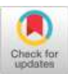

PDF Download
3754455.pdf
25 March 2026
Total Citations: 0
Total Downloads: 573

Latest updates: https://dl.acm.org/doi/10.1145/3754455

Published: 21 January 2026
Online AM: 28 July 2025
Accepted: 17 July 2025
Revised: 18 May 2025
Received: 16 November 2024

Citation in BibTeX format

RESEARCH-ARTICLE

Firmware Secure Updates Meet Formal Verification

ALBERTO TACCHELLA, University of Trento, Trento, TN, Italy

EMANUELE BEOZZO, University of Trento, Trento, TN, Italy

BRUNO CRISPO, University of Trento, Trento, TN, Italy

MARCO ROVERI, University of Trento, Trento, TN, Italy

Open Access Support provided by:

University of Trento

Firmware Secure Updates Meet Formal Verification

ALBERTO TACCHELLA, EMANUELE BEOZZO, BRUNO CRISPO, and MARCO ROVERI,
Università degli Studi di Trento, Trento, Italy

Industrial Internet of Things (IIoT) systems require robust mechanisms for secure firmware updates. Existing approaches are often inadequate due to vendor fragmentation, network limitations, and the safety-critical nature of many IIoT applications. In this article, we address these challenges by extending the IETF SUIT (Software Updates for Internet of Things) framework to enhance the security and assurance of firmware updates. Our contributions include the integration of Software Bill of Materials (SBOM) mechanisms and a Behavioral Certification Manifest into the SUIT architecture to increase transparency and provide formal guarantees about the content of the update. The approach is validated by a prototype implementation that demonstrates its feasibility and scalability using real-world benchmarks.

CCS Concepts: • **Security and privacy → Logic and verification**; **Systems security**; *Software security engineering*; *Network security*;

Additional Key Words and Phrases: Secure Update, IIoT, Systems Security, Firmware Distribution, Formal Methods, SBOM

ACM Reference format:
Alberto Tacchella, Emanuele Beozzo, Bruno Crispo, and Marco Roveri. 2026. Firmware Secure Updates Meet Formal Verification. *ACM Trans. Cyber-Phys. Syst.* 10, 1, Article 6 (January 2026), 26 pages.
https://doi.org/10.1145/3754455

1 Introduction

Industrial Internet-of-Things (IIoT) devices are becoming more prevalent in modern infrastructure, with applications spanning from industrial automation to safety-critical systems, such as autonomous transportation and smart grids. The native software stack running on these devices, usually called the *firmware*, must be updated regularly to introduce new functionalities or to address existing bugs and vulnerabilities. The security of the firmware update process is thus critical, as it can become a potential attack vector, especially if weak verification mechanisms are employed.

Since typical IIoT systems include devices from different vendors and each firmware can be composed of software modules provided by various developers, the approach in which each party provides its update mechanisms does not work, leading to version inconsistencies and

This work is supported by the European Union through the Sponsor Horizon Europe Research and Innovation Program, project Cross-Platform Open Security Stack for Connected Devices (CROSSCON), Grant No. 101070537, and through the Sponsor Next Generation EU PRIN 2022 Prot. No. 202297YF75.
Authors' Contact Information: Alberto Tacchella (corresponding author), Università degli Studi di Trento, Trento, Italy; e-mail: alberto.tacchella@unitn.it; Emanuele Beozzo, Università degli Studi di Trento, Trento, Italy; e-mail: emanuele.beozzo@unitn.it; Bruno Crispo, Università degli Studi di Trento, Trento, Italy; e-mail: bruno.crispo@unitn.it; Marco Roveri, Università degli Studi di Trento, Trento, Italy; e-mail: marco.roveri@unitn.it.

This work is licensed under Creative Commons Attribution International 4.0.

© 2026 Copyright held by the owner/author(s).
ACM 2378-9638/2026/1-ART6
https://doi.org/10.1145/3754455

conflicts among the devices of an IIoT system. On the other hand, reusing existing update mechanisms developed for traditional computing systems also does not work because connectivity to the Internet cannot always be guaranteed, the many parties that are required to update their software cannot always run their agents on these devices, and finally, because updating firmware is an invasive process that may require, for instance, rebooting the device, which is typically unattended. Even today, many device vendors still do not support remote firmware update mechanisms. The EU Cyber Resilience Act [35] underscores the importance of this issue by requiring the inclusion of remote update capabilities for every device placed on the EU market after 2027.

In this context, the IETF **Software Updates for Internet of Things (SUIT)** working group proposed a new standard [32] to enhance the security of software updates for **Internet of Things (IoT)** devices, among which we can also consider the IIoT domain. In the SUIT framework, an **IoT Software Maintainer (ISM)** generates an update bundle containing the new version of the firmware with associated metadata and uploads the bundle to a Firmware Server, which distributes it to affected devices using **Over-the-Air (OTA)** or wired technologies. The process is carefully designed to ensure the integrity and confidentiality of the updates, guaranteeing end-to-end security from the ISM to the device, even when an untrusted server is involved. This creates a secure communication channel between the trusted ISM and the IoT devices, safeguarding the entire update process. However, even with such a mechanism in place, there is still no guarantee about the safety of the update's content. This limitation means that an ISM could, intentionally or unintentionally, introduce a vulnerable software component that jeopardizes the safety or security of the updated device. Although the protection provided by digitally signing the update is enough for some IoT systems, a higher level of assurance is necessary in IIoT systems with more stringent requirements.

In this article, we address the problem of securing updates of IIoT devices by means of the following contributions. First, we propose an extension of the SUIT standard architecture that integrates within the SUIT manifest [31] two new components, a **Software Bill of Materials (SBOM)** and a **Behavioral Certification Manifest (BCM)**.

An SBOM is a list of (i) all the components present in a codebase, (ii) all the licenses that govern those components, (iii) the versions of the components used in the codebase, and (iv) their patch status, which allows security teams to quickly identify any associated security or license risks. The SBOM offers many advantages: it increases the traceability of the various software components used by the firmware (and thus its security) and provides information for a more accurate estimation of the cyber risk. The usage of an SBOM in the context of software updates has been mandated both in the US government Executive Order 14,028 on improving cybersecurity standards [22], issued in May 2021 amid the fallout from the SolarWinds attack, and in the EU Cyber Resilience Act [35], which has been approved by the European Parliament in October 2024. The U.S. Defense Advanced Research Projects Agency has recently sponsored the E-BOSS research program[1] whose objective is to develop an Enhanced Software Bill of Material to improve SBOMs and SBOM-driven technologies by adding new types of metadata and new algorithms to determine whether flawed or sensitive code is actually reachable and triggerable.

A BCM is a structured list of formal proofs, each attesting that the update does not violate a corresponding (security or behavioral) property. Relevant properties include the correctness of configuration parameters or conditions on the control flow of the updated binaries to ensure that an attacker has not injected calls to malicious functions.

Second, we extend the SUIT update workflow to take into account the new additions to the update manifest. In particular, the new flow requires first extracting from the SUIT manifest the

[1] https://www.darpa.mil/program/enhanced-sbom-for-optimized-software-sustainment.

SBOM to verify the presence of vulnerable (or otherwise undesirable) components; if one is found, the update is rejected immediately. Otherwise, the process continues by extracting the BCM and verifying each enclosed proof for validity. If the verification of any proof fails, the update is again rejected; otherwise, the update is installed. To take into account the possibly limited computational resources of the IIoT device, two different approaches are considered to verify the proofs: directly on the device or delegated to a trusted verification server.

