ICSBoM: Uncovering Hidden Supply Chain Vulnerabilities in ICS Firmware

Yongyu Xie[1][0009-0001-6346-5759], Daniel Khoshkhoo[1][0009-0009-1537-2249], Hithem Lamri[2][0009-0008-5269-8718], Constantine Doumanidis[2][0000-0003-2479-8187], Brian Davidson[1][0009-0006-1416-5502], Burak Sahin[1][0009-0000-8701-9211], Ryan Pickren[1][0009-0001-0705-5667], Raheem Beyah[1][0000-0002-9188-3464], Katherine Davis[3][0000-0002-1603-1122], Michail Maniatakos[2][0000-0001-6899-0651], and Saman Zonouz[1][0009-0009-1585-4192]

[1] Georgia Institute of Technology, Atlanta, USA
{yxie405, dkhoshkhoo3, bdavidson40, buraksahin, rpickren3, szonouz6}@gatech.edu, rbeyah@coe.gatech.edu
[2] New York University Abu Dhabi, Abu Dhabi, UAE
{hl5743, cd3294, michail.maniatakos}@nyu.edu
[3] Texas A&M University, College Station, USA
katedavis@tamu.edu

Abstract. Industrial Control Systems (ICS) play a vital role in managing critical infrastructure like power grids and nuclear facilities. As ICS increase in complexity, the integration of Third-Party Components (TPCs) supporting advanced functionality into ICS firmware becomes inevitable. Unfortunately, this rapid expansion of the software supply chain introduces new attack surfaces in the ICS ecosystem. While products in the IT domain also regularly include TPCs, constraints specific to the ICS domain like stringent uptime requirements, force ICS vendors to incorporate TPCs with chaotic patching practices.

In this paper, we present evidence that traditional TPC version-based vulnerability scanners fail to adequately characterize risks in modern ICS firmware due to non-standard integration practices, such as backporting official fixes and deploying custom patches. These unconventional integration methods, combined with the closed-source nature of hardened embedded systems, make accurate ICS supply chain vulnerability assessment exceedingly challenging. We address this issue by developing the first end-to-end framework, ICSBoM, to identify firmware vulnerabilities hidden in unconventionally imported TPCs. Our experiments using real-world ICS firmware show that ICSBoM achieves 100% accuracy with TPC version identification, 96% accuracy in locating impacted functions, and 86% accuracy with security patch detection. Our solution directly aids in critical infrastructure security by enabling end-users, for the first time, to accurately assess the security posture of their own devices without vendor cooperation or source code disclosure.

Keywords: Industrial Control System · Software Supply Chain · Vulnerability Analysis.

1 Introduction

Industrial Control Systems (ICS) are a collection of hardware and software that collaboratively monitor and control industrial processes such as manufacturing, transportation, power generation, and other critical infrastructure sectors. ICS devices range from foundational Programmable Logic Controllers (PLCs), which process sensor data and issue actuator commands, to emerging Industrial IoT (IIoT) gateways, which communicate between physical systems and cloud platforms, collectively supporting industrial automation and monitoring. As ICS evolve in complexity, advanced Information Technology (IT) capabilities such as network communication and data analytics are increasingly incorporated into traditional Operational Technology (OT) devices. This integration of Third-Party Components (TPCs) into ICS devices reduces the cost of in-house development and maintenance of these capabilities while improving the overall service quality of the devices.

Despite these benefits, this integration practice introduces a spectrum of software supply chain security risks to the ICS domain, which can compromise ICS devices, sabotage physical processes, and even threaten national security, considering the widespread deployment of these devices in critical infrastructure. An example is the Log4Shell vulnerability (CVE-2021-44228) found in the Apache *Log4j* logging library, which attackers can exploit to gain unauthorized control over affected systems [19]. Because *Log4j* is used by a variety of ICS vendors, Siemens listed more than 100 affected products, while other vendors such as Rockwell Automation and Schneider Electric also announced that multiple products were impacted [30]. Another recent research demonstrates that web services incorporated in PLCs facilitate malicious JavaScript injection attacks, thereby allowing manipulation of physical processes, and all major ICS vendors are vulnerable to this new attack targeting ICS software supply chain [39].

To mitigate software supply chain security risks, the U.S. government issued Executive Order 14028, promoting the release of a Software Bill of Materials (SBoM) [48]. An SBoM is an inventory of all software components that constitute a software application, along with detailed information such as component versions and dependencies. This encourages the development of commercial tools such as [12, 34, 43] that generate SBoM documents during the creation of software artifacts. However, these commercial tools rely on proprietary techniques and closed implementations, which limit transparency and hinder a clear understanding of their underlying methodologies. Meanwhile, the research community has proposed numerous supply chain vulnerability scanners that focus on identifying TPC versions and reporting version-associated vulnerabilities in domains such as Android applications and IoT devices. These studies identify TPC versions by leveraging semantic and structural features [72, 73, 75, 76], as well as multi-level matching across application classes, methods, and call graphs [63]. Recent work further improves robustness by addressing challenges introduced by code optimization [64] and shrinking [74]. These approaches then report supply chain vulnerabilities by mapping identified TPC versions to known vulnerabilities, as TPC vulnerability statuses align with upstream software versions. Un-

fortunately, this assumption does not hold in the ICS domain, rendering these vulnerability scanners inapplicable to ICS firmware.

Due to uninterrupted operation requirements of ICS [44], shutting down systems to update TPCs to their latest versions and perform mandatory time-consuming testing is often impractical. Instead, ICS vendors mitigate disclosed vulnerabilities through *backporting*, a practice in which security fixes from newer software versions are applied to older, stable versions without performing full upgrades. However, this patching practice results in a *chaotic ICS ecosystem* from a supply chain vulnerability analysis perspective, as TPC versions are no longer trustworthy to determine whether version-associated vulnerabilities are present in ICS firmware. Furthermore, there is often a delay between the release of patches from TPC developers and their adoption by ICS vendors to devices, resulting in some vulnerabilities being patched by vendors while others remain unaddressed. Consequently, existing TPC version-based vulnerability scanners are insufficient for this domain. These characteristics call for an end-to-end solution that can accurately analyze ICS supply chain vulnerabilities, and enable end-users to assess the security posture of their devices independently without relying on vendor cooperation or source code access.

In this paper, we first demonstrate the unique and chaotic ICS ecosystem. We then propose ICSBoM[4], the first end-to-end solution that accurately analyzes supply chain vulnerabilities for ICS firmware within such ecosystem. ICSBoM currently focus on analyzing dynamically linked TPCs, while statically linked ICS firmware is outside the scope of this paper. We break down the comprehensive and challenging task into the following major steps. (1) Given an ICS firmware, ICSBoM preprocesses the firmware by discovering all executable files and collecting file metadata. (2) ICSBoM identifies integrated TPCs as well as their versions. (3) ICSBoM looks up Common Vulnerabilities and Exposures (CVEs) affecting each TPC of the identified version in a prebuilt enriched vulnerability database. (4) ICSBoM locates the target function(s) associated with each vulnerability within stripped TPC-related executables. (5) Finally, ICSBoM determines whether each vulnerability remains unaddressed or has been fixed by backporting the security patch to the target function(s). Our contributions are summarized as follows:

- We discover the unique and chaotic ICS ecosystem in supply chain vulnerability assessment, highlighting the mismatch between TPC version-associated vulnerabilities and backported security patches.
- We propose an end-to-end framework, ICSBoM, for automated ICS firmware vulnerability analysis by identifying TPCs and their versions, looking up CVEs, locating target functions, and detecting security patches.
- We evaluate the performance of ICSBoM on a large-scale patch dataset and through detailed firmware case studies. Based on a total of 1646 CVE analyses, ICSBoM achieves an overall accuracy of 100%, 96%, and 86% on

[4] The code for ICSBoM is available at the anonymous GitHub repository https://github.com/unaxiee/ICSBoM.

TPC identification, target function locating, and security patch detection, respectively.
- We observe a substantial number of ICS vendor custom patches applied to integrated TPCs, and characterize their nature and impact on ICS devices. This study spans across 1213 unique patches from 111 different TPCs.

2 Problem Statement

In this section, we first demonstrate the presence of outdated TPCs integrated in ICS devices, and then illustrate the problematic patching practices adopted by ICS vendors to mitigate disclosed vulnerabilities, which directly result in the chaotic ICS ecosystem.

2.1 Outdated TPC Integration

The unique constraints facing ICS such as uninterrupted operations and costly mandatory testing, often force ICS vendors to push back TPC module updates and continue using outdated TPCs [21]. To underline the staleness of TPCs integrated in ICS firmware, we investigate multiple firmware versions from four PLC products covering two ICS vendors, i.e., Siemens and WAGO. Fig. 1 depicts the Cumulative Distribution Function (CDF) of the time difference between the release dates of ICS firmware-integrated TPCs and the release dates of the most recently available TPCs prior to firmware release, following the definition:

$$F_X(x) = P(X \leq x) \tag{1}$$

where the variable X is the day difference and P shows the TPC percentage where X is less than or equal to a given time period x. By calculating the expected value of X as:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x \cdot f_X(x)dx \tag{2}$$

the average time difference is 1384 days across 300 TPCs for Siemens and 494 days across 70 TPCs for WAGO. This indicates that, despite the availability of much newer versions, ICS vendors continue using outdated TPCs that were even released a few years ago. Additionally, WAGO releases firmware updates every 110 days on average while Siemens publishes updates every 134 days. Compared to the firmware update frequency, the integrated TPCs are apparently outdated in ICS firmware.

