capabilities = app_AutoLockafterXminutes.state attribute = cap_state_attr_runIn value = cap_state_attr_runIn_val_on	attribute = app_AutoLockafterXminutes.lock1 attribute = cap_lock_attr_lock value = cap_lock_attr_lock_val_lock
AutoLockDoorsv2	lock1 + state	R0	capabilities = app_AutoLockDoorsv2.lock1 attribute = cap_lock_attr_lock no value	capabilities = app_AutoLockDoorsv2.lock1 attribute = cap_lock_attr_lock value = cap_lock_attr_lock_val_locked	attribute = app_AutoLockDoorsv2.state attribute = cap_runIn_attr_runIn value = cap_runIn_attr_runIn_val_on
		R1	no triggers	capabilities = app_AutoLockDoorsv2.state attribute = cap_runIn_attr_runIn value = cap_runIn_attr_runIn_val_on	attribute = app_AutoLockDoorsv2.lock1 attribute = cap_lock_attr_lock value = cap_lock_attr_lock_val_locked
DoorAutoLock	lock1 + state	R0	capabilities = app_DoorAutoLock.lock1 attribute = cap_lock_attr_lock no value	capabilities = app_DoorAutoLock.lock1 attribute = cap_lock_attr_lock value = cap_lock_attr_lock_val_unlocked	attribute = app_DoorAutoLock.state attribute = cap_runIn_attr_runIn value = cap_runIn_attr_runIn_val_on
		R1	no triggers	capabilities = app_DoorAutoLock.state attribute = cap_runIn_attr_runIn value = cap_runIn_attr_runIn_val_on	attribute = app_DoorAutoLock.lock1 attribute = cap_lock_attr_lock value = cap_lock_attr_lock_val_locked
DoorsUnlocked	presence1 + lock1	R0	capabilities = app_DoorsUnlocked.presence1 attribute = cap_presenceSensor_attr_presence no value	capabilities = app_DoorsUnlocked.presence1 attribute = cap_presenceSensor_attr_presence value = cap_presenceSensor_attr_presence_val_null	attribute = app_DoorsUnlocked.lock1 attribute = cap_lock_attr_lock value = cap_lock_attr_lock_val_lock
IfFloodTurnValveOff	alarm + valve	R0	capabilities = app>IfloodTurnValveOff.alarm attribute = cap_waterSensor_attr_water value = cap_waterSensor_attr_water_val_wet	capabilities = app>IfloodTurnValveOff.valve attribute = cap_valve_attr_any value = cap_valve_attr_any_val_no_value	attribute = app>IfloodTurnValveOff.valve attribute = cap_valve_attr_valve value = cap_valve_attr_valve_val_closed
ItsTooCold	temperatureSensor1 + switch1	R0	capabilities = app_ItsTooCold.temperatureSensor1 attribute = cap_temperatureMeasurement_attr_temperature no value	capabilities = app_ItsTooCold.temperatureSensor1 attribute = cap_temperatureMeasurement_attr_temperature value = range_0//cap_temperatureMeasurement_attr_temperature_val_lte_tooCold	attribute = app_ItsTooCold.switch1 attribute = cap_switch_attr_switch value = cap_switch_attr_switch_val_on
LightOnCold	temperatureSensor1+ switch1	R0	capabilities = app_LightOnCold.temperatureSensor1 attribute = cap_temperatureMeasurement_attr_temperature no value	capabilities = app_LightOnCold.temperatureSensor1 attribute = cap_temperatureMeasurement_attr_temperature Value = cap_temperatureMeasurement_attr_temperature_val_lte_tooCold	attribute = app_LightOnCold.switch1 attribute = cap_switch_attr_switch value = cap_switch_attr_switch_val_on

Table 3. Cont.

Application Name	Capabilities	Rule No.	Triggers	Conditions	Actions
Motion TriggersLock	motion1 + lock1	R0	capabilities = app_MotionTriggersLock.motion1 attribute = cap_motionSensor_attr_motion value = cap_motionSensor_attr_motion_val_active	capabilities = app_MotionTriggersLock.lock1 attribute = cap_lock_attr_lock value = cap_lock_attr_lock_val_locked	attribute = app_MotionTriggersLock.lock1 attribute = cap_lock_attr_lock value = cap_lock_attr_lock_val_unlocked
DelayedCommandExecution	contact1 + switch1	R0	capabilities = app_DelayedCommandExecution.contact1 attribute = cap_contactSensor_attr_contact no value	capabilities = app_DelayedCommandExecution.contact1 attribute = cap_contactSensor_attr_contact value = cap_contactSensor_attr_contact_val_open	attribute = app_DelayedCommandExecution.switch1 attribute = cap_switch_attr_switch value = cap_switch_attr_switch_val_on
		R1	capabilities = app_DelayedCommandExecution.contact1 attribute = cap_contactSensor_attr_contact no value	capabilities = app_DelayedCommandExecution.contact1 attribute = cap_contactSensor_attr_contact value = cap_contactSensor_attr_contact_val_open - cap_contactSensor_attr_contact_val_open capabilities = app_DelayedCommandExecution.contact1 attribute = cap_contactSensor_attr_contact value = cap_contactSensor_attr_contact_val_closed	attribute = app_DelayedCommandExecution.switch1 attribute = cap_switch_attr_switch value = cap_switch_attr_switch_val_off
UnlockitWhenitOpens	contact1 + Lock1	R0	capabilities = app_UnlockitWhenitOpens.contact1 attribute = cap_contactSensor_attr_contact no value	capabilities = app_UnlockitWhenitOpens.contact1 attribute = cap_contactSensor_attr_contact value = cap_contactSensor_attr_contact_val_open	attribute = app_UnlockitWhenitOpens.lock1 attribute = cap_lock_attr_lock value = cap_lock_attr_lock_val_unlock

The HSAS-MD focuses on the value of actions. However, it uses the word2vec model before using CNN to convert the Th, Ch, and Ah from words to vectors. The rule model, including its elements (capabilities, attributes, and values), will then be fed as an input to the DL phase (CNN) as features. From the training dataset of the IoT apps, when the condition elements (capabilities, attributes, and values) are matched, the value of actions can be detected using CNN. The early detection of whether the app has malware or not saves time of analysis.

4.4. Deep-Learning (DL) Phase in the Proposed HSAS-MD

The proposed HSAS-MD analyzer uses the Convolutional Neural Network (CNN). CNN is one of the DL algorithms that can typically classify text [52–55] by taking a feature vector, assigning significance (trainable weights and biases) to various aspects in the input feature, and distinguishing one from the other. The preprocessing time needed for CNN is lower than for other different classifiers. CNN is capable of understanding these filters/characteristics.