Third, we consider in detail three broad classes of behavioral properties: the correctness of the configuration parameters, the control flow invariance, and the control flow preservation. The first property aims to ensure that the update preserves security configuration constraints; the other two are concerned with the control flow of the updated component. Control flow invariance checks if there is a graph isomorphism between a "golden" **Control Flow Graph (CFG)** and the CFG of the received update. Control flow preservation relaxes this constraint by allowing the update to add new execution paths (without altering the old ones). For each of the behavioral properties considered, we provide a formal characterization and an encoding into a **Satisfiability Modulo Theory (SMT)** problem [7], to enable the verification and generation of a proof with any state-of-the-art SMT solver, and check of the generated proof with existing third party proof checkers. A few remarks are in order: (a) The proposed approach does not necessarily require the source code and, therefore, can also be applied to third-party applications available only in binary form. (b) The proposed approach does not depend on the hardware that will host and run the application: resource capabilities can be used to decide whether to check the proofs locally on the device or remotely on a trusted server.

Fourth, to demonstrate the applicability of our proposal, we developed a prototype implementation by extending an existing reference implementation of the SUIT framework. To this end, we developed a toolchain to generate proof certificates for the three classes of behavioral properties mentioned above and independently verify them, using the proof extraction capability of the cvc5 SMT solver [5].

Finally, we validated our prototype on several real-world benchmark applications of different sizes and kinds taken from the OpTEE Operating System distribution to demonstrate the approach's feasibility and assess its scalability. The results show that our framework can be applied to real use cases of different sizes.

The article is organized as follows: In Section 2, we briefly summarize the existing research literature on the topic and the leading industry solutions, highlighting the differences with our approach. In Section 3, we describe the main points of the SUIT standard, on which our proposal is based, discuss its problems and limitations, and show how our proposal improves on the current situation. In Section 4, we explain in detail the architecture of our proposal, including a detailed workflow for the update generation and installation phases. In Section 5, we focus on the generation and verification of Behavioral Certification artifacts, explaining in detail how our proposal works for two exemplary properties: *static partitioning of resources* and *control flow preservation*. In Section 6, we describe our implementation and report on the evaluation of its performance. Finally, Section 7 provides some concluding remarks.

2 Related Work

Firmware update is a critical process for the security of IoT/IIoT devices. Not being able to update a firmware is one of the most common sources of vulnerability during the lifecycle of a device. As many recent surveys have revealed (e.g., [42]), it is very common to find IoT devices in the wild that lack a secure firmware update system. In [18], the authors present an analysis performed on a total of 1.06 million devices accessible through the public Internet and show that the average age of the installed firmware is 19.2 months. Thus, the time window to exploit any vulnerability is very large.

Firmware update mechanisms fall into two main categories: *wired* updates (e.g., through some interface such as JTAG, USB) or *OTA* updates. Moreover, the update can be either *full* or *partial* (e.g., the update is a replacement of the entire firmware or it will replace or add single parts of the installed firmware), it may include signature and integrity verification, and it may require an attestation of delivery and installation. Hereafter, we will focus on the **Firmware Over-the-Air (FOTA)** update process, which is most relevant for IoT devices.

Seshadri et al. [40] propose a **Secure Code Update By Attestation (SCUBA)** framework for sensor networks to detect and repair compromised sensors through firmware updates. However, the (software-based) attestation technique heavily relies on the timing characteristics of the measurement process and the use of an optimal checksum function; due to these assumptions, SCUBA has limited applicability in generic IIoT settings.

The Update Framework (TUF) [39] provides an open source specification for FOTA updates that focuses in particular on key security issues. However, this approach does not consider supply chain attacks in which no signing key is ever compromised, but the content of an update package is altered by tampering with its dependencies.

In [25], the authors propose Securing Software Updates for Automobiles (Uptane). It is based on TUF for automotive systems by adding a director repository, which allows an Original Equipment Manufacturer to have more control of software images deployed in individual **Engine Control Units (ECUs)**. In [29], the authors propose a secure FOTA update scheme for automotive ECUs based on the verification of the integrity and authenticity of update packages, together with versioning and entitlement schemes for each vehicle. The main limitation of these two approaches is that they are designed specifically for the automotive scenario and cannot be easily generalized to other domains.

In [10], the authors present an FOTA update system that provides a low-power mesh protocol, route discovery, and establishment. The solution is based on the Lightweight Mesh network protocol (LWMesh) over a peer-to-peer mesh architecture (P2PMesh). However, in addition to the proprietary aspect, the proposal misses the security aspects.

Doddapaneni et al. [16] propose an FOTA update scheme for IoT devices by defining a new secure object called Firmware Object Signing and Encryption. The object is encoded using a secure format such as JSON Object Signing and Encryption. The main goal is to solve the problems of integrity (in case of network loss or break) and security (by encrypting the payload). The paper also proposes a procedure for OTA updates. However, it addresses only the case where there is an intermediary application between the end-device and the device manager.

In [4], the authors design an architecture (ASSURED) for a secure update framework of realistic embedded devices. To demonstrate its feasibility, ASSURED is instantiated and evaluated on two commodity hardware platforms, HYDRA and Arm TrustZone-M. The proposal is an enhancement of the TUF architecture and adopts the same terminology of the SUIT Working Group. However, this proposal lacks formal guarantees on the content of the update package.

The UpKit proposal [27] is a lightweight and portable software update framework for restricted IoT devices that aims to cover all phases of the update process, from generation to installation. This scheme guarantees the freshness and integrity of the firmware image, but does not cover the functional properties of the deployed update.

Giaretta et al. [20] propose a *security by contract* framework (S×C4IoT) which allows to formally bind a behavioral description of a device to its firmware. This proposal is conceptually very similar to ours; however, in their scheme, the proof verification step must be performed by the network each time a new device joins it, whereas in our scheme, this step is performed either by the affected device only or by a specific trusted server specified by the firmware author.

Dejon et al. [15] propose an automated security analysis framework for (binary) software updates centered on the use of the ANGR binary analysis tool (to extract the CFG from a binary image) and the Prism probabilistic model checker (to analyze whether the generated CFG is compliant with some specified security policy). Our approach is more general than that of IoTAV, in that it takes into account also the source code of the software update (which is assumed not to be available in IoTAV) and at the same time keeps the possibility of proving properties by direct binary analysis, which is useful in case the compiler is not fully trusted (e.g., when aggressive optimizations are enabled) or when the update embeds components whose source is not available. Furthermore, IoTAV always relies on an external server to carry out the verification, while we also allow the possibility of executing the checks locally.

Among the most significant solutions adopted by industry, we can mention the following: *Cloud IoT Core*[2] is a cloud-based product provided by Google that allows for the connection and management of secure devices from a few to millions of IoT devices. *AWS IoT device management*[3] is a solution provided by Amazon to register, organize, monitor, and remotely manage IoT devices. Among its functionalities, it provides means to query the states of managed IoT devices and to send FOTA update, mainly for FreeRTOS devices. *OTA Download*[4] is an ecosystem provided by Texas Instruments that allows the update of the firmware image that runs on BLE devices wirelessly. *Pelion Device Management*[5] is a component of the Mbed OS operating systems for ARM platforms, which enables the provisioning of IoT end nodes and provides secure and reliable OTA software updates. However, all these industry solutions are proprietary and therefore lack the transparency required for interoperability and for an auditable secure solution.

3 Background

Currently, there is no consensus on a standard for firmware updates in the IoT/IIoT world [9]. Many solutions have been proposed, with various use cases and scenarios in mind. In industry practice, different manufacturers follow different approaches depending on their tools, infrastructures, and strategies [19].

3.1 The SUIT Standard

The IETF SUIT working group has developed an Internet standard, described in RFC 9019 [32], for a new firmware update architecture that uses state-of-the-art security and can be deployed even on very constrained devices (Class 1, as defined in RFC 7228). The architecture is based on a metadata structure, known as the *manifest*, which is sent to the devices affected by the update. The information contained in the manifest is used for authentication and authorization purposes, to retrieve and validate the new firmware image, and to fulfill any other requirement that the process might have. The architecture is designed to be flexible enough to be applied in many situations, but at the same time can be made robust with regard to functionality and security.