Even more concerning is that some TPCs are used in ICS firmware beyond their End-of-Life (EoL), after which TPC developers typically stop addressing security issues. We take *OpenSSL* [5], an open-source cryptography library, as an example because it is commonly used by a variety of ICS vendors. In Fig. 2, each sub-figure represents a major release of *OpenSSL* (i.e., 0.9.8, 1.0.2, 1.1.0, and 1.1.1), where the solid line shows the development cycle, and the vertical

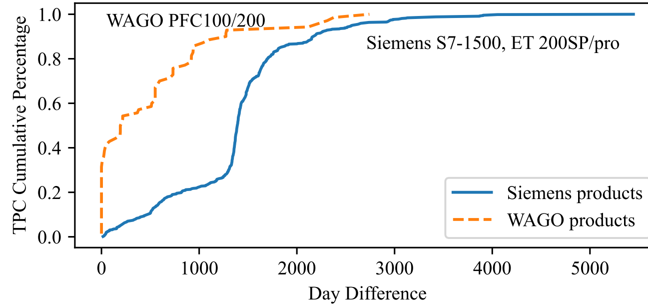

Fig. 1: Cumulative distribution function of the release time difference between ICS firmware-integrated TPCs and the most recently available TPCs prior to firmware release. Siemens shows an average delay of 1384 days across 300 TPCs, and WAGO reports a 494-day delay across 70 TPCs.

line marks the EoL date. The dashed lines indicate the versions of *OpenSSL* integrated in different PLC series. As Fig. 2 shows, Siemens S7-1200 (the first sub-figure from the top) used *OpenSSL* 0.9.8 for four years after the EoL date. Similarly, WAGO PFC100/200 (the second sub-figure) relied on *OpenSSL* 1.0.2 beyond the EoL for two years. Moreover, Fig. 2 shows that the versions of ICS firmware-integrated *OpenSSL* are several releases behind the development cycle, further exemplifying the staleness of integrated TPCs.

2.2 Chaotic Patching Practices

We next present several real-world examples that demonstrate the chaotic patching practices that ICS vendors adopt to mitigate vulnerabilities in outdated TPCs. These practices result in the chaotic ICS ecosystem where TPC versions are no more reliable to report supply chain vulnerabilities.

Incomplete and Unnecessary Security Patch. Siemens SIMATIC IOT-2000 version 3.1.17 was released in July 2022 [42], with *curl* [3] as one of the integrated TPCs supporting data transfer. But the integrated *curl* version 7.69.1 was released two years earlier, in March 2020, and by May 2022, *curl* had published 21 CVEs affecting version 7.69.1 [4]. To address these vulnerabilities, Siemens backported 16 official security patches [67]. However, after carefully reviewing these backported patches, we discover that (1) three security patches for vulnerabilities (CVE-2021-22922, CVE-2021-22923, and CVE-2022-27774) present in the firmware-integrated *curl* version are missing, and (2) three backported patches (for CVE-2020-8286, CVE-2021-22898, and CVE-2021-22925) are unnecessary, as the impacted functionality e.g., the Telnet communication protocol, is not enabled on the device.

ICS Vendor Custom Patch. Similarly, a recent WAGO PFC200-G2 firmware, released in November 2024 [54], integrated an extremely outdated version

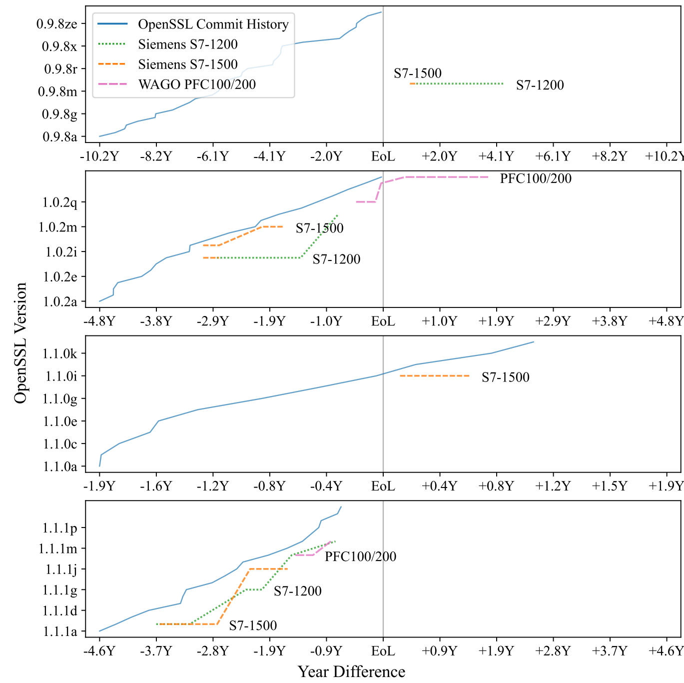

Fig. 2: Comparison of ICS firmware-integrated *OpenSSL* versions with the development cycle and the EoL date of major *OpenSSL* releases. Siemens S7-1200 used *OpenSSL* 0.9.8 for four years beyond the EoL, and WAGO PFC 100/200 used *OpenSSL* 1.0.2 for two years after the EoL.

of *libmodbus* [40], a communication protocol, which was released 13 years ago, in 2013. The vendor continues using this outdated version because they have developed a total of 41 custom patches to expand a variety of vendor-specific features for *libmodbus* [53]. Upon studying the 41 custom patches, we find that the security fixes for two vulnerabilities (CVE-2019-14462 and CVE-2019-14463) are combined into one of the custom patches, further complicating the security patch detection. This complication arises because the impacted functions are not only altered by security fixes but also modified with vendor-specific features. In contrast, the security fix for another vulnerability (CVE-2022-0367) found in *libmodbus* does not appear in any of the custom patches. This finding reinforces our discovery that TPC versions are no longer trustworthy, as some version-associated vulnerabilities are fixed while others remain unaddressed.

3 ICSBoM Design

We first provide an overview of ICSBoM design as depicted in Fig. 3, and describe each component in detail, highlighting the challenges encountered and solutions developed.

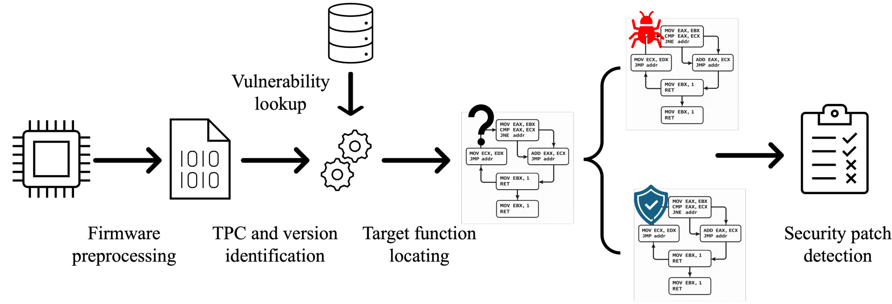

Fig. 3: Overview of ICSBoM. The major steps include firmware preprocessing, TPC and version identification, vulnerability lookup, target function locating, and security patch detection.

3.1 Design Overview

Firmware Preprocessing. Given an ICS firmware, the first step is to dissect the firmware and discover all files that are possibly compiled from any TPCs. The files of interest include executable binaries and shared libraries, which are two major file types that contain executable code. These executable files play a critical role in identifying integrated TPCs and their versions, as well as determining whether TPC-introduced vulnerabilities are still present or corresponding security patches have been backported in subsequent steps.

TPC and Version Identification. We develop a two-layer approach to associate each executable file with a version number. (1) The first layer signature-based version matching leverages the fact that version numbers of some commonly seen executables are embedded in their binary strings. Thus, we create "version signatures", implemented as regular expressions, to capture version numbers from these binary strings. (2) The second layer repository-based version querying is designed for the remaining executables that do not have version signatures. We look up their file names in some online package maintaining repositories, and generate similar version patterns based on version formats observed in the repositories. For TPC identification, we further rely on the package repositories that list all files produced by every package/TPC. We use the relationship between a TPC and its generated files to match each executable file to a proper TPC (e.g., library *libcrypto* to TPC *OpenSSL*).

CVE Database Construction and Lookup. We prebuild an enriched vulnerability database (referred to as the CVE database) by collecting specific CVE information that cannot be directly obtained from existing CVE databases. Each CVE entry in the database includes (1) the fixed version, indicating the software release in which the vulnerability is patched, and (2) the function(s) modified by the associated security patch, referred to as patch-altered functions. We query the database using identified TPCs and their versions, and retrieve the fixed version and patch-altered functions for each TPC-introduced vulnerability.

Target Function Locating. To locate patch-altered functions within stripped executable files extracted from firmware, we first build a reference TPC that matches the identified TPC version. This same-version reference TPC, referred to as the vulnerable reference TPC, contains vulnerabilities associated with that version. Additionally, we build several patched reference TPCs corresponding to the fixed versions. The process of locating patch-altered (target) functions begins by retrieving their counterpart reference functions from the vulnerable reference TPC. We then develop a two-layer hashing-based algorithm. (1) The first layer fuzzy hashing selects several function candidates exhibiting comprehensive similarity to the reference function. (2) The second layer crypto hashing determines the final target function by checking basic block-level identicalness.

