Published: 10 September 2025
Online AM: 19 June 2025
Accepted: 12 June 2025
Revised: 30 April 2025
Received: 24 December 2024

SURVEY

Static Code Analysis for IoT Security: A Systematic Literature Review

DIEGO R GOMES, University of Coimbra, Centre for Informatics and System, Coimbra, Portugal

EDUARDO F FELIX, University of Coimbra, Centre for Informatics and System, Coimbra, Portugal

FERNANDO AIRES, Federal Rural University of Pernambuco, Recife, PE, Brazil

MARCO PAULO AMORIM VIEIRA, The University of North Carolina at Charlotte, Charlotte, NC, United States

Static Code Analysis for IoT Security: A Systematic Literature Review

DIEGO GOMES, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal
EDUARDO FELIX, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal
FERNANDO AIRES, Department of Applied Computing, Federal Rural University of Pernambuco, Recife, Brazil
MARCO VIEIRA, Department of Computer Science, The University of North Carolina at Charlotte, Charlotte, United States

The growth of the Internet of Things (IoT) has provided significant advances in several areas of the industry, but security concerns have also increased due to this expansion. Many IoT devices are the target of cyber attacks due to various firmware, source code, and software vulnerabilities. In this context, static code analysis, leveraging various techniques, has emerged as an effective approach to examine and identify security vulnerabilities, including insecure functions, buffer overflows, and code injection. However, recent research has shown several challenges associated with this approach, such as limited understanding of vulnerabilities, inadequate threat detection, and insufficient semantic analysis of IoT device source code. Consequently, several IoT security research studies integrate static analysis with other methods, such as dynamic analysis, machine learning, and natural language processing, to enhance vulnerability analysis and detection. To provide a comprehensive understanding of the current state of static analysis in IoT security, this systematic literature review explores existing vulnerabilities, techniques, and methods while highlighting the challenges that hinder the extraction of meaningful insights from such analyses.

CCS Concepts: • **Security and privacy → Software security engineering;**

Additional Key Words and Phrases: Static code analysis, internet of things, IoT security, vulnerability detection, OWASP

This work was partially funded by FCT grants 2024.06022.BD, 2024.06014.BD, and I.P./MCTES through national funds (PIDDAC), within the scope of the CISUC R&D Unit - UIDB/00326/2020 and project code UIDP/00326/2020. Content produced within the scope of the Agenda "NEXUS - Pacto de Inovação – Transição Verde e Digital para Transportes, Logística e Mobilidade", financed by the Portuguese Recovery and Resilience Plan (PRR), with no. C645112083-00000059 (investment project no. 53).
Authors' Contact Information: Diego Gomes, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, University of Coimbra, Coimbra, Coimbra, Portugal; e-mail: drg@dei.uc.pt; Eduardo Felix, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, University of Coimbra, Coimbra, Coimbra, Portugal; e-mail: effelix@dei.uc.pt; Fernando Aires, Department of Applied Computing, Federal Rural University of Pernambuco, Recife, Pernambuco, Brazil; e-mail: fernandoaires@ufrpe.br; Marco Vieira, Department of Computer Science, The University of North Carolina at Charlotte, Charlotte, North Carolina, United States; e-mail: marco.vieira@charlotte.edu.

ACM 0360-0300/2025/09-ART65
https://doi.org/10.1145/3745019

ACM Reference Format:
Diego Gomes, Eduardo Felix, Fernando Aires, and Marco Vieira. 2025. Static Code Analysis for IoT Security: A Systematic Literature Review. *ACM Comput. Surv.* 58, 3, Article 65 (September 2025), 47 pages. https://doi.org/10.1145/3745019

1 Introduction

The **Internet of Things** (**IoT**) market is advancing rapidly, with a projected annual expansion of 10.49% from 2024 to 2029. This growth is observed across multiple sectors, including automotive, industrial, consumer, smart finance, healthcare, smart cities, and other areas of IoT [113]. However, over 76% of IoT device manufacturers and retailers must comply with new security and privacy legislation and requirements [53]. Given the inherent complexity of IoT systems, meeting these security and privacy requirements is a significant challenge, with persistent security issues arising at each layer of the IoT architecture. According to Mohindru and Garg [76], IoT systems typically follow a three-tier architecture: Perception, Network, and Application. The Application layer, in particular, is frequently linked to high-level security vulnerabilities, including insecure software, firmware, and interfaces.

A vulnerability is a flaw that an attacker can exploit to harm interested parties. In this regard, the **Open Web Application Security Project** (**OWASP**)[1] provides widely recognized security guidelines, including the OWASP IoT Top 10, which highlights critical vulnerabilities in hardware, software, and IoT device systems. OWASP also highlights the importance of mitigating vulnerabilities to improve the protection of these devices [88]. If security requirements such as authentication, privacy, integrity, and control are met, the IoT system can be considered highly secure [55]. To achieve this, it is essential to use a combination of **Static Application Security Testing** (**SAST**) and **Dynamic Application Security Testing** (**DAST**), among other techniques, to identify security flaws and provide insights that support developers in implementing necessary fixes.

Static Code Analysis, a core component of SAST, consists of analyzing software artifacts, such as source code, binaries, and firmware, without executing them and is generally performed during the early phases of the secure **software development life cycle** (**SDLC**). This method is considered essential in software engineering both for identifying defects [104] and improving code quality [84]. Additionally, static analysis enables the detection of coding patterns that can be maliciously exploited [25], allowing the developers to take preventive measures to increase system security and protect sensitive data against cyber threats [49]. To achieve these goals, static code analysis involves executing tools to identify vulnerabilities in source code. These tools apply several techniques, such as ***Data Flow Analysis (DFA)***, ***Control Flow Analysis (CFA)***, *Taint Analysis*, and *Lexical Analysis*, to detect possible flaws and weaknesses in the code [31].

Existing solutions for identifying vulnerabilities in IoT devices face several significant challenges. These include achieving comprehensive detection of diverse vulnerabilities within the code, enhancing precision and recall to reduce false positives, and developing a deeper semantic understanding and contextualization of IoT device source code. While static analysis remains a key solution for vulnerability detection and security enhancement, there is a pressing need for thorough reviews and surveys that critically evaluate its application in the context of IoT security.

This **Systematic Literature Review** (**SLR**) on static code analysis for IoT security brings four main contributions:

[1]Open Web Application Security Project (OWASP), available at https://owasp.org.

— We systematically identify and analyze 94 of the most relevant articles published between 2015 and 2025, providing a comprehensive overview of existing studies on IoT security vulnerability detection using static code analysis.
— We categorize and discuss critical security vulnerabilities identified through static analysis techniques, including unsafe functions, privacy violations, **buffer overflows** (**BO**), injection attacks, and improper data validation. This helps researchers and practitioners understand which vulnerabilities are most prevalent in IoT environments.
— We examine the primary static analysis techniques used for vulnerability detection, including CFA, DFA, taint analysis, and semantic analysis. Additionally, we detail how static analysis has been integrated with hybrid approaches such as dynamic analysis, **machine learning** (**ML**), and **natural language processing** (**NLP**) to enhance vulnerability detection in IoT environments.
— We outline the key challenges and limitations of applying static analysis to IoT security, focusing on comprehensive vulnerability coverage, architectural and resource constraints, and precision and recall issues. Furthermore, we highlight open problems and future directions, providing insights to advance the field.

The article is divided into five sections. Section 2 discusses related work. Section 3 discusses the SLR process and its steps. Sections 4–6 present the analysis of the selected studies, discussing the identified IoT security vulnerabilities, the static analysis techniques and approaches used in vulnerability detection, and the main challenges encountered in improving IoT security through static analysis. These sections highlight the results of the reviewed studies and provide insights into the field's current state. Section 7 presents a broader discussion on the current IoT security landscape, the role of stakeholders, and future research directions. Section 8 highlights the threats to validity. Finally, Section 9 concludes this research and indicates possible future directions.

Appendix A provides definitions of technical terms used throughout the article. Appendix B lists studies that exclusively use static analysis techniques, Appendix C presents studies combining static and dynamic analysis, and papers integrating static analysis with ML/NLP techniques are in Appendix D. Appendix E lists the challenges and approaches in static, dynamic, and ML/NLP-based analysis for IoT Security in the literature.

2 Related Work

Xie et al. [123] conducted a literature review focusing on identifying vulnerabilities in IoT firmware, categorizing papers into groups such as Static Analysis, Symbolic Execution, Fuzzing on Emulators, and Comprehensive Testing. The research highlights vulnerabilities of IoT devices to security flaws, such as *Cross-site scripting(XSS)*, *injection*, *authentication bypass*, and *BO*. Furthermore, the study presents a new method that combines fuzzing with static analysis to detect and verify authentication bypass vulnerabilities in binary servers integrated into IoT systems.

Ngo et al. [81] compared 10 papers that present methods and techniques for detecting malware in IoT devices by analyzing static features. These methods encompass a variety of features, such as ELF-header, String, Opcode, Opcode Graph, Grayscale Image, **Control Flow Graphs** (**CFGs**), and PSI Graph. The techniques employed in these studies include neural networks, fuzzing, ML, clustering, and graph theory. The evaluation primarily focused on the effectiveness of malware detection, the occurrence of false positives, and computational demands.

Qasem et al. [97] present comprehensive research on automatically detecting embedded devices and firmware vulnerabilities. The study encompasses a detailed analysis of dynamic analysis, symbolic execution, and static vulnerability detection, proposing layered taxonomies to map methods and characteristics, application domains, and binary analysis techniques. Additionally, it conducts

quantitative and qualitative comparisons of 27 papers on vulnerability detection in embedded systems, and discusses the challenges embedded systems face due to outdated software and libraries, emphasizing the importance of vulnerability detection, particularly in IoT devices.

Tang et al. [117] investigate vulnerabilities in intelligent contracts, focusing on static analysis, dynamic analysis, and formal verification techniques to examine 15 security vulnerabilities, highlighting the vulnerable areas and their underlying causes. In the context of static analysis, the authors present nine papers that utilize program code scanning tools, which employ lexical analysis, syntax analysis, control flow, and DFA. The authors emphasize that static analysis is an interesting approach because it can detect problems before deploying a smart contract on the blockchain. However, the results reveal that some vulnerabilities in smart contracts are not detected, even with these available tools.

Li et al. [60] identified and analyzed vulnerability detection methods in Linux-based IoT devices. The research is grounded in an SLR, which identifies the advantages, disadvantages, and limitations of different analysis methods. The authors presented five studies using static analysis techniques for backdoor detection. Additionally, 18 studies were examined that combined static analysis with dynamic analysis and fuzzing to identify undisclosed vulnerabilities. Lastly, eight papers delved into static analysis alongside ML techniques. The analysis results highlight the prevalence of static and dynamic analysis as the primary techniques for vulnerability detection.

Sun et al. [115] conducted a review on privacy and security in IoT environments, proposing a general protection framework and analyzing threats and countermeasures. The study analyzed 13 papers on firmware vulnerabilities, covering authentication bypass, memory vulnerabilities, logic flaws, command injection, and firmware modification. In addition, it reviewed six studies on automated vulnerability detection techniques, categorizing them into static analysis, symbolic execution, fuzzing, and ML. In the context of static analysis, the article identifies lexical, syntactic, control flow, and DFA as key techniques for detecting security vulnerabilities. The study also discusses challenges in firmware protection and highlights the need to improve automated detection and integrate hybrid approaches to increase the effectiveness of vulnerability identification.

Previous literature reviews have examined various aspects of static analysis in IoT security, including detecting vulnerabilities in firmware, embedded devices, smart contracts, and Linux-based IoT devices. However, these studies often focus on specific domains or particular applications without providing a unified and comprehensive assessment of their effectiveness across different IoT components. For instance, while some reviews have focused on firmware analysis, others have explored static techniques for smart contracts or embedded devices. Moreover, most existing studies do not compare static analysis approaches with hybrid approaches, such as integrating static and dynamic analysis, ML, or NLP for IoT vulnerability detection. Similarly, previous research does not provide a detailed study of the specific static analysis techniques employed, nor does it assess their effectiveness across different aspects of IoT.

To address these gaps, we conduct **a comprehensive SLR that explores the effectiveness of static analysis methods in identifying vulnerabilities across the broader IoT domain.** We investigate three fundamental components (source code, binaries, and firmware), examining the security vulnerabilities associated with each. Additionally, we explore static analysis methods and hybrid approaches, evaluating their effectiveness in different IoT contexts. We also identify the most commonly used static analysis techniques, highlighting their applicability and limitations. Finally, we discuss the main challenges in applying these techniques, considering IoT systems' complexity, the devices' diversity, and their dynamic interactions. This is the central objective of our study. By addressing this gap in the literature, we aim to provide critical guidance for future research and advances in IoT security.

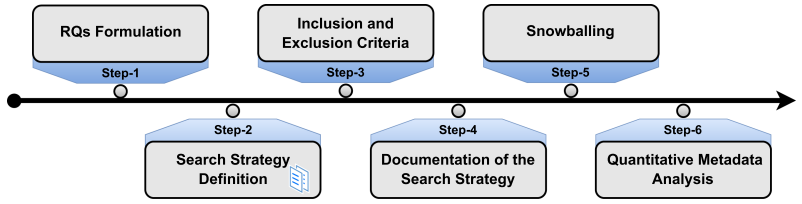

Fig. 1. SLR process.

3 Systematic Literature Review Process

The main objective of this article is to provide a comprehensive review of the application of static analysis for security in the IoT domain. To achieve this, we conducted an SLR following established guidelines and methodologies, drawing inspiration from prior studies such as [1, 58, 62]. This review encompasses six steps, as shown in Figure 1, which will be discussed in detail in the following sections.

3.1 RQs Formulation

The scope of this research is to provide a holistic view of the vulnerabilities that can be identified through static analysis, the techniques used for this identification, and the challenges and concerns related to the application of static analysis in IoT security. To this end, three research questions were defined. The motivation and descriptive details about these questions are provided next.

RQ1. What IoT security vulnerabilities have been identified through static analysis? The IoT is susceptible to numerous security vulnerabilities, including unsafe functions, BO, injection attacks, and memory leaks. Identifying the most commonly detected vulnerabilities through static analysis is essential for understanding the security challenges in IoT systems and guiding efforts to address these issues effectively. *This research question is addressed in Section 4.*

RQ2. What static analysis techniques and approaches are used in IoT vulnerability detection? Static analysis encompasses a range of techniques, including CFA, DFA, taint analysis, and semantic analysis. Identifying and understanding the most commonly used techniques and approaches in IoT static analysis is critical for enhancing the effectiveness of vulnerability detection. *RQ2 is discussed in Section 5.*

RQ3. What are the challenges of static analysis for IoT security? Both IoT and static analysis are research fields characterized by numerous challenges and unresolved issues. Understanding the specific barriers and limitations associated with applying static analysis to IoT vulnerability identification is crucial. Addressing these challenges can help identify gaps in current methodologies and provide valuable guidelines for shaping future research. *This research question is discussed in Section 6.*

3.2 Search Strategy Definition

The search strategy was structured into four activities: keyword definition, source selection, search string formulation, and search process definition, which are detailed below.

3.2.1 Keyword Definition. Specific keywords in static analysis and the IoT were identified and analyzed. However, due to various acronyms associated with IoT, a comprehensive effort was made to identify and examine all IoT-related acronyms. The goal of this specific approach was to thoroughly identify all relevant studies in both areas, as shown in Table 1.

Table 1. Keywords Used

Group	Keywords
Static Application Security Testing	`"SAST" OR "static application security testing" OR "security test*" OR "static analy*" OR "analy* static" OR "static code" OR "software analy*" OR "code review" OR "code inspection" OR "code quality" OR "secure code"`
Internet of Things	`"IoT" OR "internet of things" OR "IIoT" OR "IoTaaS" OR "IoE" OR "IoMT" OR "IoV" OR "IoP" OR "IoH" OR "IoA" OR "IoR" OR "WoT"`
Definition of IoT Acronyms:	
IoA: Internet of Agriculture, **IoE:** Internet of Everything, **IoR:** Internet of Retail, **IoV:** Internet of Vehicles, **IoP:** Internet of People, **IIoT:** Industrial Internet of Things, **IoH:** Internet of Healthcare, **IoMT:** Internet of Medical Things, **IoTaaS:** IoT as a Service, **WoT:** Web of Things	

Table 2. Search Results for Different Databases

Database Source	Title (A) AND Title (B)	Title (A) AND Abstract (B)	Title (B) AND Abstract (A)	Abstract (A) AND Abstract (B)	Duplicated	Result
ACM Digital Library	1	1	3	11	-5	11
Engineering Village	23	84	144	508	-218	541
IEEE Xplore	20	33	88	160	-127	174
Science Direct	0	10	20	51	-43	38
Scopus	39	72	264	402	-303	474
SpringerLink	60	17	24	0	-24	77
Web of Science	24	47	136	246	-195	258
Wiley Online Library	0	0	1	4	0	5

3.2.2 Source Selection. Eight search databases were selected to identify the most significant papers. These databases, illustrated in Table 2, offer appropriate search mechanisms for automated queries [118].

3.2.3 Search String Formulation. The search string formulation was guided by the keywords derived from the domains of static analysis and the IoT (see Table 1). These keywords were organized into two categories to streamline the process: Group A, referred to as SAST, and Group B, related to IoT. The search criteria were specifically designed to identify relevant papers by focusing on the title and abstract of the documents.

Four approaches were defined to construct the search string, as presented in Table 2. The first step involved searching for Group A and Group B terms in the title (Title (A) AND Title (B)). In the second step, Group A was searched in the title while Group B was searched in the abstract (Title (A) AND Abstract (B)), and vice versa (Title (B) AND Abstract (A)). Finally, a broader search was performed by applying both groups to the abstract (Abstract (A) AND Abstract (B)). This systematic approach ensured the inclusion of the most relevant papers in the review.

3.2.4 Search Process Definition. This activity aims to identify relevant primary studies. Therefore, manual searches were conducted to validate the accuracy of the search terms. Subsequently, the documents were automatically searched across all search databases. Initially, **inclusion and exclusion criteria** (**IC/EC**) were applied, such as selecting studies written in English (IC1/EC2) and those published between 2015 and 2025 (IC2/EC3). These criteria, which are summarized in Table 3, are discussed in detail in Section 3.3. The results of the four search approaches described

Table 3. Inclusion and Exclusion Criteria

Inclusion Criteria	Exclusion Criteria
IC1 Studies that are written in English and available in full text.	EC1 Exclude duplicate studies.
IC2 Research papers from 2015 to 2025 were included in the studies.	EC2 Exclude studies not in English unless a translation is available.
IC3 Studies that have been published in peer-reviewed conferences or journals.	EC3 Exclude studies published before a certain period.
IC4 Studies that are directly relevant to static analysis in IoT security.	EC4 Exclude works not directly related to static analysis in IoT security.
	EC5 Exclude works or studies that cannot be accessed.
	EC6 Exclude documents such as proceedings, chapters, thesis, and books.

above are summarized in Table 2.The Rayyan Web platform [98] was used to automatically identify duplicate papers across search databases, while the final confirmation and exclusion of duplicates were performed manually.

The initial search process retrieved a total of 1,578 documents across all databases. Engineering Village (541), Scopus (474), and IEEE Xplore (174) contributed the largest number of results. In contrast, Springer (77), ScienceDirect (38), the ACM Digital Library (11), and Wiley Online Library (5) had more limited outputs.

3.3 Inclusion and Exclusion Criteria

This section details the IC/EC. These criteria are detailed in Table 3. Initially, redundant papers were removed (EC1). Only peer-reviewed papers (IC3) that addressed the context of static analysis in IoT security (IC4/EC4) were included. Finally, inaccessible papers (EC5) and those categorized as proceedings, chapters, theses, and books (EC6) were excluded. These IC/EC were established to align with the study objectives, emphasizing the reliability, relevance, and accessibility of the selected studies. Peer-reviewed papers were prioritized due to their validated findings, while inaccessible papers and non-journal publications were excluded to promote consistency and reproducibility [56].

3.4 Documentation of the Search Strategy

The search strategy consisted of five steps described in Figure 2. Initially, 1,578 papers were collected from various databases. Then, 771 duplicated documents were identified (EC1), resulting in 807 unique papers.

In the second stage, 417 articles were excluded due to acronyms unrelated to IoT or failure to meet the criterion (EC6). Meanwhile, 31 articles, identified as research articles, systematic reviews, or literature reviews, were separated for inclusion in the related work section. These papers also underwent IC/EC, resulting in six articles ultimately detailed in the related work section (see Section 2). With this, 359 articles were selected for the next stage.