An information model for the SUIT manifest has been described in RFC 9124 [30], and a proposal for a concrete encoding scheme is currently in an advanced draft stage [31]; it is based on the CBOR binary serialization format (described in RFC 8949) and the associated COSE packaging mechanism with cryptographic support (described in RFC 8152). The standard allows for extensions implementing optional capabilities such as firmware encryption, trust domains, update management, inclusion of a Manufacturer Usage Description specification, and secure methods to report the update status.

[2] https://cloud.google.com/iot-core.
[3] https://aws.amazon.com/iot-device-management/.
[4] https://software-dl.ti.com/lprf/simplelink_cc2640r2_sdk/1.00.00.22/exports/docs/blestack/html/oad/oad.html.
[5] https://www.pelion.com/.

The architecture of the SUIT update process is schematically depicted in Figure 1 and involves the following four main components:

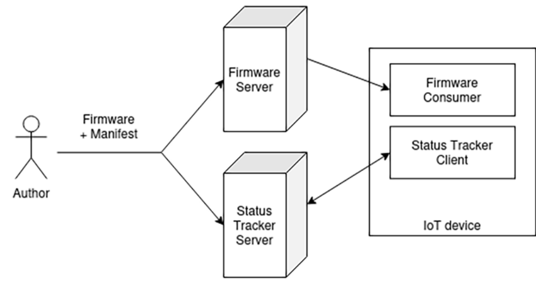

Fig. 1. The SUIT secure update architecture.

- an IoT device, or more specifically, a component inside the device that needs to receive the firmware update, called the **Firmware Consumer (FC)**;
- a *Firmware Server* that can store manifests and firmware images and make them available upon request;
- a *Status Tracker Server* that keeps track of software and hardware information about each device on the network and the availability of updates;
- a *Status Tracker Client* running on the IoT device and communicating with the Status Tracker Server.

The update process can be triggered by the Status Tracker Server (push mode) as soon as a new firmware image is available, or by the Status Tracker Client (pull mode) by periodically querying the server. Hybrid approaches are also possible, and the Status Tracker Client can implement a more complex logic to decide on a time that does not disrupt the device workflow. Once the update is initiated, the FC receives the manifest and must validate the signature to assert its authenticity. The standard does not cover how the signature is generated and checked, and assumes that a trust anchor is already present in the device. Once the signature is verified, the FC must parse the manifest to check the validity of the update, identify if it applies to the device, and perform the required integrity checks. The manifest can also specify how to perform the update, where to store the firmware, and so on. Finally, the firmware image is fetched depending on the capabilities of the device; it can be downloaded using a **Uniform Resource Identifier (URI)** contained in the manifest, it can be embedded in the manifest itself, or it can be delivered through physical means. The obtained image is then verified and installed according to the instructions contained in the manifest.

The SUIT manifest is hosted in an enclosing structure called the *envelope*, which (in addition to the manifest itself) contains an *Authentication Block*, zero or more *Severable Elements*, and zero or more *Integrated Payloads*. Severable elements are elements of the update package which are needed only in some phase of the update process and can be discarded (or "severed") after that phase. These elements are authenticated by including a cryptographic digest in the update manifest, so that they can be removed without changing the manifest's signature. This mechanism allows to save resources (storage space, bandwidth) on the device that receives the update.

3.2 Threat Model

The SUIT workflow has been designed to enforce integrity and confidentiality in the software update process even when an untrusted server is involved. This ensures end-to-end security between the Firmware Author and the device. However, it provides no guarantees on the effective content of the update. For instance, there is no mechanism to prevent the distribution of a firmware image containing security vulnerabilities or malicious behaviors unbeknownst to the Firmware Author. Similarly for any scenario in which an attack is carried out by injecting malicious software components inside the firmware directly at the source level, as in the recent xzutils backdooring attempt [1]. Furthermore, there is no protection against an attacker who manages to compromise

the signing key used by the firmware author to protect the integrity and authenticity of its updates. While in theory these attacks can and should be prevented, in practice they occur much more often than expected. The code-signing keys of Nvidia [12], Samsung [3], and MSI [21] were exposed in recent years. If such attacks are possible with big vendors, they are even more likely to happen with small developers. Attackers sometimes only need simple exploitation skills, such as keyword search in Docker Hub [14].

More in detail, we consider a threat model similar to what the state of the art considers [25], where the main objectives of the attacker are any of the following:

— *Reading the Updates.* Attackers aim to learn the contents of software updates to steal intellectual property rights. This can be done by mounting eavesdrop attacks where the attacker controlling the network is able to intercept the content of the update message on the route. We do not consider in scope the attack implemented by breaking into the server of the Firmware Author and stealing the updates. This attack should be prevented by implementing regular and well-known security practices.
— *Denying Updates.* Attackers want to prevent devices from updating and fixing known software vulnerabilities. We consider here the attack that by manipulating the content of the update (i.e., version number, class of device to which the update applies, etc.) makes the device fail to install the update. This is a denial of service attack mounted by altering the genuine content of the update.
— *Rollback Attacks.* Attackers aim at pushing old but valid versions of the update containing a vulnerability that has been fixed in newer versions. The attacker can play the role of a device that needs to be updated to receive all genuine copies of the update, then try to mount the attack by sending old updates to the victim device.
— *Modify Updates.* Attackers manipulate the content of the update by deleting, adding, and/or altering functionalities of the updates, thus causing the device to stop functioning or preventing it from functioning correctly. The attacker can mount this attack by intercepting update messages in clear and altering their content.

The SUIT standard, on which our proposal is based, includes a threat model that already considers the above attacks [30] and implements the security requirements to address them. However, we extend the SUIT threat model by assuming a more powerful attacker which is also capable of corrupting the image of the update at her will and generating a valid signature of such altered update (e.g., because she managed to compromise the update signing key or because she managed to inject somehow malicious content in the process of the update generation). Thus, our scheme considers a more powerful attacker model comprising the following additional threat:

— *Fraudulent Updates.* Attackers replace authentic updates with those generated by them. Attackers manipulate the content of the update by deleting, adding, and/or altering functionalities of the updates and if required signing them with the compromised key.

We consider denial of service attacks against the various servers for preventing the update to ever reach a device to be out of scope. These network level attacks are indeed important, but they are more related to connectivity aspects, so they can be addressed using existing techniques that aim at building redundancy in the servers and in the network [41]. We also consider out of scope the attacks that target single devices. By compromising and taking full control of a device, those attacks can prevent any update to be installed. We consider here only the security goals specific to the update protocol and mechanism, and we demand to other existing mechanisms [2, 23] the local protection of the devices.

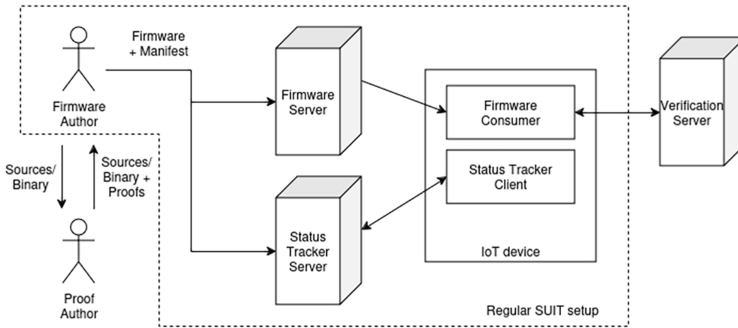

Fig. 2. Extended SUIT secure update architecture.

This article does not consider advanced adversarial scenarios, such as adaptive attackers that, for example, may use a game-theoretic approach to try to manipulate the SBOM or the proof verification. We leave the exploration of whether such attacks are feasible, considering the constraints dictated by the context (i.e., the attack must produce still exploitable and running firmware) as future work.