Security Patch Detection. For each located target function, we perform a comparative analysis against two reference functions, one from the vulnerable reference TPC and the other from the patched reference TPC. For each target–reference function pair, we first identify the differing basic blocks within the pair, and then quantify the similarity of the pair by analyzing the assembly instructions contained in these differing basic blocks. Finally, by comparing the quantified similarity values of the two pairs, we determine whether the target function more closely resembles the vulnerable or the patched reference function, further deciding whether the vulnerability persists or has been patched.

3.2 Firmware Preprocessing

Firmware Extraction. We observe that ICS firmware can be delivered in a variety of formats ranging from simple archives to filesystem images and other obscure schemes. Processing firmware of various formats and extracting all files from an input firmware are fundamental to subsequent supply chain vulnerability analysis. To that end, we first arm our approach with binary analysis tools like `binwalk` [1] to recursively extract all files within the firmware.

Executable Discovery. Not all files extracted from firmware are relevant to our target. For instance, images, HTML files, PDF files, and text-based user documents are less important in identifying supply chain vulnerabilities and detecting security patches. Therefore, we filter and select files that contain executable code. We identify such executable files using `libmagic` [49] which infers the Multipurpose Internet Mail Extension (MIME) [62] type of a file. We also collect metadata from selected executable files including not only basic information such as file name, size, and path, but also less obvious data such as the MIME type and other attributes identified by `libmagic`.

3.3 TPC and Version Identification

Signature-based Version Matching. We first associate each executable file with a version number using a two-layer version identification approach. Based on our preliminary investigation of commonly observed executable files in ICS firmware across multiple vendors, such as shared libraries from *OpenSSL*, *curl*, and *libmodbus* (these TPCs are briefly discussed in Section 2), we observe that their version numbers are embedded in printable strings within the executables following consistent patterns. Consequently, we develop regular expressions, referred to as version signatures, for a total of 121 commonly encountered executable files. Their precise version numbers can be obtained by matching the version signatures against the output of the `strings` [26] utility. This version signature development is a one-time offline effort.

Repository-based Version Querying. For executable files whose version signatures cannot be determined, or for previously unseen executables for which we do not have the chance to create version signatures, we develop a second layer repository-based version querying approach. We download, parse, and organize package information from the official Arch Linux repository [51] and Arch User Repository (AUR) [11] into a local `MongoDB` [35] database instance. This database contains information for more than 12,000 packages. For each executable file without a predefined version signature, we extract several version number candidates from its printable strings, such as strings in x.y or x.y.z formats. We then query the package database using the executable file name to find the entry with the most similar package name, and retrieve its version number. Finally, the retrieved version number is used as a matching pattern to determine the closest version number among all candidates.

Executable to TPC Pairing. Now we are able to identify the versions of executable files. However, an executable file is not equivalent to a TPC, as a single TPC can produce multiple executables. For example, *OpenSSL* includes the binary tool *openssl* as well as libraries *libcrypto* and *libssl*. Thus, to match each executable file to the appropriate TPC, we reuse the package database, where each package record contains not only the package name and version number, but also the list of files generated by the package. Therefore, we query the database using the executable file name, and search for the entry whose file list contains the most similar file name. Consequently, the package name in the retrieved entry is assigned as the TPC to the executable file.

3.4 CVE Database Construction and Lookup

Publicly accessible CVE databases such as the National Vulnerability Database (NVD) [36], the CVE Program [46], and Bugzilla [2] provide general CVE information. However, these databases do not directly supply the specific data required by our solution. In particular, for each CVE, we need (1) the fixed version in which the vulnerability is patched, and (2) the functions that are altered by the corresponding security patch, referred to as patch-altered functions. Additionally, existing CVE data scrapers, such as cve-search [18] and

Algorithm 1 CVE Data Scraping

```
1: Input: TPC_name, git_repo_name
2: Output: CVEs_info_list
3: CVEs_info_list ← ∅
4: CVEs_NVD_list ← NVD.search(TPC_name)
5: for cve in CVEs_NVD_list do
6:     commit_url ← None
7:     for link in cve.references do
8:         if link.tag == "Patch" and git_repo_name in link.url then
9:             if "commit" in link.url then
10:                commit_url ← link.url
11:            else if "pull" in link.url then
12:                commit_url ← get_commit_from_pull(link.url)
13:            else
14:                add exception handling for special cases
15:                continue
16:            end if
17:            break
18:        end if
19:    end for
20:    if commit_url then
21:        if excluding_version in cve.metadata then
22:            fixed_version ← excluding_version
23:        else
24:            fixed_version ← get_version_from_commit(commit_url)
25:        end if
26:        function_names_list ← get_function_names_list(commit_url)
27:        CVEs_info_list ← CVEs_info_list ∪
                   {(cve.id, fixed_version, function_names_list)}
28:    else
29:        add exception handling for special cases
30:        continue
31:    end if
32: end for
33: return CVEs_info_list
```

OpenCVE [7], mainly reorganize the CVE database contents without extracting this fine-grained information. Consequently, we develop a custom scraper (outlined in Algorithm 1) tailored to extract the fixed versions and patch-altered functions from NVD as well as various version control systems.

Each CVE entry in NVD consists of a brief description, a reference section listing external resource links with tags (e.g., "Patch", "Vendor Advisory", "Third Party Advisory", etc.), and information on affected software versions. During CVE database construction, the scraper processes each CVE entry by identifying all reference links tagged as "Patch", and selecting the links pointing to commits in version control systems (lines 8-18). The scraper then navigates to the corresponding repository, and uses regular expressions to extract the earliest version tag associated with this commit (line 24), to handle cases where the affected version information is incomplete or missing in NVD (i.e., when line 22 fails). Finally, the scraper extracts the names of modified functions in the commit (line 26). Although the construction process may seem straightforward, it must overcome several practical challenges: (1) Not all CVE entries in NVD include links to external patch details. (2) Even listed in NVD, many patch links are broken. (3) Different TPCs use diverse version control platforms such as GitHub, GitLab, and GNU Savannah. (4) The Git protocol implemented in

these platforms only displays text changes, rather than directly providing required patch-altered function names. We address these challenges by extending scraper functionality, implementing exception handling, and applying necessary corrections.

Through this process, we collect 487 CVE entries and confirm the correctness of their fixed versions and patch-altered functions. The CVE database construction is a one-time effort. While running the scraper to enlarge the database is easy, manual verification and correction of CVE information remain time-intensive. Therefore, we select 21 TPCs that are widely used in ICS firmware across multiple vendors, balancing the database coverage and evaluation scale with the manual verification workload. If a new TPC becomes relevant, the scraper can be rerun to automatically extract CVE data, and information verification is again a one-time effort.

3.5 Target Function Locating

Reference TPC Building. Unlike traditional TPC version-based vulnerability scanners, we perform a deeper analysis to determine whether a TPC-introduced vulnerability has been patched. This patch detection focuses on patch-altered functions. The first step is to locate these functions (target functions) within stripped executable files. To this end, we build a reference TPC matching the identified TPC version, referred to as the vulnerable reference TPC, and additional patched reference TPCs corresponding to the fixed versions retrieved from the CVE database. During the build process, all optional features specified in the TPC build script are enabled to ensure the coverage of all potential patch-altered functions. Mutually exclusive features remain at default settings. When a build error occurs, we automatically revert to the default configuration. The target architecture is aligned with the input ICS firmware. For outdated TPC versions, we create historical software environments using snapshots of older Linux distributions to mitigate dependency conflicts and deprecated package issues. After compilation, the entire build directory is scanned, and regular expressions are used to find counterpart executable files in the built reference TPCs.

Assembly Sanitization. We refer to the executable file extracted from firmware as the target binary and assume it contains no debugging information. The counterpart executable file extracted from the vulnerable reference TPC is denoted as the reference binary. Next, we disassemble both the target and reference binaries to obtain their assembly code. To mitigate inconsistencies introduced by differing build environments, such as unknown or unavailable compilers used to generate firmware, which prevent rebuilding the reference TPCs with identical toolchains, we design an assembly sanitization algorithm. The algorithm performs three "specific-to-general" conversions to normalize the assembly code: (1) All immediate values, regardless of base or format, are replaced with the same string "imm". (2) All specific memory addresses are replaced with string "addr". (3) All specific function names are replaced with string "func". This normalization reduces build environment-induced variations, improving the

robustness of target function locating. The effectiveness of the sanitization algorithm is demonstrated in the ablation study presented in Section 4.

Fuzzy Hashing Selection. Our preliminary analysis shows that a security patch is likely to alter only a subset of all basic blocks contained in a target function, while the remaining basic blocks stay unchanged. Therefore, the function-level similarity between the target and reference functions should be preserved to some extent. Meanwhile, they are expected to share several identical basic blocks. Therefore, we first employ TLSH [37], a fuzzy hashing algorithm that generates similar hash values for similar inputs. We apply the algorithm to calculate fuzzy hash values for the reference functions and all functions in the target binary, using the sanitized function-level assembly code as input. For each reference function, we select a predefined number (proportional to the total number of functions in the target binary) of function candidates from the target binary whose hash values are closest to that of the counterpart reference function.