Features are extracted from a feature dataset using trained TextCNN models. TextCNN layers are used to learn distinctive malware representations to enhance the efficiency of the proposed HSAS-MD analyzer. Therefore, CNN's architecture is designed to extract the features from the dataset by using a five-layered CNN model [56], as shown in Figure 9. The five layers are:

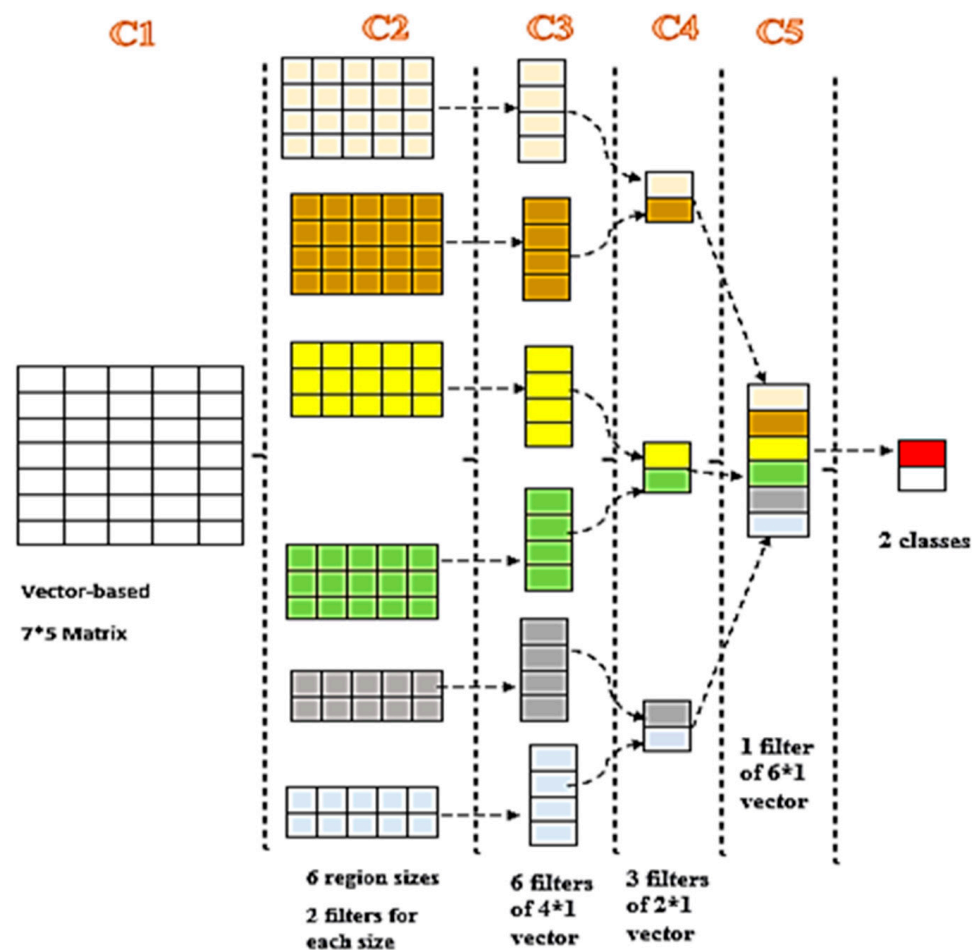


Figure 9. The DL phase in the HSAS-MD.

The input layer (C1) is the input feature in the vector-based 7×5 matrix. This layer begins with triggers, conditions, and actions, which are then converted into a sentencing matrix; the rows are word-vector representations of each. The sentence length is s and the dimension of the word vector by d ; the dimension of the sentence matrix is $s \times d^2$.

The first hidden layer (C2) is the layer that filters features into six filters. There are three filter region sizes: 2, 3, and 4, each with two filters. The weight matrix w makes the filter with region size z . Then, $z \cdot d$ parameters were obtained. The sentence matrix by $k \in \mathbb{R}^{s \times d}$, $k [m: n]$ represents the sub-matrix of k from row m to row n . The output sequence $X \in \mathbb{R}^{s-z+1}$ of the convolution operator is obtained by repeatedly applying the filter on sub-matrices of k .

$$X_i = w \cdot k [m: m + z - 1] \quad (1)$$

where $m = 1$. $s - z + 1$, then adds a bias term $b \in \mathbb{R}$ and an activation function f to each X_i , inducing the $c \in \mathbb{R}^{s-z+1}$ for this filter:

$$c_i = f(X_i + b) \quad (2)$$

The second hidden layer (C3) aims to resample and transform the second layer result. Thus, for each feature map in C2, six filters of 4×1 vectors are obtained. This feature map was obtained after applying many filters for the same region size z .

In the third hidden layer (C4), the six filters in $c3$ are merged, and three filters of 2×1 are obtained. Each feature map is applied to 1-max pooling, which means that the most significant number from each feature map is collected. Thus, a simple feature vector is formed from all six maps, and these six features are combined to build a feature vector for the final layer.

The output layer (C5): this is a final layer of CNN, which receives a C4 feature vector to classify the sentences. It will classify them into two classes.

4.5. Filtration Phase in the Proposed HSAS-MD

This critical phase is used to improve CNN's training and enhance the accurate decision for the output. "Binary cross-entropy" function used as the output of CNN consists of two classes (the class with the number 1 indicates the malware class, and the class with the number 0 shows the benign class). The following Equation (1) is used in binary cross-entropy. B is the predicted probability, and A is the binary indicator (0 or 1). The output of this phase is the final decision if there is malware or not.

$$BCE = -(A \log(B) + (1 - A) \log(1 - B)) \quad (3)$$

4.6. Model-Checking Technique in the Proposed HSAS-MD

The model-checking technique (MCT) is the core analysis process in the proposed HSAS-MD analyzer for verifying the behavior of IoT apps. MCT has systematically verified that all states of the given system meet the given properties to verify the app's behavior. The proposed HSAS-MD verifies the rule model in the static analysis phase as the same analyzers [6,45,46] and verifies CFG in the dynamic analysis phase. For the static analysis phase, which depends on the rule model, MCT depends on rewriting logic (the logic that happens due to the changes of the state and concurrent computations). These computations are formulated from logic (L) to rule model (R), meaning that every state that matches the L will be converted to R [57,58]. In [59], the security rules to LTL were formulated, which were verified using MCT. For the dynamic phase, MCT verifies CFG dynamically using DynAUoy. It enhances the Alloy specification system, allowing models' dynamic properties to be specified and analyzed [51].

5. Implementation, Evaluation, and Discussion of HSAS-MD

This section presents the implementation, evaluation, and discussion of the proposed HSAS-MD analyzer using different evaluation metrics. The central core of evaluating the proposed HSAS-MD is the ability to (i) extract static and dynamic features from the source code of IoT applications; (ii) detect malware from features' datasets; (iii) detect APP behavior using MCT.