4 Proposed Architecture

To overcome the limitations of the SUIT standard explained above, we extend it along two different directions: (1) the integration of an *SBOM*, in order to improve the transparency of the update process and ease the tracking of dependencies and vulnerabilities, and (2) the addition of a *BCM*, whose purpose is to host formal proofs of some selected behavioral properties of the firmware update. Moreover, we supplement the high-level architecture of the SUIT mechanism described in Figure 1 with: (1) a new agent, the *Proof Author*, whose purpose is to generate and cryptographically sign the proof certificates related to the update, and (2) an additional (trusted) engine to perform the verification of computationally intensive proof steps when needed. The engine can be hosted by third-party services among which the FC can freely select one, if she does not have her own engine. The resulting overall scheme is depicted in Figure 2.

The rest of this section is structured as follows: In Section 4.1, we define the extensions to the SUIT standard that are needed to support our framework. In Section 4.2, we present in detail the proposed workflows for the update generation and the update installation phases.

4.1 Extensions to the SUIT Information Model

To describe our extensions to the SUIT manifest, we adopt the same terminology as in RFC 9124, which defines the information model for the original SUIT manifest; therefore, we speak of *manifest information elements*, leaving their concrete encoding to be determined by the particular data model adopted.

4.1.1 SBOM. The first new required information element we propose is a SBOM for the update package. The SBOM is a structured list of all the software components involved in the construction of the update package. When an update is received by the device, as a preliminary check, this structured list is parsed and for each listed component a check for the existence of possible reported vulnerability related to it is performed. If a vulnerability is found, the update process is aborted and an appropriate notification is sent. We shall explain this process in more detail in Section 4.2 below. In this way, it is possible to quickly identify vulnerable software components before even starting their installation. Since the SBOM is only used in the initial phase of the update process,

it can be treated as a Severable Element that can be safely discarded once the absence of known vulnerabilities is established.

There are three main standards currently in use for specifying an SBOM: SPDX (an ISO/IEC standard), SWID (an older ISO/IEC standard), and CycloneDX (a more recent standard developed by OWASP). For implementation purposes, we selected the CycloneDX standard, which seems particularly suitable for IIoT devices; however, nothing in our proposal depends on this choice, and any other relevant standard could be used as well.

There are many tools and platforms that support vulnerability management using CycloneDX, including OWASP Dependency-Track,[6] often cited as a reference implementation for consuming and analyzing SBOMs. The identification of known vulnerabilities can be achieved in particular through the use of the *Package URL* field, whose aim is to standardize how software package metadata is represented, the *Common Platform Enumeration* specification, designed to handle operating systems, applications, and hardware devices, and the *Software ID* field, which is defined in the ISO/IEC standard 19770-2:2015 and is used primarily to identify installed software.

4.1.2 BCM. The second new required information element is the BCM. The BCM is a structured list of formal properties satisfied by the contents of the update package, together with the corresponding proof certificates that enable the device to directly verify the validity of such properties before performing the update. We shall distinguish between a *proof certificate*, which is a textual representation of a formal proof written in some specified formal language [6], and a *proof descriptor*, which is a data structure whose purpose is to encapsulate a proof certificate together with all the information needed to verify it.

The BCM consists of a list of proof descriptors. Each proof descriptor is a record consisting of the following elements:

- A *property identifier*, which is used to uniquely identify the formal property that the proof descriptor refers to. Each property identifier must appear only once in the BCM.
- A list of *component identifiers*, which specifies what components affected by the update are targeted by the considered proof. Each element of this list must match the identifier of a corresponding software component affected by the update, as defined in the basic SUIT standard.
- A *language identifier*, which is used to specify the formal language in which the proof is expressed. By requiring each proof descriptor to define the language used in the corresponding proof, our proposal can remain fully agnostic about the particular tools used to generate proof certificates.
- The *proof certificate* itself, expressed in the formal language singled out by the language identifier.
- A *locality constraint*, which is a Boolean flag whose setting requires the proof verification step to be performed on the device, without involving any communication with the external world. This possibility can be useful for sufficiently powerful devices that, nevertheless, have no or very restricted access to the Internet, or when a verification server cannot be set up in a trusted way.
- A list of suggested *verification servers* (which is only considered when the locality constraint flag is not set). Each element of this list is the URI of an external server that can be queried to verify the associated proof certificate. Again, specifying the servers on a proof-by-proof basis is useful because some languages and/or proof techniques may be supported only by some of

[6]https://dependencytrack.org/.

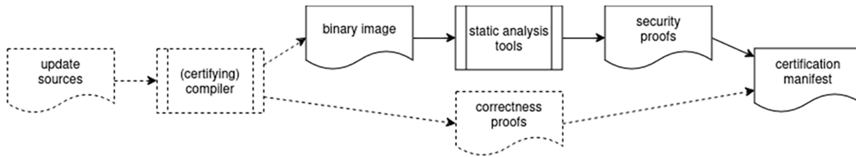

Fig. 3. Workflow for the generation of the Behavioral Certification Manifest. The dashed lines represent optional steps.

the available backends. This list is meant to be advisory only; the owner of the device can always opt for a trusted server of his choice when it is available.

Proof certificates can be generated in many ways. In this article, we focus on two options in particular: *certifying compilers* and *static analysis tools*. A *certifying compiler* is a compiler that is able to generate, in addition to the object code, also a proof certificate attesting that the compiled code satisfies some chosen correctness and/or security property. The concept has been introduced by Necula and Lee in 1998 [34]; similar notions in the literature include *proof generating compilers* [37] and *translation validation* [33, 36]. An application to embedded systems can be found in [8]. *Static analysis tools* are tools that analyze a program without (explicitly) executing it. Popular implementation techniques include symbolic execution [26], abstract interpretation [13], and model checking [11]. We refer the reader to [17] for an in-depth survey of these and other related topics.

4.2 Update Workflows

In this subsection, we describe in detail the proposed workflow for update generation and update delivery.

4.2.1 Update Generation. We shall divide the update generation process in two phases: generation of the BCM and generation of the update package. The first phase must be performed by two different entities: the agent responsible for producing the update (the *Firmware Author*, in the SUIT standard terminology) and a trusted third party, the *Proof Author*, whose presence is needed to independently verify the proof generation process.

The workflow for generating the BCM is depicted in Figure 3. The process starts by compiling the source code of the update, if it is available, possibly using a certifying compiler as explained in Section 4.1. The results of this step are the binary and zero or more proof certificates generated by the compilation process. This binary image, together with every other binary component whose sources are not available, is then subject to the appropriate static analysis tools, which generate further proof certificates to be bundled with the update. Finally, all the proof artifacts are collected in the BCM, together with the metadata needed to independently verify them.

When the Proof Author has completed these steps, he authenticates his artifacts and sends them to the Firmware Author, who compares the received artifacts with her own to check the Proof Author's trustworthiness. Notice that in the absence of an independent Proof Author, the Firmware Author could simply package the update binary image with made-up (but correct) proofs, unrelated to the actual content of the update. Therefore, the presence of the Proof Author's signature is essential to ensure the reliability of the overall scheme.

The workflow for the construction of the update package is summarized in Figure 4. The inputs are the binary image of the update (signed by the Proof Author), the SBOM, and the BCM (also signed by the Proof Author). To build the final update package, the Firmware Author will add the BCM to the regular SUIT manifest, along with a digest of the SBOM and all other metadata necessary for the update process. The manifest is then signed and packaged along with all the

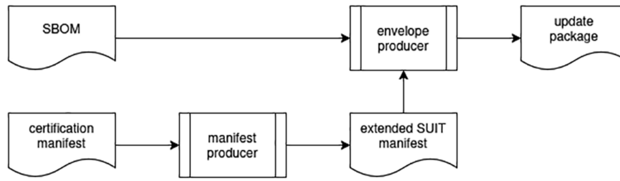

Fig. 4. Workflow for the generation of the Update Package.