Crypto Hashing Locating. We do not rely solely on the fuzzy hashing algorithm because certain security patches may significantly modify a target function, causing unrelated functions in the target binary to exhibit closer fuzzy hash values than the true match. To address this, we calculate MD5 [60] hash values for all basic blocks within the reference function and each function candidate selected by the fuzzy hashing algorithm. This crypto hash algorithm takes as input the basic block-level sanitized assembly code, and only generates identical hash values for exactly matching inputs. The final target function is determined as the candidate with the highest number of matching crypto hash values compared to those within the reference function. The two-layer hashing-based target function locating algorithm jointly considers function-level similarity and basic block-level identicalness. Notably, the algorithm remains effective when the corresponding security patch has not been applied, as unpatched target functions exhibit greater similarity to their counterpart reference functions and therefore are easier to locate.

3.6 Security Patch Detection

The final step takes as input a triplet consisting of a target function, a vulnerable function, and a patched function. The target function is located from the target binary, and the latter two are extracted from the vulnerable and patched reference TPCs. The key idea is to measure if the target function more closely resembles the vulnerable or patched function by analyzing both function-level structural changes and basic block-level assembly differences. We collect basic block-level information, including assembly code and control flow relationships (i.e., parent and child blocks) for all input functions. Actually, this is completed during the previous step, as they share the same assembly code extraction and sanitization processes, requiring only to store additional basic block relationships for each function.

Different Basic Blocks Discovery. The next two steps are inspired by [66], which detects vulnerabilities across different versions in general software, but overlooks the chaotic patching practices specific to the ICS domain. Therefore,

we improve and design ICSBoM to address the chaotic ICS ecosystem. We first identify differing basic blocks between the target function and the reference function by removing identical basic blocks. This process is conducted twice, once for target-vulnerable pair and once for target-patched pair. In cases where a function contains duplicated basic blocks, we further compare their surrounding contexts, i.e., the assembly of their parent and child basic blocks, to accurately locate the true match. The remaining unmatched blocks consequently represent the different basic blocks that distinguish the function pair.

Similarity Measuring. We then model the relationships of these differing blocks using directed graphs, where nodes represent basic blocks and edges capture their parent–child relationships. For each target–reference pair, two graphs are constructed, one for the target function and one for the reference function, as unmatched basic blocks remain on both sides. For each directed graph, we collect all simple paths that start from a root node and end at a leaf node, representing all possible execution paths, thereby capturing as many control flows among these blocks as possible. Each path is then converted into an instruction stream by concatenating the sanitized assembly code from all basic blocks along the path. Finally, for a pair of target and reference functions, we generate two instruction stream sets. The similarity between the function pair is quantified by computing the average Levenshtein edit distance [61] between instruction streams from the two sets. We justify the choice of Levenshtein distance through an ablation study in Section 4.

Detection Principles. Since a vulnerability may impact multiple target functions and different target-reference function pairs may yield conflicting similarity results, we design three high-level principles, as described in Algorithm 2, to more efficiently and accurately determine whether a security patch has been backported. (1) *Structural Change.* If a security patch introduces new functions or removes existing ones, the algorithm directly returns the final decision by checking for the presence or absence of these functions across the target, vulnerable, and patched binary triplet (lines 9-15). (2) *Majority Voting.* If a security patch alters multiple functions, and these functions produce conflicting decisions on whether the target function is closer to the vulnerable function ("V") or the patched one ("P"), the algorithm counts the occurrences of "V" and "P" decisions (lines 20-24), and choose the final decision that has the larger number of votes (lines 30-34). (3) *Confidence Maximization.* If the numbers of "V" and "P" decisions are equal, the algorithm returns the decision that reports the largest absolute difference between the target-vulnerable and target-patched similarity values (lines 25-28), reflecting the most confident decision.

4 Evaluation

We first evaluate the performance of ICSBoM on a large-scale patch dataset constructed by backporting incomplete security patches to TPCs, thereby reflecting the chaotic ICS ecosystem. Next, we conduct end-to-end ICS firmware case studies spanning across multiple vendors and widely used devices in the in-

Algorithm 2 Security Patch Detection Principles

```
1: Input: CVE_number
2: Output: final_decision
3: function_triplet_list ← CVE_database.search(CVE_number)
4: vulnerable_num ← 0
5: patched_num ← 0
6: decision_list ← ∅
7: max_sim_diff ← 0
8: for function_triplet in function_triplet_list do
9:     if function_triplet.target and function_triplet.vulnerable and
       not function_triplet.patched then
10:        final_decision ← "V"
11:        return final_decision
12:    else if function_triplet.target and function_triplet.patched and
       not function_triplet.vulnerable then
13:        final_decision ← "P"
14:        return final_decision
15:    end if
16:    decision ← patch_detection(function_triplet.target,
                  function_triplet.vulnerable, function_triplet.patched)
17:    decision_list ← decision_list ∪ {decision}
18: end for
19: for decision in decision_list do
20:    if decision.result == "V" then
21:        vulnerable_num = vulnerable_num + 1
22:    else if decision.result == "P" then
23:        patched_num = patched_num + 1
24:    end if
25:    if decision.sim_diff > max_sim_diff then
26:        max_sim_diff = decision.sim_diff
27:        final_decision = decision.result
28:    end if
29: end for
30: if vulnerable_num > patched_num then
31:    final_decision = "V"
32: else if vulnerable_num < patched_num then
33:    final_decision = "P"
34: end if
35: return final_decision
```

dustry. Finally, we conclude the section by analyzing ICS vendor custom patches and characterizing their nature and impact, which supplements the discussion in Section 2.

4.1 Patch Dataset Evaluation

Facing the challenge of limited availability of open-source ICS firmware build tools and closed-source downloadable firmware images, in order to perform a large-scale analysis, we collect a patch dataset based on the widely used Yocto Poky project [68], which provides a prepackaged build environment for custom embedded device images for the industry. The Poky maintainers actively backport security patches for disclosed vulnerabilities. However, it is inevitable that some newly discovered vulnerabilities remain unpatched when certain Poky project releases are published. Therefore, by building firmware images sourced from different versions of the Poky project, our dataset finally contains 168 backported (BP) and 300 non-backported (NB) security patches involving more than

Table 1: Accuracy of target function locating and security patch detection.

Function Locating		Patch Detection					
ICSBoM				ICSBoM		BinXray	
Tol_{FL}	Corr_{FL}	Tol_{BP}	Tol_{NB}	Corr_{BP}	Corr_{NB}	Corr_{BP}	Corr_{NB}
1132	1097	168	300	126	271	70	160
Acc_{FL}	**0.969**	Acc_{PD}		**0.750**	**0.903**	0.417	0.533

1100 patch-altered functions. For preprocessing, we use IDA Pro [27] to disassemble binaries and collect basic block-level information (e.g., assembly code) for each function.

Accuracy. As shown in the first two columns in Table 1, the patch dataset includes a total of 1132 patch-altered functions (Tol_{FL}), of which ICSBoM correctly locates 1097 (Corr_{FL}), achieving an accuracy of around 97%. This result demonstrates the effectiveness of the two-layer hashing-based algorithm and provides a solid foundation for the subsequent security patch detection step. The third and fourth columns summarize the total numbers of BP (Tol_{BP}) and NB (Tol_{NB}) security patches in the patch dataset. The next two columns list the total numbers of correctly detected NB (Corr_{NB}) and BP (Corr_{BP}) patches by ICSBoM. The overall detection accuracies achieve above 90% and 75%, respectively. It is reasonable that NB patch detection accuracy is higher than that of BP patches, as a patched reference function contains not only security fixes but also potential new features or other functional changes. Finally, we reimplement the state-of-the-art patch detection algorithm BinXray [66] and evaluate it on our patch dataset. BinXray extracts find-grained code differences between vulnerable and patched functions as signatures, and detects patch presence by matching these signatures against target functions. This design assumes that patch-induced changes remain stable and localized, such that signatures extracted from reference functions can generalize to target functions. However, as shown in the last two columns in Table 1, BinXray performs poorly in the chaotic ICS ecosystem, while ICSBoM improves the overall accuracy by nearly 30%. This is because chaotic patching practices often introduce additional changes beyond the original security fix, causing the extracted signatures to deviate from actual patch patterns in ICS firmware.

Efficiency. We summarize the execution time of ICSBoM and BinXray in Table 2, and visualize the results as Fig. 4. Both solutions utilize control flow graphs generated based on basic blocks, which significantly affect the running efficiency. Therefore, the maximum number of basic blocks that the solution can handle must be carefully chosen, to guarantee the solution not only achieves desired detection accuracy but also completes the execution within a reasonable time frame. For ICSBoM, we set the threshold to 100 basic blocks, and the resulting execution time is on average 0.393 seconds. The authors of BinXray choose 40 basic blocks as the threshold, reporting shorter execution time than

Table 2: Efficiency of security patch detection.

Solution	Max # of Basic Blocks	Avg_{time}(s)	Skip_{PD}	Acc_{PD}
BinXray	40	**0.156**	0.597	0.491
	50	2.419	0.561	0.517
	60	4.939	0.525	0.538
	70	136.259	0.482	0.551
ICSBoM	**100**	0.393	–	**0.848**

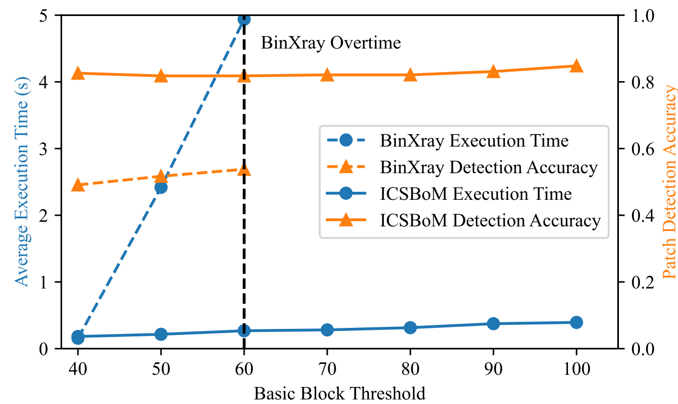

Fig. 4: Execution time and detection accuracy of ICSBoM and BinXray as basic block threshold increases. The lines stop for BinXray when the threshold equals 60, because it takes more than 24 hours to finish the experiments when the threshold is larger than 60.