During the third stage, the titles, keywords, and abstracts of the papers were analyzed to identify those aligned with the research objectives, applying the exclusion criterion (EC4). As a result, 101 papers were partially chosen. However, doubts existed regarding the adherence to the inclusion criteria (IC4) for 17 of these papers. To address this, a collaborative review was conducted using the

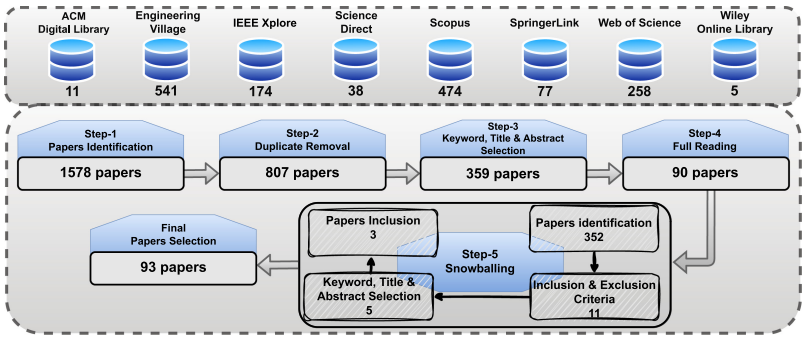

Fig. 2. Search strategy processes.

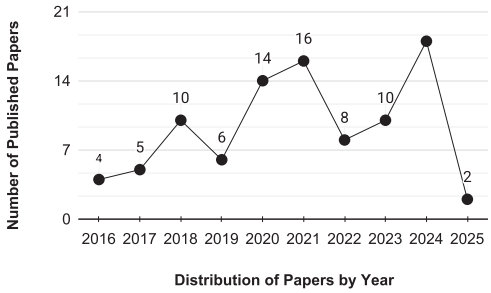

Fig. 3. Papers publications distribution per year.

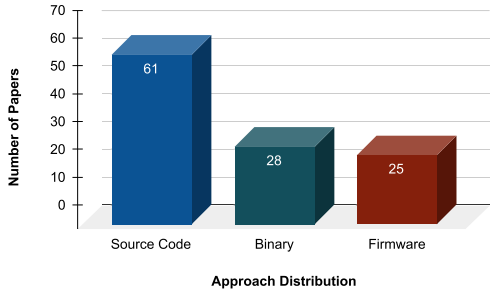

Fig. 4. Papers count by source code, binary, and firmware.

Rayyan Web platform, which facilitated blind screening and conflict resolution. Each researcher independently assessed the papers, and a consensus was reached through discussion to determine whether they met the IC. After this process, we found that only 7 of the 17 papers met the criteria. Based on meeting the IC/EC, 90 papers were chosen.

3.5 Snowballing

The snowballing process relies on reviewing relevant studies to identify additional ones [58, 120, 121]. We used the ResearchRabbitApp tool [103] to assist in discovering new articles by exploring the citations included in the selected articles. The stages of this process are detailed in Figure 2. The snowballing was applied to the 90 papers selected through the systematic process described in Section 3.4, resulting in 352 new articles identified from their reference lists. To ensure relevance, we applied the IC/EC outlined in Table 3, which reduced the number to 11. After a detailed screening based on titles, keywords, and abstracts, five papers were shortlisted for full review, and three of them ultimately met all selection criteria. As a result, the final set of selected studies comprises 93 papers relevant to the scope of this research.

3.6 Quantitative Metadata Analysis

The metadata analysis enables the identification of patterns, relationships among data, and trends, contributing to a more comprehensive understanding of the research results. Figure 3 illustrates the timeline of papers related to static analysis in IoT in the context of security.

The distribution of publications from 2016 to 2025 reveals specific trends: there was a significant growth between 2016 and 2018, from 4 to 10 papers; in 2019, publications declined, but in 2020 and

Table 4. Distribution of Articles by Methods

Methods	Paper identifiers per year									
	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025
Static Analysis only	[9, 105]	[27, 86] [10, 23]	[15, 16] [17, 36] [74, 94] [96]	[33, 49] [108]	[84, 114] [29, 106] [14, 18] [35, 70] [129]	[5, 8] [12, 68] [109, 110]	[57, 124] [79, 87] [93]	[6, 116] [7, 19] [50]	[65, 128] [30, 130] [47, 100] [13, 82] [69, 72]	-
Static and Dynamic Analysis	[40, 80]	[71]	[25, 63] [95, 119]	[107, 112] [132]	[20, 102] [127]	[32, 104] [37, 45] [48, 64] [66, 101]	[39, 46]	[54, 99]	[3, 61] [85, 131]	-
Static Analysis and ML/NLP	-	-	-	-	[28, 83]	[67]	[4]	[22, 125] [126]	[75, 133] [24, 73]	[21, 122]

2021, it peaked at 14 to 16 papers; however, in 2022, the number of publications dropped again but rose shortly after, indicating a growth trend. This trend continued in 2024, reaching 18 publications, while in 2025, 2 papers have been recorded so far.

The data shows that static analysis plays a crucial role in several approaches related to IoT, with specific emphasis on evaluating components such as source code, binary, and firmware, as illustrated in Figure 4.

Several papers focused on static analysis of source code [27, 86]. On the other hand, other studies applied static analysis to binaries [23, 114], while some publications highlighted their analyses on firmware [40, 112]. However, certain research efforts have integrated static analysis into multiple approaches [29, 106, 116, 124].

The distribution of papers based on different methods allows for a clearer understanding of studies, as illustrated in Table 4. The first group includes papers focusing exclusively on static analysis or using it as part of their approach. The following groups include hybrid methods that combine static analysis with dynamic analysis or ML/NLP.

Most studies are exclusively related to static analysis. Specifically, 39 papers focus only on static analysis, as per the work of [23, 27, 36, 50], while 12 papers use it as part of a broader approach [14, 93, 110]. However, a hybrid approach was adopted by 29 studies that combined static and dynamic analysis methods, as demonstrated in [20, 25, 63]. Of these, nine studies used the fuzzing technique, according to studies [112, 132]. Finally, 13 studies used static analysis with the ML method, such as the [28, 125] and NLP studies, as evidenced in the studies [67, 126]. The dataset supporting the review is publicly available at [41].

In the following sections, we discuss the main findings identified during the proposed systematic review, addressing the research questions presented in Section 3.1. Specifically, **RQ1** is addressed in Section 4, **RQ2** in Section 5, and **RQ3** in Section 6. This discussion highlights gaps in the state of the art and identifies open challenges for future investigations in static analysis for IoT security.

4 RQ1. What IoT Security Vulnerabilities have been Identified through Static Analysis?

A vulnerability is a flaw or weakness that an attacker can exploit to cause harm. OWASP provides a detailed framework categorizing 61 recognized vulnerabilities [91]. Based on the analysis of the 51 articles that applied only static analysis, we identified the vulnerabilities discussed and correlated them with the OWASP framework. In several cases, multiple vulnerabilities were identified in a single article. However, eight articles used static analysis to focus on other issues,

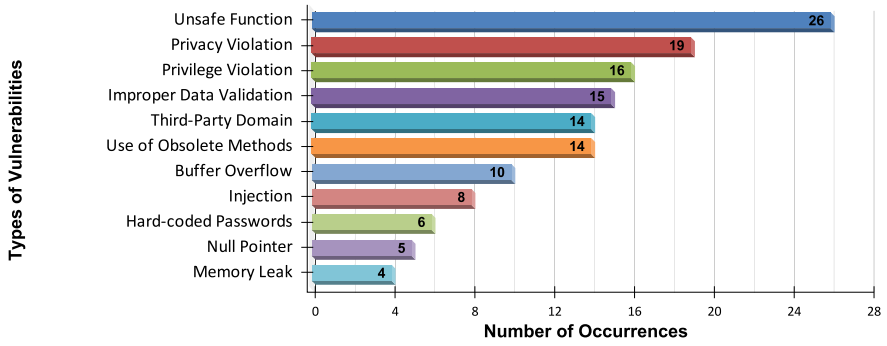

Fig. 5. Total occurrences of vulnerabilities across papers.

such as evaluating smart contract analysis tools [8, 94], modeling dataflow and network configurations in IoT systems [14, 19], analyzing firmware structure [13, 128], extracting arguments from stripped binaries [109], or assessing code quality metrics [57]. Figure 5 presents the 11 identified vulnerability types and the number of their occurrences across the reviewed papers.

As illustrated in Figure 5, the most frequent vulnerability addressed in the analyzed works is Unsafe Function, followed by Privacy and Privilege Violation. Although less common, vulnerabilities such as Memory Leak and Null Pointer are also focused on. This distribution highlights critical security areas in IoT systems development that require more effort. It is essential to implement Secure Coding Practices [92] to significantly improve the integrity of these systems [9].

The **Unsafe Function** vulnerability is cited 26 times, highlighting the importance of examining functions that can compromise system security. Functions such as *gets*, *strcpy*, *memcpy*, and *strlen* lack buffer limit checks, making them prone to memory-related vulnerabilities [50, 69, 74]. For example, Rai and Grover [100] highlight the use of *memcpy* in cryptographic modules due to performance benefits, but warn that its improper use can introduce vulnerabilities. According to Al-Boghdady et al. [5], the use of these unsafe functions in low-level programming can lead to memory corruption, BO, and improper memory management, posing significant threats, especially in IoT operating systems such as RIOT, Contiki, and FreeRTOS, where resource constraints and real-time execution amplify security risks. These vulnerabilities often stem from a lack of secure programming awareness, failure to implement proper boundary checks when handling buffers in user-defined functions, and the use of unsafe functions that weaken the security of RTOS-based IoT device firmware [124].

Concerns about **Privacy Violations** are the second most cited vulnerability in the selected studies. In the IoT context, these violations occur when personal information is collected or used without user awareness, consent, or clear justification. Although privacy violations are technically the result of underlying security flaws, we adopt this terminology in line with OWASP usage, which frames such issues as a specific type of vulnerability due to their frequency and critical impact. Mobile apps used to interact with IoT devices (particularly those in categories such as Health and Fitness, Medical, Navigation, and Tools) pose privacy risks due to the sensitive nature of the data involved. If not properly managed, these practices can result in the misuse or exposure of personal information, ultimately violating user privacy [110].

Additionally, advertising trackers in pre-installed apps can collect user data without consent [116]. Privacy risks can also be inferred from application metadata, such as declared permissions, presence of advertising libraries, and components defined in files like *AndroidManifest.xml*, as explored by the App-PI ecosystem [72]. Unencrypted communication between IoT devices in domestic environments also facilitates spying on network traffic [49]. According to Carvalho

and Eler [15], smart toys that collect personal data through cameras, microphones, and sensors are also a concern, as this information can be exposed. For instance, CloudPets leaked voice messages and user data due to poor access controls, while toys like Hello Barbie and My Friend Cayla posed serious privacy risks by allowing unprotected wireless connections and recording children's conversations. Several methods have been suggested to mitigate this vulnerability, such as the IoTAV framework, which employs the MQTT protocol and local file storage to support the encryption and protection of information [29]. Similarly, Ferrara et al. [35] proposed an extension of the industrial tool Julia,[2] which aims to detect privacy violations related to GDPR. This extension can identify sensitive data leaks [35, 36].

Sixteen papers highlighted concerns about the vulnerability **Privilege Violations**, which occur when applications retain elevated privileges longer than necessary after performing privileged operations, particularly in the context of mobile and IoT platforms. SmartThings and SmartApp applications often declare access to more resources required, which results in the assignment of excessive privileges [18]. These resources may result in unauthorized access to sensitive devices and information [6, 16, 79]. The IoTCom system found insecure interactions between IoT applications that could lead to unauthorized actions, such as triggering devices without permission [7]. Yuan et al. [130] report privilege violations in MQTT brokers, where flawed authorization allows message delivery in *QoS 2* even after clients lose the required permissions. Therefore, applying the principle of least privilege is crucial to limit access to essential resources, preventing these vulnerabilities [96].

Improper Data Validation, the fourth most cited vulnerability in the selected studies, occurs when user input is not checked for validity, completeness, or consistency before being processed by the application. This flaw supports malicious code execution in IoT systems, compromising the application [18]. Issues such as integer overflow and unvalidated timestamps are usual examples of this type of failure [94]. DFA is essential for identifying vulnerabilities, as highlighted by Costin [27] since input validation failures can create breaches when untrusted data reaches critical functions without proper sanitization. Proper validation helps promote the integrity of the system and user data [108]. A practical example is taint analysis frameworks, which examine whether user input reaches sensitive components, such as SQL queries, without validation, preventing attacks [70]. Rai and Grover [100] show that when preconditions like pointer validity or variable correctness are not fully verified, functions may be invoked with unexpected or incorrect parameters, particularly in critical components like cryptographic modules. Inadequate input validation can expose IoT systems to serious security risks, particularly in medical devices such as insulin pumps or heart monitors, where insufficient checks may lead to device malfunction or compromise. However, static analysis can detect these flaws early and prevent malicious data from compromising the system [47].

Identifying vulnerabilities in **Third-party Domains** is crucial for IoT security, as they occur when an application includes or interacts with third-party resources, such as external services, cloud APIs, or embedded content, without proper validation or sanitization. As Nobakht et al. [84] highlighted, third-party applications on IoT platforms, such as Google Fit-enabled applications, often request access to user data from cloud servers or external services. For example, some IoT fitness applications request excessive permissions beyond their intended functionality, such as access to biometric data, activity, and location, which could lead to data leaks. Interactions between different applications and the interconnection of IoT devices generate new vulnerabilities and increase security risks [12, 105]. Additionally, outdated third-party libraries in firmware can introduce vulnerabilities. Nino et al. [82] identified 14 N-day flaws in 191 firmware images, show-

[2]The Julia Static Analyzer for Java, available at http://doi.org/10.1007/978-3-662-53413-7_3.

ing persistent weaknesses caused by a lack of updates or remediation. Therefore, the importance of secure ecosystem interfaces in IoT systems [70] is highlighted, leading to the use of tools such as OWASP Dependency Checker to identify vulnerabilities in third-party components [30, 35].

Obsolete Methods represent one of the most critical vulnerabilities in IoT systems and occur when deprecated or outdated functions are used in the code, indicating a lack of maintenance. For example, using outdated firmware, such as BusyBox v0.60, can expose systems to known security vulnerabilities, such as CVE-2017-13772, which consists of multiple stack-based BO flaws that could be exploited to compromise device security [106]. Liu et al. [65] show that a considerable number of IoT binaries still include insecure protocols like SSL (2.0 and 3.0) and TLS (1.0 and 1.1), known for vulnerabilities such as POODLE. Their static analysis also revealed legacy API calls (e.g., *SSLv23_client_method()*) that enable insecure fallback behaviors, exposing devices to downgrade attacks. Obsolete methods also appear at the source code level. For instance, incorrect implementation of the *equals* and *hashCode* methods in Java classes can introduce flaws in the application logic [86]. Similarly, studies such as Alnaeli et al. [9], McBride et al. [74] emphasize the importance of replacing insecure commands in C and C++ with more secure alternatives, as recommended by newer language standards.

IoT systems also face security issues such as BO, which occur when excessive data is written to a buffer, exceeding its limits. This issue is prevalent in programs written in C due to manual memory management, which makes them vulnerable to attacks [50]. Ma et al. [69] identified BO as the main vulnerability in IoT-embedded firmware, reporting 13 cases, both heap and stack overflows, caused by missing input length checks, such as in the use of *_isoc99_sscanf*. As highlighted by Sachidananda et al. [106], this vulnerability has been found in IoT software such as BitThunder and Lepton due to the inappropriate use of functions such as *strcpy* and *strncpy*. IoT devices running FreeRTOS are also susceptible to this type of vulnerability since improper memory management can execute malicious code [5]. Additionally, *Stack Overflow*, a specific kind of BO, can cause crashes and unpredictable behavior in systems [23], and in WebAssembly environments, it can lead to arbitrary code execution [114].

Injection vulnerabilities occur when untrusted input is interpreted as code or commands. Common types include SQL injection, command injection, format string injection, and log injection. When combined with exposed HTTP endpoints and excessive privileges, attackers can execute malicious commands [79]. Specific studies state that code injection vulnerabilities in IoT operating systems arise when improperly validated data is passed to an interpreter as part of a command or query [5, 27]. Similarly, SQL injection vulnerabilities have been found in more than 70% of the analyzed IoT APKs due to creating queries without parameterized instructions [106]. However, static analysis can be a solution for detecting security vulnerabilities such as SQL [36, 47] and XSS [35, 47] injections. Industrial tools, such as the Julia static analyzer, are used to analyze contamination, verifying adequate user information hygiene before carrying out confidential operations [70].

Although the first vulnerability in the OWASP IoT Top 10 is the use of **Weak, Guessable, or Hardcoded Passwords,owasp2018iot**, this issue ranked only ninth among the vulnerabilities discussed in the analyzed studies. Passwords are a central concern for IoT security, as device misconfigurations can lead to large-scale attacks, such as the Mirai botnet [109]. Hardcoded credentials in firmware also pose a significant risk [106]; for instance, Sutter and Tellenbach [116] reported the detection of GitHub credentials embedded in firmware samples using static analysis, even when the tool presented a high false positive rate. While passwords are essential for enabling server communication, their inclusion directly in the source code, through hardcoding, is a well-known vulnerability, as defined by CWE-798 [49]. However, this flaw can be detected using static analysis techniques [35, 129].

Null Pointer vulnerabilities occur when a program attempts to access memory or invoke operations on an invalid reference or pointer, expecting it to be valid. These vulnerabilities pose a significant code security concern if not addressed correctly [18]. According to Lyu et al. [68], the SparrowHawk tool detected 340 instances of *null pointer dereferences* (CWE-476). After a manual analysis, 42 of them were confirmed. Rai and Grover [100] also highlight risks associated with unverified pointer validity when using the WP plugin of Frama-C, noting that it may fail to guarantee that pointers always reference valid objects, raising concerns about potential crashes or unintended behavior in IoT devices. Static analyzers are essential to detect potential pointer errors [36]. In Java-based systems, null pointer vulnerabilities can be mitigated by specifying conditions such as *EQ null* using **Java Modeling Language** (**JML**), which supports the correct handling of *null* values in methods like *equals* [86].

Finally, the **Memory Leak** vulnerability was addressed in four articles. This flaw occurs when memory is not released correctly, leading to wasted resources and potential software crashes, and is especially critical in IoT devices due to their limited resources [33, 106]. Estimating the maximum heap memory usage is crucial to prevent memory leaks and support proper resource release [23, 47].

5 RQ2. What Static Analysis Techniques and Approaches are Used in IoT Vulnerability Detection?

Several static analysis techniques and approaches have been used in the literature to identify vulnerabilities in IoT systems. For example, the articles in references [36, 82, 114] demonstrate how control flow, dataflow, and taint analysis techniques contribute to identifying specific vulnerabilities. However, several studies mention static analysis techniques more implicitly; although they do not provide explicit details about the adopted methods and procedures, it is possible to infer that these techniques were used based on the employed approaches [100, 106].

Several static analysis techniques exhibit interdependencies and typically require an integrated use of these techniques. For example, syntax analysis often requires a preliminary lexical analysis [87]. Similarly, semantic analysis typically relies on syntax analysis as a prior phase [110]. Taint analysis, in turn, depends on DFA to track the propagation of tainted data through the program [8]. However, some studies focus on the explicitly described analysis techniques without directly detailing how these auxiliary techniques and their interconnections were employed.

Section 5.1 covers the papers that exclusively applied static analysis techniques for IoT security. Hybrid methods combining static and dynamic analysis are discussed in Section 5.2, and works on static analysis with ML/NLP are in 5.3. Although these studies integrate different methods, our discussion focuses on the static analysis techniques. Techniques that use other hybrid approaches will not be detailed as they are outside the scope of this study.

5.1 Static Analysis Techniques in IoT Security

The analyzed studies include a variety of contributions, such as the development of tools, frameworks, or complete systems, as well as detailed analyses and the creation of innovative models, methods, and approaches. Table 5 presents such studies, the frequency of the techniques employed, and the corresponding research proposals.