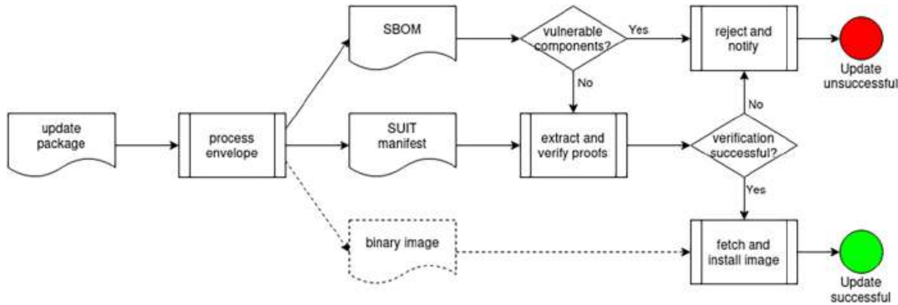

Fig. 5. Workflow for update installation. The dashed lines represent optional steps.

severable elements (including the SBOM itself) and other payloads (including the binary image, in case it should be distributed together with the manifest) to create the SUIT envelope. Finally, the Firmware Author sends the envelope to the Firmware Server, in order to enable distribution of the update, and notifies the Status Tracker Server of the availability of the update.

4.2.2 Update Installation. The corresponding workflow for the installation of an update package is depicted in Figure 5. At update reception time, the FC starts processing the SUIT envelope by verifying its cryptographic signature. If this basic integrity check passes, the FC can proceed further and extract the SBOM, the manifest itself, and the integrated firmware image (if present). The FC then checks the SBOM for the presence of vulnerable components by querying an external (trusted) database of known vulnerabilities, as explained in Section 4.1. If some vulnerable or otherwise undesirable component (e.g., for licensing reasons) is found, the update is rejected. If no vulnerable components are found, the FC extracts the formal proofs packaged in the manifest and verifies them by feeding each proof to an appropriate proof checker. This step can either be performed directly on the device (if the available resources allow it) or offloaded to an external server, as described in more detail in Section 5.3. If any of the proofs fails to verify, the update is rejected. If all the proofs are successfully verified, the FC can proceed with the regular SUIT installation workflow, either using the binary image embedded in the update envelope or fetching it from the URI in the manifest.

When an update is rejected as invalid, an appropriate notification must be sent to the Status Tracker Server and to any relevant authority in charge of the update process. In particular, the device owner is expected to register a way to receive notifications about the results of the update. These notifications must contain both the identifier of the component/property whose proof certificate check failed, and the reason for the failure (e.g., vulnerable component, bad proof certificate, server timeout). A detailed log may also be attached; alternatively, the log can also be put on a private or public blockchain.

Figure 6 shows an example of a sequence diagram depicting a successful update installation that follows the guidelines of the present proposal. This particular setup involves a server-initiated

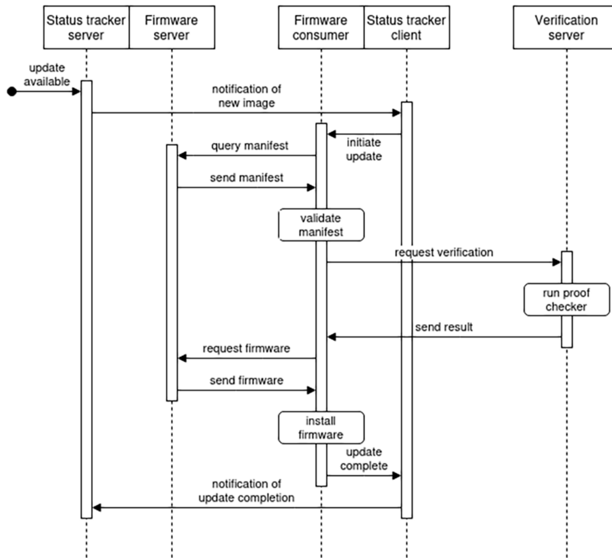

Fig. 6. A possible sequence diagram for firmware distribution. Here, we adopt a push model (update is initiated by the Status Tracker Server) in which the manifest and the firmware image are sent separately, and proof checking is performed remotely.

update, a proof verification step performed remotely, and a binary image retrieved separately from the manifest. Similar diagrams handling, e.g., local proof checking, or binary images bundled with the manifest can be easily derived from the one in Figure 6.

4.3 Security Analysis

This section analyses the security of the proposed scheme against the threats considered in Section 3.2.

— *Reading the Updates.* This attack is prevented by encrypting the content of the update. Encryption can be applied either to the update source or to the binary itself. This helps prevent the leakage of sensitive information related to the update. The manifest is left in clear, but does not contain any Intellectual Property-sensitive information.
— *Denying Updates.* To mitigate this attack, the update must include all the metadata required to complete the update installation (e.g., the identifier of the device class to which the update applies, the location where the update package has been saved on the device, the vendor identifier, the format). Metadata is signed to protect its integrity and authenticity.
— *Rollback Attacks.* This attack can be mitigated by including in each update a monotonic version number. However, this alone may not be enough, since the attacker may target a device that has been offline for a long time and runs an old firmware version. The attacker sends an old, but valid, manifest to a device with an old and vulnerable, but valid, firmware image. This attack is mitigated by requiring the device to confirm the version number of the just installed update to the firmware server in the last notification message.
— *Modify or Generate Completely Fraudulent Updates.* To mitigate manipulation of the content of the update by attackers able to eavesdrop messages, firmware update images are signed by the Firmware Author's key. However, this is not sufficient in case the signing key is compromised. To mitigate this scenario, our scheme uses proofs to certify the behavior of the

update. Manifests attesting to different certified behaviors can be generated either from the source code, if available, or directly from the update binary. It is essential to establish two separate security domains: one for generating and signing the firmware update, and another for computing and signing the proof using a different key. In some cases, the entity responsible for generating and signing the BCM is different from the Firmware Author. In other cases (such as when intellectual property constraints prevent this separation), the Firmware Author must implement both security domains internally. By introducing the separation of concerns, the attacker now has to compromise two different processes to mount a successful attack.

5 Generating and Checking Behavioral Certification Artifacts

A behavioral property is any property that concerns the safety, security, or other behavioral aspects of the code affected by an update. Examples of typical properties are that the received update has not been modified by adding new malicious code or calls to malicious functions, or that a configuration possibly associated to the update does not violate predefined security policies like confidentiality or non-interference conditions. This section describes how to formally specify a behavioral property, how to check it on a given program (either in source or object code form), and how to produce a proof certificate that witnesses the truth of that property.

5.1 Formal Verification of Behavioral Properties

We shall discuss in detail two broad classes of security properties which are amenable to a formal verification process. The properties in the first class apply already at the source code level and are thus suitable to be implemented as part of a certifying compilation approach. The properties in the second class are concerned with the object code and are thus more suitable to be checked with the aid of a static binary analysis framework.

5.1.1 Static Checking of Configuration Parameters. In many applications, especially in IIoT and embedded settings, the firmware update will involve a set of statically determined configuration parameters, like for instance memory maps, partitioning of resources, and so on. The correctness of these parameters can be checked at compile time by defining a formal specification for their allowed values as an SMT formula [7], whose validity is then verified by an SMT solver as part of the compilation pipeline. The resulting proof certificate can be used as a guarantee of the correctness of the configuration deployed by the update.

As a paradigmatic example of this kind of property, we consider the case of a *separation kernel* like, e.g., BAO [28]. A separation kernel is a trusted component whose purpose is to partition the available memory of a device into a finite number n of disjoint and nonempty regions $R_1, \dots, R_n$, where each region is assigned to a different **Virtual Machine (VM)** running on the device. Each region's physical location and size is specified by a memory map statically determined at configuration time.

Figures 7 and 8 show two examples of this kind of setup. There are two VMs, labeled 1 and 2; the first VM reserves three different (disjoint) memory regions $R_{1,1}$, $R_{1,2}$, and $R_{1,3}$, whereas the second VM reserves a single memory region R_2. Figure 7 represents an example of a valid configuration: all regions pertaining to VM 1 are disjoint, and the union of all regions reserved by VM 1 is disjoint from the region reserved by VM 2. In contrast, Figure 8 represents an example of an invalid configuration: here, the region R_2 overlaps with region $R_{1,3}$ reserved by VM 1, so this configuration should be rejected by the separation kernel at compilation time.