5.1. Description of Tools Used

Experiments of program analysis (IoT APPs) were performed with Eclipse IDE and IntelliJ IDEA 2021.1.1 ×64 on a 2.6 GHz Intel i5-4300M processor and 8 GB RAM laptop, using Oracle's Java Runtime version 1.8 (64 bit) in its default settings. The Alloy tool used for the model-checking test applied the Graphviz tool for state-model representation. The output of the hybrid analysis will be the input of the CNN model. This CNN model used PyTorch, based on torch library runs with Google Colaboratory.

5.2. Implementation of the Proposed HSAS-MD

The proposed HSAS-MD depends on reusing a hybrid analysis analyzer that depends on the model-checker tool, then filtering its output features using the CNN model. This method is the fastest, most accurate, and most applicable to be used. The proposed HSAS-MD implementation reused IoTCOM [6] and then filtered its features using the CNN model. There are two stages to verify the proposed HAS-MD analyzer before using the model-checker tool.

5.2.1. Extracting Rule Model from Static Analysis

This stage depends on reusing the analyzer "IoTCOM" [6]. It depends on using the potential of our formal abstractions by creating a modular model extractor. This depends on Alloy, which is a model-checker tool for representing the standard specifications of IoT apps, such as reflecting the behavior of IoT apps to safely detect security risks and respect cyber and physical channels by using IoT apps' PA code. It consists of two phases: a behavioral rule extractor and a formal analyzer. The output of the static phase "rule extractor," is reused in HSAS-MD in the CNN model, and the dynamic analysis output is verified directly to the final phase of MCT.

5.2.2. Testing the Deep-Learning "CNN Model"

The CNN model's task is to filter fake actions (malware) in the proposed HSAS-MD. It takes the static analysis output: triggers, conditions, and actions to be trained and tested. As shown in Table 3, each application has its own rules. The CNN model deals with triggers, conditions, and actions as sentences. These sentences are represented in a word-embedding vector [56]. The testing phase focuses on the action vector, if the trigger vector and condition vector match the same previously restored trigger and condition vector to the same rule in the considered application. The action must match the new action. If the two actions are matched, it is a benign action, and it will be moved to the next phase (MCT). Otherwise, it will be malware.

Figure 10 shows the following test result; when the input action from the considered IoT application "Door Auto-Lock" to CNN is matched, the output of CNN will detect that this application "Door Auto-Lock" is benign. However, Figure 11 shows that the test result is malware. When the input action from this application "Door Unlock X minutes" to the CNN does not match the stored action, it will stop this application from continuing and transferring to the next verifying phase, MCT.


```

# Test
test_text = 'Door Auto Lock '
tests = [np.asarray([word_dict[n] for n in test_text.split()])]
test_batch = Variable(torch.LongTensor(tests))

# Predict
predict = model(test_batch).data.max(1, keepdim=True)[1]
if predict[0][0] == 0:
    print(test_text,"is Malware ...")
else:
    print(test_text,"is Bengin!!")

Epoch: 1000 cost = 0.646730
Epoch: 2000 cost = 0.196541
Epoch: 3000 cost = 0.135146
Epoch: 4000 cost = 0.117257
Epoch: 5000 cost = 0.036940
Door Auto Lock is Bengin!!

```

Figure 10. The test of the CNN phase at the HSAS-MD analyzer.

```

# Test
test_text = 'Door Unlock Xminutes '
tests = [np.asarray([word_dict[n] for n in test_text.split()])]
test_batch = Variable(torch.LongTensor(tests))

# Predict
predict = model(test_batch).data.max(1, keepdim=True)[1]
if predict[0][0] == 0:
    print(test_text,"is Malware ...")
else:
    print(test_text,"is Bengin!!")

Epoch: 1000 cost = 0.742987
Epoch: 2000 cost = 0.436481
Epoch: 3000 cost = 0.307618
Epoch: 4000 cost = 0.169658
Epoch: 5000 cost = 0.108155
Door Unlock Xminutes is Malware ...

```

Figure 11. The second test of the CNN phase at the HSAS-MD analyzer.

5.3. Evaluation Metrics

The performance of the proposed analyzer utilizes the confusion matrix to evaluate the output of the HSAS-MD. The confusion matrix can be made using defined indicators, as shown in Table 4, which provide a helpful way to measure the proposed analyzer's performance against other SAS analyzers. In this paper, the analyzers were evaluated based on commonly used accuracy, precision, recall, and F1 score, as in Equations (4)–(7), respectively.

Table 4. Description of the Confusion Matrix.

Indicator	Description
True positive (TP)	The number of malware samples was detected correctly and labeled as malware.
True negative (TN)	The number of benign samples was correctly detected and labeled as benign.
False positive (FP)	The number of benign samples was wrong and labeled as malicious.
False negative (FN)	The number of malware samples was wrong and labeled as benign.

Based on the above indicators, the following performance analysis equations were used to obtain the confusion matrix as shown in Figure 12:

- Accuracy (ACC) is the ratio representing the number of correctly identified analyzers.

		Truth data			
		Class 1	Class 2	Classification overall	User's accuracy (Precision)
Classifier results	Class 1	80	5	85	94.118%
	Class 2	4	100	104	96.154%
	Truth overall	84	105	189	
	Producer's accuracy (Recall)	95.238%	95.238%		

Figure 12. Confusion matrix of the proposed HSAS-MD.

$$ACC = (TP + TN)/(TP + TN + FP + FN) \quad (4)$$

- Precision (PRC) is the number of accurately predicted true positives.

$$PRC = TP/(TP + FP) \quad (5)$$

- Recall (RCL) is the ratio of correctly identified malware relative to the total quantity of malware.

$$RCL = TP/(TP + FN) \quad (6)$$

- F-Measure (F-MS) is the ratio that reflects a mixture of accuracy and the recall efficiency of the system.

$$F-MS = 2 \times (\text{Precision} \times \text{Recall})/(\text{Precision} + \text{Recall}) \quad (7)$$

5.4. Evaluation of the Proposed Analyzer

To ensure the efficiency of the HSAS-MD analyzer, the evaluation is based on a comparison with competitors' analyzers. Competitors have been carefully selected to have the same goal to detect malware or to detect abnormal behavior, such as violations due to communication/device failures, physical safety violations, accounts for app interactions, etc., in the source code of the IoT app.

The comparison was established using the same target dataset, "IoTMAL" [60]. This dataset creates erroneous SmartApps with property violations in multi-app or single-app environments. Moreover, more public repository "SmartThings Apps" are used to

detect malware. The HSAS-MD used the same test cases of violation detection properties [4,6,10,46] to evaluate the performance, using Table 4.