The **CFA** technique is the most frequent among the approaches described. Several articles highlight using CFGs to understand the paths the data can follow through the program [33, 36, 69]. The study by Choi et al. [23] uses **Call Graphs** (**CGs**) to evaluate the maximum stack memory usage, analyzing which routes consume the most stack memory. In another approach, Nino et al. [82] constructed CGs in Jimple IR, performed backward DFA to trace string construction, and applied forward simulation to recover final URL values from program traces. Other articles, such as the

Table 5. Distribution of Static Analysis Techniques in IoT Security Research

	Static Analysis Techniques									
Authors	Control Flow Analysis	Data Flow Analysis	Syntax Analysis	Taint Analysis	Semantic Analysis	Symbolic Analysis	Lexical Analysis	Points-to Analysis	Not Available	**Proposal**
Alnaeli et al. [9]			✓				•			Analysis
Rodríguez-Mota et al. [105]									✗	Tool
Costin [27]	•	•	✓	✓	•		•			Tool
Okano et al. [86]					•	✓				Approach
Choi et al. [23]	✓	•								Method
Alnaeli et al. [10]			✓				•			Analysis
Carvalho and Eler [15]									✗	Analysis
Celik et al. [16]	✓	✓	•	✓	•		•			Tool
Celik et al. [17]	✓	✓	•	✓	✓	✓	•			System
Ferrara and Spoto [36]	✓	✓		✓						Analysis
McBride et al. [74]									✗	Analysis
Parizi et al. [94]	✓	✓	✓	✓	✓	✓	✓			Analysis
Poroor [96]									✗	Approach
Hur et al. [49]									✗	Method
Dong et al. [33]	✓	✓	•		•	✓	•	✓		Model
Schmeidl et al. [108]	✓	✓	•	✓	•					Tool
Nobakht et al. [84]	✓	✓								Tool
Stievenart and Roover [114]	✓	✓		✓						Approach
Dejon et al. [29]	✓	•	•		•	✓				Framework
Sachidananda et al. [106]	•	•		•						Framework
Bodei and Galletta [14]	✓	✓	✓	✓	✓		•			Framework
Chang et al. [18]			✓							Method
Ferrara et al. [35]	✓	✓		✓	✓					Tool
Mandal et al. [70]	✓	✓		✓	✓					Framework
Yoda et al. [129]	✓	✓								Method
Ali et al. [8]	✓	•	✓	✓			✓			Tool
Al-Boghdady et al. [5]									✗	Analysis
Balliu et al. [12]		✓	✓		✓					Framework
Lyu et al. [68]			•		✓	✓	✓			System
Sivakumaran and Blasco [109]	✓		✓		✓		•			Framework
Son et al. [110]	•	•	•	•	✓		•			Approach
Xie et al. [124]	✓	•				✓				Tool
Klima et al. [57]	✓	•	•		✓		✓			Analysis
Nazzal and Alalfi [79]	•	✓	✓	✓	•		•			Tool
Oliveira and Mattos [87]			✓				•			Tool
Paccamiccio and Mostarda [93]	✓	✓						✓		Method
Sutter and Tellenbach [116]	✓	•		•						Framework
Alalfi et al. [6]	•	•	✓		✓		•			Tool
Alhanahnah et al. [7]	✓	✓	✓		•					System
Cheminod and Seno [19]									✗	Model
Inácio and Medeiros [50]	•	✓	✓	✓	•	•	•			Tool
Yoda et al. [128]									✗	Tool
Liu et al. [65]	✓	✓	•	✓	•					Tool
Yuan et al. [130]	✓	✓				✓		✓		Tool
Delaitre and Pulgar Gutiérrez [30]	•	✓	✓	•	✓		•			Tool
Rai and Grover [100]	•		•		•	•	•	•		Analysis
Hassan et al. [47]	•									Approach
Bianco et al. [13]	•				•					Tool
Nino et al. [82]	✓	✓		✓		✓				Tool
Martínez-González et al. [72]									✗	Analysis
Ma et al. [69]	✓	✓	✓	✓	•		•			Approach
Total	**25**	**22**	**16**	**16**	**12**	**9**	**4**	**3**	**9**	

A ✓ indicates that the approach explicitly uses the corresponding technique, • indicates that the technique is mentioned implicitly, an empty field indicates it is not used, and ✗ means the information is not available.

one by Alhanahnah et al. [7], employed **Inter-procedural Control Flow Graphs** (**ICFGs**), which combine CFGs with CGs for a more comprehensive analysis of the interactions between different parts of the code. In a related direction, Liu et al. [65] constructed ICFGs and derive **API-centric control flow graphs** (**ACFGs**) to extract SSL/TLS API sequences from binaries, applying reduction techniques to improve efficiency and mitigate path-explosion. Yuan et al. [130] analyzed the control flow of MQTT brokers by converting source code to LLVM IR and traversing ICFGs to identify key basic blocks affecting system state.

DFA is used to monitor data movement across different parts of a program [35], helping to detect tainted flows from sources to sinks [79]. This technique is critical for tracking how data propagates within functions that interact with security-critical APIs. For instance, Liu et al. [65] applied DFA to examine the propagation of parameters in functions calling SSL/TLS APIs, enabling the detection of API misuses. Nino et al. [82] described a similar use in the OTACap tool, which traces how sensitive values (e.g., URLs, credentials) propagate through the app's code using backward analysis from network-related sinks (HTTP, FTP, MQTT). Information flow analysis represents a more specialized approach, as demonstrated by Stievenart and Roover [114], which aims to prevent transmitting sensitive data to unintended parts of the system. Furthermore, using information-flow policies, as discussed by Balliu et al. [12], involves implementing rules governing the circulation of sensitive information, thus ensuring that data exchanges between applications do not lead to security vulnerabilities.

Syntax Analysis plays a vital role in code analysis by verifying if the code structure complies with the rules of the language [50]. Several studies demonstrate different applications of this technique. For example, Nazzal and Alalfi [79] used an **Abstract Syntax Tree** (**AST**) to display the program structure, while Costin [27] developed a **Concrete Syntax Tree** (**CST**), which was converted to AST for further analysis. Likewise, Delaitre and Pulgar Gutiérrez [30] extracted ASTs during compilation and used them as part of a semantic graph construction pipeline, integrating syntactic information to support vulnerability detection. A similar syntactic approach is used to parse AST nodes from decompiled firmware code and extract structural elements and parameter keywords, which assist in identifying dangerous back-end functions [69]. Furthermore, Balliu et al. [12] set syntax-based conditions to support secure application interactions, improving compliance with security policies. Syntax analysis methods have also been enhanced by analyzing code structure: Alnaeli et al. [10] used SrcML to represent code syntax in XML, and Ali et al. [8] generated an XML tree from the syntax of Solidity code. From another perspective, Bodei and Galletta [14] defined the syntax for *terms*, *values*, and *sensors* to predict data paths in IoT systems. Several studies have used automated tools, such as [87], which employed JSGuide to generate an AST from JavaScript code and detect *code smells*, and [18], which used CodeNarc to analyze the code structure with AST. IoTCom was used in [7] to scan the AST and extract critical elements such as API method calls (*subscribe, schedule, runIn, runOnce*). Finally, the ChYP approach was applied in [6] to identify excessive privileges and monitor complex functions and loops.

Taint Analysis is essential for identifying sensitive data streams. It operates by marking input values as tainted and tracking their influence through the application to detect if they reach defined sinks, revealing potential data exposure [82]. According to Celik et al. [16], the SAINT tool was designed to meet this requirement. Ma et al. [69] applied lightweight taint analysis during static analysis to trace the dataflow along function call sequences and verify whether tainted parameters can lead to unsafe function execution. Liu et al. [65] integrated taint analysis into their detection mechanism to identify both T-II.a (API argument misuse) and T-II.b (API execution result verification misuse) vulnerabilities by performing backward analysis to trace the source of API arguments and forward analysis to track the use of API results. However, the SOTERIA system, by Celik et al. [17], performs reverse taint analysis to identify sources that may affect changes to numeric attributes in

an application's state model. Tools such as AndroGuard and Exodus have been proposed to identify advertising trackers and permissions in pre-installed applications [116]. Taint analysis has also been used to support GDPR compliance by detecting potential data breaches in the build process [36]. Furthermore, this technique has been applied to identify vulnerabilities in WebAssembly programs [114], Smart Contracts [8], SmartThings IoT applications [108], and in IoT APKs [106].

Semantic Analysis is essential for verifying syntactic correctness and context in programming languages. In this regard, Lyu et al. [68] used this technique to detect flaws and accurately understand function meanings. Klima et al. [57] explored identifying code duplication, including semantic duplication, where blocks of code have the same function but different syntax. The study by Son et al. [110] developed a taxonomy to analyze the semantic similarity between various data items collected by Android applications. Additionally, the literature mentions several tools, such as the Julia static analyzer, that aid in deep semantic analysis to identify security flaws [70]. Another example is the DocSpot tool, which incorporates a semantic analyzer to examine application logic and identify issues such as insecure method usage, using a semantic graph representation of the code [30]. Finally, Sivakumaran and Blasco [109] discussed the Jira semantic tool, which computes jump tables in ELF binaries using techniques similar to those employed by the ArgXtract tool for binary analysis.

Symbolic Analysis uses symbolic values to represent variables and perform program or system analysis. This technique is widely applied in both dynamic and static analysis. However, the main focus is static analysis, which does not require code execution. Yuan et al. [130] applied symbolic execution to prune unreachable paths and enforce constraints over extracted control flows, ensuring that only feasible execution paths are modeled in formal security verification. Symbolic execution has also been used to generate concrete value candidates for tainted variables by solving execution path constraints; e.g., a variable like *LocalSysver* may yield six possible values from six constraints [82]. In more specialized applications, Dejon et al. [29] presented the IoTAV framework, which validates security policies by analyzing the binary code of IoT applications. To this end, the framework uses the Angr platform, which performs symbolic execution to identify potential vulnerabilities. On the other hand, Xie et al. [124] introduced the RTOSExtracter tool, which employs the Unicorn simulation execution technique[3] to generate parameter values from statically extracted code paths.

Lexical Analysis, essential for converting source code into tokens, is the first stage of code analysis. According to Klima et al. [57], this step improves code readability. For example, Ali et al. [8] used ANTLR to tokenize Solidity code, while Lyu et al. [68] applied CharBPETokenizer to split function names into subwords, making code easier to understand and annotate.

Techniques such as **Points-to Analysis** were used in the studies of Dong et al. [33] and Paccamiccio and Mostarda [93] to identify which pointers reference memory blocks, improving the accuracy of code analysis and detecting problems such as memory leaks. This type of analysis has also been applied to map how pointers influence state variables in the source code, using context-sensitive techniques implemented through the SVF tool [130].

At this point, we should highlight some studies from the broader set in the table to illustrate the application of the static analysis techniques discussed above. Parizi et al. [94] analyzed the effectiveness and accuracy of static security testing tools, which apply various static analysis techniques to identify vulnerabilities in Ethereum smart contracts written in Solidity. In contrast, the study by Chang et al. [18] proposed a method to evaluate the quality of SmartThings applications, using syntax analysis to examine the code structure and promote compliance with coding standards. Additionally, it is essential to consider research such as that of McBride et al. [74], which, despite not explicitly mentioning any specific technique, used static analysis tools to evaluate the security

[3]Unicorn Engine: A Platform for Code Simulation, available at https://www.unicorn-engine.org/.

structure of the Contiki IoT operating system. Finally, both [33] and [50] used seven static analysis techniques on C code. Dong et al. [33] developed a model to improve memory leak detection, while Inácio and Medeiros [50] created a tool to detect and fix BO vulnerabilities automatically.

To better understand how techniques are applied across different targets and contexts, the previous comparison highlighted key patterns and the rationale behind their adoption. The use of static analysis techniques in IoT security varies according to the analysis target. Studies focused on source code frequently adopt syntax, taint, and semantic analysis, as these leverage structured representations to detect insecure functions, data leaks, and logic flaws. In contrast, firmware and binary analyses often rely on control flow and symbolic analysis to compensate for the absence of metadata and source context, enabling the reconstruction of low-level logic and exploration of execution paths. *Points-to* analysis also appears in binary-level studies to resolve indirect references. These trends indicate that the choice of techniques is closely aligned with the software layer and the nature of the available artifacts.

Appendix B: Table 8 lists studies that exclusively use Static Analysis techniques for IoT security, providing a detailed overview of the methods and tools used to identify vulnerabilities and highlights each paper's approaches, application domains, and contributions.

5.2 Integration of Static and Dynamic Analysis

Some studies demonstrate the effectiveness of combining static and dynamic analyses in identifying vulnerabilities. Table 6 presents an overview of the static analysis techniques employed in those studies, highlighting the predominance of CFA. However, dataflow and points-to analysis were not explicitly referenced. Furthermore, although all studies incorporate dynamic analysis, several primarily rely on fuzzing techniques to uncover vulnerabilities.

CFA is used to understand the possible program execution paths [63]. Chen et al. [20] applied this technique to construct the CFG and CG of firmware code, while He et al. [48] used CFG generation to identify functions that perform vulnerable operations. Zhang et al. [131] advanced this line of work by reconstructing the CFG directly from PLC binary programs, combining static analysis with simulated execution logic to deduce indirect jumps and link code blocks. Several tools have integrated these analyses, such as Liu et al. [64], which employed FlowDroid to map the interactions between application functions and the firmware. In contrast, Classen et al. [25] used IDA Pro to examine binaries of different versions of Fitbit. Furthermore, Fowze and Yavuz [37] introduced the SEESAW tool, which employs graph analysis to evaluate interrelationships within code components and uses **Points-to Analysis** to determine the possible values pointers can reference.

DFA was used in Qin et al. [99] to trace data movement in back-end code, identify constraints, and produce efficient fuzzing test cases for SOHO routers. The DIANE tool was employed in Redini et al. [101] to detect potential fuzzing triggers that could bypass application validation and access the IoT device. Similarly, Yao et al. [127] presented the prototype tool Aric, which uses this technique to extract program execution paths and evaluate behavior under varying conditions. In **Taint Analysis**, Zheng et al. [132] proposed a gray-box fuzzing technique to detect vulnerabilities by tracking infected variables in instructions and functions. Srivastava et al. [112] introduced the FirmFuzz tool for intraprocedural taint analysis on PHP scripts to monitor user-controlled variables and locate insecure functions. Guo et al. [46] presented the iootfuzzer prototype, which analyzes the external input data stream into firmware binaries of IoT devices based on ARM and MIPS architectures.

Syntax Analysis was used in Ding et al. [32] to validate syntactic rules related to *device IDs*, *states*, and *trigger conditions*, ensuring compliance with security policies. Similarly, Long et al. [66] used the JADX tool to decompile APK files and convert the code to Java, facilitating the structural

Table 6. Integration of Static Analysis Techniques with Dynamic Analysis Approaches in IoT

	Static Analysis Techniques									
Authors	Control Flow Analysis	Data Flow Analysis	Taint Analysis	Syntax Analysis	Semantic Analysis	Symbolic Analysis	Points-to Analysis	Lexical Analysis	Not Available	**Proposal**
Ge and Xue [40]									✗	Method
Neugschwandtner et al. [80]	✓	•				•				System
Mangano et al. [71]									✗	Approach
Classen et al. [25]	•	•								Analysis
Liu et al. [63]	✓									Method
Peyrard et al. [95]	•	•			•	•				Approach
Visoottiviseth et al. [119]				•				•		Tool
Chen et al. [20]	✓									Tool
Redini et al. [102]	✓	✓	✓	✓	✓	✓		•		Approach
Yao et al. [127]	•	•	✓		•					Tool
Ding et al. [32]	✓	•		✓						System
Fowze and Yavuz [37]	✓	✓					✓			Tool
Guo et al. [45]									✗	Method
He et al. [48]	✓	•								System
Liu et al. [64]	✓	✓								Framework
Long et al. [66]	•	•		•						Analysis
Kapoor et al. [54]									✗	Analysis
Ajith et al. [3]									✗	Analysis
Nyzhnyk et al. [85]									✗	Analysis
Zhang et al. [131]	✓	•			✓	•				System
Srivastava et al. [112]✻			✓							Framework
Sachidananda et al. [107]✻									✗	Framework
Zheng et al. [132]✻	✓	✓	✓				•	•		Method
Robin et al. [104]✻	•	•	•							Approach
Redini et al. [101]✻	✓	✓				✓				Tool
Gao et al. [39]✻									✗	Approach
Guo et al. [46]✻	•	✓	✓	✓						Tool
Qin et al. [99]✻	✓	✓	•		✓					Tool
Li et al. [61]✻	•	•		•	•		•			Method
Total	**12**	**7**	**5**	**3**	**3**	**2**	**1**	**0**	**8**	

A ✻ indicates that the article uses dynamic analysis with fuzzing, ✓ indicates that the approach explicitly uses the corresponding technique, • indicates that the technique is mentioned implicitly, an empty field indicates it is not used, and ✗ means the information is not available.

understanding of applications. Visoottiviseth et al. [119] employed tools such as PHP_CodeSniffer and PHPCS-Security-Audit v2, which combine **Lexical Analysis** and Syntax Analysis to verify the structure of PHP code and identify syntax-related vulnerabilities.

Qin et al. [99] used the **Semantic Analysis** technique to extract detailed information from the back-end code, which allowed generating *seeds* that can bypass front-end checks and explore deeper paths in the code, making fuzzing more efficient. Furthermore, Peyrard et al. [95] used the same technique to verify the absence of runtime errors in the AES and CCM cryptographic modules within Contiki.Zhang et al. [131] further extended the use of semantic analysis by converting PLC binary instructions into the VEX intermediate representation and then into the *.smv* formal verification language, preserving the semantics of the original binary for accurate formal verification of control logic. On the other hand, Neugschwandtner et al. [80] applied **Symbolic Analysis** to resolve symbols in executable binaries, focusing on return addresses. The study verified whether these addresses were correctly exported or imported by the binary, ensuring they were adequately handled during execution.

To illustrate the integration of the aforementioned techniques with dynamic analysis, we examine a selection of relevant studies. Redini et al. [102] developed an approach that identified insecure interactions between different firmware binaries using a combination of static analysis techniques. Although lexical analysis was not explicitly mentioned in this study, it facilitated the segmentation of binaries into smaller components. On the other hand, Liu et al. [63] proposed an automated method that integrates CFG with **Dynamic Symbolic Execution** (**DSE**) to detect vulnerabilities in embedded devices.

An approach developed by Robin et al. [104] implicitly integrated several techniques through the Klocwork tool to detect flaws in source code and, subsequently, employed dynamic testing to validate these findings in the compiled software. Finally, Sachidananda et al. [107] introduced the modular PIT framework, which combines Penetration Testing, Static Analysis, Fuzzing, and Dynamic Analysis to identify vulnerabilities at multiple layers, such as application, communication, and hardware.

This synthesis highlights key usage trends and the rationale behind selecting techniques across different analysis contexts. The type of artifact often drives the integration of static and dynamic analysis in studied IoT security. In firmware and binary analysis, control flow and taint analysis are commonly applied to trace execution paths and monitor data propagation, particularly in stripped or undocumented code. Combined with fuzzing or DSE, these techniques help uncover vulnerabilities such as BO and unsafe input handling. In contrast, source code analysis employs syntax and semantic techniques to verify structural and logical correctness before deployment. The combination of approaches mitigates the limitations of individual techniques and enhances the overall coverage of vulnerability detection in complex and resource-constrained IoT environments.

Appendix C: Table 9 presents the research that combines static analysis techniques with dynamic analysis approaches for IoT security. This table details studies that use different methods and tools to identify vulnerabilities, providing information about the approaches adopted, the focus of the studies, and their specific contributions.