ICSBoM. However, this threshold causes their solution to skip nearly 60% of the located target functions (Skip_{PD}) that contain more than 40 basic blocks, thereby making no contribution to patch detection. Consequently, we increase the threshold to allow their solution to detect more target functions. However, as Table 2 shows, increasing the threshold only slightly decreases Skip_{PD} with execution time increasing rapidly and detection accuracy improving marginally. This limitation stems from the mismatch between extracted patch signatures and actual patch patterns, which significantly increases the number of basic blocks involved in matching while fundamentally limiting the detection accuracy. In comparison, ICSBoM is able to handle functions containing a large number of basic blocks. Meanwhile, the execution time does not change much as the threshold increases, and the accuracy remains high even with low thresholds as Fig. 4 shows.

Ablation Studies. Since both target function locating and security patch detection are conducted at binary level, the effect of the compiler must be con-

Table 3: Ablation studies on effect of compiler and edit distance algorithm.

ICSBoM Variation	Acc_{FL}	Acc_{PD}	Avg_{time}(s)
Baseline (`x86_64-poky-gcc` + Levenshtein distance)	**0.969**	**0.848**	0.393
Replaced by `gcc`	0.937	0.825	**0.235**
Replaced by Jaro distance	–	0.814	0.799
Replaced by Jaro-Winkler distance	–	0.797	0.809

sidered. The TPCs integrated in firmware images are compiled using the Poky-provided cross-compiler [69], and each firmware image is built with a unique compiler version. We first build the reference TPCs using the latest version of [69] as our baseline, achieving function locating and patch detection accuracies of 96.9% and 84.8%, respectively; this demonstrates that ICSBoM is robust against different compiler versions. We then recompile all reference TPCs using the built-in `gcc` 11.4.0 on Linux Ubuntu 22.04 and repeat the experiments. As Table 3 shows, when the target and reference TPCs are built with different compilers, the performance of ICSBoM only drops by approximately 2.5% on average, confirming the effectiveness of the assembly sanitization algorithm in mitigating compiler-induced influences.

We also justify the choice of Levenshtein distance as the metric for quantifying assembly differences. The baseline utilizes Levenshtein distance [61], which counts the minimum number of insertion, deletion, and substitution operations to convert one string to another. For comparison, we implement two alternative metrics, i.e., Jaro distance and its variant Jaro-Winkler distance [59], both of which rely solely on transposition operations. Table 3 shows Levenshtein distance outperforms the other two in terms of accuracy and efficiency. This is because assembly-level changes introduced by security patches are more effectively captured by string insertions and deletions rather than by character transpositions within strings.

4.2 Detailed Firmware Case Studies

Next, we conduct end-to-end case studies on 14 firmware covering various ICS vendors and devices. In the first two analyses, the vendors release the source code of firmware build tools, allowing us to compare to the ground truth. The subsequent case studies are performed in a black-box manner, as the firmware are completely closed-source. We focus on TPCs contained in our CVE database and these case studies encompass a total of 680 CVE analyses and involve more than 1100 patch-altered functions.

Siemens. We first test ICSBoM on three Siemens IOT2000 firmware as listed in Table 4. All firmware images are built using the IOT2000 build tool with source code available in the Siemens GitHub repository [42]. Thus, we have full knowledge of their TPC usage such as TPC versions and patch statuses.

Table 4: Analysis of multiple versions of Siemens IOT2000.

Third-party	3.1.17					3.1.1					2.6.0				
Component	Ver_{TPC}	TF_{VI}	Acc_{FL}	Acc_{NB}	Acc_{BP}	Ver_{TPC}	TF_{VI}	Acc_{FL}	Acc_{NB}	Acc_{BP}	Ver_{TPC}	TF_{VI}	Acc_{FL}	Acc_{NB}	Acc_{BP}
curl	7.69.1	✓	40/43	16/17	9/13	7.69.1	✓	42/43	28/30	0/0	7.61.0	✓	40/43	28/30	2/3
dbus	1.12.20	✓	4/4	3/3	0/0	1.12.16	✓	10/10	5/5	0/0	1.2.10	✓	11/11	6/6	0/0
e2fsprogs	1.45.7	✓	2/2	1/1	0/0	1.45.4	✓	3/3	1/1	1/1	1.44.3	✓	6/7	3/3	0/0
expat	2.2.9	✓	21/22	1/4	12/15	2.2.9	✓	23/23	16/19	0/0	2.2.6	✓	25/27	16/22	0/0
libarchive	3.4.2	✓	6/6	4/4	1/1	3.4.2	✓	6/6	4/5	0/0	3.3.3	✓	16/16	11/13	0/0
libgcrypt	1.8.5	✓	9/10	1/1	3/3	1.8.5	✓	9/10	3/3	1/1	1.8.4	✓	10/10	4/4	0/0
libpcap	1.9.1	✓	–	–	–	1.9.1	✓	–	–	–	1.8.1	✓	1/1	1/1	0/0
libtirpc	1.2.6	✓	1/1	1/1	0/0	1.2.5	✓	1/1	1/1	0/0	–	–	–	–	–
libxml2	2.9.10	✓	42/46	9/9	7/9	2.9.10	✓	46/47	16/16	1/2	2.9.8	✓	49/50	18/19	1/2
ncurses	6.2	✓	14/14	2/2	2/2	6.2	✓	14/14	4/4	0/0	6.1	✓	14/14	4/4	0/0
openssh	8.2p1	✓	29/29	6/6	1/2	8.2p1	✓	29/29	8/8	0/0	7.8p1	✓	35/35	8/10	0/0
openssl	1.1.1o	✓	10/11	7/8	0/0	1.1.1g	✓	19/22	14/17	0/0	1.1.1b	✓	31/32	19/21	1/1
util-linux	2.35.1	✓	3/3	0/0	2/3	2.35.1	✓	3/3	3/3	0/0	2.32.1	✓	1/1	1/1	0/0
zlib	1.2.11	✓	9/9	0/1	1/1	1.2.11	✓	9/9	1/2	0/0	1.2.11	✓	8/9	1/2	0/0
Sum/Acc	14	1.000	0.950	0.895	0.776	14	1.000	0.973	0.912	0.750	13	1.000	0.965	0.882	0.667

As listed in the first column in Table 4, we focus on 14 TPCs. For each firmware version, the first two columns (Ver_{TPC} and TF_{VI}) present the TPC versions identified by ICSBoM as well as whether the results are correct or not. The next column (Acc_{FL}) reports the accuracy of target function locating. Specifically, the numbers before and after the forward slash represent the correctly located functions and the total functions, respectively. The final two columns (Acc_{NB} and Acc_{BP}) list the accuracy of patch detection for NB and BP security patches in a similar format. Overall, ICSBoM achieves 96% accuracy on target function locating, and patch detection accuracies of 90% for NB patches and 76% for BP patches. Additionally, the dash symbol in Table 4 indicates either the TPC version is new enough so that there is no CVEs associated with the TPC, or the TPC is not integrated into the firmware.

WAGO. We next evaluate ICSBoM on multiple WAGO devices. All firmware showcased in Table 5 and Table 6 are downloaded from the WAGO GitHub repository [56] where the source code of firmware build tools is available, providing us a full understanding of their TPC usage.

Table 5 follows the same structure as Table 4, listing three different WAGO PLC product lines, namely CC, TP, and PFC. All firmware are built from a recent release (FW-26). Notably, different PLC series actually integrate identical versions of TPCs except that *libmodbus* uniquely appears in the PFC product. Although TPC versions remain the same across different PLC series, the specific firmware build environment varies, resulting in slight differences in the performance of ICSBoM. As Table 5 shows, the accuracy of target function locating achieves around 93% across all three PLC series. The accuracy of security patch detection is 93% for the CC and TP products. However, for the PFC device, the patch detection accuracy decreases to 88% because ICSBoM fails to detect two backported patches in *libmodbus*.

Table 6 serves as a longitudinal supplement to Table 5 with each sub-table presenting a specific version of WAGO PFC firmware and providing detailed

Table 5: Analysis of multiple devices of WAGO.

