WIP: Runtime Consistency Enforcement Between SBOM and Software Execution

Yuta Shimamoto*, Hiroyuki Uekawa†, Mitsuaki Akiyama† and Toshihiro Yamauchi*
* Okayama University, Okayama, Japan
Email: shimamoto@s.okayama-u.ac.jp yamauchi@okayama-u.ac.jp
†NTT Social Informatics Laboratories, Tokyo, Japan
Email: h.uekawa@ntt.com akiyama@ieee.org

***Abstract*—A Software Bill of Materials (SBOM) enables rapid understanding of software composition and improves the efficiency of vulnerability management. However, inconsistencies between the components described in the SBOM and those that actually exist on a device can result in missed detections or false positives during SBOM-based vulnerability analysis, thereby increasing the risk of executing unknown threats. This study proposes SBOM-based Access Control (SBOM-AC), a mechanism that determines whether a program may be executed by enforcing access control policies derived from the SBOM. By denying the execution of programs that do not match the SBOM, SBOM-AC reduces security risks arising from the runtime execution of unmanaged programs. Denial logs can also be used to improve the completeness and accuracy of the SBOM, thereby reducing missed detections and false positives in SBOM-based vulnerability management and enabling the identification of unexpected execution attempts. SBOM-AC can be implemented as a Linux Security Module (LSM), making it suitable for deployment on Linux-based IoT devices and compatible with existing Mandatory Access Control systems. Experimental results show that SBOM-AC introduces a maximum latency of only 0.14 ms. Based on this measurement, the estimated performance impact of SBOM-AC on device services is negligible.**

I. Introduction

The increasing complexity of the software supply chain has exposed significant security risks, as demonstrated by attacks on software vendors such as the SolarWinds supply chain attack [1] and the widespread vulnerabilities in common dependencies such as Log4j [2]. One reason for the delayed identification and response in such incidents is the insufficient visibility into the internal components of software systems. For example, the firmware of a typical home router—a representative IoT device—contains an average of 662 components [3], making it difficult to rapidly and accurately understand its full software composition. A Software Bill of Materials (SBOM) addresses this visibility issue and contributes to faster vulnerability management and incident response.

An SBOM is formal, machine-readable metadata that uniquely identifies software packages and their components, enabling consistent sharing of component information among stakeholders in the software supply chain. SBOM adoption is advancing globally and is expected to become a standard exchange format for software composition data. For example, in the European Union, the Cyber Resilience Act requires the creation of an SBOM for all products with digital elements, including IoT devices [4]. In the United States, Executive Order 14028 mandates the creation of SBOMs for software delivered to the federal government as a part of the broader efforts to enhance the software supply chain [5].

However, SBOMs do not always comprehensively or accurately reflect the actual composition of a device. This is because SBOMs may not contain complete or accurate information at the time of creation, and they can become outdated because they provide only information about software components at a specific point in time.

When an SBOM does not match the real system state, vulnerability detection may produce false positives or miss actual vulnerabilities, potentially leading to inadequate impact assessment and delaying incident response. In particular, the execution of programs not listed in the SBOM indicates that unmanaged programs are running, which exposes attack surfaces that have not been identified. Therefore, the programs that are executed on the device must remain consistent with those listed in the SBOM. Ensuring such consistency requires a mechanism that guarantees that only SBOM-listed programs may be executed.

Mandatory Access Control (MAC) provides a promising foundation for enforcing this requirement.However, existing MAC systems are not designed to compare program information during runtime, i.e., during device use, with SBOM entries, making them unsuitable for direct enforcement based on SBOM-based execution policies. Specialized policies and mechanisms capable of verifying SBOM consistency at runtime are needed, and such controls must coexist with existing security policies without conflicts.

This study proposes SBOM-based Access Control (SBOM-AC), which allows or denies program execution based on consistency with the SBOM. SBOM-AC uses SBOM-derived policies to prevent or merely log the execution of programs not listed in the SBOM, and denial logs help improve SBOM accuracy and completeness. This approach enhances the reliability of SBOM-based vulnerability management. Additionally,

Workshop on Security and Privacy in Standardized IoT (SDIoTSec) 2026
23 February 2026, San Diego, CA, USA
ISBN 978-1-970672-01-5
https://dx.doi.org/10.14722/sdiotsec.2026.23019
www.ndss-symposium.org

SBOM-AC helps prevent the execution of malicious programs introduced after product release. Evaluation results show that SBOM-AC introduces a maximum latency of only 0.14 ms, estimating a negligible performance impact on device services.