5.3 Integration of Static Analysis, Machine Learning, and Natural Language Processing

Several studies integrated static analysis with ML and NLP to improve IoT vulnerability detection. Table 7 summarizes the studies, emphasizing the static analysis techniques employed. Syntax analysis appeared in seven studies, followed by semantic and control flow analyses, each present in six. Taint analysis was applied in five studies, while lexical and symbolic analyses were the least

Table 7. Integration of Static Analysis Techniques with ML Algorithms and NLP in IoT

	Static Analysis Techniques									
Authors	Syntax Analysis	Semantic Analysis	Control Flow Analysis	Taint Analysis	Lexical Analysis	Symbolic Analysis	Data Flow Analysis	Points-to Analysis	Not Available	**Proposal**
Cui et al. [28]									✗	KNN, LR, RF, DT, SVM, GBDT
Niu et al. [83]	✓	✓	✓	✓	✓	✓	•			RNN, LSTM, BLSTM, CNN-BLSTM, Word2Vec
Luo et al. [67]	✓	✓	✓		•					CNN, Part-of-Speech (POS) Tagging, Word2Vec
Al-Boghdady et al. [4]				•	•					RF, CNN, RNN, Tokenization, TF-IDF
Cheng et al. [22]			✓	✓						BiLSTM
Yang et al. [125]	✓	✓	✓	✓	✓	✓				FNN, SVM, DNN, GAN, MCTS
Yang et al. [126]	•	•								SVM, CNN, Word2Vec
Minani et al. [75]									✗	LLM(ChatGPT-4)
Zhou et al. [133]	✓	✓			•					LLM(ChatGPT-4)
Maruf et al. [73]	✓	✓								LLM(BERT/CodeBERT), Tokenization, TF-IDF
Chu et al. [24]	✓	✓			•					LeakGAN, SVM
Chen et al. [21]	✓		✓	✓			•			LLM(ChatGPT-4)
Xiang et al. [122]		•	✓	✓			•			GRU, Structure2Vec, Siamese Network, Word2Vec
Total	**7**	**6**	**6**	**5**	**2**	**2**	**0**	**0**	**2**	

A ✓ indicates that the approach explicitly uses the corresponding technique, • indicates that the technique is mentioned implicitly, an empty field indicates it is not used, and ✗ means the information is not available.

frequent, each referenced in two. Although dataflow and points-to-analysis were less frequently and explicitly referenced, the studies implemented various ML and NLP algorithms, including **convolutional neural networks (CNNs)** and **support vector machines (SVMs)**, and more recently, **large language models (LLMs)** such as BERT and GPT-4.

Syntax and Lexical Analysis were applied to examine the grammatical structure of Diff files [83], verify trigger-action dependencies and *if* conditions in the source code [67], and convert the code into a sequence of tokens and recognize its syntactic structure [125]. Syntax analysis has also been used to parse Lua code into ASTs, providing structured representations of functions and statements for further analysis [21]. In the context of Solidity, ASTs were generated to capture the hierarchical syntactic organization of smart contracts [24, 75]. Similarly, Python's AST module segmented non-modular code into syntactic units, enabling structured analysis and modularization [73].

Complementing these approaches, Luo et al. [67] used **Semantic Analysis** to compute the similarity between sentences in application descriptions and input prompts. In contrast, Niu et al. [83] adopted *Word2Vec embeddings* to capture the semantic meaning and relationships between different parts of the code. Chu et al. [24] further explored semantic relationships by analyzing smart contracts by extracting logical links between code components and their intended functionality. Minani et al. [75] incorporated GPT-4 to interpret pseudocode semantically, evaluating logical consistency and coherence as part of the analysis process. Additionally, semantic analysis using BERT-based models has been applied to understand contextual meaning in source code and comments, supporting the extraction of functional and security-relevant information [73].

CFA was used by Niu et al. [83] to describe the construction of a CG based on the relationships between function calls within the program. Luo et al. [67] employed an ICFG to map the event paths that trigger action commands, and control flow was also explored through the construction of CFGs from ASTs, traversing syntactic nodes to establish execution order and control structures within Lua code [21]. Xiang et al. [122] introduced **CISCFG** (**Command-Injection-Suspected CFG**), a CFG variation tailored to command injection detection, generated from firmware binaries using backward taint analysis to isolate exploit chains.

Additionally, **Taint Analysis** was applied to trace the dataflow in the program, identifying which parts of the code are influenced by external inputs [22, 125]. Chen et al. [21] proposed a static taint analysis framework that identifies *sources, sinks, and sanitizers*, and performs backtracking based on the arguments of sensitive functions. The analysis includes field-sensitive tracking by mapping keys in data structures to their corresponding values, allowing precise differentiation between trusted and untrusted inputs. Backward taint analysis was also used to trace user-controlled data from vulnerable sinks to their origins, narrowing the focus to potentially exploitable paths [122]. Both Niu et al. [83] and Yang et al. [125] used **Symbolic Analysis**. Niu et al. [83] used this technique to transform taint propagation paths into symbolic representations. In contrast, Yang et al. [125] employed it to verify the AST model of the code and compare it with known vulnerability patterns.

A selection of relevant studies is reviewed to better understand the techniques used. Niu et al. [83] presented a method that combines static analysis techniques with deep learning models, specifically CNNs and **Bidirectional Long Short-Term Memory** (**BLSTM**), to improve the accuracy in identifying vulnerabilities and reducing false positives and negatives. On the other hand, Luo et al. [67] employ NLP methods to extract sensitive data and match application fingerprints. The study used the *SpaCy Library* to perform ***Part-Of-Speech (POS)*** *tagging*, analyze word relationships, and compare word vectors generated by *Word2Vec*.

The approaches presented by Al-Boghdady et al. [4] and Yang et al. [126] used static analysis techniques implicitly. The development of the iDetect tool by Al-Boghdady et al. [4] uses the RF algorithm in combination with NLP techniques such as tokenization to divide source code into smaller parts and **Term Frequency-Inverse Document Frequency** (**TF-IDF**) vectorization to analyze the frequency and importance of each token. Similarly, the **Secure Coding Support Vector Machines** (**SCSVM**) method proposed by Yang et al. [126] integrates NLP techniques such as *Word2Vec* with ML models such as SVM and CNN to improve vulnerability detection in Smart Contracts.

Zhou et al. [133] and Maruf et al. [73] applied LLMs to enhance static analysis through syntax and semantic understanding. Zhou et al. [133] used GPT-4 to analyze pseudocode derived from Solidity contracts, improving vulnerability detection by combining direct analysis with iterative refinement of static analysis results. In contrast, Maruf et al. [73] used BERT and CodeBERT to extract features and requirements from Python code and documentation using tokenization and TF-IDF, supporting code modularization through contextual embeddings.

Finally, both Cui et al. [28] and Minani et al. [75] explored the use of ML and NLP techniques to enhance software quality and security. Cui et al. [28] developed a prediction model that uses ML algorithms such as **Random Forest (RF)**, **Decision Tree (DT)**, and **Gradient Boosted Decision Trees (GBDT)** to classify security risks in Android applications based on static code metrics. Furthermore, the article also discusses additional techniques, including **K-Nearest Neighbors** (**KNN**), **Logistic Regression** (**LR**), and SVM, to assess the risk levels of these applications. In turn, Minani et al. [75] applied an LLM (ChatGPT-4) to automatically generate test scripts for IoT wearable applications lacking manual tests. The model iteratively analyzed the source code and

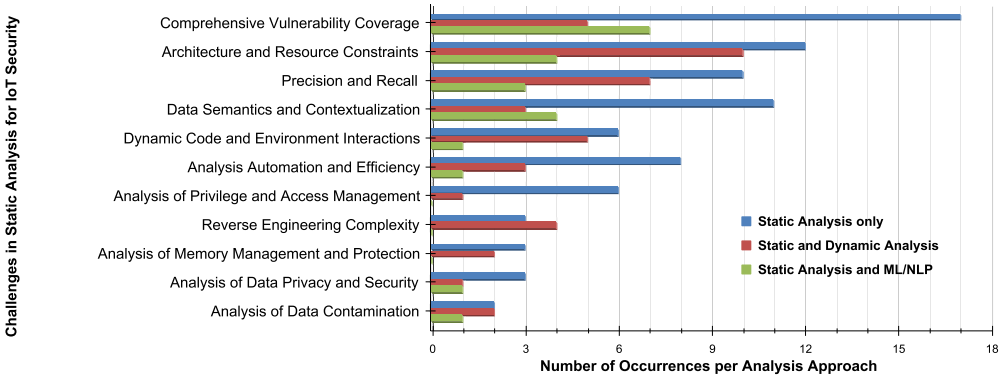

Fig. 6. Distribution of challenges by analysis approach.

refined the generated test cases until achieving full code coverage, demonstrating its potential to address testability challenges in resource-constrained environments.

The comparison above summarizes key trends and justifications for applying each technique across different analysis scopes. The integration of static analysis with ML/NLP in IoT security reveals a clear alignment between techniques and their evaluation goals. Syntax and lexical analyses are commonly used when transforming source code into structured or tokenized formats, enabling effective input for ML models and embeddings. Semantic and control flow analyses are applied to capture application logic and behavior, supporting accurate classification and vulnerability localization. Although less frequent, taint and symbolic analyses are valuable in firmware scenarios to trace dataflows or emulate execution paths in the absence of source-level context. These combinations indicate that the selection of static techniques depends not only on the software layer (e.g., source code vs. firmware) but also on the specific nature of the ML/NLP task (e.g., pattern recognition, semantic matching, or vulnerability inference).

Appendix D: Table 10 presents the papers that integrate static analysis with ML and NLP, showing approaches, focuses, and contributions of the studies that employ various methods and tools to identify vulnerabilities.

6 RQ3. What are the Challenges of Static Analysis for IoT Security?

Static analysis plays a crucial role in identifying vulnerabilities in IoT devices. Still, it faces significant challenges due to the complexity of these systems, such as the diversity of devices and their dynamic interactions. These obstacles have encouraged researchers to explore solutions beyond static analysis through hybrid approaches integrating dynamic analysis, ML, and NLP. Figure 6 illustrates the main security challenges discussed in papers that use static or hybrid analysis approaches, either implicitly or explicitly, highlighting the frequency with which each challenge is mentioned.

The challenges of static analysis in IoT security stem from the technique's inherent characteristics (e.g., comprehensive vulnerability coverage; architecture and resource constraints) and the complexity of addressing specific vulnerability types (e.g., analysis of privilege and access management; analisis of data privacy and security). The predominant challenges, *Comprehensive Vulnerability Coverage* and *Architectural and Resource Constraints*, were mentioned in 29 and 26 studies, respectively, highlighting their relevance to IoT security. In contrast, challenges such as *Analysis of Privilege and Access Management*, *Reverse Engineering Complexity*, *Analysis of Memory Management and Protection*, *Analysis of Data Privacy and Security*, and *Analysis of Data Contamination* were mentioned in only seven or five studies, suggesting that, while pertinent, they emerge in specific contexts.

The lack of a frequent reference to specific challenges in hybrid approaches, such as *Analysis of Privilege and Access Management*, should not be interpreted as a complete resolution of these issues. This gap may reflect the lack of explicit discussion of these topics in the reviewed studies, suggesting that they may not have been directly addressed. Therefore, a more in-depth analysis of these challenges is essential for a more comprehensive understanding of the obstacles that impact the effectiveness of these approaches.

The following sections explore each of the challenges identified in the reviewed works, organizing the discussion into three parts (as applicable): (1) the challenge as addressed in works that focus exclusively on static analysis, (2) the challenge as approached in works integrating static and dynamic analysis, and (3) the challenge as discussed in works combining static analysis with ML and/or NLP.

Appendix E: Table 11 provides an overview of the challenges in static analysis for IoT security, highlighting the papers that address each challenge and the corresponding analysis approaches employed.

6.1 Comprehensive Vulnerability Coverage

Comprehensive vulnerability coverage identifies multiple vulnerabilities in systems and code, including known security flaws and emerging threats.

Static Analysis. Static analysis employed in the IoT context may not successfully detect all potential security vulnerabilities. According to Bodei and Galletta [14], challenges include the inability to scan native libraries [116], the complexities associated with removing known insecure commands and functions [9, 10], and the identification of vulnerabilities in mixed programming styles within a single logical block [57]. Furthermore, detecting complex threats usually requires advanced techniques [105]. For example, second-order vulnerabilities are difficult to identify because they involve storing an exploitation payload, which is later used in a security-sensitive operation [27]. Extracting relevant information from large codebases, such as MQTT brokers, is challenging: insufficient detail may prevent the identification of vulnerabilities, while excessive information can lead to false positives [130].

Paccamiccio and Mostarda [93] point out that concrete execution may not provide complete code coverage. On the other hand, symbolic execution may not be suitable for specific tasks due to its inherent complexity. Furthermore, software logical evaluation often relies on model checking, which can lead to inconsistencies between the model and the implementation.

According to Sachidananda et al. [106], the effectiveness of static analysis tools may be limited by the need for specialized security expertise. Furthermore, these tools may fail to identify vulnerabilities present in IoT applications [15, 87]. Tradeoffs in the functionalities of these tools may also compromise complete coverage of security issues [79], often requiring multiple tools to cover IoT vulnerabilities. For instance, Karonte detects only a subset, such as BO and DDoS attacks, which limits comprehensive coverage [128]. In IoT, static analysis must address vulnerabilities across code, dependencies, and configurations, requiring tools that handle heterogeneous data without missing edge cases [30]. In blockchain-based smart contracts, using practical tools and methodologies to identify vulnerabilities is essential [94]. However, Ali et al. [8] highlight that their tool relies on vulnerability patterns specific to the Ethereum community, which limits its ability to detect zero-day exploits.

Static and Dynamic Analysis. Identifying vulnerabilities related to sensitive data, such as weak passwords, is particularly challenging because the static analysis does not capture runtime data handling [40, 119]. Also, it is difficult to detect vulnerabilities that occur only during runtime or when the application is processing data [66]. Static analysis tools also face limitations in accuracy

and efficiency, generating a large volume of alerts per device, which requires significant processing time and makes it challenging to identify vulnerabilities accurately [102, 112].

Static Analysis and ML/NLP. Static analysis of smart contracts can identify vulnerabilities and security weaknesses, but it does not cover all attack vectors or complex vulnerabilities in IoT environments [126]. While the Slither tool detects many issues in smart contracts, it misses or misclassifies others, requiring complementary checks; integration with LLMs helps cover vulnerabilities beyond its native scope, such as those aligned with SWC standards [133]. The effectiveness of such analyses is limited by the reliance on non-commercial SATs, which may fail to detect vulnerabilities and generate false negatives [4, 28]. Another limitation is the lack of a robust database that relates versions and vulnerabilities, which is essential due to inconsistencies between the **Common Platform Enumeration** (**CPE**) and the current of vulnerable versions [22]. Tools like SonarQube, although used to assess code quality and security in IoT applications, fail to detect certain vulnerabilities and error-prone segments [75]. In firmware analysis, static methods struggle with vulnerabilities that span multiple functions or involve complex exploit chains, making detecting certain command injection paths difficult, weakening the overall coverage [122].

6.2 Architecture and Resource Constraints

This section discusses the challenges posed by architectural and resource constraints in IoT systems, including limited processing, memory, and storage, as well as heterogeneity of hardware, operating systems, and protocols.

Static Analysis. Static analysis techniques struggle to identify vulnerabilities in IoT devices due to their complexity and diversity. The variety of devices and protocols makes it difficult to accurately and efficiently detect vulnerabilities [19, 87, 114]. Limitations in processing power, memory, and difficulty in updates [100], combined with the nature of components such as kernels, protocol stacks, APKs, and firmware, intensify the analysis challenges [106].

Static analysis tools such as SAW face challenges in analyzing programs containing loops, which is also relevant in IoT software development [86]. Furthermore, using APIs in IoT programming makes it challenging to identify excessive privileges [84]. According to Poroor [96], optimizing power consumption in IoT devices poses a challenge for static analysis, requiring the identification of code inefficiencies that impact runtime costs. Furthermore, covering all potential environmental conditions in an IoT environment can be complex with conventional methods [93].

Specific challenges involve automating the identification of task creation APIs in RTOS firmware, which include complex parameters like task addresses, names, stack sizes, and priorities [124]. Limitations in modeling physical interactions at the application level result in overapproximation of physical channels, thus impacting accuracy [7].

Static and Dynamic Analysis. Static analysis of IoT devices is limited by the complexity of dealing with target binaries, which makes it challenging to identify vulnerabilities effectively [20]. Such analysis faces several challenges, including the difficulty in collecting information from files [40]; limitations in detecting vulnerabilities at the IoT system layers [107]; and the inability to provide a complete understanding of the energy optimization needs of IoT security algorithms [45].

Some works highlight limitations associated with the firmware on the devices [101]. For example, the effectiveness of static analysis for identifying vulnerabilities in Fitbit trackers is limited by the need for in-depth knowledge of the memory architecture and chip design [25]. According to Yao et al. [127], the static analysis tool Aric is restricted to ARM firmware and requires specific operating and programming conditions. Furthermore, variations in ARM, MIPSEL, and MIPSEB

architectures, especially in devices such as cameras and routers, impose additional constraints due to the emulation techniques employed [46, 132]. Similar architectural limitations are observed when integrating modern cloud-based security solutions with legacy SCADA and IIoT systems, due to differences in system design and resource constraints [85].

Static Analysis and ML/NLP. Implementing static analysis in IoT is hampered by code complexity and state space expansion in large-scale detection programs [125]. Furthermore, this approach must be adjusted to address all security concerns, considering the dynamic nature of IoT environments [126]. Finally, static analysis may result in incomplete vulnerability identifications in complex IoT systems with interconnected components, as it may not capture all interactions between system parts [83]. These challenges are amplified in embedded systems that integrate ML and fog computing, where legacy and inflexible architectures limit the adaptability of static analysis techniques [73].

6.3 Precision and Recall

This section discusses the challenges in improving precision and recall when applying static analysis to IoT systems. These challenges stem from the prevalence of false positives and false negatives, which hinder accurate vulnerability detection and compromise the reliability of the analysis results.

Static Analysis. Static analysis tools for IoT face significant challenges, mainly related to false positives and negatives that prevent accurate vulnerability detection [8, 36]. These false positives, in particular, avoid identifying real vulnerabilities in the source code [30, 50]. A specific problem occurs when the design of the tools leads to the incorrect classification of all external HTTP calls as violations [18]. The SAINT tool exemplifies this problem by considering all user input and state variables tainted, compromising the vulnerability identification accuracy [16].

According to Al-Boghdady et al. [5], the Cppcheck tool has limitations in vulnerability detection, while Flawfinder generates several false positives due to its analysis based on plain text patterns. These accuracy issues undermine the effectiveness of vulnerability analyses, as highlighted by Hassan et al. [47], Nazzal and Alalfi [79]. Furthermore, inconsistencies between the results produced by different tools and studies hinder consensus on bug density, making accurate assessments in static analysis for IoT more challenging [74].

Static and Dynamic Analysis. Existing static analysis techniques often have high false positive and false negative rates, compromising vulnerability detection accuracy [48, 112, 119]. Furthermore, there is an overreliance on manual rules, which makes it challenging to balance accuracy and efficiency [39]. According to Qin et al. [99], false positives require considerable manual effort to verify them and to develop **proof-of-concept** (**PoC**) scripts.

Another challenge is the context limitation of static analysis, which can generate false positives when detecting memory errors due to imprecise calling contexts that do not accurately represent all execution paths [37]. On the other hand, false negatives arise when interactions are captured by dynamic testing but not detected by static analysis, creating potential unnoticed security risks [32].

Static Analysis and ML/NLP. According to Cui et al. [28], static analysis tools can generate false positives, wasting resources for non-existent security vulnerabilities. These challenges also affect smart contracts and firmware. In smart contracts, false positives and negatives hinder detection accuracy, despite efforts to improve results using LLMs [133]. In firmware, overtainting and undertainting impact precision and recall. While techniques like field-sensitive analysis and LLM pruning help, balancing both metrics remains challenging [21].

6.4 Data Semantics and Contextualization

Analyzing data semantics and contextualization in IoT systems is challenging due to the difficulty in capturing code's meaning and behavior, especially in ambiguous structures, implicit flows, and a lack of contextual information.

Static Analysis. Ambiguity in natural language function names makes it difficult to determine semantic meaning automatically [68]. Furthermore, complex logical structures prevent automated tools from recognizing the core functionality of a function. Static analysis tools often miss implicit behaviors like automatic *toString()* calls and common string operations (*replace, split*), hindering accurate reconstruction of runtime values [82]. More broadly, the focus of static analysis on syntax can lead to an inadequate understanding of software semantics [6]. Another challenge is capturing semantic interactions between actions and triggers, which exceed syntactic correspondence and can compromise system security [12]. Similarly, identifying data entry points in embedded web services depends on understanding implicit frontend/backend interactions, requiring context-aware DFA [69].

The complexity of domain-specific languages and their limited testability make it challenging to create reliable smart contracts [94]. Furthermore, improvements in NLP techniques and lexer grammar are needed to increase translation efficiency [7]. Static analysis faces challenges with reflection calls, leading to inaccurate CGs and false positives in property checking [17]. This method can be somewhat conservative in analyzing software components of IoT systems, failing to represent the real environment accurately [70].

There are challenges in recognizing security-related *code smells*, including improper module interaction and improper API versioning [57]. Another notable challenge is the need for input configurations to define secure input sanitization functions since their security cannot be easily verified through static analysis alone [27].