Third-party Component	Ver_{TPC}	TF_{VI}	CC		TP		PFC	
			Acc_{FL}	Acc_{PD}	Acc_{FL}	Acc_{PD}	Acc_{FL}	Acc_{PD}
curl	8.0.1	✓	19/20	6/7	19/20	6/7	18/20	6/7
dbus	1.12.20	✓	5/5	4/4	5/5	4/4	5/5	4/4
e2fsprogs	1.45.6	✓	1/2	1/1	1/2	1/1	2/2	1/1
expat	2.5.0	✓	7/7	2/2	7/7	2/2	7/7	2/2
libarchive	3.6.1	✓	2/2	1/1	2/2	1/1	2/2	1/1
libmodbus	3.0.5	✓	–	–	–	–	1/1	1/3
libssh2	1.9.0	✓	1/3	3/3	1/3	3/3	1/3	3/3
libxml2	2.10.3	✓	7/7	3/4	7/7	3/4	7/7	3/4
ncurses	6.2	✓	6/6	4/4	6/6	4/4	6/6	4/4
openssl	1.1.1u	✓	1/1	2/2	1/1	2/2	1/1	2/2
util-linux	2.36	✓	3/3	2/2	3/3	2/2	3/3	2/2
Sum/Acc	11	1.000	0.929	0.933	0.929	0.933	0.930	0.879

results on TPCs whose versions differ from those in the immediately subsequent firmware release. Taking the sub-table 6(a) as an example, FW-25 integrates an older version of *OpenSSL* compared to the subsequent release FW-26, while the rest 10 TPCs listed in Table 5 share the same versions and patch statuses with those in FW-26 (summarized as “Non-changed TPCs" row in Table 6(a)). Across different firmware versions, the accuracy of target function locating remains stable at around 93%, and the security patch detection accuracy ranges from 82% to 90%. Note that FW-23 is excluded from Table 6 due to the absence of downloadable firmware image in the WAGO GitHub repository.

ABB. In the following case studies, all firmware are analyzed in a black-box manner, as they are closed-source flahsable images downloaded from vendor repositories without any build information. We first test two AC500 firmware from ABB [6], and summarize key testing statistics in Table 7. ICSBoM identifies the versions for 47 executable files. After matching these executable files to their corresponding TPCs, a total of 41 TPCs are identified. For target function locating, we focus on patch-altered functions that have identifiable names after disassembling the stripped binaries, enabling both accuracy reporting and subsequent patch detection. Among 40 target functions with identifiable names, ICSBoM correctly locates 37 of them. Finally, ICSBoM determines the patch status for 18 CVEs, and discovers one possible backported security patch (for CVE-2021-37600) in both firmware.

Eaton. We also test two firmware from Eaton [22], a vendor specializing in providing industrial solutions to power systems, and present the results in Table 7. Similarly, ICSBoM identifies version information for 52 executable files, which are subsequently matched to 32 TPCs. ICSBoM successfully locates all target functions with identifiable names, achieving an overall target function locating accuracy of 93% across the two vendors. It is worth noting that many

Table 6: Analysis of multiple versions of WAGO PFC.

(a) WAGO PFC FW-25.

TPC	Ver_{TPC}	TF_{VI}	Acc_{FL}	Acc_{PD}
openssl	1.1.1t	✓	6/6	4/5
Non-changed TPCs	–	–	52/56	27/31
Sum/Acc	11	1.000	0.935	0.861

(b) WAGO PFC FW-24.

TPC	Ver_{TPC}	TF_{VI}	Acc_{FL}	Acc_{PD}
curl	7.87.0	✓	24/25	13/14
openssl	1.1.1q	✓	9/11	8/9
Non-changed TPCs	–	–	34/36	21/24
Sum/Acc	11	1.000	0.931	0.894

(c) WAGO PFC FW-22.

TPC	Ver_{TPC}	TF_{VI}	Acc_{FL}	Acc_{PD}
curl	7.81.0	✓	30/33	20/23
dnsmasq	2.83	✓	12/12	1/1
expat	2.4.7	✓	9/9	2/4
libarchive	3.3.3	✓	14/15	10/12
libxml2	2.9.13	✓	28/30	7/8
openssl	1.1.1n	✓	8/11	9/9
zlib	1.2.11	✓	9/9	2/2
Non-changed TPCs	–	–	18/20	15/17
Sum/Acc	13	1.000	0.921	0.868

(d) WAGO PFC FW-21.

TPC	Ver_{TPC}	TF_{VI}	Acc_{FL}	Acc_{PD}
expat	2.2.9	✓	19/22	14/19
libpcap	1.8.1	✓	1/1	1/1
libxml2	2.9.10	✓	45/46	14/18
openssl	1.1.1l	✓	9/12	7/10
Non-changed TPCs	–	–	83/89	48/55
Sum/Acc	14	1.000	0.924	0.816

Table 7: Analysis on ABB and Eaton firmware.

ICS	TPC and Version Identification		Function Locating		Patch Detection	
Vendor	# of Executables	# of TPCs	Tol_{FL}	$Corr_{FL}$	Tol_{case}	Tol_{PD}
ABB	47	41	40	37	40	18
Eaton	52	32	4	4	6	6
Sum/Acc	99	73	0.932		46	24

target functions have non-inferable names, preventing subsequent security patch detection, and therefore are excluded from these case studies. Based on detection on located target functions, ICSBoM does not discover any backported security patches in Eaton firmware.

4.3 ICS Vendor Custom Patch Study

Finally, we perform a large-scale study spanning across 111 different TPCs and 1213 unique patches from WAGO PFC devices [52, 55] to provide insights into vendor custom patch characteristics.

Platform, Configuration, and Building. A large number of patches are created to configure TPCs used in firmware to adjust their respective build processes. These patches vary in scope, from minor adjustments, such as *acl*'s *0001-*

builddefs-save-C-and-CPPFLAGS-from-configure-and-use.patch, which merely adds some compiler flags, to more extensive modifications, such as *php*'s *0001-Fix -dl-cross-compiling-issue.patch* and *0002-Fix-strcasestr-cross-compiling-issue.patch*, which add libraries and accommodate for cross-compilation. Platform-specific configuration patches are also present. Notably, several hundred such patches are applied while building the Linux kernel and the *Barebox* boot loader for firmware images.

Feature-adding and Functionality-altering. Another significant portion of patches focus on expanding the features of TPCs, or specializing their functionality for PLCs. For instance, as discussed in Section 2, *libmodbus* is extensively patched to introduce additional capabilities, such as UDP support (*0001-udp-extension.patch*), multi-connection test for clients (*0002-multi-connection-client-test.patch*), and extended function support (*0035-add-extended-function-support.patch*). Functionality-altering patches are also common. For example, *pure-ftpd*'s *ftp_ change_ anonymous .patch* modifies the handling of anonymous users, treating them as normal users to align with vendor default settings. Additionally, patches that bring back functionality that is disabled by TPC developers, have been spotted, as is the case with *busybox*'s *0200-reactivate-check-for-tty.patch* that restores a previously disabled testing condition.

Bug-fixing and Maintenance. Patches for bug fixes are also prevalent across many TPCs, addressing issues of varying complexity. These range from simple one-line fixes, such us *fakeroot*'s *0003-xattr-increase-MAX_ IPC_ BUFFER _ SIZE-to-1024.patch*, which increases a buffer size, to more complex patches, such as addressing a memory leak in *Azure*'s IoT C SDK (*0002_ fix_ memory_ leak _ of_ github_ issue_ 190.patch*). Some patches address rare issues, like the loss of synchronization in *libModbus*, resolved by *0012-fix-rare-case-of-out-of-sync-parser.patch* through buffer flushing. Maintenance patches are also present. For example, *disable_ deprecated_ ulog_ extension.diff* in *ebtables* disables the deprecated ULog extension, which is a file format used for logging messages.

Logging, Debugging, and Error-handling. There are also numerous patches that add or modify the logging capabilities of TPCs. For instance, *OpenVPN*'s *0001-always-append-to-logfile.patch* changes the log output to append mode, ensuring compatiblility with logrotate's copytruncate option, which manages the size of log files. Similarly, *Ipwatchd*'s *extend-action-script.patch* introduces a new log file type, while both *net-snmp* and *libModbus* have patches that modify their logging mechanisms. Patches aiming at enhancing debugging are also present, as is the case with *busybox*'s *0203-scripts-trylink-honour-SKIP_ STRIP-and-don-t-strip-if.patch* which skips symbol stripping during the compilation process, aiding in degugging. Additionally, patches that add error handling code can be found in TPCs like *libmodbus* and *mosquitto*.

5 Discussion

Putting it All Together and Takeaways. Integrating TPCs to extend functionality and enhance service quality is common across different domains. How-

ever, this integration inevitably introduces supply chain vulnerabilities into the respective domains. In general software ecosystems, these vulnerabilities can be mitigated by promptly updating TPCs to their latest versions, as services are interruptible and updates are easy to perform. Even in practical the updates cannot be performed timely, the TPC versions are reliable indicators to associate supply chain vulnerabilities, which inspire a lot of TPC version-based vulnerability scanners.

In contrast, ICS face stringent uptime requirements where conducting mandatory and time-consuming testing after regular updates is impractical. Therefore, the seemingly optimal backporting practice is adopted by vendors to mitigate disclosed vulnerabilities. Unfortunately, this backporting practice creates a fundamental mismatch between TPC versions and version-associated vulnerabilities, leading to the chaotic ICS ecosystem. This situation is further exacerbated by the selective nature of patching, such as released patches may not be backported promptly, or may fail to apply due to compatibility issues. Such chaotic patching patterns that some vulnerabilities are patched while others remain unaddressed, are observed across many TPCs in various devices from well-known vendors, highlighting the chaos in the ICS firmware supply chain.

Therefore, we develop ICSBoM to improve supply chain transparency. As the first solution of its kind, ICSBoM enables end-users to independently evaluate the security posture of their devices, thereby establishing trust from critical infrastructure down to devices without requiring vendor cooperation or intellectual property disclosure. Moreover, the detected vulnerabilities and corresponding threats can proactively alert vendors to issue timely updates and strengthen their security maintenance practices.