The main contributions of this paper are as follows:

1) We propose SBOM-AC, which derives access control policies from the SBOM and uses them to control program execution.
2) We implement an SBOM-AC prototype as a Linux Security Module (LSM), enabling integration into common Linux-based IoT platforms.
3) We evaluate the latency introduced by SBOM-AC and demonstrate that its overhead is estimated to have a negligible impact on device services.

II. Background

A. Threats in Vulnerability Management Using SBOM

This section identifies potential causes of inconsistencies in SBOMs during both pre-release and post-release phases, along with the resulting threats. These causes are summarized in Figure 1.

CISA defines the SBOM creation process as follows [6]: First, as many SBOMs as possible are collected for the components on which the target component depends. New SBOMs are generated for components without available SBOMs. Next, the collected and newly generated SBOMs are integrated to construct the SBOM of the target component. At this stage, the SBOM creation is performed on a best-effort basis and does not require completeness.

However, this process makes it difficult to comprehensively and accurately represent a device's composition. For example, collected SBOMs may contain errors, or the retrieved SBOM may correspond to a different version than the one deployed on the device. Such cases lead to inconsistencies between the SBOM and the actual system composition. Additional challenges arise during SBOM generation: even with automated tools, accuracy and completeness remain insufficient in many cases [7], [8].

Furthermore, inherent limitations of SBOMs must be considered. SBOMs capture static component information at a specific point in time. Therefore, composition changes or program updates after SBOM creation are not automatically reflected. For example, if a vendor using third-party equipment updates its firmware but does not or cannot reacquire the corresponding SBOM, continued use of the outdated SBOM results in referencing information that no longer matches the actual composition.

Vulnerability management based on outdated, inaccurate or incomplete SBOMs risks missing vulnerabilities or threats and generating false positives. Executing programs excluded from vulnerability management introduces significant risk. Therefore, a mechanism is required to ensure that the SBOM covers all components necessary for vulnerability management before product release and to prevent unlisted programs from affecting the device after release.

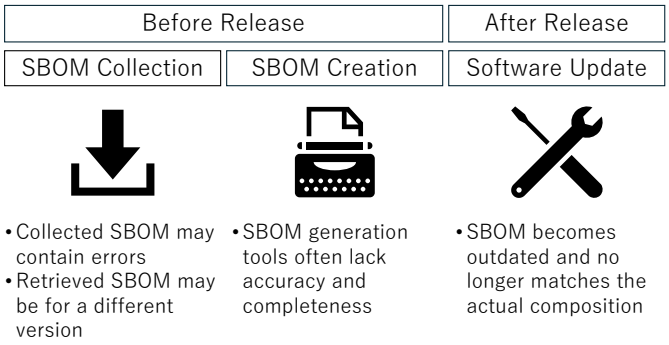

Fig. 1. Causes of Inconsistencies in SBOMs

B. Limitations of Existing MAC for SBOM-AC

MAC is an access control mechanism in which system administrators enforce access permissions. Unlike Discretionary Access Control, MAC systems enforce access based on predefined security policies, meaning that users and file owners cannot arbitrarily modify permissions.

Representative MAC systems includes SELinux and AppArmor. SELinux is a label-based MAC system in which files are associated with security labels, and access control is enforced based on label relationships. AppArmor is path-based and defines access permissions for each program using file paths.

However, SBOM-AC relies on verifying that runtime program information matches the SBOM. SELinux and AppArmor cannot enforce policies based on runtime program metadata. Since they rely solely on associating labels with files or managing access by file paths, they cannot incorporate program information during runtime into control decisions. Consequently, they cannot correctly enforce program execution constraints in cases where the program content changes after policy creation.

Furthermore, SELinux and AppArmor already provide well-established operational policies. SBOM-AC policies would introduce numerous new rules for many programs. Attempting to merge SBOM-based policies with existing MAC rules is likely to cause numerous conflicts, and multiple policies cannot coexist for the same program. In practice, enforcing SBOM-based execution control using these systems would require disabling existing policies, thereby weakening overall system security.