The following research questions have been created in the evaluation phase of HSAS-MD with related SAS analyzers using Table 5. SAS analyzers were selected based on MCT. Some only use MCT [4,6,10,43,44], while [45,46] use other techniques with MCT.

Table 5. A comparison between various SASs based on MCT.

Paper	SAS Analyzers	Analysis Technique	Analysis Type	Static Analysis Model
[10]	Soteria	MCT	Static	State
[4]	IoTSan	MCT	Static	State
[43]	ForeSee	MCT	Static	State
[6]	IoTCOM	MCT	Static	Rule
[44]	SIFT	MCT	Static	Rule
[45]	iRuler	MCT & NLP	Static	Rule
[46]	TAPInspector	MCT & Slicing	Static	Rule
This Paper	HSAS-MD	MCT & CNN	Static & Dynamic	State & Rule

The proposed HSAS-MD uses two techniques, which are MCT and DL. The following figures [9–11,15–18] are configured to represent the results of the research questions.

RQ1. What is the accuracy, precision, recall, and F-measure of the proposed HSAS-MD compared to other analyzers?

The HSAS-MD gives the best results in accuracy (ACC), precision (PRC), recall (RCL), and F-measure (F-MS), as it depends on using hybrid analysis, not relying only on static or dynamic analysis. This impact plays an essential role in covering all vulnerabilities, as shown in Figure 13. Another impact that makes the proposed HSAS-MD the best is that the state model and rule model are used together in the static phase. It is noted that the analyzers which relied on rule models such as TAPInspector, SIFT, iRuler, and IoTCOM gave the best result than others, such as Soteria, IoTSan, and ForeSee, that depend on the state model.

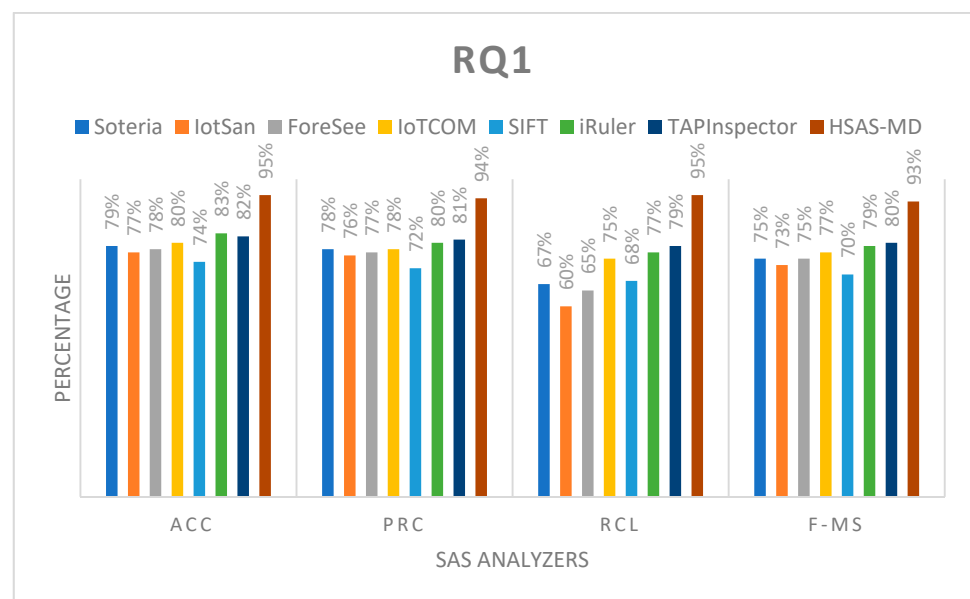


Figure 13. The comparison of the proposed HSAS-MD analyzer with other related analyzers based on ACC, PRC, RCL, and F-MS.

RQ2. What is the performance, according to the analysis time, to detect abnormal behavior?

The time of analysis is the essential impact to indicate the performance of analyzers. As shown in Figure 14, The proposed HSAS-MD gives the best performance as it has a fast analysis time for the early detection of malware. The deep-learning CNN model has an essential role in detecting malware early.

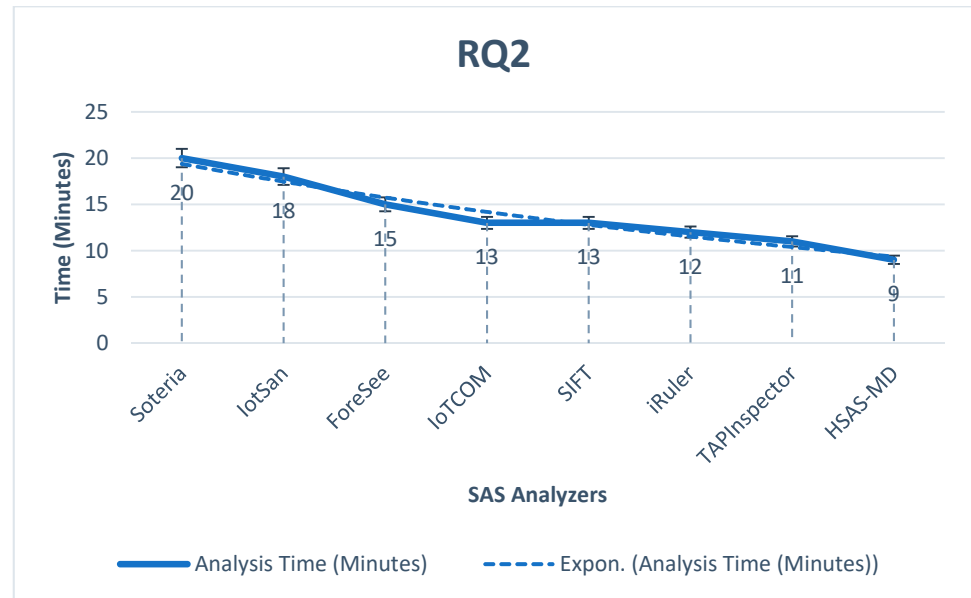


Figure 14. The analysis time of the proposed HSAS-MD analyzer with other related analyzers.

RQ3. What is the performance according to analysis type (static, dynamic and hybrid) in the SAS?

The hybrid analysis achieves the best result, as it combines static and dynamic analysis. As shown in Figure 15, the proposed HSAS-MD gives the best results. In contrast, the other analyzers depend on static analysis only.