Static and Dynamic Analysis. Static analysis aids dynamic execution but lacks semantic richness regarding program behavior [63]. According to Qin et al. [99], front-end fuzzing methods restrict efficiency and vulnerability discovery due to the limitation of narrow seeds and the absence of semantic data from the back-end code. These constraints arise from challenges in identifying communication interfaces and ensuring the correct composition of data fields. Analysts must also manage the automatic insertion of intrinsics (low-level functions or instructions) to prevent disrupting program semantics [104].

Static Analysis and ML/NLP. Semantic interpretation remains a key challenge in static analysis, especially in heterogeneous and non-standardized code. In smart contracts, redundant code and excessive comments hinder semantic understanding, even with the support of language models [133]. In embedded software, models like BERT help extract contextual information, but varied semantics still limit accurate interpretation [73]. Dynamic languages like Lua complicate DFA in firmware due to flexible typing and syntax [21]. Static analysis of binaries also struggles to preserve semantic context, and although models like Word2Vec and GRU offer improvements, maintaining deep semantic understanding remains difficult [122].

6.5 Dynamic Code and Environment Interactions

A challenge arises from static analysis's inability to deal with execution states and dynamic environmental conditions, such as runtime behavior and interactions between components.

Static Analysis. Static analysis fails to capture the dynamics of system behavior as it focuses on predicting data trajectories and assessing vulnerabilities rather than considering real-time

data interactions [14, 70]. Furthermore, limitations in tracking user input across different layers result in incomplete coverage and decreased accuracy [35]. This challenge is compounded by the inability of current techniques to represent the runtime memory states of IoT applications adequately [33]. Static analysis also fails to capture behaviors dependent on hardware configurations or runtime inputs, such as sensors and networks in IoT systems [65]. In Android applications, implicit dataflows between components, like Intents with global parameters, are likewise difficult to trace statically [82].

Static and Dynamic Analysis. Smart home applications can induce complex interactions that neglect the constraints of the physical environment [32]. In this context, the effectiveness of static analysis techniques for IoT systems is significantly limited by factors such as call stack depth, forwarding emulation, and link address spoofing, making vulnerability detection harder [80]. Many techniques are limited by their focus on individual programs or modules, which restricts the modeling of interactions between binaries [48]. A significant challenge lies in the inadequacy of static analysis to capture runtime behavior [64]. Furthermore, tools often struggle to identify vulnerabilities in IoT devices when they cannot access the firmware [101].

Static Analysis and ML/NLP. Static analysis of IoT applications can have limitations in detecting vulnerabilities at runtime because it cannot capture dynamic behaviors or interactions during execution [83].

6.6 Analysis Automation and Efficiency

This section discusses the challenges of efficiently automating static analysis in IoT systems, as the complexity of code structures, limitations of current tools, and the large volume of heterogeneous data hinder the scalability and practicality of fully automated analysis approaches.

Static Analysis. Static analysis tools face scalability challenges, requiring long timelines for application analysis due to technical limitations [116]. Yoda et al. [128] described how manual steps such as disassembly, configuration extraction, and ELF file handling contribute to inefficiencies in firmware analysis. Additionally, path explosion in IoT binaries makes extracting API call sequences difficult, often causing timeouts during analysis [65]. Loops pose a challenge, as their complexity usually prevents complete static analysis; techniques like loop unrolling can help, but only when iteration counts are known [82]. Ma et al. [69] highlighted the difficulty of balancing accuracy and efficiency in firmware taint analysis, which must remain lightweight without losing precision. Moreover, static tools must adapt to evolving languages and frameworks to handle modern code patterns, while still struggling to explore all execution paths due to complexity and uncommon logic constructs [47]. There is a significant lack of automated tools aimed at the security assessment of smart contracts [94]. For example, the IoTAV tool limits its assessment to firmware images with unextracted executables, restricting the analysis to binaries that contain symbols [29].

Static and Dynamic Analysis. Challenges include difficulties in automating processes, such as creating *harnesses* to prepare test targets and improving compatibility with test frameworks. In addition, using obsolete tools reduces the effectiveness of static analysis in IoT security [104]. Zhang et al. [131] addressed the state explosion problem in PLC binaries, proposing optimizations to improve verification efficiency. Li et al. [61] pointed out that dynamic taint tracking is costly and limits automation, particularly in embedded systems with constrained resources.

Static Analysis and ML/NLP. Static analysis faces high time costs from symbolic execution and feature extraction, making it hard to balance broad analysis and efficiency, mainly in large-scale firmware under resource limits [122].

6.7 Analysis of Privilege and Access Management

Analyzing privilege and access management in IoT systems is challenging due to complex permission models and dynamic configurations, making identifying excessive privileges and access control issues difficult.

Static Analysis. Static evaluation of security policies in IoT can result in incomplete analyses [29]. Additionally, static analysis struggles to assess firmware trustworthiness based solely on permission count, as signature permissions may grant access similar to dangerous permissions [116]. Other challenges include application over-privileging [84], detecting malicious applications that circumvent permission systems to access protected data through side channels [110], and extracting permission data from unstructured text in software [6]. Finally, the dynamic nature of device permissions and configurations adds more complexity to the analysis [17].

Static and Dynamic Analysis. According to Kapoor et al. [54], static analysis applied to IoT devices may face challenges in detecting security vulnerabilities associated with inadequate authentication management, insufficient input validation, and excessive permissions from third-party libraries.

6.8 Reverse Engineering Complexity

The challenge of applying static analysis to reverse engineer code and firmware from IoT devices lies in their binary complexity, obfuscation, and lack of documentation, which make it difficult to recover software functionality and identify vulnerabilities.

Static Analysis. Yoda et al. [128] noted that reverse engineering firmware is inherently complex, involving decoding binary files, interpreting control flow, and correlating disparate components. Still, static analysis faces significant challenges as attackers can quickly reverse engineer binaries of IoT devices to identify vulnerabilities [49, 69].

Static and Dynamic Analysis. Static analysis faces challenges in IoT security due to firmware binary complexity and code obfuscation or encryption. These factors make reverse engineering difficult, compromising exploits and vulnerability identification [25, 64]. Zhang et al. [131] noted that analyzing undocumented PLC binaries requires deep reverse engineering to recover control logic and runtime behavior. Similarly, identifying service handlers in embedded binaries without debug symbols demands semantic understanding of numerous functions, highlighting the complexity of reverse engineering in these environments [61].

6.9 Analysis of Memory Management and Protection

The challenge of analyzing memory management and protection in IoT systems lies in accurately detecting issues such as memory leaks and out-of-bounds access, which are often caused by dynamic memory allocation, low-level pointer operations, and unsafe functions.

Static Analysis. Applying static analysis in IoT faces challenges. One difficulty is accurately analyzing dynamically allocated memory blocks and arrays [33]. Another challenge is the reliance on **Executable and Linkable Format** (**ELF**) files to extract the necessary data and estimate maximum heap memory usage [23]. Additionally, the use of unsafe functions like `memcpy`, often adopted for performance reasons, introduces difficulties in verifying memory safety and correct handling of allocation and deallocation [100].

Static and Dynamic Analysis. The complexity of pointer arithmetic and type conversions in generic module implementations can make formal verification of C software difficult [71].

Furthermore, static analysis in IoT is significantly compromised by runtime errors, particularly those arising from the heavy use of pointers and arrays in the AES module code [95].

6.10 Analysis of Data Privacy and Security

The challenge of analyzing data privacy and security in IoT systems is detecting sensitive data exposure due to the lack of standardized protocols, reliance on implicit dataflows, and limited support for privacy inference at the application level.

Static Analysis. The absence of standardized security protocols in IoT devices compromises the effectiveness of static analysis in identifying vulnerabilities and implementing security measures [114]. Furthermore, static analysis techniques in IoT systems face security-related constraints, including privacy concerns, which can lead to inaccuracies in vulnerability detection [36]. In turn, the SAINT static analysis tool presents limitations in not flagging encoded sequences with sensitive information unless an implicit flow reveals potential data leaks [16].

Static and Dynamic Analysis. Nyzhnyk et al. [85] highlighted the difficulty of protecting sensitive data in SCADA and IIoT cloud environments, stressing the need for robust privacy mechanisms and regular audits to meet regulations like GDPR, HIPAA, and NIST.

Static Analysis and ML/NLP. According to Luo et al. [67], static analysis in IoT applications faces limitations in accurately inferring sensitive information, as current privacy inference is focused on individual events (specific actions) rather than at the application level.

6.11 Analysis of Data Contamination

Analyzing data contamination in IoT systems is challenging due to the difficulty of tracking tainted data across complex and implicit flows, which affects the precision of vulnerability detection and compromises system integrity.

Static Analysis. According to Schmeidl et al. [108], the main challenge in IoT static analysis is tracing dependencies to identify statements that affect tainted data, a complex task due to the need for dependency graphs and algorithms to manage tainted flow. Celik et al. [16] state that the SAINT tool produces an inaccurate CG when dealing with reflection calls, leading to excessive taint and requiring a more thorough analysis to refine the call set.

Static and Dynamic Analysis. Limitations of static analysis in IoT vulnerability mining include the difficulty of identifying tainted data sources, locating suspicious functions, extracting relevant program execution paths [127], and the accuracy in detecting tainted sinks [132].

Static Analysis and ML/NLP. Chen et al. [21] highlighted the difficulty of tracking tainted data through complex flows and identifying sanitizers, especially in static contexts with implicit contamination paths.

7 Discussion

The discussion section is essential for reflecting on this study's findings, examining static analysis's role in IoT software security, and relating them to current practices and future developments. Section 7.1 presents the current IoT security landscape and the role of static analysis across different software layers. Section 7.2 describes the stakeholder ecosystem and how static analysis is adopted in development and testing. Finally, Section 7.3 outlines future research directions, including taxonomy development, tool evaluation, and hybrid analysis approaches.

7.1 Current IoT Security Landscape for Static Analysis

The IoT security landscape is heterogeneous and dynamic, encompassing everything from simple sensors to complex industrial systems with distinct hardware and software architectures. In addition to this technological diversity, IoT security faces structural challenges, including the lack of a widely accepted standardized definition. According to Franklin et al. [38], four key factors hinder the security of these devices: **(i) ubiquity**, due to the large number of connected devices; **(ii) diversity**, caused by variations in manufacturers and versions of hardware, firmware, and software; **(iii) ecosystem**, with multiple suppliers involved in the development of each device; and **(iv) standardization**, due to the absence of widely adopted norms for access and communication security.

IoT security analysis can be applied at different software levels: source code, firmware, and binary, each presenting specific challenges. Static analysis at the source code level is helpful in identifying vulnerabilities during development but can be insufficient, as flaws may emerge in firmware or binaries due to optimizations, external libraries, and compilation modifications. This makes analysis at these levels essential. Firmware is a critical component in this context, as it is often proprietary and closed-source, making direct inspection difficult. In such cases, binary analysis becomes fundamental for detecting vulnerabilities in compiled code, although it presents challenges such as the need for reverse engineering and the lack of structural metadata. For a more comprehensive evaluation, approaches are combined, including **Software Composition Analysis** (**SCA**), Fuzz Testing, and DAST, which complement static analysis to cover multiple attack vectors [26].

Integrating these approaches into secure development is essential to identify vulnerabilities from the early stages of the IoT software lifecycle. The **Secure Software Development Framework** (**SSDF**) recommends using automated tools to detect flaws and ensure compliance with security standards, complementing the analysis with manual review for triaging and fixing identified issues [111].

7.2 Stakeholder Ecosystem and Static Analysis Adoption in IoT

The IoT ecosystem consists of various stakeholders, each playing a crucial role in mitigating risks and ensuring the security of connected devices and services. Among others, the key stakeholders include **IoT Service Providers, IoT Device Manufacturers, IoT Developers, Network Operators, Regulators, and Testers** [43]. In the following, we detail how each stakeholder contributes to the security of IoT systems, emphasizing their role in promoting or adopting static code analysis practices across different development and deployment phases.

IoT Service Providers – Companies or organizations that develop innovative connected products and services. These providers operate in various sectors, including smart homes, smart cities, automotive, transportation, healthcare, utilities, and consumer electronics [43].

IoT Device Manufacturers – Responsible for producing IoT devices that enable services provided by IoT Service Providers. To enhance security, devices should announce their functionalities and network restrictions, reducing the attack surface. An example is the adoption of **Manufacturer Usage Description** (**MUD**), as specified in IETF RFC 8520, which defines secure communication policies [34].

IoT Developers – Responsible for developing IoT services for providers, ensuring the secure implementation of solutions in compliance with industry standards [51, 92]. They must follow best security practices and utilize frameworks and documentation-based validation to verify requirements during project reviews and audits [52].

Network Operators – Network providers that deliver or implement IoT services for service providers. They are responsible for the continuous delivery and maintenance of software and IoT

systems, including administration, monitoring, and optimization, performed by Operations, Administrators, DevOps, and SRE teams [34].

Regulators – National or regional authorities that establish security and interoperability regulations for IoT, aiming to minimize market fragmentation. To do so, they rely on guidelines and standards from organizations such as ENISA, GSMA, ATIS, OASIS, IEEE, and NIST, which develop frameworks and security recommendations for the IoT ecosystem [42]. Additionally, regulators may require the adoption of security analysis methods, including static code analysis, as part of certification and compliance processes.

Testers – Manufacturers, operators, and third-party laboratories that perform security testing on IoT devices, products, and services. Their goal is to identify vulnerabilities throughout the software development and execution lifecycle by using various tools, including SAST and DAST . Static analysis is commonly adopted by manufacturers and developers to detect security flaws in the source code before deployment. Dynamic analysis is more frequently used by operators and third-party testers, as it evaluates software behavior in a runtime environment. The choice between these approaches depends on factors such as regulatory requirements, system criticality, and implementation costs [78, 89, 90].

Each stakeholder plays a vital role in ensuring the security and functionality of IoT systems. Adopting security practices from the early stages of development, combined with the integration of static and dynamic methods, is essential for risk mitigation and strengthening IoT implementations. Static analysis is more widely used during development, whereas dynamic analysis is applied during testing and operation. The adoption of these practices is influenced by factors such as sector-specific regulations, cost-effectiveness, and availability of automation tools.

7.3 Future Directions in IoT Security

We identified significant gaps and challenges that require attention in future research, particularly:

(i) There is a lack of detailed and structured taxonomies that map specific vulnerabilities found in the source code of IoT software. Creating such a taxonomy, based on sources like CVE, CWE, and NVD and aligned with standards such as the OWASP IoT Top 10, would support researchers and developers in understanding the nature and context of common vulnerabilities.

(ii) Current landscape of IoT vulnerability detection tools is fragmented, and their evaluation is often inconsistent. Future research should focus on developing a structured benchmark framework, including precision, recall, and coverage criteria, to evaluate the effectiveness of both general-purpose and IoT-specific tools.

(iii) While static analysis is a critical technique, combining it with dynamic analysis, fuzzing, and AI-based methods, especially ML, NLP, and LLMs, could enhance the precision and scalability of vulnerability detection. Research must integrate these approaches into automated pipelines, minimizing false positives while increasing coverage.

(iv) Although guidelines such as OWASP exist, many IoT systems still fail to adopt secure coding practices effectively. This highlights the need to align static analysis methodologies with these principles to enable early detection of insecure patterns and strengthen the overall resilience of IoT software.

8 Threats to Validity

8.1 Internal Validity

Systematic Review Process: Study objectives, research questions, search strategies, and IC/EC may influence the obtained results. To address these threats, we systematically planned and reviewed all steps to improve the methodological rigor and results consistency.

Collaborative Review: To maintain impartiality in evaluating articles, we adopted a collaborative review methodology using the Rayyan Web platform, where researchers independently assessed study eligibility. This reduced potential bias in inclusion/exclusion decisions.

Implicit Use of Static Analysis Methods: Several studies applied static analysis techniques without explicitly describing or naming them, making their identification dependent on technical interpretation. The researchers' experience was crucial in recognizing these methods, although insufficient details may lead to erroneous conclusions. Furthermore, clarity regarding the interrelationships between the techniques may make it easier to understand the integration processes. To address these concerns, we established an online feedback platform using the **Open Science Framework** (**OSF**), allowing readers to provide corrections and insights. This platform provides a publicly available dataset, contributing to the accuracy of the results.

8.2 External Validity

Database Limitations: The data quality in online databases is a significant factor. Incomplete searches or poor indexing may lead to an incomplete set of articles. Therefore, eight reliable online databases commonly used in relevant studies were employed to mitigate this concern.

Mitigating Missing Studies: To reduce the risk of excluding relevant articles not captured through initial database searches, we applied the snowballing method and leveraged the ResearchRabbitApp to identify relevant studies from cited references.

8.3 Construct Validity

Terminology in Static Analysis and IoT: The literature on static analysis and the IoT uses varied terminology, leading to excessive or inappropriate search results. To minimize this effect, we organized the keywords into two groups, one for *Static Analysis* and one for the *IoT*, to build effective search strings. These groups included the most relevant terms and acronyms commonly used in each area. We used a manual testing approach to verify the selected terms and acronyms. Additionally, we applied four complementary search strategies to reduce the risk of missing significant studies.

Exclusion of Quality Assessment: We chose not to assess the quality of the identified studies to avoid potential bias in the results. This indicates that the results may have a subjective bias that favors less rigorous research. This decision is supported by related studies [59], which acknowledge the challenges in assessing the quality of selected works. Consequently, we employed a conservative approach by including only peer-reviewed publications.

9 Conclusions

This article presents a systematic review of static analysis for IoT security. To achieve this goal, we analyzed 73 articles from eight computer science databases spanning eight years. The identified articles were categorized into three distinct groups: the first group includes those that focus exclusively on static analysis or incorporate it as part of their approach. The following groups include hybrid methods that integrate static analysis with dynamic analysis, and finally, static analysis with ML and NLP techniques.

To guide a comprehensive investigation, we structured this work around three key research questions (RQs) directly relevant to IoT security. These questions focus on the vulnerabilities identified through static analysis (RQ1), the techniques and approaches used in IoT vulnerability detection (RQ2), and the challenges associated with applying static analysis to IoT security (RQ3). For **RQ1**, we identified 11 commonly discussed vulnerabilities in the reviewed literature, including unsafe functions, privacy violations, and privilege escalation issues. Regarding **RQ2**, we uncovered eight distinct static analysis techniques utilized in the papers: CFA, DFA, syntax

analysis, taint analysis, semantic analysis, symbolic analysis, lexical analysis, and points-to analysis. For **RQ3**, our research highlighted 11 significant challenges, such as achieving comprehensive vulnerability coverage, addressing architectural and resource constraints, and balancing precision and recall effectively.

In summary, this review consolidates the current knowledge on static analysis for IoT security by mapping relevant studies, classifying key vulnerabilities, examining the techniques used, and outlining the main technical challenges. By synthesizing these insights, the study provides a solid foundation for advancing research and practical solutions in secure IoT system development.

10 Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to impact the research presented in this article.