Therefore, neither SELinux nor AppArmor is suitable for SBOM-driven access control. A new method is required that can verify consistency by comparing the composition information recorded in the SBOM with the actual system state.

III. Threat Model

This study aims to prevent threats that cannot be identified due to inconsistencies between the SBOM and the actual composition in SBOM-based vulnerability management. Therefore, attacks that exploit known vulnerabilities in programs already listed in the SBOM are outside the scope of this study. Furthermore, this study is premised on the assumption that

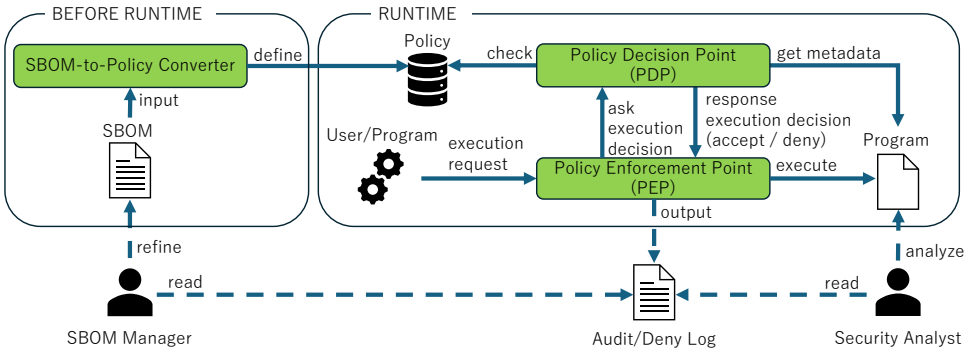

Fig. 2. Overview of SBOM-AC

vendors make their best effort to create SBOMs that comprehensively include all necessary components by correcting identified inconsistencies.

IV. Proposed Method

A. Overview

If all programs that affect device operations are comprehensively listed in the SBOM, threats arising from inconsistencies between the SBOM and the actual system composition can be mitigated. SBOM-AC therefore focuses on identifying and controlling the program execution, specifically the execution of program files, based on whether they belong to components accurately represented in the SBOM. By allowing such programs to run while appropriately handling the execution of programs not listed in the SBOM, SBOM-AC prevents security risks caused by mismatches between the SBOM and the real system state.

The design principles of SBOM-AC are as follows:

Reference program information at execution time: To accommodate program modifications or replacements after policy definition, SBOM-AC must reference the latest program metadata at execution time.

Avoiding disruption of essential device services: Essential programs must not be blocked; therefore, the design must guarantee reliable execution for required components.

Coexistence with existing MAC systems: SBOM-AC must operate without interfering with existing MAC systems or their established policies, ensuring that current security measures remain intact.

The overall workflow is shown in Figure 2. Before runtime, the SBOM-to-Policy Converter defines policies using metadata extracted from the SBOM. At runtime, when a program execution request occurs, the Policy Enforcement Point (PEP) permits or blocks execution and logs all SBOM-inconsistent execution attempts based on decisions made by the Policy Decision Point (PDP). Runtime operations are designed to remain compatible with existing MAC systems. The following sections describe each component in detail.

B. SBOM-to-Policy Converter

This component takes an SBOM as input, extracts the metadata of the programs it contains, and defines a corresponding policy. To ensure that files listed in the SBOM exist and have not been modified, the policy must be expressed in a verifiable format. Therefore, in addition to each file's path, the policy includes the file's hash value and its hash algorithm.

C. Policy Enforcement Point (PEP)

When a program execution is requested by a user or another program, the PEP consults the PDP. If the PDP determines that execution should be allowed, the PEP permits the program to run. If the PDP determines that execution should be denied, the PEP performs one of the following actions:

Audit mode: The program is executed, but a log entry is generated indicating that a program not listed in the SBOM was executed.

Deny mode: The program is not executed, and a log entry is generated indicating an attempt to run a non-listed program.

During development, audit mode allows SBOM managers to iteratively refine SBOM entries, thereby strengthening the effectiveness of vulnerability management. Policies generated from the corrected SBOMs can then be safely used in deny mode without interrupting essential device operations. After release, the deny mode reliably blocks potentially harmful program executions. An additional benefit is that deny mode also prevents the execution of malicious programs introduced after deployment such as IoT malware (for example, Mirai [9]).