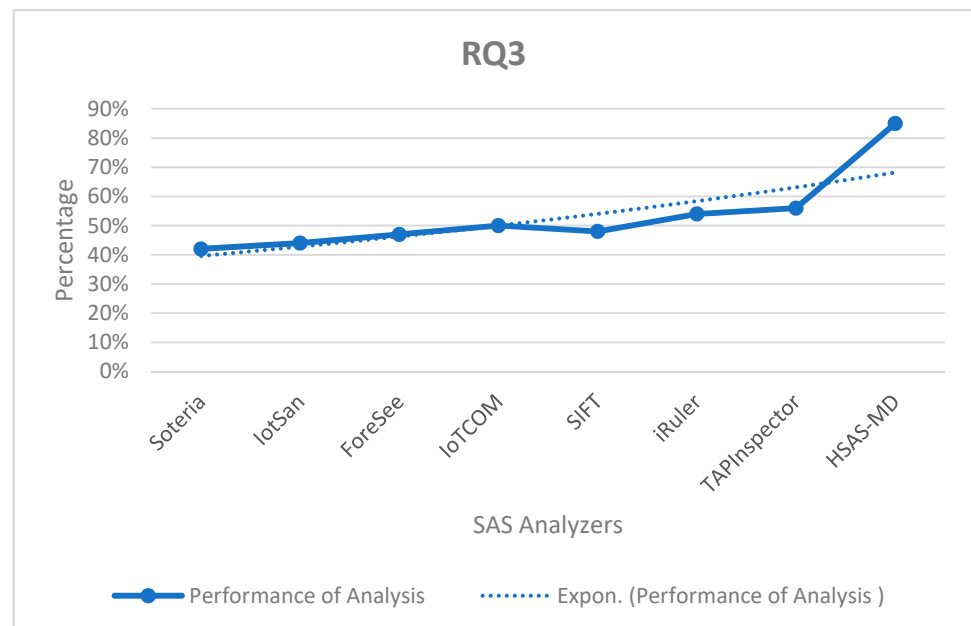


Figure 15. The performance of the proposed HSAS-MD analyzer with other related analyzers based on the analysis type.

RQ4. What is the best the static analysis phase to be used (state model, rule model, and hybrid model that combines state and rule)?

There are three SASs based on the state model: Soteria, IotSan, and ForeSee. Additionally, three SASs are based on the rule model: IoTCOM, SIFT, and iRuler, compared to the proposed HSAS-MD which is based on both the rule and state models. The performance of analyzers increases more using the rule model than the state model. Figure 16 shows that the performance of three SASs based on the rule model is better than the other three SASs based on the state model; the proposed HSAS-MD gives the best performance as it is a hybrid using the two models.

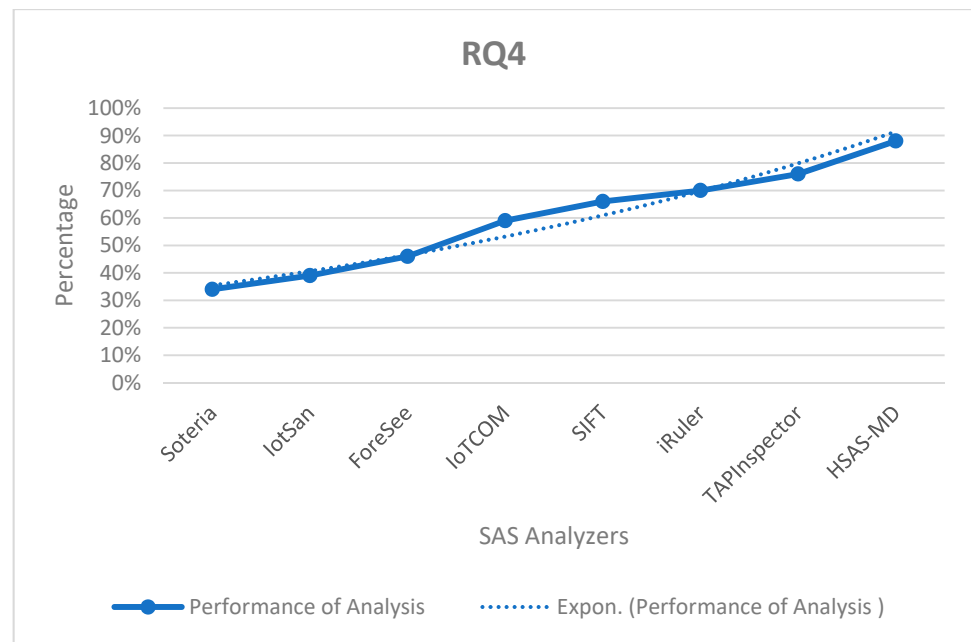


Figure 16. The performance of the proposed HSAS-MD analyzer with other related analyzers based on the static analysis model.

RQ5. What is the verification time of CNN for detecting conflict rules?

It is essential to measure the efficiency of the CNN model. According to rules training and testing in the proposed HSAS-MD, measuring the verification time of CNN is essential. Figure 17 shows that the rate of verification time is suitable to the number of rules used. The verification time increases with the increase in rules at the appropriate time rate. Various CNN models such as RNN [61], TCN [62], RNN with an attention layer [63], and CNN-1D [34] are compared with the proposed CNN to measure the verification time of rules as shown in Figure 18. The proposed HSAS-MD give the best ACC, PRC, RCL, and F-MS to verify and classify rules that may have a fake condition (malware).

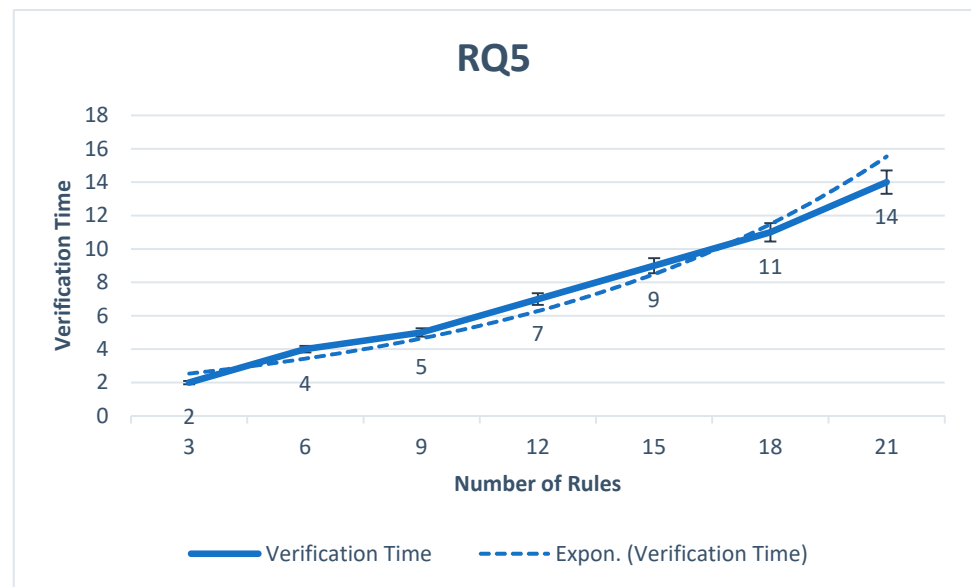


Figure 17. The verification time of the CNN model to verify rules.