References

[1] Philip Achimugu, Ali Selamat, Roliana Ibrahim, and Mohd Naz'ri Mahrin. 2014. A systematic literature review of software requirements prioritization research. *Information and Software Technology* 56, 6 (June 2014), 568–585. DOI: https://doi.org/10.1016/j.infsof.2014.02.001

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Vishnu Ajith, Tom Cyriac, Chetan Chavda, Anum Tanveer Kiyani, Vijay Chennareddy, and Kamran Ali. 2024. Analyzing Docker vulnerabilities through static and dynamic methods and enhancing IoT security with AWS IoT core, CloudWatch, and GuardDuty. *IoT* 5, 3 (Sept. 2024), 592–607. DOI: https://doi.org/10.3390/iot5030026

[4] Abdullah Al-Boghdady, Mohammad El-Ramly, and Khaled Wassif. 2022. iDetect for vulnerability detection in internet of things operating systems using machine learning. *Scientific Reports* 12, 1 (Oct. 2022), 1–12. DOI: https://doi.org/10.1038/s41598-022-21325-x

[5] Abdullah Al-Boghdady, Khaled Wassif, and Mohammad El-Ramly. 2021. The presence, trends, and causes of security vulnerabilities in operating systems of IoT's low-end devices. *Sensors* 21, 7 (March 2021), 2329. DOI: https://doi.org/10.3390/s21072329

[6] Manar H. Alalfi, Atheer Abu Zaid, and Ali Miri. 2023. A model-driven-reverse engineering approach for detecting privilege escalation in IoT systems. *The Journal of Object Technology* 22, 1 (2023), 1:1. DOI: https://doi.org/10.5381/jot.2023.22.1.a1

[7] Mohannad Alhanahnah, Clay Stevens, Bocheng Chen, Qiben Yan, and Hamid Bagheri. 2023. IoTCom: Dissecting interaction threats in IoT systems. *IEEE Transactions on Software Engineering* 49, 4 (April 2023), 1523–1539. DOI: https://doi.org/10.1109/tse.2022.3179294

[8] Amir Ali, Zain Ul Abideen, and Kalim Ullah. 2021. SESCon: Secure ethereum smart contracts by vulnerable patterns' detection. *Security and Communication Networks* 2021 (Sept. 2021), 1–14. DOI: https://doi.org/10.1155/2021/2897565

[9] Saleh M. Alnaeli, Melissa Sarnowski, Md Sayedul Aman, Ahmed Abdelgawad, and Kumar Yelamarthi. 2016. Vulnerable C/C++ code usage in IoT software systems. In *Proceedings of the 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. IEEE, 637–642. DOI: https://doi.org/10.1109/WF-IoT.2016.7845497

[10] Saleh Mohamed Alnaeli, Melissa Sarnowski, Md Sayedul Aman, Ahmed Abdelgawad, and Kumar Yelamarthi. 2017. Source code vulnerabilities in IoT software systems. *Advances in Science, Technology and Engineering Systems Journal* 2, 3 (2017), 1502–1507. DOI: https://doi.org/10.25046/aj0203188

[11] Andrew W. Appel and Jens Palsberg. 2003. *Modern Compiler Implementation in Java* (2nd ed.). Cambridge University Press, USA.

[12] Musard Balliu, Massimo Merro, Michele Pasqua, and Mikhail Shcherbakov. 2021. Friendly fire: Cross-app interactions in IoT platforms. *ACM Transactions on Privacy and Security* 24, 3 (April 2021), 1–40. DOI: https://doi.org/10.1145/3444963

[13] Giuseppe Marco Bianco, Luca Ardito, and Michele Valsesia. 2024. A tool for IoT firmware certification. In *Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES 2024)*. ACM, New York, NY, USA, 1–7. DOI: https://doi.org/10.1145/3664476.3670469

[14] Chiara Bodei and Letterio Galletta. 2020. Analysing the provenance of IoT data. In *Information Systems Security and Privacy*, P. Mori, S. Furnell, and O. Camp (Eds.). Springer International Publishing, Cham, Switzerland, 358–381. DOI: https://doi.org/10.1007/978-3-030-49443-8_17

[15] Luciano Gonçalves Carvalho and Marcelo Medeiros Eler. 2018. Security requirements and tests for smart toys. In *Enterprise Information Systems ICEIS 2017*, S. Hammoudi, M. Śmiałek, O. Camp, and J. Filipe (Eds.). Lecture Notes in Business Information Processing, Vol 321. Springer International Publishing, Cham, Switzerland, 291–312. DOI: https://doi.org/10.1007/978-3-319-93375-7_14

[16] Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive information tracking in commodity IoT. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, USA, 1687–1704.

[17] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. SOTERIA: Automated IoT safety and security analysis. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, USA, 147–158.

[18] Byeong-Mo Chang, Janine Cassandra Son, and Kwanghoon Choi and. 2020. A GQM approach to evaluation of the quality of smartthings applications using static analysis. *KSII Transactions on Internet and Information Systems* 14, 6 (June 2020), 2354–2376. DOI: https://doi.org/10.3837/tiis.2020.06.003

[19] Manuel Cheminod and Lucia Seno. 2023. Static analysis of packet forwarding and filtering configurations in industrial networks. In *Proceedings of the 2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*. IEEE, Brussels, Belgium, 1–8. DOI: https://doi.org/10.1109/WFCS57264.2023.10144115

[20] Chen Chen, Weikong Qi, Wenting Jiang, and Peng Sun. 2020. *A Dynamic Instrumentation Technology for IoT Devices*. Springer International Publishing, Poland, 304–312. DOI: https://doi.org/10.1007/978-3-030-50399-4_29

[21] Xiao Chen, Letian Sha, Jincheng Wang, Fu Xiao, and Jiankuo Dong. 2025. SFO-CID: Structural feature optimization based command injection vulnerability discovery for internet of things. *IEEE Transactions on Industrial Informatics* 21, 2 (Feb. 2025), 1429–1438. DOI: https://doi.org/10.1109/tii.2024.3477563

[22] Yiran Cheng, Shouguo Yang, Zhe Lang, Zhiqiang Shi, and Limin Sun. 2023. VERI: A large-scale open-source components vulnerability detection in IoT firmware. *Computers & Security* 126, C (March 2023), 103068. DOI: https://doi.org/10.1016/j.cose.2022.103068

[23] Kiho Choi, Seongseop Kim, Moon Gi Seok, Jeonghun Cho, and Daejin Park. 2017. *Maximum Stack Memory Monitoring Method Assisted by Static Analysis of the Stack Usage Profile*. Springer Singapore, Taiwan, 756–765. DOI: https://doi.org/10.1007/978-981-10-7605-3_121

[24] Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, and Shunhui Ji. 2024. SGDL: Smart contract vulnerability generation via deep learning. *Journal of Software: Evolution and Process* 36, 12 (July 2024), 24 pages. DOI: https://doi.org/10.1002/smr.2712

[25] Jiska Classen, Daniel Wegemer, Paul Patras, Tom Spink, and Matthias Hollick. 2018. Anatomy of a vulnerable fitness tracking system: Dissecting the fitbit cloud, app, and firmware. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 1 (March 2018), 1–24. DOI: https://doi.org/10.1145/3191737

[26] Open Networking Community. 2021. *Open Networking & the Security of Open Source Software Deployment*. Technical Report. GSMA. Retrieved March 11, 2025 from https://www.gsma.com/security

[27] Andrei Costin. 2017. Lua code: Security overview and practical approaches to static analysis. In *Proceedings of the 2017 IEEE Security and Privacy Workshops (SPW)*. IEEE, 31–36. DOI: https://doi.org/10.1109/spw.2017.38

[28] Jianfeng Cui, Lixin Wang, Xin Zhao, and Hongyi Zhang. 2020. Towards predictive analysis of android vulnerability using statistical codes and machine learning for IoT applications. *Computer Communications* 155 (April 2020), 125–131. DOI: https://doi.org/10.1016/j.comcom.2020.02.078

[29] Nicolas Dejon, Davide Caputo, Luca Verderame, Alessandro Armando, and Alessio Merlo. 2020. *Automated Security Analysis of IoT Software Updates*. Springer International Publishing, France, 223–239. DOI: https://doi.org/10.1007/978-3-030-41702-4_14

[30] Sabine Delaitre and José Maria Pulgar Gutiérrez. 2024. Vulnerability detection tool in source code by building and leveraging semantic code graph. In *Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES 2024)*. ACM, New York, NY, USA, 1–9. DOI: https://doi.org/10.1145/3664476.3670942

[31] Ryan Dewhurst. 2024. Static Code Analysis. Retrieved February 16, 2024 from https://owasp.org/www-community/controls/Static_Code_Analysis

[32] Wenbo Ding, Hongxin Hu, and Long Cheng. 2021. IoTSafe: Enforcing safety and security policy with real IoT physical interaction discovery. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*. Internet Society, Virtual. DOI: https://doi.org/10.14722/ndss.2021.24368

[33] Yukun Dong, Wenjing Yin, Shudong Wang, Li Zhang, and Lin Sun. 2019. Memory leak detection in IoT program based on an abstract memory model SeqMM. *IEEE Access* 7 (2019), 158904–158916. DOI: https://doi.org/10.1109/access.2019.2951168

[34] ENISA. 2019. *Good Practices for Security of IoT: Secure Software Development Lifecycle*. Technical Report. European Union Agency for Cybersecurity (ENISA). Retrieved March 11, 2025 from https://www.enisa.europa.eu/publications/good-practices-for-security-of-iot

[35] Pietro Ferrara, Amit Kr Mandal, Agostino Cortesi, and Fausto Spoto. 2020. Static analysis for discovering IoT vulnerabilities. *International Journal on Software Tools for Technology Transfer* 23, 1 (Nov. 2020), 71–88. DOI: https://doi.org/10.1007/s10009-020-00592-x

[36] Pietro Ferrara and Fausto Spoto. 2018. Static analysis for GDPR compliance. In *Proceedings of the 2nd Italian Conference on Cybersecurity* . ITASEC, IT.

[37] Farhaan Fowze and Tuba Yavuz. 2021. SEESAW: A tool for detecting memory vulnerabilities in protocol stack implementations. In *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '21)*. ACM, Virtual Event, 99–108. DOI: https://doi.org/10.1145/3487212.3487345

[38] Joshua M. Franklin, Tony Krzyzewski, Maurice Turner, Kathleen Moriarty, and Robin Regnier. 2021. *CIS Controls v8 Internet of Things Companion Guide*. Technical Report. Center for Internet Security (CIS). Retrieved March 10, 2025 from https://www.cisecurity.org/insights/white-papers/cis-controls-v8-internet-of-things-companion-guide

[39] Yifei Gao, Xu Zhou, Wei Xie, Baosheng Wang, Enze Wang, and Zhenhua Wang. 2022. Optimizing IoT web fuzzing by firmware infomation mining. *Applied Sciences* 12, 13 (June 2022), 6429. DOI: https://doi.org/10.3390/app12136429

[40] Guojian Ge and Qingshu Xue. 2016. Security detection and research of intelligent hardware system. In *Proceedings of the 2016 4th International Conference on Electrical & Electronics Engineering and Computer Science (ICEEECS 2016) (ICEEECS-16)*. Atlantis Press, Beijing, China, 15–20. DOI: https://doi.org/10.2991/iceeecs-16.2016.3

[41] Diego R. Gomes, Fernando A. Aires Lins, and Marco Vieira. 2024. Static Analysis for IoT Security: A Systematic Literature Review - Dataset. DOI: https://doi.org/10.17605/OSF.IO/H9JRV

[42] GSMA. 2020. *Open Source Software Security: A Research Summary*. Technical Report. GSMA. Retrieved March 11, 2025 from https://www.gsma.com/solutions-and-impact/technologies/security/wp-content/uploads/2020/12/Open-Source-Software-Security-Research-Summary-v1.1.pdf

[43] GSMA. 2024. *IoT Security Guidelines Overview - Version 1.0*. Technical Report. GSM Association. Retrieved March 11, 2025 from https://www.gsma.com/solutions-and-impact/technologies/internet-of-things/wp-content/uploads/2024/07/FS.60.pdf

[44] GSMA. 2025. *Network Equipment Security Assurance Scheme - Audit Guidelines - Version 3.0*. Technical Report. GSM Association. Retrieved March 11, 2025 from https://www.gsma.com/solutions-and-impact/technologies/security/wp-content/uploads/2025/04/FS.46-v3.0.pdf

[45] Chen Guo, Yang Yang, Yanglin Zhou, Kuan Zhang, and Song Ci. 2021. A quantitative study of energy consumption for embedded security. In *Proceedings of the 2021 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 1–6. DOI: https://doi.org/10.1109/WCNC49053.2021.9417382

[46] Guangxin Guo, Chao Wang, Jiahan Dong, Bowen Li, and Xiaohu Wang. 2022. An iootfuzzer method for vulnerability mining of internet of things devices based on binary source code and feedback fuzzing. *International Journal of Advanced Computer Science and Applications* 13, 7 (2022), 692–697. DOI: https://doi.org/10.14569/ijacsa.2022.0130781

[47] Hassan Bapeer Hassan, Qusay Idrees Sarhan, and Árpád Beszédes. 2024. Evaluating python static code analysis tools using FAIR principles. *IEEE Access* 12 (2024), 173647–173659. DOI: https://doi.org/10.1109/access.2024.3503493

[48] Daojing He, Hongjie Gu, Tinghui Li, Yongliang Du, Xiaolei Wang, Sencun Zhu, and Nadra Guizani. 2021. Toward hybrid static-dynamic detection of vulnerabilities in IoT firmware. *IEEE Network* 35, 2 (March 2021), 202–207. DOI: https://doi.org/10.1109/mnet.011.2000450

[49] Ara Hur, Jooeun Kim, and Yeonseung Ryu. 2019. Hiding vulnerabilities of internet of things software using anti-tamper technique. In *Proceedings of the 2019 4th International Conference on Intelligent Information Technology*. ACM. DOI: https://doi.org/10.1145/3321454.3321466

[50] João Inácio and Ibéria Medeiros. 2023. CorCA: An automatic program repair tool for checking and removing effectively C flaws. In *Proceedings of the 2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 71–82. DOI: https://doi.org/10.1109/icst57152.2023.00016

[51] IoT Security Foundation. 2019. IoT Security Foundation - Secure Design Best Practice Guides - Release v2. Retrieved March 10, 2025 from https://www.iotsecurityfoundation.org/wp-content/uploads/2019/12/Best-Practice-Guides-Release-2_Digitalv3.pdf

[52] IoT Security Foundation. 2021. IoT Security Foundation - IoT Security Assurance Framework - Release v3. Retrieved March 10, 2025 from https://www.iotsecurityfoundation.org/wp-content/uploads/2021/11/IoTSF-IoT-Security-Assurance-Framework-Release-3.0-Nov-2021-1.pdf

[53] IoT Security Foundation. 2023. The State of Vulnerability Disclosure (VDP) Usage in Global Consumer IoT in 2023. Retrieved April 12, 2024 from https://iotsecurityfoundation.org/best-practice-guidelines/

[54] Pranay Kapoor, Rohan Pagey, Mohammad Mannan, and Amr Youssef. 2023. *Silver Surfers on the Tech Wave: Privacy Analysis of Android Apps for the Elderly*. Springer Nature Switzerland, Virtual Event, 673–691. DOI: https://doi.org/10.1007/978-3-031-25538-0_35

[55] Naqash Azeem Khan, Azlan Awang, and Samsul Ariffin Abdul Karim. 2022. Security in internet of things: A review. *IEEE Access* 10 (2022), 104649–104670. DOI: https://doi.org/10.1109/access.2022.3209355

[56] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report. EBSE Technical Report, Keele University and University of Durham. Version 2.3.

[57] Matej Klima, Miroslav Bures, Karel Frajtak, Vaclav Rechtberger, Michal Trnka, Xavier Bellekens, Tomas Cerny, and Bestoun S. Ahmed. 2022. Selected code-quality characteristics and metrics for internet of things systems. *IEEE Access* 10 (2022), 46144–46161. DOI: https://doi.org/10.1109/access.2022.3170475

[58] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A systematic review on software robustness assessment. *ACM Computing Surveys* 54, 4 (May 2021), 1–65. DOI: https://doi.org/10.1145/3448977

[59] Mathieu Lavalleee, Pierre-N. Robillard, and Reza Mirsalari. 2014. Performing systematic literature reviews with novices: An iterative approach. *IEEE Transactions on Education* 57, 3 (Aug. 2014), 175–181. DOI: https://doi.org/10.1109/te.2013.2292570

[60] Xixing Li, Qiang Wei, Zehui Wu, and Wei Guo. 2023. A comprehensive survey of vulnerability detection method towards Linux-based IoT devices. In *Proceedings of the 2023 2nd International Conference on Networks, Communications and Information Technology (CNCIT 2023)*. ACM, New York, NY, United States. DOI: https://doi.org/10.1145/3605801.3605808

[61] Xixing Li, Lei Zhao, Qiang Wei, Zehui Wu, Weiming Shi, and Yunchao Wang. 2024. SHFuzz: Service handler-aware fuzzing for detecting multi-type vulnerabilities in embedded devices. *Computers & Security* 138 (March 2024), 103618. DOI: https://doi.org/10.1016/j.cose.2023.103618

[62] Bin Liao, Yasir Ali, Shah Nazir, Long He, and Habib Ullah Khan. 2020. Security analysis of IoT devices by using mobile computing: A systematic literature review. *IEEE Access* 8 (2020), 120331–120350. DOI: https://doi.org/10.1109/access.2020.3006358

[63] Danjun Liu, Yong Tang, Baosheng Wang, Wei Xie, and Bo Yu. 2018. *Automated Vulnerability Detection in Embedded Devices*. Springer International Publishing, New Delhi, India, 313–329. DOI: https://doi.org/10.1007/978-3-319-99277-8_17

[64] Kaizheng Liu, Ming Yang, Zhen Ling, Huaiyu Yan, Yue Zhang, Xinwen Fu, and Wei Zhao. 2021. On manually reverse engineering communication protocols of Linux-based IoT systems. *IEEE Internet of Things Journal* 8, 8 (April 2021), 6815–6827. DOI: https://doi.org/10.1109/jiot.2020.3036232

[65] Kaizheng Liu, Ming Yang, Zhen Ling, Yuan Zhang, Chongqing Lei, Lan Luo, and Xinwen Fu. 2024. Samba: Detecting SSL/TLS API misuses in IoT binary applications. In *IEEE INFOCOM 2024 - Proceedings of the IEEE Conference on Computer Communications*. IEEE, 2029–2038. DOI: https://doi.org/10.1109/infocom52122.2024.10621138

[66] Stephanie Long, Richard Dill, and Barry Mullins. 2021. Security analysis of a medical IoT device: Data leakage to an eavesdropper. In *Proceedings of the 54th Hawaii International Conference on System Sciences (HICSS)*. Hawaii International Conference on System Sciences, Kauai, Hawaii, USA. DOI: https://doi.org/10.24251/hicss.2021.827

[67] Yuan Luo, Long Cheng, Hongxin Hu, Guojun Peng, and Danfeng Yao. 2021. Context-rich privacy leakage analysis through inferring apps in smart home IoT. *IEEE Internet of Things Journal* 8, 4 (Feb. 2021), 2736–2750. DOI: https://doi.org/10.1109/jiot.2020.3019812

[68] Yunlong Lyu, Wang Gao, Siqi Ma, Qibin Sun, and Juanru Li. 2021. *SparrowHawk: Memory Safety Flaw Detection via Data-Driven Source Code Annotation*. Springer International Publishing, Virtual Event, 129–148. DOI: https://doi.org/10.1007/978-3-030-88323-2_7

[69] Xiaocheng Ma, Chenyv Yan, Yunchao Wang, Qiang Wei, and Yunfeng Wang. 2024. A vulnerability scanning method for web services in embedded firmware. *Applied Sciences* 14, 6 (March 2024), 2373. DOI: https://doi.org/10.3390/app14062373

[70] Amit Mandal, Pietro Ferrara, Yuliy Khlyebnikov, Agostino Cortesi, and Fausto Spoto. 2020. Cross-program taint analysis for IoT systems. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC'20)*. ACM, New York, NY, United States. DOI: https://doi.org/10.1145/3341105.3373924

[71] Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. 2017. *Formal Verification of a Memory Allocation Module of Contiki with Frama-C: A Case Study*. Springer International Publishing, Roscoff, France, 114–120. DOI: https://doi.org/10.1007/978-3-319-54876-0_9

[72] M. Mercedes Martínez-González, Alejandro Pérez-Fuente, Amador Aparicio, and Pablo A. Criado-Lozano. 2024. *Using the Metadata-Based App-PI Ecosystem to Assess the Privacy Impact of Health Apps*. Springer Nature Switzerland, Belfast, United Kingdom, 522–533. DOI: https://doi.org/10.1007/978-3-031-77571-0_50

[73] Md Al Maruf, Akramul Azim, Nitin Auluck, and Mansi Sahi. 2024. FeaMod: Enhancing modularity, adaptability and code reuse in embedded software development. In *Proceedings of the 2024 IEEE International Conference on Information Reuse and Integration for Data Science (IRI)*. IEEE, San Jose, CA, USA, 246–251. DOI: https://doi.org/10.1109/iri62200.2024.00058

[74] Jack McBride, Budi Arief, and Julio Hernandez-Castro. 2018. Security analysis of Contiki IoT operating system. In *Proceedings of the 2018 International Conference on Embedded Wireless Systems and Networks (EWSN '18)*. Junction Publishing, USA, 278–283.