In order to formalize this requirement, suppose that the configuration of the separation kernel involves an array vms which contains the configuration parameters for each VM. Among these parameters, there is an array regions whose elements are records containing two unsigned integer

Fig. 7. A possible memory map for a simple configuration with two Virtual Machines.

Fig. 8. Another (incorrect) memory map for the same configuration (regions R_2 and $R_{1,3}$ overlap, red part).

fields start and end (with start $<$ end) that specify the start and end addresses of each memory region allocated to the given VM.

Given the ith VM and an integer variable x corresponding to an address of the memory, we define a predicate $\mu_{i,j}(x)$ expressing the fact that x belongs to the jth memory region for the ith VM, and a predicate $\mu_i(x)$ expressing the fact that x belongs to the union of all the memory regions for ith VM as follows:

$$\mu_{i,j}(x) := \mathrm{vm}[i].\mathrm{regions}[j].\mathrm{start} \leq x < \mathrm{vm}[i].\mathrm{regions}[j].\mathrm{end}, \tag{1}$$

$$\mu_i(x) := \bigvee_j \mu_{i,j}(x). \tag{2}$$

The correctness specification for the configuration parameters of the ith VM is then expressed by the formula:

$$\psi_i := \forall x.\forall j, j'(j < j' \rightarrow \neg(\mu_{i,j}(x) \wedge \mu_{i,j'}(x))) \tag{3}$$

formalizing the requirement that no pair of distinct regions overlap.

Similarly, the correctness of the whole configuration, i.e., no pair of distinct regions for a VM overlap and that no pair of memory regions assigned to the different VMs overlap, can be expressed by the formula:

$$\varphi := \forall i.\psi_i \wedge \forall x.\forall i, i'(i < i' \rightarrow \neg(\mu_i(x) \wedge \mu_{i'}(x))). \tag{4}$$

Note that the indices i and j always range over finite domains, so every universal quantification over them can be replaced with a finite conjunction. Thus, the only essential variable in Equations (1)–(4) is the integer variable x.

5.1.2 Policies on the Control Flow of Updated Binaries. A security policy may require that every update of the software stack of a high-assurance IIoT system does not alter the control flow of the replaced components. This general requirement can be spelled out in many different ways; let us focus on formalizing some of them.

The CFG of a program is a directed graph that encodes all possible execution paths that the program can take during its runtime. It can be formally defined as a pair $\Gamma = (V, E)$, where:

- V is a set of *vertices*; each vertex $v \in V$ represents a *basic block*, i.e., a sequence of instructions that do not alter the control flow of the program.
- E is a set of *edges*; each edge $e \in E$ is an ordered pair of vertices (v_0, v_1), representing a possible control flow transfer from the end of the "source" block v_0 to the beginning of the "target" block v_1.

At the object code level, only two kinds of control flow transfer instructions are available: conditional or unconditional branch instructions (also known as jumps), and function call/return instructions. Thus, each basic block of a binary program ends either with one of these instructions or immediately

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
    char *target = NULL;
    if (argc > 1) {
        target = argv[1];
    } else {
        target = "world";
    }
    printf("Hello,␣%s!\n", target);
    return 0;
}
```

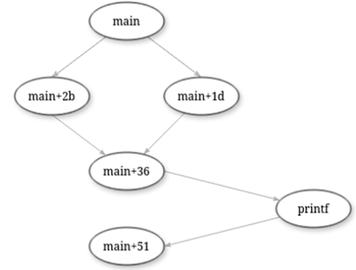

Fig. 9. Left: the hello program. Right: the control flow graph of the object code for the `main` function.

before the target of a branch. It is important to emphasize that in general the execution path of any nontrivial program cannot be determined statically, since the sequence of instructions to be executed typically depends on data which are only available at runtime.

In the left-hand pane of Figure 9, we show a very simple example of a C program whose control flow depends on input data. The program prints a different greeting depending on whether at least one argument has been supplied to it or not. In the right-hand pane, we depict the CFG of the compiled version of the `main` function. The two different execution paths (which are taken according to whether `argc` is greater than 1 or not) are clearly noticeable, as well as the call to the standard library function `printf`. In this regard, note that for the present purposes, it is natural to stop the process of control flow reconstruction at the C standard library interface, treating standard library functions like `printf` as atomic primitives. This is because we are typically not interested in considering the control flow inside the C standard library; rather, we *assume* that the standard library code is safe and correct, and base the proofs about our code on this assumption.

The vertices of the CFG displayed in Figure 9 have been labeled with symbolic names associated with the starting address of each basic block. These names come either directly from the source code, in case of function entry points (e.g., `main`, `printf`), or have the form `name+offset` where `name` is a function name and `offset` is a (hexadecimal) offset from the function's start address. As we shall see later, it is important to keep track of these names because they can help the SMT solver in the search for a proof.

The strictest reading of the informal "control flow preservation" policy is just to require that the CFG of the update is exactly the same as the CFG of the replaced binary. Formally, this corresponds to the notion of *isomorphic graphs*.

Definition 1. Given two directed graphs $\Gamma_1 = (V_1, E_1)$ and $\Gamma_2 = (V_2, E_2)$, an *isomorphism from* Γ_1 *to* Γ_2 is a map $f\colon V_1 \longrightarrow V_2$, which is:

$$\text{Injective:} \qquad \forall v, v' \in V. f(v) = f(v') \Rightarrow v = v', \tag{5}$$

$$\text{Surjective:} \qquad \forall w \in V_2\ \exists v \in V_1. f(v) = w, \tag{6}$$

$$\text{Homomorphic:} \qquad \forall v_1, v_2 \in V_1.(v_1, v_2) \in E_1 \Leftrightarrow (f(v_1), f(v_2)) \in E_2. \tag{7}$$

When such a map exists, the two graphs Γ_1 and Γ_2 are said to be *isomorphic.*

We remark that, in this setting, the existence of an isomorphism between CFGs is a very strong requirement: indeed, it implies a one-to-one correspondence between the basic blocks (and, consequently, between the control flow transfer instructions) of the two binaries considered. Of course,

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
  char *target = NULL;
  if (argc == 2) {    // Modified condition
      target = argv[1];
  } else {
      target = "world";
  }
  printf("Hello,␣%s!\n", target);
  return 0;
}
```

Fig. 10. The hello-0 program.

this does not mean that the update is necessarily functionally equivalent in every respect to the replaced binary. For instance, the author of the hello program of Figure 9 may decide that the program should only display the personalized greeting when it is executed with *exactly one* argument. This corresponds to replacing the program in Figure 9 with the program in Figure 10, whose CFG is clearly isomorphic to the one in Figure 9.

On the other hand, the author may decide to keep the original behavior of hello but display a warning in the case where more than one argument has been specified. A version of hello that conforms to this new specification is displayed in Figure 11, along with the CFG of the corresponding `main` function. As a side note, notice how the C compiler used (gcc version 12.2.0) replaced the second call to `printf` with a call to the more efficient `puts` library function. This highlights the importance of reasoning about control flow at the object code level even when the source is available, as compilers can typically substantially alter the "naive" control flow that can be inferred just by looking at the source code (especially when optimizations are turned on).

It is clear that the new desired behavior cannot be implemented without altering the CFG of the original program: at the very least, a further branch to distinguish between the cases `argc == 2` and `argc > 2` is unavoidable. To cover situations of this kind, it is thus desirable to relax a bit the previous formalization of the control flow preservation property by allowing the update to contain additional code that *does not modify the control flow of existing code*. One way to formalize this requirement is to replace the very strong isomorphism condition with the weaker requirement of the existence of an *embedding* of the original CFG into the CFG of the update.