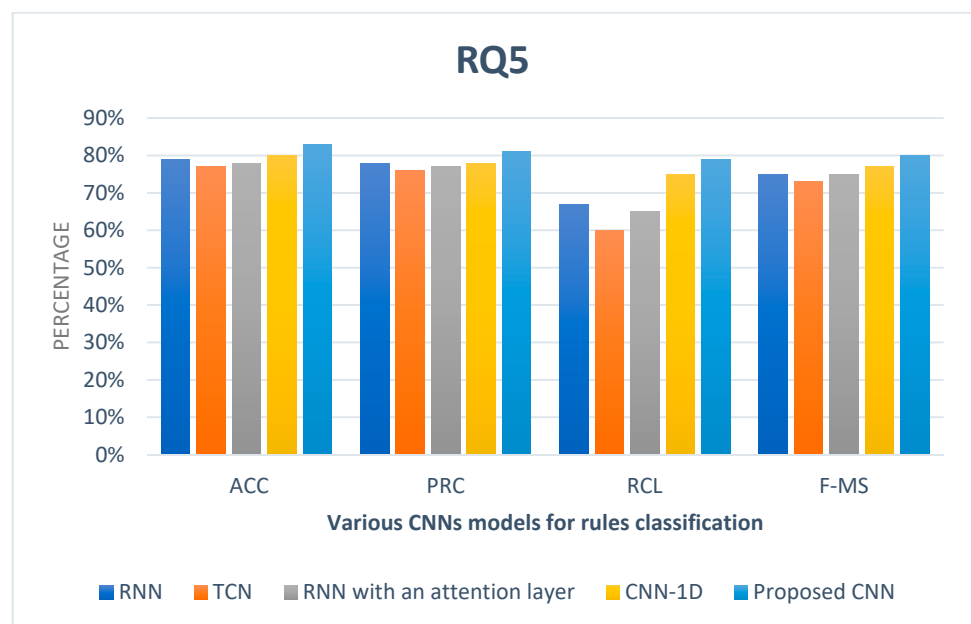


Figure 18. The comparison between various CNN models for rule classification.

6. Conclusions, Limitations, and Future Research Trends

The competition between security experts and developers has been a never-ending battle against malware complexities, improving rapidly as innovation advances. It is, therefore, necessary to move towards the development of SASs to ensure the security and safety of IoT applications. This paper presented a hybrid security analysis system (HSAS) based on MCT and DL for IoT applications. The main advantages of the proposed HSAS-MD analyzer are the ability to (i) convert the source code of the target applications to a format of a model checker that can deal with it; (ii) detect any abnormal behavior in the IoT application; (iii) extract the main static features from it to be tested and classified using a deep-learning algorithm; (iv) decide whether the app behavior is safe or not.

It is evident from the experimental results of HSAS-MD that the applied hybrid analysis is better than the static or dynamic analysis, as it provides reliable results and covers all the advantages of static and dynamic analysis. Moreover, the results show that MCT is a more accurate professional analysis technique, particularly with the applied path

of sensitivity compared with similar SASs that use various PA techniques. HSAS-MD gives the best results in the detection of physical vulnerabilities and malware.

A conclusion was made about the effectiveness of using hybrid analysis for malware detection, as it covers all issues for detecting malware. It is better than using one type of analysis (static or dynamic analysis). MCT is an efficient technique for detecting malware in applications. In addition, using CNN with MCT give the best results.

There is a limitation in HSAS-MD, which is that malware detection depends on the spatial context, which does not focus on IoT devices' locations. The physical channels or connections are affected by the location of IoT devices. The second limitation is that HSAS-MD focuses on one type of action conflict, which depends on the value of the actuator. There are two other types of action which rely on time or static attributes, such as sending notifications.

There are more points in future research trends. The first main point addresses the mentioned limitations by focusing on the IoT devices' locations. Additionally, all types of action conflict will be resolved. The second is enhancing the "program-slicing technique", which relies on slicing one program into a multi-statement. Using the program-slicing technique with MCT and CNN will provide more accuracy and time. The third point is testing the proposed HSAS-MD with different datasets to increase the training of the detection of new IoT malware. The fourth is that the proposed HSAS-MD will be applied with Android apps for the IoT platform by modifying the specifications that match these Android apps.

Author Contributions: Conceptualization; methodology; software; validation; formal analysis; investigation; resources; visualization, A.A.H., and A.M.B.-E.; data curation, A.M.B.-E.; writing—original draft preparation, A.A.H.; writing—review and editing, A.A.H.; supervision, I.T.A.H., M.A.S. and A.M.B.-E. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data supporting reported results can be found by contacting authors.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

Symbol	Description
APP	Application
AST	Abstract Syntax Tree
BCE	Binary Cross-Entropy
CFG	Control Flow Graph
CNN	Convolutional Neural Networks
DCR	Device Capability Reference
DGR	Dependency Graph Representation
DL	Deep Learning
ICFG	Inter-procedural control flow graph
IR	Intermediate Representation
MCT	Model-Checking Technique
PA	Program Analysis
SAS	Security Analysis Systems
LTL	Linear Temporal Logic
CTL	Computing Tree Logic
SMT	Satisfiability Modulo Theories