[75] Jean Baptiste Minani, Yahia El Fellah, Sanam Ahmed, Fatima Sabir, Naouel Moha, and Yann-Gaël Guéhéneuc. 2024. An exploratory study on code quality, testing, data accuracy, and practical use cases of IoT wearables. In *Proceedings of the 2024 7th Conference on Cloud and Internet of Things (CIoT)*. IEEE, 1–5. DOI: https://doi.org/10.1109/ciot63799.2024.10756966

[76] Vandana Mohindru and Anjali Garg. 2021. *Security Attacks in Internet of Things: A Review*. Springer Singapore, India, 679–693. DOI: https://doi.org/10.1007/978-981-15-8297-4_54

[77] Ibrahim Nadir, Haroon Mahmood, and Ghalib Asadullah. 2022. A taxonomy of IoT firmware security and principal firmware analysis techniques. *International Journal of Critical Infrastructure Protection* 38, C (Sept. 2022), 100552. DOI: https://doi.org/10.1016/j.ijcip.2022.100552

[78] National Institute of Standards and Technology (NIST). 2022. Source Code Security Analyzers. Retrieved March 11, 2025 from https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers

[79] Bara' Nazzal and Manar H. Alalfi. 2022. An automated approach for privacy leakage identification in IoT apps. *IEEE Access* 10 (2022), 80727–80747. DOI: https://doi.org/10.1109/access.2022.3192562

[80] Matthias Neugschwandtner, Collin Mulliner, William Robertson, and Engin Kirda. 2016. *Runtime Integrity Checking for Exploit Mitigation on Lightweight Embedded Devices*. Springer International Publishing, Vienna, Austria, 60–81. DOI: https://doi.org/10.1007/978-3-319-45572-3_4

[81] Quoc-Dung Ngo, Huy-Trung Nguyen, Van-Hoang Le, and Doan-Hieu Nguyen. 2020. A survey of IoT malware and detection methods based on static features. *ICT Express* 6, 4 (Dec. 2020), 280–286. DOI: https://doi.org/10.1016/j.icte.2020.04.005

[82] Nicolas Nino, Ruibo Lu, Wei Zhou, Kyu Hyung Lee, Ziming Zhao, and Le Guan. 2024. Unveiling IoT security in reality: A firmware-centric journey. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 5609–5626. Retrieved from https://www.usenix.org/conference/usenixsecurity24/presentation/nino

[83] Weina Niu, Xiaosong Zhang, Xiaojiang Du, Lingyuan Zhao, Rong Cao, and Mohsen Guizani. 2020. A deep learning based static taint analysis approach for IoT software vulnerability location. *Measurement* 152, 5 (Feb. 2020), 107139. DOI: https://doi.org/10.1016/j.measurement.2019.107139

[84] Mehdi Nobakht, Yulei Sui, Aruna Seneviratne, and Wen Hu. 2020. PGFit: Static permission analysis of health and fitness apps in IoT programming frameworks. *Journal of Network and Computer Applications* 152, C (Feb. 2020), 102509. DOI: https://doi.org/10.1016/j.jnca.2019.102509

[85] Andrii Nyzhnyk, Andrii Partyka, and Michal Podpora. 2024. Increase the cybersecurity of SCADA and IIoT devices with secure memory management. In *Proceedings of the 2024 Cyber Security and Data Protection (CSDP 2024) (CEUR Workshop Proceedings, Vol. 3800)*. CEUR-WS, Lviv, 32–41. Code 203583.

[86] Kozo Okano, Satoshi Harauchi, Toshifusa Sekizawa, Shinpei Ogata, and Shin Nakashima. 2017. Equivalence checking of Java methods: Toward ensuring IoT dependability. In *Proceedings of the 2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, Beijing, China. DOI: https://doi.org/10.1109/icccn.2017.8038505

[87] Fernando L. Oliveira and Júlio C. B. Mattos. 2022. JSGuide: A tool to improve JavaScript algorithms focusing on IoT devices. In *Proceedings of the 2022 Symposium on Internet of Things (SIoT)*. IEEE, Lima, Peru. DOI: https://doi.org/10.1109/siot56383.2022.10070155

[88] OWASP Foundation. 2018. OWASP Internet of Things (IoT) Project. Retrieved February 16, 2024 from https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project

[89] OWASP Foundation. 2022. Source Code Analysis Tools. Retrieved March 11, 2025 from https://owasp.org/www-community/Source_Code_Analysis_Tools/

[90] OWASP Foundation. 2022. Vulnerability Scanning Tools. Retrieved March 11, 2025 from https://owasp.org/www-community/Vulnerability_Scanning_Tools/

[91] OWASP Foundation. 2024. OWASP Community Pages: Vulnerability. Retrieved February 20, 2024 from https://owasp.org/www-community/vulnerabilities/

[92] OWASP Foundation. 2024. OWASP Secure Coding Practices - Quick Reference Guide. Retrieved February 16, 2024 from https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/

[93] Mattia Paccamiccio and Leonardo Mostarda. 2022. *Reasoning About Inter-procedural Security Requirements in IoT Applications*. Springer International Publishing, Viena, Áustria, 245–254. DOI: https://doi.org/10.1007/978-3-030-99619-2_24

[94] Reza M. Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON '18)*. IBM Corp., USA, 103–113. DOI: https://doi.org/10.5555/3291291.3291303

[95] Alexandre Peyrard, Nikolai Kosmatov, Simon Duquennoy, and Shahid Raza. 2018. Towards formal verification of Contiki: Analysis of the AES-CCM* modules with frama-C. In *Proceedings of the European Conference/Workshop on Wireless Sensor Networks*. Junction Publishing, United States. Retrieved from https://api.semanticscholar.org/CorpusID:51614156

[96] Jayaraj Poroor. 2018. Work-in-Progress: VerticalThings - a language-based microkernel for constrained IoT devices. In *Proceedings of the 2018 International Conference on Embedded Software (EMSOFT)*. IEEE, 1–4. DOI: https://doi.org/10.1109/EMSOFT.2018.8537193

[97] Abdullah Qasem, Paria Shirani, Mourad Debbabi, Lingyu Wang, Bernard Lebel, and Basile L. Agba. 2021. Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies. *ACM Computing Surveys* 54, 2 (March 2021), 1–42. DOI: https://doi.org/10.1145/3432893

[98] Qatar Computing Research Institute. 2023. Rayyan. Retrieved January 15, 2024 from https://www.rayyan.ai/

[99] Chuan Qin, Jiaqian Peng, Puzhuo Liu, Yaowen Zheng, Kai Cheng, Weidong Zhang, and Limin Sun. 2023. UCRF: Static analyzing firmware to generate under-constrained seed for fuzzing SOHO router. *Computers & Security* 128, C (May 2023), 103157. DOI: https://doi.org/10.1016/j.cose.2023.103157

[100] Nirnai Rai and Jyoti Grover. 2024. Analysis of crypto module in RIOT OS using Frama-C. *The Journal of Supercomputing* 80, 13 (May 2024), 18521–18543. DOI: https://doi.org/10.1007/s11227-024-06171-0

[101] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2021. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. DOI: https://doi.org/10.1109/sp40001.2021.00066

[102] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. DOI: https://doi.org/10.1109/sp40000.2020.00036

[103] Research Rabbit Technologies. 2023. ResearchRabbitApp. Retrieved February 16, 2024 from https://www.researchrabbit.ai/

[104] David Robin, Jonathan Salwan, and Justin Bourroux. 2021. From source code to crash test-case through software testing automation. In *Automation in Cybersecurity Appel à communications* (Rennes, FR). C&ESAR, FR.

[105] Abraham Rodríguez-Mota, Ponciano Jorge Escamilla-Ambrosio, Jassim Happa, and Eleazar Aguirre-Anaya. 2016. *GARMDROID: IoT Potential Security Threats Analysis Through the Inference of Android Applications Hardware Features Requirements*. Springer International Publishing, Puebla, Mexico, 63–74. DOI: https://doi.org/10.1007/978-3-319-49622-1_8

[106] Vinay Sachidananda, Suhas Bhairav, and Yuval Elovici. 2020. OVER: Overhauling vulnerability detection for IoT through an adaptable and automated static analysis framework. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC'20)*. ACM, New York, NY, United States, 729 – 738. DOI: https://doi.org/10.1145/3341105.3373930

[107] Vinay Sachidananda, Suhas Bhairav, Nirnay Ghosh, and Yuval Elovici. 2019. PIT: A probe into internet of things by comprehensive security analysis. In *Proceedings of the 2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE)*. IEEE, New York, USA. DOI: https://doi.org/10.1109/trustcom/bigdatase.2019.00076

[108] Florian Schmeidl, Bara Nazzal, and Manar H. Alalfi. 2019. Security analysis for SmartThings IoT applications. In *Proceedings of the 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. DOI: https://doi.org/10.1109/mobilesoft.2019.00013

[109] Pallavi Sivakumaran and Jorge Blasco. 2021. argXtract: Deriving IoT security configurations via automated static analysis of stripped ARM Cortex-M binaries. In *Proceedings of the Annual Computer Security Applications Conference*. ACM, New Orleans, LA, USA. DOI: https://doi.org/10.1145/3485832.3488007

[110] Ha Xuan Son, Barbara Carminati, and Elena Ferrari. 2021. A risk assessment mechanism for android apps. In *Proceedings of the 2021 IEEE International Conference on Smart Internet of Things (SmartIoT)*. IEEE, 237–244. DOI: https://doi.org/10.1109/smartiot52359.2021.00044

[111] Murugiah Souppaya, Karen Scarfone, and Donna Dodson. 2022. *Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities*. Technical Report. National Institute of Standards and Technology (NIST). DOI: https://doi.org/10.6028/NIST.SP.800-218 Accessed: 2025-03-11.

[112] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. 2019. FirmFuzz: Automated IoT firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things (CCS'19)*. ACM, New York, NY, USA, 15–21. DOI: https://doi.org/10.1145/3338507.3358616

[113] Statista. 2023. Internet of Things - Worldwide. Retrieved March 10, 2024 from https://www.statista.com/outlook/tmo/internet-of-things/worldwide

[114] Quentin Stievenart and Coen De Roover. 2020. Compositional information flow analysis for webassembly programs. In *Proceedings of the 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Amsterdam, Netherlands. DOI: https://doi.org/10.1109/scam51674.2020.00007

[115] Panjun Sun, Yi Wan, Zongda Wu, Zhaoxi Fang, and Qi Li. 2025. A survey on privacy and security issues in IoT-based environments: Technologies, protection measures and future directions. *Computers & Security* 148, C (Jan. 2025), 104097. DOI: https://doi.org/10.1016/j.cose.2024.104097

[116] Thomas Sutter and Bernhard Tellenbach. 2023. FirmwareDroid: Towards automated static analysis of pre-installed android apps. In *Proceedings of the 2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, Seoul, South Korea. DOI: https://doi.org/10.1109/mobilsoft59058.2023.00009

[117] Xiangyan Tang, Ke Zhou, Jieren Cheng, Hui Li, and Yuming Yuan. 2021. *The Vulnerabilities in Smart Contracts: A Survey*. Springer International Publishing, Dublin, Ireland. DOI: https://doi.org/10.1007/978-3-030-78621-2

[118] Raymon Van Dinter, Bedir Tekinerdogan, and Cagatay Catal. 2021. Automation of systematic literature reviews: A systematic literature review. *Information and Software Technology* 136, C (Aug. 2021), 106589. DOI: https://doi.org/10.1016/j.infsof.2021.106589

[119] Vasaka Visoottiviseth, Pongnapat Jutadhammakorn, Natthamon Pongchanchai, and Pongjarun Kosolyudhthasarn. 2018. Firmaster: Analysis tool for home router firmware. In *Proceedings of the 2018 15th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 1–6. DOI: https://doi.org/10.1109/jcsse.2018.8457340

[120] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 1–10. DOI: https://doi.org/10.1145/2601248.2601268

[121] Claes Wohlin, Marcos Kalinowski, Katia Romero Felizardo, and Emilia Mendes. 2022. Successful combination of database search and snowballing for identification of primary studies in systematic literature studies. *Information and Software Technology* 147, C (2022), 106908. DOI: https://doi.org/10.1016/j.infsof.2022.106908

[122] Jiahui Xiang, Lirong Fu, Tong Ye, Peiyu Liu, Huan Le, Liming Zhu, and Wenhai Wang. 2025. LuaTaint: A static analysis system for web configuration interface vulnerability of internet of things devices. *IEEE Internet of Things Journal* 12, 5 (March 2025), 5970–5984. DOI: https://doi.org/10.1109/jiot.2024.3490661

[123] Wei Xie, Yikun Jiang, Yong Tang, Ning Ding, and Yuanming Gao. 2017. Vulnerability detection in IoT firmware: A survey. In *Proceedings of the 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, Vol. 51 9. IEEE, 769–772. DOI: https://doi.org/10.1109/icpads.2017.00104

[124] Xinguang Xie, Junjian Ye, Lifa Wu, and Rong Li. 2022. RTOSExtracter: Extracting user-defined functions in stripped RTOS-based firmware. In *Proceedings of the 2022 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. IEEE, New York, USA. DOI: https://doi.org/10.1109/cyberc55534.2022.00024

[125] Hao Yang, Junfeng Zhang, Jun Li, and Xin Xie. 2023. Mining method of code vulnerability of multi-source power IoT terminal based on reinforcement learning. *International Journal of Network Security* 25, 3 (May 2023), 436–448. DOI: https://doi.org/10.6633/IJNS.202305_25(3).07

[126] Zhongju Yang, Weixing Zhu, and Minggang Yu. 2023. Improvement and optimization of vulnerability detection methods for ethernet smart contracts. *IEEE Access* 11 (2023), 78207–78223. DOI: https://doi.org/10.1109/access.2023.3298672

[127] Min Yao, Baojiang Cui, and Chen Chen. 2020. *Research on IoT Device Vulnerability Mining Technology Based on Static Preprocessing and Coloring Analysis*. Springer International Publishing, Poland, 254–263. DOI: https://doi.org/10.1007/978-3-030-50399-4_25

[128] Minami Yoda, Shigeo Nakamura, Yutaka Matsuno, Yuichi Sei, Yasuyuki Tahara, and Akihiko Ohsuga. 2024. YODA: Middleware for improving setup and preprocessing in static analysis of IoT firmware. In *Proceedings of the 2024 16th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, Vol. 117. IEEE, 1–8. DOI: https://doi.org/10.1109/iiai-aai63651.2024.00010

[129] Minami Yoda, Shuji Sakuraba, Yuichi Sei, Yasuyuki Tahara, and Akihiko Ohsuga. 2020. Detection of the hardcoded login information from socket symbols. In *Proceedings of the 2020 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*. IEEE, 33–38. DOI: https://doi.org/10.1109/iccece49321.2020.9231177

[130] Bin Yuan, Zhanxiang Song, Yan Jia, Zhenyu Lu, Deqing Zou, Hai Jin, and Luyi Xing. 2024. MQTTactic: Security analysis and verification for logic flaws in MQTT implementations. In *Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2385–2403. DOI: https://doi.org/10.1109/sp54263.2024.00013

[131] Xuankai Zhang, Jianhua Li, Jun Wu, Guoxing Chen, Yan Meng, Haojin Zhu, and Xiaosong Zhang. 2025. Binary-level formal verification based automatic security ensurement for PLC in industrial IoT. *IEEE Transactions on Dependable and Secure Computing* 22, 3 (2025), 2211–2226. DOI: https://doi.org/10.1109/tdsc.2024.3481433

[132] Yaowen Zheng, Zhanwei Song, Yuyan Sun, Kai Cheng, Hongsong Zhu, and Limin Sun. 2019. An efficient grey-box fuzzing scheme for linux-based IoT programs through binary static analysis. In *Proceedings of the 2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. IEEE, San Diego, CA, USA, 1–6. DOI: https://doi.org/10.1109/IPCCC47392.2019.8958740

[133] Yun Zhou, Junpeng Qi, and Jin Zhu. 2024. Improving smart contract analysis with large language models: The SLLM system. In *Proceedings of the 2024 IEEE 17th International Conference on Signal Processing (ICSP)*. IEEE, 243–246. DOI: https://doi.org/10.1109/icsp62129.2024.10846030

Appendices

Appendix A

In this appendix, we identify and summarize the main concepts used throughout this work, particularly those related to static analysis, vulnerability detection, and IoT security. The following paragraphs present the terminology adopted for the security concepts, vulnerability types, and techniques discussed in the main sections of this article.

Binaries: Refers to compiled versions of software code that consist of machine-level instructions, which can be executed directly by a processor. Examples include executables, libraries, and firmware images [44].

Dynamic Application Security Testing (DAST): Tests the software during execution by simulating attacks to detect vulnerabilities in integrated components [34].

Firmware: Refers to embedded software that interacts directly with hardware components and typically operates at the highest level of privilege [77].

Fuzzing Test: Involves generating or mutating large volumes of input data and feeding it to the application to observe its behavior [34].

Penetration Testing: Simulates real-world attacks on a software system to evaluate its security. The effectiveness depends on the tester's skill level [34].

Software: Digitally encoded instructions written in a formal language to be executed by machines. It includes binaries, source code, scripts, and can serve as firmware, applications, or system components [44].

Software Composition Analysis (SCA): Focuses on identifying vulnerabilities in third-party components and libraries used within the software [34].

Source Code: Human-readable instructions written in high-level programming languages (e.g., C++) that represent the logic of a program before being compiled into machine-executable code [26].

Static Application Security Testing (SAST): Analyzes the software without executing it, examining source code, bytecode, and binaries. It aims to identify coding and design flaws that may lead to known vulnerabilities [34].

Vulnerability: A design flaw or implementation bug that allows an attacker to compromise an application or harm its stakeholders [91].

The following definitions reflect the terminology adopted from the OWASP community and are used consistently in the context of vulnerability detection [91].

Buffer Overflow: Arises when a program attempts to write more data to a memory buffer than it can accommodate. This can corrupt adjacent memory and lead to code execution or crashes, as seen in both stack-based and heap-based BO.

Hard-coded Passwords: Embedding credentials directly in the source code, such as fixed passwords or cryptographic keys, makes them difficult to change and easy to discover, significantly increasing the risk of unauthorized access.

Improper Data Validation: Happens when user input or external data is not properly checked for type, length, format, or range before being processed.

Injection: A class of vulnerabilities in which untrusted input is interpreted as part of a command or query. This includes SQL injection, command injection, and OS command injection, where an attacker manipulates input to alter the control flow of a program.

Memory Leak: Refers to an unintentional form of memory consumption that occurs when the application fails to release an allocated memory block after it is no longer needed.

Null Pointer: Occurs when the application tries to access or manipulate an object or memory location through a null pointer. This typically results in a crash, such as a NullPointerException in high-level languages or a segmentation fault in low-level code.

Privacy Violation: Involves mishandling sensitive information such as user credentials, location data, or device identifiers.

Privilege Violation: Occurs when an application retains elevated privileges beyond what is necessary. For example, functions like chroot() should be executed with elevated privileges, but these should be dropped immediately afterward to minimize exposure.

Third-Party Domain: Describes situations where an application includes content or resources from external sources (such as scripts or media from third-party domains) without proper validation or sanitization.

Unsafe Function: Refers to invoking potentially dangerous functions that may introduce vulnerabilities if misused. Although such functions (e.g., strcpy, gets) can be used safely, they require strict input validation and careful handling to avoid security flaws.

Use of Obsolete Methods: Refers to deprecated or outdated programming constructs, such as obsolete cryptographic functions or insecure APIs, which suggest that the code has not been actively reviewed or maintained.

The following paragraphs define the terminology used for different vulnerability analysis techniques, identified in the literature and used throughout this review.

Abstract Syntax Tree (AST): A tree structure representing the semantic structure of source code, where each internal node denotes a programming construct or operator, and the children represent its components. It omits grammar-specific helper nodes [2].

Backward Taint Analysis: Traces the dataflow backwards from the sink (e.g., an API argument) to its source. It propagates tainted information in reverse to determine where a value originated [65].

Call Graph (CG): A CG is a directed graph where nodes represent functions or procedures, and edges represent calls from one function to another. It is used to model the calling relationships within a program [2].

Concrete Syntax Tree (CST): Also known as a parse tree, it represents the complete syntactic structure of code according to a grammar, including all nonterminal symbols and helper constructs [2].

Control Flow Analysis (CFA): Examines and models a program's control flow to identify and understand how instructions are executed, decisions are reached, and control moves between code segments. It uses components such as the CFG, the CG, and the ICFG to perform this task [2, 11, 31].

Data Flow Analysis (DFA): Refers to collecting runtime information about data in software to understand the behavior of data and the interaction between various code elements during software execution [2, 31].

Forward Taint Analysis: Traces the flow of data forward from a source (e.g., the return value of an API call) to its usage points, verifying if the value was properly checked or used securely [65].