The generated logs are also valuable for security analysis. By examining these logs, security analysts can determine whether an event resulted from insufficient SBOM maintenance, intentionally omitted programs, or malicious programs introduced externally, enabling appropriate countermeasures.

D. Policy Decision Point (PDP)

For each program execution request, the PDP determines whether execution should be allowed or denied. Specifically, it compares the program information allowed by the policy with the latest metadata of the program requesting execution. If they match, the PDP instructs the PEP to allow execution. If any mismatch is detected, it instructs the PEP to deny it.

The PDP retrieves the program's current path and hash value for each execution request. The hash value is computed for the target program using the policy-specified algorithm.

E. Limitations

SBOM-AC has the following limitations:

1) The file path of the program to be executed and its hash value must be included in the SBOM.
2) SBOM-AC cannot detect or prevent attempts to bypass its enforcement mechanism; therefore, it should be used together with other security mechanisms.
3) SBOM-AC logs cannot be used to identify programs that have been uninstalled from the device.

V. Evaluation and results

A. Prototype Implementation

A prototype of SBOM-AC was implemented to verify its core functions: permitting the execution of programs listed in the SBOM, denying non-compliant programs, and recording audit or deny logs. All functions were successfully confirmed. The prototype consists of (i) a Policy Loader that defines and loads policies from the SBOM, and (ii) an Execution Enforcer that verifies program information and enforces execution decisions. Although the prototype does not yet support the per-program hashing algorithm selection described in Section IV, and currently limits execution control to programs within a specific directory, it is sufficient to confirm that SBOM-AC operates on Linux-based distributions. The design principles of each prototype component and the method used to create the SBOM are described below.

Policy Loader: A simplified prototype of the SBOM-to-Policy Converter was implemented. It extracts file paths and file hash values from the SBOM and converts each pair into an internal policy format. In this prototype, all hash algorithms were fixed to SHA-256. The resulting policy is loaded into kernel space. File paths are stored in a hash table within the Execution Enforcer to enable fast lookup.

Execution Enforcer: This component implements both the PEP and PDP as an LSM and integrates them into Ubuntu 24.04.3. It supports both the audit and deny modes. Whenever there is an attempt to execute a program in the `/home/user/sbomac_test/` directory, its path and hash value are retrieved and compared with the preloaded policy to determine whether execution should be allowed or denied. Denials are logged to dmesg. Experiments were conducted using VMware (25H2) running on a Windows 11 host. The virtual machine was configured with four logical cores of an Intel® Core™ i7-8700T CPU @ 2.40 GHz and 8 GB of RAM.

SBOM Generation: The SBOM generation tool Syft [10] v1.38.0 was used to generate SBOMs for use by the Policy Loader. When configured to collect as many files as possible, Syft comprehensively scans all files within a specified directory and its subdirectories, enabling the extraction of file path and hash-value pairs. The resulting SBOMs were formatted in SPDX 2.3, which supports representing both file paths and hash values, making it suitable for use in this prototype.

B. Performance Evaluation

To estimate the runtime impact of SBOM-AC on IoT device services, we measured the execution time of a lightweight program that outputs a simple string, under conditions where SBOM-AC was enabled or disabled. One program was included in the SBOM, representing allowed execution, and the other was excluded, representing denied execution.

TABLE I
Execution Time Under Each Condition

Condition	Entries	Minimum	Average	Maximum
Disabled	-	2.39 ms	2.42 ms	2.50 ms
Allowed execution	1	2.46 ms	2.49 ms	2.53 ms
	475,953	2.46 ms	2.49 ms	2.51 ms
Denied execution	1	2.44 ms	2.46 ms	2.51 ms
	475,953	2.46 ms	2.48 ms	2.52 ms

Measurements were taken under the following three conditions:

1) Program executed with SBOM-AC disabled
2) Allowed program executed with SBOM-AC enabled
3) Denied program executed with SBOM-AC enabled

SBOM-AC was fixed to audit mode for all measurements, as deny mode blocks program execution and therefore cannot be compared with the SBOM-AC-disabled baseline.