References

1. Al-Garadi, M.A.; Mohamed, A.; Al-Ali, A.K.; Du, X.; Ali, I.; Guizani, M. A survey of machine and deep learning methods for internet of things (IoT) security. *IEEE Commun. Surv. Tutor.* **2020**, *22*, 1646–1685. [CrossRef]
2. Atlam, H.F.; Wills, G.B. IoT security, privacy, safety and ethics. In *Digital Twin Technologies and Smart Cities 2020*; Springer: Cham, Switzerland, 2020; pp. 123–149. [CrossRef]
3. Sengupta, J.; Ruj, S.; Bit, S.D. A comprehensive survey on attacks, security issues and blockchain solutions for IoT and IIoT. *J. Netw. Comput. Appl.* **2020**, *149*, 102481. [CrossRef]
4. Nguyen, D.T.; Song, C.; Qian, Z.; Krishnamurthy, S.V.; Colbert, E.J.; McDaniel, P. IotSan: Fortifying the Safety of IoT Systems. In Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies 2018, Heraklion, Greece, 4–7 December 2018; pp. 191–203. [CrossRef]
5. Celik, Z.B.; Fernandes, E.; Pauley, E.; Tan, G.; McDaniel, P. Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–30. [CrossRef]
6. Alhanahnah, M.; Stevens, C.; Bagheri, H. Scalable Analysis of Interaction Threats in IoT Systems. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, 18–22 July 2020; pp. 272–285. [CrossRef]
7. Wang, T.; Bhuiyan, M.Z.A.; Wang, G.; Qi, L.; Wu, J.; Hayajneh, T. Preserving balance between privacy and data integrity in edge-assisted Internet of Things. *IEEE Internet Things J.* **2019**, *7*, 2679–2689. [CrossRef]
8. Morgan, S. 2019 Cybersecurity almanac: 100 facts, figures, predictions and statistics. *Cybercrime Magazine*, 6 February 2019.
9. Nobakht, M.; Sui, Y.; Seneviratne, A.; Hu, W. PGFit: Static permission analysis of health and fitness apps in IoT programming frameworks. *J. Netw. Comput. Appl.* **2020**, *152*, 102509. [CrossRef]
10. Celik, Z.B.; McDaniel, P.; Tan, G. Soteria: Automated Iot Safety and Security Analysis. In Proceedings of the 2018 {USENIX} Annual Technical Conference, ({USENIX}{ATC} 18), Boston, MA, USA, 11–13 July 2018; pp. 147–158.
11. Wang, Q.; Hassan, W.U.; Bates, A.; Gunter, C. Fear and Logging in the Internet of Things. In Proceedings of the Network and Distributed Systems Symposium, San Diego, CA, USA, 18–21 February 2018. [CrossRef]
12. Celik, Z.B.; Tan, G.; McDaniel, P.D. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In Proceedings of the NDSS, San Diego, CA, USA, 24–27 February 2019. [CrossRef]
13. Tian, Y.; Zhang, N.; Lin, Y.H.; Wang, X.; Ur, B.; Guo, X.; Tague, P. Smartauth: User-Centered Authorization for the Internet of Things. In Proceedings of the 26th {USENIX} Security Symposium ({USENIX} Security 17), Vancouver, BC, Canada, 5 May 2017; pp. 361–378.
14. Chen, J.; Diao, W.; Zhao, Q.; Zuo, C.; Lin, Z.; Wang, X.; Lau, W.C.; Sun, M.; Yang, R.; Zhang, K. IoTfuzzer: Discovering Memory Corruptions in IoT Through App-Based Fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 18–21 February 2018. [CrossRef]
15. Roundy, K.A.; Miller, B.P. Hybrid analysis and control of malware. In *International Workshop on Recent Advances in Intrusion Detection*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 317–338. [CrossRef]
16. Community, S. Samsung Smartthings Applications. Available online: <https://github.com/SmartThingsCommunity/SmartThingsPublic> (accessed on 27 December 2021).
17. Apple. HomeKit. Available online: <https://developer.apple.com/homekit/> (accessed on 27 December 2021).
18. Amazon. Alexa. Available online: <https://developer.amazon.com/alexa> (accessed on 27 December 2021).
19. Alam, T. A reliable Communication Framework and Its Use in Internet of Things (IoT). *SSRN* **2018**, 450–456. [CrossRef]
20. De Prado, A.G.; Ortiz, G.; Boubeta-Puig, J. CARED-SOA: A Context-Aware Event-Driven Service-Oriented Architecture. *IEEE Access* **2017**, *5*, 4646–4663. [CrossRef]
21. Su, T.; Fu, Z.; Pu, G.; He, J.; Su, Z. Combining symbolic execution and model checking for data flow testing. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1, pp. 654–665.
22. Souri, A.; Norouzi, M. A state-of-the-art survey on formal verification of the internet of things applications. *J. Serv. Sci. Res.* **2019**, *11*, 47–67. [CrossRef]
23. Aslan, Ö.A.; Samet, R. A comprehensive review on malware detection approaches. *IEEE Access* **2020**, *8*, 6249–6271. [CrossRef]
24. Clarke, E.M.; Henzinger, T.A.; Veith, H.; Bloem, R. (Eds.) *Handbook of Model Checking*; Springer: Cham, Switzerland, 2018; Volume 10.
25. Jiang, L.; Rewcastle, R.; Denny, P.; Tempero, E. CompareCFG: Providing Visual Feedback on Code Quality Using Control Flow Graphs. In Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, Online, 17–19 June 2020; pp. 493–499.
26. Das, M.; Lerner, S.; Seigle, M. ESP: Path-Sensitive Program Verification in Polynomial Time. In Proceedings of the ACM SIG-PLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, 17–19 June 2002; pp. 57–68.
27. Fernandes, E.; Jung, J.; Prakash, A. Security Analysis of Emerging Smart Home Applications. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; pp. 636–654. [CrossRef]
28. Liu, J.; Shen, S.; Yue, G.; Han, R.; Li, H. A stochastic evolutionary coalition game model of secure and dependable virtual service in sensor-cloud. *Appl. Soft Comput.* **2015**, *30*, 123–135. [CrossRef]
29. Sun, P. Security and privacy protection in cloud computing: Discussions and challenges. *J. Netw. Comput. Appl.* **2020**, *160*, 102642. [CrossRef]