Information Flow Analysis: It is used to determine how dataflows from inputs, such as function parameters and global state, to outputs like return values and modified global variables [114].

Intermediate Representation (IR): A simplified program representation used during compilation, often combining graphical notation and three-address code [2].

Lexical Analysis: Converts source code into fundamental units known as *tokens* to simplify analysis and manipulation of the source code [2, 31].

Pointer Analysis: Refers to determining which variables or memory addresses can be referenced by pointers within a program by employing code analysis techniques to predict the behavior of pointers [2].

Semantic Analysis: Uses syntax trees and symbol tables to support semantic consistency in source code with the programming language's rules [11].

Symbolic Analysis: Uses algebraic expressions to represent abstractly and track variables' values. Symbolic execution leverages this method to investigate program execution paths with symbolic variables. It can be used for program optimization and understanding in static and dynamic analysis [2].

Syntax Analysis: Also called parsing, it builds a tree-based structure from tokens to represent the program's grammatical structure, typically using an AST [2, 11].

Taint Analysis: Seeks to determine parameters tainted by user-controlled inputs and trace them back to vulnerable functions, known as *sinks*. If a tainted parameter is passed to a sink without being previously sanitized, it is flagged as a vulnerability [31].

Appendix B

Table 8. Overview of Static Analysis Methods for IoT Security
A: Unsafe Function, **B**: Privacy Violation, **C**: Privilege Violation, **D**: Third-Party Domain, **E**: Use of Obsolete Methods, **F**: Improper Data Validation, **G**: Buffer Overflow, **H**: Injection, **I**: Hard-coded Passwords, **J**: Null Pointer, **K**: Memory Leak

Ref.	Target	Application Domain	Tools Used	Vulns.	Contribution
[9]	Source Code	Operating System	UnsafeFunsDetector	A-C-J	Identify and analyze the use of insecure functions in C/C++ code used in IoT software systems.
[105]	Source Code	Android Apps	GARMDROID	B-C-D	Introduces the GARMDROID web tool that analyzes potential security threats in IoT applications.
[27]	Source Code	Code Snippet	Lua Code	A-F-H	Presents and open-sources the first static analysis tool designed specifically for Lua code.
[86]	Source Code	Code Snippet	SAW	B-E-F-J	Introduces a method to verify the correctness of *equals* and *hashCode* to support Java application security.
[23]	Binary	Executable Files	-	A-G-K	Introduces a method to monitor stack memory usage in programs to prevent stack overflow.
[10]	Source Code	Operating System	UnsafeFunsDetector	A	Examine a larger group of IoT software systems to understand how frequently insecure functions are used.
[15]	Source Code	Smart Toy	-	B-D-E-F	Performs a detailed analysis of the toy's computing environment using threat modeling.
[16]	Source Code	SmartThings	SAINT	B-C-D	Introduces SAINT, a static taint analysis tool designed to analyze IoT applications.
[17]	Source Code	SmartThings	SOTERIA	A-E	Introduces SOTERIA, a static analysis system designed to verify the safety, security, and health of IoT applications.
[36]	Source Code	Code Snippet	Julia Static Analyzer	B-H-J	Explores how static analysis techniques can support GDPR compliant software systems.
[74]	Source Code	Operating System	Cppcheck Flawfinder RATS	A-E-G	Analyzes the security of the Contiki IoT operating system using static analysis tools.
[94]	Source Code	Smart Contracts	Oyente, Mythril Securify SmartCheck	-	Provides the first empirical evaluation of static security testing tools for Ethereum smart contracts written in Solidity.
[96]	Source Code	Code Snippet	-	A-C	Features the VerticalThings programming language, a secure microkernel designed for embedded and IoT devices.
[49]	Source Code	IoT Software	-	B-I	Development of a technique to hide software vulnerabilities in IoT devices using FPGAs.
[33]	Source Code	Open-Source Project	DTSC_SeqMM Klockwork DTSC_RSTVL	A-K	Introduces SeqMM, an abstract memory model designed to enhance memory leak detection accuracy in IoT programs.
[108]	Source Code	SmartThings	Taint-Things SAINT	A-B-C-F	Introduces Taint-Things, a static analysis tool that identifies potential data leaks in SmartThings IoT applications.
[84]	Source Code	Android Apps	PGFIT	B-C-D-E	Introduces the PGFIT tool, which detects apps built on the Google Fit framework that request excessive permissions.
[114]	Binary	WebAssembly	-	A-G	Introduces an automated and compositional static analysis technique for WebAssembly, targeting information flow.
[29]	Firmware Binary	Executable Files	IoTAV	A-B-E	Introduces the IoTAV framework, which automatically checks the security of software updates for IoT devices.
[106]	Source Code Firmware	Open-Source Project Android Apps File System	OVER	A-E-G-H-I-K	Features an adaptive and automated static analysis framework designed to detect various security vulnerabilities.
[14]	-	System Design Phase	-	-	A mathematical model using the IoT-LySa language is introduced to analyze data movement in IoT systems.
[18]	Source Code	SmartThings	CodeNarc	B-C-D-F-J	Introduces a new method to review the quality of SmartApps on the SmartThings platform.
[35]	Source Code	Code Snippet	Extended Julia's	B-D-E-F-H-I	Identifies and discusses the critical security vulnerabilities in IoT systems outlined by the OWASP IoT Top 10 project.
[70]	Source Code	IoT System	Julia Static Analyzer	B-D-F-H	Presents a cross-program taint analysis method, which extends existing techniques for tracking data flow.

(Continued on the next page)

Table 9. Overview of Static Analysis Methods for IoT Security (*continued*)
A: Unsafe Function, **B**: Privacy Violation, **C**: Privilege Violation, **D**: Third-Party Domain, **E**: Use of Obsolete Methods, **F**: Improper Data Validation, **G**: Buffer Overflow, **H**: Injection, **I**: Hard-coded Passwords, **J**: Null Pointer, **K**: Memory Leak

Ref.	Target	Application Domain	Tools Used	Vulns.	Contribution
[129]	Firmware	Firmware Code	Ghidra	I	Proposes a lightweight method to detect hardcoded login information in IoT devices using static analysis.
[8]	Source Code	Smart Contracts	SESCon, Slither SmartCheck, Solhint Securify	-	Introduces SESCon, a new tool designed to detect vulnerabilities in Ethereum smart contracts.
[5]	Source Code	Operating System	Cppcheck Flawfinder RATS	A-G-H	Investigates the security status and source-code-level vulnerabilities in popular open-source IoT operating systems.
[12]	Source Code	IoT Applications	-	B-C-D-F	Presents a semantic framework for understanding and managing interactions between different IoT applications.
[68]	Source Code	Project Source Code	Clang	A-G-J	Presents SparrowHawk, a system that automatically annotates functions to detect memory-related vulnerabilities.
[109]	Binary	Bluetooth LE ANT	ArgXtract	-	Introduces the ArgXtract tool, designed to analyze Stripped ARM Cortex-M binaries automatically.
[110]	Source Code	Android Apps	-	B-C-D	Presents an approach to estimate the privacy risk of mobile applications through static analysis.
[124]	Firmware Binary	Firmware Code	RTOSExtracter	A	Introduces RTOSExtracter, an automated tool that identifies user-defined functions and their names in RTOS firmware.
[57]	Source Code	IoT System	-	-	Presents a comprehensive view of code quality metrics reviewed specifically for IoT systems.
[79]	Source Code	SmartThings	Taint-Things SAINT	A-B-C-D-H	Introduces Taint-Things, a tool that automatically detects and reports information leaks in SmartThings IoT applications.
[87]	Source Code	JavaScript Code	JSGuide	A-E	Introduces the JSGuide tool, aimed at helping developers enhance JavaScript code for IoT device software.
[93]	Source Code	IoT Applications	CClyzer	A-C	Develops a technique to verify software security properties using static analysis in IoT applications.
[116]	Firmware Binary	Android Apps	AndroGuard Exodus, APKiD QARK, APKLeaks	B-C-D-E-I	Introduces FirmwareDroid, a tool that automates extraction and analysis of pre-installed software on Android devices.
[6]	Source Code	SmartThings	MDE-ChYP ChYP	C-F	Designs and implements the automated MDE-ChYP tool to detect excessive privileges in the SmartThings application.
[7]	Source Code	SmartThings	IoTCom	A-C-D-F	Presents IOTCOM, a system that automatically detects and verifies insecure interactions between IoT applications and devices.
[19]	-	Network Configurations	-	-	Presents a model to describe industrial network infrastructures, focusing on the physical layout and device setup.
[50]	Source Code	Project Source Code	CorCA	A-F-G	Introduces the CorCA tool, which automatically detects and fixes vulnerabilities in C programs through static analysis.
[128]	Firmware Binary	Firmware Code	Ghidra	-	Proposes YODA, a middleware to streamline IoT firmware analysis via static analysis and preprocessing.
[65]	Binary	Executable Files	IDAPython	E	Presents SAMBA, the first static tool to detect SSL/TLS API misuses in IoT binary executables.
[130]	Source Code	MQTT brokers	Clang, Gollvm, Rustc SVF, Haybale	C	Introduces MQTTactic, a semi-automated tool combining static analysis and model checking to detect MQTT auth flaws.
[30]	Source Code	Project Source Code External Dependencies	DocSpot	A-B-D-E	Introduces a semantic graph-based suite combining static analysis and knowledge graphs for enhanced vulnerability detection.
[100]	Source Code	Operating System	Frama-C	A-F-J	Proposes formal verification with Frama-C to assess security in the RIOT OS crypto module.
[47]	Source Code	Python SAST tools	Sonar Python	F-G-H-K	Develops a FAIR-based approach to evaluate Python static analysis tools by Findability, Accessibility, Interoperability, and Reusability.
[13]	Firmware Binary	Executable Files	Radare2 Objdump	-	Develops manifest-producer, a static analysis tool for IoT firmware certification by verifying API behavior in ELF binaries.
[82]	Source Code Firmware	Android Apps	LeakScope, IoTFlow Binwalk, Ghidra FirmLine, FirmXRay	D	Develops OTACap, the first static analysis tool to recover URLs used in firmware update from Android apps.
[72]	Source Code	Android Apps	-	B-C	Introduces App-PI, a framework for quantifying privacy risks in health apps using metadata and static analysis.
[69]	Firmware Binary	Executable Files	SaTC, VulFi IDA Pro Angr Framework	A-F-G	Proposes WFinder, a lightweight scanner for web services in firmware, analyzing backend binaries for vulnerabilities.

Appendix C

Table 9. Overview of Works Integrating Static and Dynamic Analysis Techniques in IoT Security
A: Unsafe Function, **B**: Privacy Violation, **C**: Privilege Violation, **D**: Third-Party Domain, **E**: Use of Obsolete Methods, **F**: Improper Data Validation, **G**: Buffer Overflow, **H**: Injection, **I**: Hard-coded Passwords, **J**: Null Pointer, **K**: Memory Leak

Ref.	Target	Application Domain	Tools Used	Vulns.	Contribution
[40]	Firmware	File System	RIPS	B-C-D-F-H	Proposes a method for security analysis of intelligent hardware devices based on the embedded Linux system.
[80]	Binary	Executable Files	BINtegrity	A-F-G-H	Introduces BINtegrity, a system designed to protect embedded devices against memory corruption attacks.
[71]	Source Code	Operating System	Frama-C	A-F-G-K	Presents a case study on formal verification of the memory allocation module (memb) in Contiki OS.
[25]	Firmware Binary	Fitbit Device	IDA Pro	B-D-F-H	Analyzes how attackers exploit data from nearby Fitbit trackers to extract information and compromise privacy.
[63]	Binary	Executable Files	Angr Framework	A-G-H-K	Introduces a new automated method for detecting vulnerabilities in embedded devices.
[95]	Source Code	Operating System	Frama-C	A-E-G	Demonstrates the use of formal verification techniques to enhance the security of Contiki OS.
[119]	Source Code Firmware Binary	Firmware Code	PHP_CodeSniffer PHPCS-Sec-Audit v2	E-I	Presents the Firmaster tool, which analyzes router firmware for security vulnerabilities using static and dynamic analysis.
[20]	Firmware Binary	Firmware Code	IoTDIT	-	Introduces the IoTDIT tool that combines static and dynamic analysis to evaluate the security of IoT devices.
[102]	Firmware Binary	Firmware Code	Angr Framework BootStomp	G	Presents an approach, KARONTE, for analyzing firmware by tracing data flows between different program parts.
[127]	Firmware Binary	Firmware Code	IDA Pro	A-F-G-H	Introduces the Aric prototype tool, which combines static and color analysis to find vulnerabilities in IoT devices.
[32]	Source Code	SmartThings	IOTSAFE	A-F	Presents a system, IOTSAFE, designed to apply security and protection policies in smart home environments dynamically.
[37]	Source Code	USB Bluetooth Modules	SVF	A-G-K	Introduces SEESAW, a tool for detecting memory vulnerabilities in protocol stack implementations.
[45]	Binary	Executable Files	IDA Pro	A-E-F-K	Develops a static analysis method with dynamic tracking to detail the energy consumption of security algorithms.
[48]	Firmware Binary	Firmware Code	Binwalk, LIEF CWE Checker	A-D-F-G-I	Designs a platform that combines static and dynamic detection methods to detect vulnerabilities in firmware.
[64]	Firmware Binary	File System	IDA Pro FlowDroid	B-D	Presents the first systematic manual reverse engineering framework for discovering IoT communication protocols.
[66]	Source Code Binary	Android Apps Bluetooth LE	IDA Pro Jadx	B	Analyzes the security of the Masimo MightySat oximeter and its app, focusing on BLE data leakage.
[54]	Source Code	Android Apps	MobSF LiteRadar	B-C-D-F-H	Analyzes the security and privacy risks in Android Apps designed specifically for elderly users.
[3]	Source Code	Docker Image	Trivy	-	Proposes hybrid analysis for Docker/IoT security, integrating vulnerability detection and cloud monitoring.
[85]	Source Code	SCADA IIoT devices	SonarQube	G-J-K	Presents an analysis of memory flaws in SCADA/IIoT and proposes a multi-layered solution using static analysis, sanitizers, and Rust.
[131]	Binary	Executable Files	Angr Framework	-	Introduces VoICS, a binary-level framework that combines static and dynamic analysis to extract and verify PLC logic.
[112]*	Firmware	Firmware Code	FirmFuzz	G-H-J	Introduces FirmFuzz, a framework designed to find security weaknesses in IoT device firmware automatically.
[107]*	Source Code Firmware	Operating System IoT Devices	Cppcheck	A-F-G-H-K	Introduces a modular framework PIT, designed to conduct automated security analysis.
[132]*	Firmware Binary	Executable Files	IDA Pro IDAPython	A-K	Develops an efficient method to identify security weaknesses in Linux-based IoT programs using greybox fuzzing.
[104]*	Source Code Binary	TCP/IP Stack	Klocwork	G-H	Combines static analysis with dynamic testing methods like fuzzing and Dynamic Symbolic Execution to validate alerts.
[101]*	Source Code	Android Apps	DIANE	A-F	Introduces the tool DIANE, combining static and dynamic analysis to identify *fuzz triggers* in Android apps.
[39]*	Firmware	File System	IoTParser	C-G-H	Introduces a prototype IoTParser that enhances the ability to find security holes in IoT devices using firmware information.
[46]*	Firmware Binary	Executable Files	PyDaint	A-F	Presents iootfuzzer, a prototype for mining vulnerabilities in IoT firmware using static analysis to identify tainted paths.
[99]*	Firmware Binary	Firmware Code	UCRF	G-H	Presents the UCRF tool for fuzzing SOHO routers, leveraging back-end static analysis to uncover security vulnerabilities.
[61]*	Binary	Executable Files	SHFuzz	A-B-G-H	Develops SHFuzz, a service-aware fuzzing method for detecting multiple vulnerabilities in embedded devices.

(*)indicates that the article uses dynamic analysis with fuzzing.

Appendix D

Table 10. Overview of ML and NLP Methods for IoT Security
A: Unsafe Function, **B**: Privacy Violation, **C**: Privilege Violation, **D**: Third-Party Domain, **E**: Use of Obsolete Methods, **F**: Improper Data Validation, **G**: Buffer Overflow, **H**: Injection, **I**: Hard-coded Passwords, **J**: Null Pointer, **K**: Memory Leak

Ref.	Target	Application Domain	Tools Used	Vulns.	Contribution
[28]	Source Code	Android Apps	SonarQube	-	Develops a prediction model using machine learning and static code metrics to identify security vulnerabilities.
[83]	Source Code	IoT Applications	-	G	Introduces a deep learning method to find and locate software vulnerabilities in IoT applications automatically.
[67]	Source Code	SmartThings	ALTA	B-D-E-F	Presents ALTA, a method that analyzes application-level traffic to identify privacy-sensitive information.
[4]	Source Code	Operating System	Cppcheck Flawfinder RATS	A-F-G-H-K	Develops iDetect, a tool using ML to detect vulnerabilities in C/C++ source code of IoT operating systems.
[22]	Firmware	Open Source Components	Binwalk IDAPython	B-C-G-H	Develops the VERI system that uses deep learning to extract names of OSCs and vulnerable versions of vulnerabilities.
[125]	Source Code	Power IoT Terminals	-	G-J	Presents a new method for finding vulnerabilities in the code of energy IoT devices using reinforcement learning.
[126]	Source Code	Smart Contracts	Mythrill, Osiris Oyente, Security Slither	-	Develops effective methods to detect vulnerabilities in Ethereum smart contracts using SCSVM and SCLMF.
[75]	Source Code	IoT Applications	SonarQube Codacy	-	Introduces an LLM-based approach to automate test case generation in Fitbit apps, improving efficiency, coverage, and reliability.
[133]	Source Code	Smart Contracts	Slither	-	Develops SLLM, combining static analysis and LLMs to improve smart contracts vulnerability detection.
[73]	Source Code	Code Snippet	Python's AST	-	Proposes FeaMod, combining static analysis and BERT to automate feature extraction and modularization in legacy embedded software.
[24]	Source Code	Smart Contracts	Mythrill, Slither Oyente, SmartCheck	-	Develops SGDL, a GAN-based method using static analysis to generate diverse smart contracts vulnerabilities.
[21]	Source Code	Firmware Code	LuaCheck Semgrep TscanCode	F-G-H	Develops LuaTaint, a static analysis tool with taint tracking and LLM-based pruning for Lua IoT firmware.
[122]	Firmware Binary	Firmware Code	IDA Pro Binwalk IDAPython	H	Proposes SFO-CID, a model integrating static analysis, ML, and NLP to detect command injection in cross-platform IoT devices.

Appendix E

Table 11. Overview of Challenges and Approaches in Static, Dynamic, and ML/NLP-Based Analysis for IoT Security

Challenges	Static Analysis only	Static and Dynamic Analysis	Static Analysis and ML/NLP
Comprehensive Vulnerability Coverage	[8–10, 14, 15, 27, 30, 57, 79, 87, 93, 94, 105, 106, 116, 128, 130]	[40, 66, 102, 112, 119]	[4, 22, 28, 75, 122, 126, 133]
Architecture and Resource Constraints	[7, 19, 84, 86, 87, 93, 96, 100, 106, 114, 124, 130]	[20, 25, 40, 45, 46, 85, 101, 107, 127, 132]	[73, 83, 125, 126]
Precision and Recall	[5, 8, 16, 18, 30, 36, 47, 50, 74, 79]	[32, 37, 39, 48, 99, 112, 119]	[21, 28, 133]
Data Semantics and Contextualization	[6, 7, 12, 17, 27, 57, 68–70, 82, 94]	[63, 99, 104]	[21, 73, 122, 133]
Dynamic Code and Environment Interactions	[14, 33, 35, 65, 70, 82]	[32, 48, 64, 80, 101]	[83]
Analysis Automation and Efficiency	[47, 65, 69, 82, 94, 116, 124, 128]	[61, 104, 131]	[122]
Analysis of Privilege and Access Management	[6, 17, 29, 84, 110, 116]	[54]	–
Reverse Engineering Complexity	[49, 69, 128]	[25, 61, 64, 131]	–
Analysis of Memory Management and Protection	[23, 33, 100]	[71, 95]	–
Analysis of Data Privacy and Security	[16, 36, 114]	[85]	[67]
Analysis of Data Contamination	[16, 108]	[127, 132]	[21]

Received 24 December 2024; revised 30 April 2025; accepted 12 June 2025