Definition 2. Given two directed graphs $\Gamma_1 = (V_1, E_1)$ and $\Gamma_2 = (V_2, E_2)$, an *embedding of* Γ_1 *into* Γ_2 is a map $f : V_1 \longrightarrow V_2$, which is injective (cf. Equation (5)) and such that

$$\forall v_1, v_2 \in V_1.(v_1, v_2) \in E_1 \Rightarrow (f(v_1), f(v_2)) \in E_2. \tag{8}$$

The difference with respect to Definition 1 is twofold: first, we no longer require f to be surjective, which means that some vertices in Γ_2 may have no correspondent in Γ_1. Second, condition (7) is weakened by removing the inverse implication: we only require that each edge in Γ_1 has a matching one in Γ_2, but the latter may contain additional edges. In this way, we can formally capture the idea of a preservation of the original control flow in passing from a binary to its updated version. Figure 12 shows the obvious embedding of the CFG of the original hello program (Figure 9) into the CFG of its extended version (Figure 11).

Thus, we can formalize the informal "control flow preservation" requirement as follows:

Definition 3. Given an original program with a CFG Γ_{orig}, and another program corresponding to its update with a CFG Γ_{upd}, we define *Control Flow Invariance* and *Control Flow Preservation* as

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    char *target = NULL;
    if (argc > 1) {
        target = argv[1];
    } else {
        target = "world";
    }
    printf("Hello,␣%s!\n", target);
    if (argc > 2) {
        printf("Some␣arguments␣have"
               "␣been␣ignored\n");
    }
    return 0;
}
```

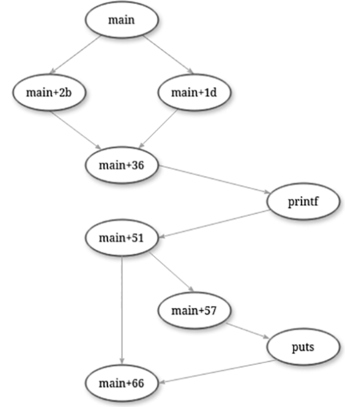

Fig. 11. Left: the hello-1 program. Right: the control flow graph of the object code for the `main` function.

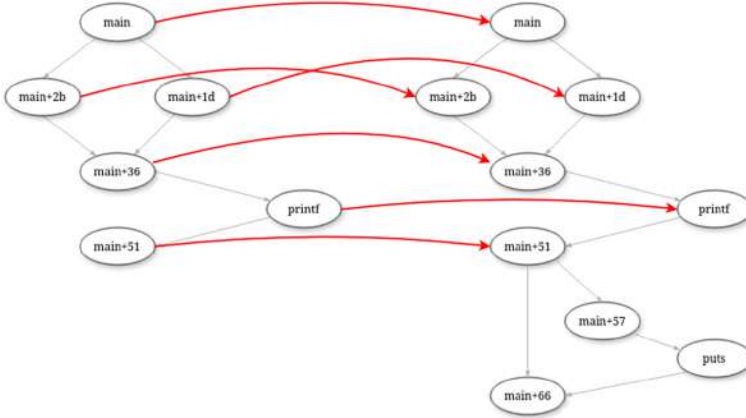

Fig. 12. An embedding f of the control flow graph of hello main function inside the control flow graph of hello-1 `main` function. The red arcs are the mappings defined by the function f.

follows:

$$\text{Control Flow Invariance : there exists a graph isomorphism } f\colon \Gamma_{\text{orig}} \longrightarrow \Gamma_{\text{upd}}, \tag{9}$$

$$\text{Control Flow Preservation : there exists a graph embedding } f\colon \Gamma_{\text{orig}} \longrightarrow \Gamma_{\text{upd}}. \tag{10}$$

5.2 Generating Proof Certificates for Behavioral Properties

We now explain how the two behavioral properties considered above can be encoded in a formal theory handled by SMT solvers, and how to produce a proof certificate that can be included in the BCM as a witness for the truthfulness of the property.

5.2.1 Memory Partitioning. It is straightforward to translate Equations (1)–(4) into the theory of Quantifier-Free Linear Integer Arithmetic, which is typically very well supported by every state-of-the-art SMT solver. As an example, we show the translation in the SMT-LIB2 format of the verification problem for the configuration depicted in Figures 7 and 8.

```
1  (set-logic QF_LIA)
2  (declare-fun x () Int)
3  (assert
4    (let ((mu_1_1 (and (<= vm1_r1_start x) (< x vm1_r1_end)))
5          (mu_1_2 (and (<= vm1_r2_start x) (< x vm1_r2_end)))
6          (mu_1_3 (and (<= vm1_r3_start x) (< x vm1_r3_end)))
7          (mu_2_1 (and (<= vm2_r1_start x) (< x vm2_r1_end))))
8      (let ((mu_1 (or mu_1_1 mu_1_2 mu_1_3))
9            (mu_2 (or mu_2_1)))
10       (let ((tau_1
11         (or (and mu_1_1 mu_1_2) (and mu_1_1 mu_1_3) (and mu_1_2 mu_1_3))))
12         (let ((tau_2 (and mu_1 mu_2)))
13           (or tau_1 tau_2))))))
14 (check-sat)
```

Fig. 13. SMT-LIB encoding of the validity check for the configuration of the separation kernel.

Since our goal is to *verify* a specification, that is, to prove that Equation (4) is *valid* for every possible value of x, we need to query the SMT solver to get a proof that the *negation* of that formula is not satisfiable. Therefore, the logical formula we need to encode is

$$\neg\varphi \equiv \big(\exists x.\neg\forall i\forall j, j'(j < j' \to \neg(\mu_{i,j}(x) \wedge \mu_{i,j'}(x)))\big) \vee (\exists x.\neg\forall i, i'(i < i' \to \neg(\mu_i(x) \wedge \mu_{i'}(x)))) . \tag{11}$$

Pushing the negations inside the quantifiers, we see that this is logically equivalent to finding a witness x for the disjunction $\tau_1(x) \vee \tau_2(x)$, where

$$\tau_1(x) \coloneqq \exists i, j, j'.j < j' \wedge \mu_{i,j}(x) \wedge \mu_{i,j'}(x), \tag{12}$$

$$\tau_2(x) \coloneqq \exists i, i'.i < i' \wedge \mu_i(x) \wedge \mu_{i'}(x). \tag{13}$$

In Figure 13, we display a possible SMT-LIB encoding of this formula. We first define the predicates $\mu_{i,j}$ for the relevant pairs $(i, j) \in \{(1, 1), (1, 2), (1, 3), (2, 1)\}$ (lines 4–7), where the constants `vm1_r1_start`, and more, are to be replaced with the concrete values extracted from the configuration of the separation kernel. Then, we define the predicates μ_1 and μ_2 as the relevant disjunctions (lines 8 and 9) and define the translations of Equation (12) (lines 10–13; the case $i = 2$ produces no clauses because there is a single memory region for VM 2) and the translation of Equation (13) (line 14). The final formula to be checked for unsatisfiability can be found in line 15.

5.2.2 Control Flow Preservation. In order to apply an SMT solver to the task of proving a control flow preservation property, we need to translate Equations (9) and (10) into one of the supported theories. Since the unknown is in both cases a function between the enumerated sets, it is natural to use the combination of the *theory of datatypes* DT (to specify the vertex sets) and the *theory of uninterpreted functions* UF (hereafter referred to as UFDT).

A generic directed graph $\Gamma = (V, E)$ can then be encoded as a term in the UFDT theory as follows:

- The set $V = \{v_1, \ldots, v_n\}$ is translated to a datatype declaration whose elements are the elements of V.
- The set E is translated to the definition of a *Boolean function* on the product $V \times V$ which is true exactly for those pairs of vertices (v_0, v_1) such that $(v_0, v_1) \in E$. This works because we are only dealing with *simple* graphs, so that between each pair of vertices there is at most 1 edge.