30. Amanullah, M.A.; Habeeb, R.A.A.; Nasaruddin, F.H.; Gani, A.; Ahmed, E.; Nainar, A.S.M.; Akim, N.M.; Imran, M. Deep learning and big data technologies for IoT security. *Comput. Commun.* **2020**, *151*, 495–517. [[CrossRef](#)]
31. Diro, A.A.; Chilamkurti, N. Distributed attack detection scheme using deep learning approach for Internet of Things. *Future Gener. Comput. Syst.* **2018**, *82*, 761–768. [[CrossRef](#)]
32. Al-Dujaili, A.; Huang, A.; Hemberg, E.; O'Reilly, U.M. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. In Proceedings of the 2018 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, 24 May 2018; pp. 76–82. [[CrossRef](#)]
33. Lopez-Martin, M.; Sanchez-Esguevillas, A.; Arribas, J.I.; Carro, B. Supervised contrastive learning over prototype-label embeddings for network intrusion detection. *Inf. Fusion* **2022**, *79*, 200–228. [[CrossRef](#)]
34. Lopez-Martin, M.; Sanchez-Esguevillas, A.; Arribas, J.I.; Carro, B. Network Intrusion Detection Based on Extended RBF Neural Network With Offline Reinforcement Learning. *IEEE Access* **2021**, *9*, 153153–153170. [[CrossRef](#)]
35. Kang, H.J.; Sim, S.Q.; Lo, D. IoTBox: Sandbox Mining to Prevent Interaction Threats in IoT Systems. In Proceedings of the 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), Porto de Galinhas, Brazil, 12–16 April 2021; pp. 182–193. [[CrossRef](#)]
36. Moser, A.; Kruegel, C.; Kirda, E. Limits of Static Analysis for Malware Detection. In Proceedings of the Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), Miami Beach, FL, USA, 10–14 December 2007; pp. 421–430. [[CrossRef](#)]
37. Wang, B.; Dou, Y.; Sang, Y.; Zhang, Y.; Huang, J. IoTcMal: Towards a Hybrid IoT HoneyPot for Capturing and Analyzing Malware. In Proceedings of the ICC 2020-2020 IEEE International Conference on Communications (ICC), Virtual Conference, 7–11 June 2020; pp. 1–7. [[CrossRef](#)]
38. Islam, R.; Tian, R.; Batten, L.M.; Versteeg, S. Classification of malware based on integrated static and dynamic features. *J. Netw. Comput. Appl.* **2013**, *36*, 646–656. [[CrossRef](#)]
39. Gibert, D.; Mateu, C.; Planes, J. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *J. Netw. Comput. Appl.* **2020**, *153*, 102526. [[CrossRef](#)]
40. Burnap, P.; French, R.; Turner, F.; Jones, K. Malware classification using self organizing feature maps and machine activity data. *Comput. Secur.* **2018**, *73*, 399–410. [[CrossRef](#)]
41. Hamza, A.A.; Abdel-Halim, I.T.; Sobh, M.A.; Bahaa-Eldin, A.M. A survey and taxonomy of program analysis for IoT platforms. *Ain Shams Eng. J.* **2021**, *12*, 3725–3736. [[CrossRef](#)]
42. Alasmay, H.; Abusnaina, A.; Jang, R.; Abuhamad, M.; Anwar, A.; Nyang, D.; Mohaisen, D. Soteria: Detecting Adversarial Examples in Control Flow Graph-Based Malware Classifiers. In Proceedings of the 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), Singapore, 29 November–1 December 2020; pp. 888–898. [[CrossRef](#)]
43. Fang, Z.; Fu, H.; Gu, T.; Qian, Z.; Jaeger, T.; Hu, P.; Mohapatra, P. A Model Checking-Based Security Analysis Framework for IoT Systems. *High-Confid. Comput.* **2021**, *1*, 100004. [[CrossRef](#)]
44. Liang, C.J.M.; Karlsson, B.F.; Lane, N.D.; Zhao, F.; Zhang, J.; Pan, Z.; Li, Z.; Yu, Y. SIFT: Building an Internet of Safe Things. In Proceedings of the 14th International Conference on Information Processing in Sensor Networks 2015, New York, NY, USA, 13–16 April 2015; pp. 298–309. [[CrossRef](#)]
45. Wang, Q.; Datta, P.; Yang, W.; Liu, S.; Bates, A.; Gunter, C.A. Charting the Attack Surface of Trigger-Action IoT Platforms. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 1439–1453. [[CrossRef](#)]
46. Yu, Y.; Liu, J. TAPInspector: Safety and Liveness Verification of Concurrent Trigger-Action IoT Systems. *arXiv* **2021**, arXiv:2102.01468.
47. Sun, P.; Garcia, L.; Salles-Loustau, G.; Zonouz, S. Hybrid Firmware Analysis for Known Mobile and IoT Security Vulnerabilities. In Proceedings of the 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Valencia, Spain, 29 June–2 July 2020; pp. 373–384. [[CrossRef](#)]
48. Ibrahim, H.; Hassan, H.; Nabil, E. A conflicts' classification for IoT-based services: A comparative survey. *PeerJ Comput. Sci.* **2021**, *7*, e480. [[CrossRef](#)] [[PubMed](#)]
49. Li, L.; Bissyandé, T.F.; Papadakis, M.; Rasthofer, S.; Bartel, A.; Outeau, D.; Klein, J.; Traon, L. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.* **2017**, *88*, 67–95. [[CrossRef](#)]
50. Rocha, C.; Meseguer, J.; Muñoz, C. Rewriting modulo SMT and open system analysis. *J. Log. Algebraic Methods Program.* **2017**, *86*, 269–297. [[CrossRef](#)]
51. Pnueli, A. The Temporal Logic of Programs. In Proceedings of the 18th Annual Symposium on Foundations of Computer Science, Providence, RI, USA, 31 October–2 November 1977; pp. 46–57. [[CrossRef](#)]
52. Kim, Y. Convolutional Neural Networks for Sentence Classification. In Proceedings of the EMNLP, Doha, Qatar, 25 October 2014; pp. 1746–1751.
53. Jin, R.; Lu, L.; Lee, J.; Usman, A. Multi-Representational convolutional neural networks for text classification. *Comput. Intell.* **2019**, *35*, 599–609. [[CrossRef](#)]
54. Zhang, X.; Zhao, J.; LeCun, Y. Character-level convolutional networks for text classification. *Adv. Neural Inf. Process. Syst.* **2015**, *28*, 649–657.

55. Lu, W.; Duan, Y.; Song, Y. Self-Attention-Based Convolutional Neural Networks for Sentence Classification. In Proceedings of the 2020 IEEE 6th International Conference on Computer and Communications (ICCC), Chengdu, China, 11–14 December 2020; pp. 2065–2069.
56. Young, T.; Hazarika, D.; Poria, S.; Cambria, E. Recent trends in deep learning based natural language processing. *IEEE Comput. Intell. Mag.* **2018**, *13*, 55–75. [[CrossRef](#)]
57. Meseguer, J. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.* **1992**, *96*, 73–155. [[CrossRef](#)]
58. Liu, S.; Ölveczky, P.C.; Zhang, M.; Wang, Q.; Meseguer, J. Automatic analysis of consistency properties of distributed trans-action systems in Maude. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer: Cham, Switzerland, 2019; pp. 40–57. [[CrossRef](#)]
59. El Maarabani, M.; Cavalli, A.; Hwang, I.; Zaïdi, F. Verification of Interoperability Security Policies by Model Checking. In Proceedings of the 2011 IEEE 13th International Symposium on High-Assurance Systems Engineering, Boca Raton, FL, USA, 10–12 November 2011; pp. 376–381. [[CrossRef](#)]
60. IoTMAL Benchmark App Repository. Available online: <https://github.com/IoTBench/IoTBench-test-suite> (accessed on 27 December 2021).
61. Chen, S.; Xi, J.; Chen, Y.; Zhao, J. Association Mining of Near Misses in Hydropower Engineering Construction Based on Convolutional Neural Network Text Classification. *Comput. Intell. Neurosci.* **2022**, *2022*, 4851615. [[CrossRef](#)]
62. Liu, Y.; Li, P.; Hu, X. Combining context-relevant features with multi-stage attention network for short text classification. *Comput. Speech Lang.* **2022**, *71*, 101268. [[CrossRef](#)]
63. Ahmed, M.; Chakraborty, P.; Choudhury, T. Bangla Document Categorization Using Deep RNN Model with Attention Mechanism. In *Cyber Intelligence and Information Retrieval*; Springer: Singapore, 2022; pp. 137–147.