Limitations and Future Directions. ICSBoM also has several limitations. First, it follows a pipeline structure in which each step relies on the output of the preceding step. Consequently, if any early step produces incorrect results, subsequent steps are unable to perform accurate assessments. This cascading effect can propagate a single error throughout the pipeline, potentially compromising the final outcome. For this reason, we exclude patch-altered functions with non-inferable names in our black-box firmware case studies, as their correctness cannot be verified. Moreover, the current implementation of ICSBoM only supports firmware that incorporate TPCs as shared objects within an operating system. This structured organization allows us to streamline the extraction of TPC-related information. In contrast, for statically linked monolithic firmware, gathering such information remains challenging because a single executable file often embeds multiple heterogeneous TPCs. As future work, we plan to extend ICSBoM to support a broader range of firmware architectures, including those employing static linking or custom runtime environments.

6 Related Work

Commercial SBoM Tools. Existing commercial SBoM tools can be broadly categorized into source-based, artifact-based, and asset visibility platforms. Source-

based tools, such as Black Duck [12], Snyk Open Source [43]. and Mend.io [34], analyze dependency manifests, build configurations, or source code to construct dependency graphs and map components to known vulnerabilities. These tools are effective when software dependencies are explicitly declared and vulnerability statuses align with upstream versions. Artifact-based tools, including Anchore [8] and JFrog Xray [28], extend to handle compiled artifacts such as container images by extracting files and applying signature-based matching to identify embedded components. While they relax the requirement for source code, their analysis still relies on recognizable file structures and known signatures. In industrial environments, asset visibility platforms such as Claroty xDome [16] and Armis Centrix [10] provide SBoM functionality by inferring software components from network traffic and device fingerprints, enabling large-scale asset discovery, vulnerability detection, and risk management. As commercial solutions, these tools rely on proprietary techniques that are not publicly disclosed.

Firmware Analysis. Firmware emulation has been studied along several directions. Automated emulation systems, such as Avatar [71], FIRMADYNE [15] and FirmAE [29], aim to execute firmware images by reconstructing runnable environments. To address the lack of hardware support, a second line of work focuses on peripheral modeling, including P2IM [24], μEmu [79], and the specification-guided approach [80]. Rehosting-based approaches execute firmware components without fully emulating hardware by adapting them to generic environments. HALuciator [17] replaces hardware abstraction layer functions. Greenhouse [45] targets user-space services, while FirmSolo [9] focuses on kernel modules. Overall, these emulation-based approaches primarily enhance emulation capacities such as functionality, scalability, and compatibility, while treating vulnerability analysis as a downstream application rather than a central objective. In parallel, fuzzing is an important approach to triggering runtime faults and uncovering security violations in firmware. Early representative work FIRM-AFL [78] enables high-throughput fuzzing by augmenting process emulation. Subsequent studies improve fuzzing fidelity. For example, Fuzzware [41] models memory-mapped I/O, while P_{EMU} [13] generates network protocol-aware inputs. ICS-specific fuzzing approaches, including ICSFuzz [47], FieldFuzz [14], and ICSQuartz [50], further incorporate domain-specific features such as I/O behaviors, network interfaces, and scan cycle semantics. Nevertheless, these approaches target specific processes, applications, or components, and are not designed to identify integrated TPCs and detect supply chain vulnerabilities.

TPC Detection Framework. On the other hand, identifying TPCs and reporting associated vulnerabilities within software ecosystems have been the focus of several recent studies. In the Android ecosystem, LibID [73] extracts obfuscation-resilient semantic features to identify TPCs and their versions. ATV-Hunter [72] focuses on precisely pinpointing TPC versions by using fine-grained features to distinguish between similar versions. LibScan [63] improves detection precision through multi-level matching based on class, method, and call-chain features. Recent work considers the impact of modern build processes. For example, LibHunter [64] studies and mitigates the effects of code optimization,

while LibSleuth [74] designs a method-level detection strategy which is robust to code shrinking. Similar efforts have been made to uncover IoT firmware supply chain security risks. For instance, FIRMSEC [75,76] identifies TPCs at version level and links them to known vulnerabilities. UVSCAN [77] studies IoT firmware supply chain vulnerabilities through a different angle, by detecting TPC usage violations. However, these approaches mainly rely on aligning identified TPC versions with known vulnerabilities and do not perform a deeper function-level vulnerability analysis. As a result, they cannot apply to the ICS domain, where backporting is a common practice and TPC version numbers are no longer reliable indicators of vulnerability statuses.

Binary Function Similarity. DiscovRE [23] uses graph matching algorithms based on Control Flow Graphs (CFGs), enabling similar binary function search but with high computational cost. To improve scalability, Genius [25] converts CFGs to high-level feature vectors using a codebook, while Gemini [65] further leverages neural networks to learn embeddings from CFGs. GMN [31] develops a graph matching network that directly computes the function similarity score for two input CFGs. OrderMatters [70] enhances CFG-based embeddings by modeling node ordering information. Asm2Vec [20] focuses on assembly code and learns function embeddings in an unsupervised manner. To enhance learning ability, SAFE [33] incorporates a self-attention mechanism over instruction sequences, while jTrans [58] leverages a transformer-based language model with control flow-aware design. Trex [38] introduces a dynamic module and incorporates function micro-traces. For better transferability, VulHawk [32] converts binaries to a known compiling environment, while CLAP [57] learns binary embeddings with the help of aligned neural language descriptions. Compared to the learning-based methods, our non-learning solution does not rely on training and can indicate the code change associated with a backported patch, which helps explain how the patch is detected.

7 Conclusion

Our study reveals the unique and chaotic ICS ecosystem in the context of supply chain vulnerability analysis, where TPC versions are no longer trustworthy to associate vulnerabilities due to the backporting practice. To address this, we propose ICSBOM, an end-to-end solution for ICS firmware supply chain vulnerability analysis. ICSBOM takes as input an ICS firmware, identifies integrated TPCs and their versions, collects TPC-introduced vulnerabilities, locates impacted functions, and finally determines whether vulnerabilities persist or have been addressed by backporting security patches. ICSBOM achieves an overall accuracy of 100%, 96%, and 86% on TPC identification, function locating, and patch detection, respectively. Furthermore, through the analysis of 1213 unique vendor custom patches across 111 TPCs, we provide insights into the characteristics and impact of ICS vendor custom patches.

References

1. binwalk: Firmware analysis tool. https://github.com/ReFirmLabs/binwalk (2024)
2. Bugzilla. https://www.bugzilla.org/ (2026)
3. curl: command line tool and library for transferring data with urls. https://curl.se/ (2026)
4. curl cves. https://curl.se/docs/security.html (2026)
5. Openssl library. https://openssl-library.org/ (2026)
6. ABB: Ac500 plcs. https://new.abb.com/plc/programmable-logic-controllers-plcs/ac500 (2026)
7. Amber Security SAS: Opencve: Opensource vulnerability management platform. https://www.opencve.io/ (2026)
8. Anchore: Software composition analysis for cloud-native applications. https://anchore.com/platform/ (2026)
9. Angelakopoulos, I., Stringhini, G., Egele, M.: {FirmSolo}: Enabling dynamic analysis of binary linux-based {IoT} kernel modules. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 5021–5038 (2023)
10. Armis Inc.: Streamline cybersecurity operations with armis centrix, the cyber exposure management platform. https://www.armis.com/platform/armis-centrix/ (2026)
11. aurweb Development Team: Arch user repository. https://aur.archlinux.org/ (2026)
12. Black Duck Software, Inc.: One platform. complete application security. https://www.blackduck.com/ (2026)
13. Bley, M., Scharnowski, T., Wörner, S., Schloegel, M., Holz, T.: Protocol-aware firmware rehosting for effective fuzzing of embedded network stacks. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 4484–4498 (2025)
14. Bytes, A., Rajput, P.H.N., Doumanidis, C., Maniatakos, M., Zhou, J., Tippenhauer, N.O.: Fieldfuzz: In situ blackbox fuzzing of proprietary industrial automation runtimes via the network. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses. pp. 499–512 (2023)
15. Chen, D.D., Woo, M., Brumley, D., Egele, M.: Towards automated dynamic analysis for linux-based embedded firmware. In: NDSS. vol. 1, pp. 1–1 (2016)
16. Claroty: See everything. secure anything. https://claroty.com/platform (2026)
17. Clements, A.A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., Payer, M.: {HALucinator}: Firmware re-hosting through abstraction layer emulation. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1201–1218 (2020)
18. cve-search Team: cve-search project. https://www.cve-search.org/ (2026)
19. Cybersecurity & Infrastructure Security Agency: Apache log4j vulnerability guidance. https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance (April 2022)
20. Ding, S.H., Fung, B.C., Charland, P.: Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 ieee symposium on security and privacy (sp). pp. 472–489. IEEE (2019)
21. Doumanidis, C., Xie, Y., Rajput, P.H., Pickren, R., Sahin, B., Zonouz, S., Maniatakos, M.: Dissecting the industrial control systems software supply chain. IEEE Security & Privacy **21**(4), 39–50 (2023)