Furthermore, for each condition, comparisons were made using two different numbers of SBOM entries:

1) Evaluation-focused SBOM containing only one entry
2) Broader SBOM with 475,953 entries, including non-evaluated files

For each condition and entry count, the average execution time was computed from 100,000 executions. This procedure was repeated 100 times to obtain the maximum, average, and minimum of the resulting averages. The results are summarized in Table I. Across all entry sizes, only denied executions exhibited a slight increase in execution time which remained smaller than the inherent execution-time variability. Allowed executions showed no clear difference, likely because the allowed program lookup position increased by only about 200 entries. Moreover, the enabled state incurred an additional latency of at most 0.14 ms compared with the disabled state.

These measurements include the additional processing overhead introduced by SBOM-AC. Comparing the maximum difference with the baseline minimum execution time (2.39 ms) shows that SBOM-AC adds approximately 6% to the execution time. Given that IoT device services typically involve more complex operations such as network communication [11], the resulting runtime overhead, and thus the performance impact, are expected to be negligible.

VI. Conclusion

This study proposes SBOM-AC to mitigate risks caused by inconsistencies between SBOMs and actual device compositions, where unknown programs may be executed. By identifying and recording execution requests for programs missing from or inaccurately described in the SBOM, SBOM-AC supports SBOM correction and detection of programs outside SBOM-based vulnerability management. Blocking such executions also prevents overlooked threats. Experimental results

demonstrate that SBOM-AC introduces a maximum latency of only 0.14 ms, expecting a negligible impact on device services.

However, several challenges remain for future work. First, we will identify additional elements that influence device operation beyond executed programs and expand the control scope of SBOM-AC. We will also refine SBOM-to-policy conversion by identifying which elements of widely used SBOM formats can reliably provide the policy information required by SBOM-AC. To evaluate overhead more rigorously, we will measure standalone latency, memory usage, and CPU utilization. We further plan to assess SBOM-AC's effectiveness in achieving its intended security objectives. Finally, we will investigate the feasibility and expected benefits of broader adoption of an SBOM-AC through vendor surveys.

ACKNOWLEDGMENT

This work was partially supported by JST BOOST (Japan Grant Number JPMJBS2403) and by Okayama Foundation for Science and Technology.

REFERENCES

[1] S. Peisert, B. Schneier, H. Okhravi, F. Massacci, T. Benzel, C. Landwehr, M. Mannan, J. Mirkovic, A. Prakash, and J. B. Michael, "Perspectives on the solarwinds incident," *IEEE Security & Privacy*, vol. 19, no. 2, pp. 7–13, 2021.

[2] CISA, "Apache Log4j Vulnerability Guidance." [Online]. Available: https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance

[3] Forescout, "Popular OT/IoT Router Firmware Images Contain Outdated Software and Exploitable N-Day Vulnerabilities Affecting the Kernel." [Online]. Available: https://www.forescout.com/press-releases/ot-iot-router-firmware-outdated-software-vulnerabilities/

[4] EU, "The Cyber Resilience Act." [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32024R2847

[5] White House, "Improving the Nation's Cybersecurity." [Online]. Available: https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity

[6] CISA, "Framing Software Component Transparency: Establishing a Common Software Bill of Materials (SBOM) Third Edition." [Online]. Available: https://www.cisa.gov/sites/default/files/2024-10/SBOM Framing Software Component Transparency 2024.pdf

[7] N. Kawaguchi, C. Hart, and H. Uchimura, "Understanding the effectiveness of sbom generation tools for manually installed packages in docker containers," *Journal of Internet Services and Information Security*, vol. 14, no. 3, pp. 191–212, 2024.

[8] S. Yu, W. Song, X. Hu, and H. Yin, "On the correctness of metadata-based sbom generation: A differential analysis approach," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024, pp. 29–36.

[9] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1093–1110.

[10] Anchor, "Syft." [Online]. Available: https://github.com/anchore/syft

[11] O. Ali, M. K. Ishak, M. K. L. Bhatti, I. Khan, and K.-I. Kim, "A comprehensive review of internet of things: Technology stack, middlewares, and fog/edge computing interface," *Sensors*, vol. 22, no. 3, p. 995, 2022.