22. Eaton: Eaton ups, rack pdu and network connectivity firmware downloads. https://www.eaton.com/us/en-us/products/backup-power-ups-surge-it-power-distribution/firmware-downloads.html (2026)
23. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E., et al.: Discovre: Efficient cross-architecture identification of bugs in binary code. In: Ndss. vol. 52, pp. 58–79 (2016)
24. Feng, B., Mera, A., Lu, L.: {P2IM}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1237–1254 (2020)
25. Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H.: Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. pp. 480–491 (2016)
26. Free Software Foundation: strings. https://sourceware.org/binutils/docs/binutils/strings.html (2026)
27. Hex-Rays: Ida pro: Powerful disassembler, decompiler & debugger. https://hex-rays.com/ida-pro (2026)
28. JFrog: Detect, prioritize and remediate open source risks across your sdlc. https://jfrog.com/xray/ (2026)
29. Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., Kim, Y.: Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In: Proceedings of the 36th Annual Computer Security Applications Conference. pp. 733–745 (2020)
30. Kovacs, E.: Ics vendors respond to log4j vulnerabilities. https://www.securityweek.com/ics-vendors-respond-log4j-vulnerabilities/ (January 2022)
31. Li, Y., Gu, C., Dullien, T., Vinyals, O., Kohli, P.: Graph matching networks for learning the similarity of graph structured objects. In: International conference on machine learning. pp. 3835–3845. PMLR (2019)
32. Luo, Z., Wang, P., Wang, B., Tang, Y., Xie, W., Zhou, X., Liu, D., Lu, K.: Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. In: NDSS (2023)
33. Massarelli, L., Di Luna, G.A., Petroni, F., Baldoni, R., Querzoni, L.: Safe: Self-attentive function embeddings for binary similarity. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 309–329. Springer (2019)
34. Mend.io: Software bill of materials (sbom) - know exactly where you stand with your open source components. https://www.mend.io/sbom/ (2026)
35. MongoDB, Inc.: Mongodb. https://www.mongodb.com/ (2026)
36. National Institute of Standards and Technology (NIST): National vulnerability database. https://nvd.nist.gov/ (2026)
37. Oliver, J., Cheng, C., Chen, Y.: Tlsh–a locality sensitive hash. In: 2013 Fourth Cybercrime and Trustworthy Computing Workshop. pp. 7–13. IEEE (2013)
38. Pei, K., Xuan, Z., Yang, J., Jana, S., Ray, B.: Trex: Learning execution semantics from micro-traces for binary similarity. arXiv preprint arXiv:2012.08680 (2020)
39. Pickren, R., Shekari, T., Zonouz, S., Beyah, R.: Compromising industrial processes using web-based programmable logic controller malware. In: Network and Distributed System Security Symposium (NDSS) (2024)
40. Raimbault, S.: libmodbus: Open source library to communicate with modbus devices. https://libmodbus.org/ (2026)
41. Scharnowski, T., Bars, N., Schloegel, M., Gustafson, E., Muench, M., Vigna, G., Kruegel, C., Holz, T., Abbasi, A.: Fuzzware: Using precise {MMIO} modeling for effective firmware fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 1239–1256 (2022)

42. Siemens: Siemens simatic iot2000 yocto board support package. https://github.com/siemens/meta-iot2000 (March 2025)
43. Snyk: Open source risk management made for developers. https://snyk.io/product/open-source-security-management/ (2026)
44. Stouffer, K., Falco, J., Scarfone, K., et al.: Guide to industrial control systems (ics) security. NIST special publication **800**(82), 16–16 (2011)
45. Tay, H.J., Zeng, K., Vadayath, J.M., Raj, A.S., Dutcher, A., Reddy, T., Gibbs, W., Basque, Z.L., Dong, F., Smith, Z., et al.: Greenhouse:{single-service} rehosting of {linux-based} firmware binaries in {user-space} emulation. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 5791–5808 (2023)
46. The MITRE Corporation: Cve program. https://cve.mitre.org/ (2026)
47. Tychalas, D., Benkraouda, H., Maniatakos, M.: {ICSFuzz}: Manipulating {I/Os} and repurposing binary code to enable instrumented fuzzing in {ICS} control applications. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2847–2862 (2021)
48. U.S. General Services Administration: Executive order on improving the nation's cybersecurity. https://www.gsa.gov/technology/it-contract-vehicles-and-purchasing-programs/technology-products-services/it-security/executive-order-14028 (October 2025)
49. Veselov, D.: Python bindings to libmagic. https://github.com/dveselov/python-libmagic (2021)
50. Villa, C., Doumanidis, C., Lamri, H., Rajput, P.H.N., Maniatakos, M.: Icsquartz: Scan cycle-aware and vendor-agnostic fuzzing for industrial control systems. In: NDSS (2025)
51. Vinet, J., Griffin, A., Polyák, L.: Arch linux packages. https://archlinux.org/packages/ (2026)
52. WAGO: Firmware sdk for pfc200-g1 and pfc100 family. https://github.com/WAGO/pfc-firmware-sdk (2022)
53. WAGO: Wago pfc-g2 firmware sdk patches for libmodbus. https://github.com/WAGO/pfc-firmware-sdk-G2/tree/5760a7f1add2e7c69bca6ae38373b1be8754f941/ptxproj/patches/libmodbus-3.0.5 (October 2023)
54. WAGO: Wago pfc200-g2 release fw28-v04.06.01. https://github.com/WAGO/pfc-firmware-sdk-G2/releases/tag/FW28-V04.06.01 (November 2024)
55. WAGO: Firmware sdk for pfc200-g2 family. https://github.com/WAGO/pfc-firmware-sdk-G2 (2025)
56. WAGO: Open source projects of wago kontakttechnik. https://github.com/WAGO (2026)
57. Wang, H., Gao, Z., Zhang, C., Sha, Z., Sun, M., Zhou, Y., Zhu, W., Sun, W., Qiu, H., Xiao, X.: Clap: Learning transferable binary code representations with natural language supervision. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 503–515 (2024)
58. Wang, H., Qu, W., Katz, G., Zhu, W., Gao, Z., Qiu, H., Zhuge, J., Zhang, C.: Jtrans: Jump-aware transformer for binary code similarity detection. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis. pp. 1–13 (2022)
59. WIKIPEDIA: Jaro–winkler distance. https://en.wikipedia.org/wiki/Jaro-Winkler_distance (2025)
60. WIKIPEDIA: Md5. https://en.wikipedia.org/wiki/MD5 (2025)
61. WIKIPEDIA: Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance (2026)

62. WIKIPEDIA: Media type. https://en.wikipedia.org/wiki/Media_type (2026)
63. Wu, Y., Sun, C., Zeng, D., Tan, G., Ma, S., Wang, P.: {LibScan}: Towards more precise {Third-Party} library identification for android applications. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 3385–3402 (2023)
64. Xie, Z., Wen, M., Li, T., Zhu, Y., Hou, Q., Jin, H.: How does code optimization impact third-party library detection for android applications? In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. pp. 1919–1931 (2024)
65. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. pp. 363–376 (2017)
66. Xu, Y., Xu, Z., Chen, B., Song, F., Liu, Y., Liu, T.: Patch based vulnerability matching for binary programs. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 376–387 (2020)
67. Yocto Project: Poky build tool patches for curl. https://github.com/yoctoproject/poky/tree/yocto-3.1.17/meta/recipes-support/curl/curl (May 2022)
68. Yocto Project: Poky build tool and metadata. https://git.yoctoproject.org/poky/ (2026)
69. Yocto Project: Yocto project application development and the extensible software development kit (esdk). https://docs.yoctoproject.org/sdk-manual/working-projects.html (2026)
70. Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., Wu, S.: Order matters: Semantic-aware neural networks for binary code similarity detection. In: Proceedings of the AAAI conference on artificial intelligence. vol. 34, pp. 1145–1152 (2020)
71. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., et al.: Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In: NDSS. vol. 14, pp. 1–16 (2014)
72. Zhan, X., Fan, L., Chen, S., We, F., Liu, T., Luo, X., Liu, Y.: Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1695–1707. IEEE (2021)
73. Zhang, J., Beresford, A.R., Kollmann, S.A.: Libid: reliable identification of obfuscated third-party android libraries. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 55–65 (2019)
74. Zhang, J., Wu, J., Ling, X., Luo, T., Zhou, B., Yang, M.: Shrunk, yet complete: Code shrinking-resilient android third-party library detection. In: 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 3557–3568. IEEE (2025)
75. Zhao, B., Ji, S., Xu, J., Tian, Y., Wei, Q., Wang, Q., Lyu, C., Zhang, X., Lin, C., Wu, J., et al.: A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 442–454 (2022)
76. Zhao, B., Ji, S., Xu, J., Tian, Y., Wei, Q., Wang, Q., Lyu, C., Zhang, X., Lin, C., Wu, J., et al.: One bad apple spoils the barrel: Understanding the security risks introduced by third-party components in iot firmware. IEEE Transactions on Dependable and Secure Computing **21**(3), 1372–1389 (2023)
77. Zhao, B., Ji, S., Zhang, X., Tian, Y., Wang, Q., Pu, Y., Lyu, C., Beyah, R.: {UVSCAN}: Detecting {Third-Party} component usage violations in {IoT} firmware. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 3421–3438 (2023)

78. Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., Sun, L.: {FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1099–1114 (2019)
79. Zhou, W., Guan, L., Liu, P., Zhang, Y.: Automatic firmware emulation through invalidity-guided knowledge inference. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2007–2024 (2021)
80. Zhou, W., Zhang, L., Guan, L., Liu, P., Zhang, Y.: What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 3269–3283 (2022)