```
(set-logic UFDT)
(declare-datatype V1 ((V1_main) (V1_main+2b) (V1_main+1d) (V1_main+36) (V1_main+51)
  (V1_printf)))
(define-fun g1 ((x V1) (y V1)) Bool
  (or (and (= x V1_main) (= y V1_main+2b)) (and (= x V1_main) (= y V1_main+1d))
      (and (= x V1_main+1d) (= y V1_main+36)) (and (= x V1_main+36) (= y V1_printf))
      (and (= x V1_printf) (= y V1_main+51))))
(declare-datatype V2 ((V2_main) (V2_main+2b) (V2_main+1d) (V2_main+36) (V2_main+51)
  (V2_printf) (V2_main+66) (V2_main+57) (V2_puts)))
(define-fun g2 ((x V2) (y V2)) Bool
  (or (and (= x V2_main) (= y V2_main+2b)) (and (= x V2_main) (= y V2_main+1d))
      (and (= x V2_main+1d) (= y V2_main+36)) (and (= x V2_main+36) (= y V2_printf))
      (and (= x V2_printf) (= y V2_main+51)) (and (= x V2_main+51) (= y V2_main+66))
      (and (= x V2_main+51) (= y V2_main+57)) (and (= x V2_main+57) (= y V2_puts)
      (and (= x V2_puts) (= y V2_main+66)))))
(declare-fun f (V1) V2)
(assert (and (= (f V1_main) V2_main) (= (f V1_printf) V2_printf)))
(assert (and (forall ((va V1) (vb V1)) (=> (g1 va vb) (g2 (f va) (f vb))))
             (forall ((va V1) (vb V1)) (=> (= (f va) (f vb)) (= va vb)))))
(check-sat)
(get-model)
```

Fig. 14. SMT-LIB encoding for the existence of an embedding between the control flow graphs of the hello and hello-1 programs.

For example, a directed graph with three vertices $V = \{v_1, v_2, v_3\}$ and two edges $E = \{(v_0, v_1), (v_1, v_2)\}$ is translated into the following definitions:

```
(declare-datatype V ((v1) (v2) (v3)))
(define-fun g ((x V) (y V)) Bool
  (or (and (= x v1) (= y v2))
      (and (= x v2) (= y v3))))
```

Now that we have a way to define the graphs, let us consider the translation of the two properties (9) and (10).

Let us start from the simpler embedding property. We assume to have encoded the two CFGs as the datatypes V1 and V2 and Boolean functions g1 and g2, as described above, and seek a formula whose models are the embeddings of Γ_1 into Γ_2. The translation of Definition 2 into the language of UFDT theory is straightforward and reads as follows:

```
(declare-fun f (V1) V2)
(assert    ; injectivity property
  (forall ((va V1) (vb V1)) (=> (= (f va) (f vb)) (= va vb))))
(assert    ; graph homomorphism property
  (forall ((va V1) (vb V1)) (=> (g1 va vb) (g2 (f va) (f vb)))))
```

Then, any model for this set of formulae (assuming it is satisfiable) gives a function f, which is the desired embedding. In Figure 14, we show the SMT-LIB file for the existence of the embedding depicted in Figure 12.

It is important to note that an SMT problem of this form may have more than one solution in general. In order to ensure that the right solution is picked by the solver, it is useful to impose some additional constraints involving the symbolic names of the vertices in the CFG. At the very least, we want to impose that *every function entry point* is preserved by embedding f. In the example of

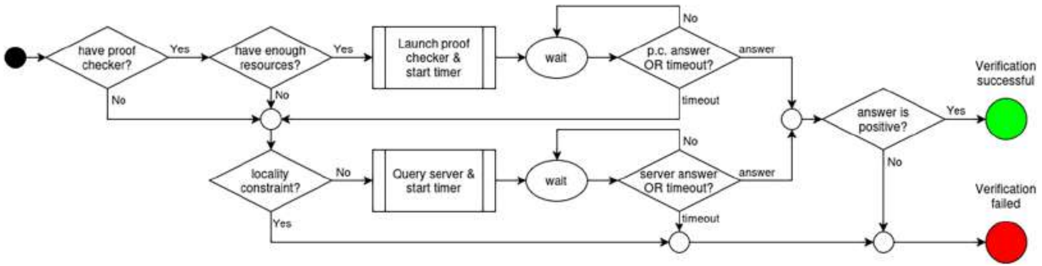

Fig. 15. Workflow for the verification of a single proof.

Figure 14, this is done by the assertion at line 17. One may also supply other hints of this form to the solver; however, notice that a blanket identification of symbolic names involving addresses is unsound in general, since code additions in the updated binary may alter the starting address of a basic block and make it coincide with the starting address of a *different*, unrelated block in the original binary.

Let us now consider the CFG isomorphism property (9). In order to translate Definition 1 into our language, we need to add two ingredients: the surjectivity property (6) and the inverse graph homomorphism property. However, the surjectivity test can be omitted as soon as we check that the two vertex sets V1 and V2 have the same cardinality (a necessary condition for the existence of a graph isomorphism), since a map between equally numerous sets is surjective if and only if it is injective. This check can be trivially done during the generation phase for the datatype definitions.

We conclude that the only necessary addition is to turn the single implication (8) into the double implication (7), that is, to add the following constraint:

```
1 (assert          ; inverse graph homomorphism property
2   (forall ((va V1) (vb V1)) (=> (g2 (f va) (f vb)) (g1 va vb))))
```

Of course, also in this case, it is possible to add additional constraints on the mapping of particular vertices, which can help the solver in finding a solution.

5.3 Checking Proof Certificates

Figure 15 shows the suggested activity diagram for the verification of each proof contained in the BCM. The task can be decomposed into the following steps. First, the FC must decide whether it can perform the check locally or not. To do it, the following two conditions must be met: (1) a proof checker for the formal language selected by the language identifier field in the corresponding proof descriptor must be locally available, and (2) the device must have sufficient resources to perform the check. The second condition involves both the device's current state and the proof certificate's characteristics. For example, the check could be as simple as a single condition "available memory $\geq k \cdot \text{size}(\text{proof})$," where proof denotes the proof certificate and the constant k is chosen using some appropriate heuristics, perhaps involving the formal language used and/or the kind of property that must be verified.

If the check can be performed locally, the appropriate proof checker is launched and a timer is set. If the proof checker terminates within the chosen time bound, we read its answer and act accordingly; otherwise, the proof checker process is killed. In all the cases in which the local verification is not successful, we still have the chance of checking the proof using a remote server, assuming that this has been allowed by the update author (i.e., the locality constraint flag is not set). In this case, we query either a known server or one of the remote server(s) specified by the

proof descriptor itself and wait for the result. Again, the process should abort if the answer does not arrive within a given time limit, which now should also take into account network latency effects.

It is important to emphasize that, in general, it is very difficult to decide in advance the amount of resources needed to verify a proof certificate. Proof checkers typically contain sophisticated heuristics to manage the memory used depending on the available hardware resources [24, 38]. The size of the proof certificate, as well as other easily measured metrics like the number of intermediate steps, are not reliable indicators of the computational complexity of the task. A relatively large proof certificate expressed in a formal theory T may need fewer resources than a smaller one based on a different formal theory T'. The best approach that the Firmware Author can take is probably to verify, ahead of deployment, whether each of the proof certificates contained in the BCM can be successfully verified on a representative target device under typical load conditions. If the test does not go through, he can make sure that the locality constraint for that proof is not set and specify one or more suitable verification servers. The flexibility of the BCM specification is essential in allowing such leeway in the preparation of each update package.

6 Implementation and Evaluation

To verify the viability of our proposal, we implemented a prototype of our framework[7] by extending a reference implementation of the SUIT standard with new code to produce and verify the extended update package.

6.1 Toolchain

For the update production phase, we need a tool to generate the new components described in Section 4.2 and package them in the SUIT Envelope. After evaluating the available implementations for the creation of SUIT Manifests and Envelopes, we chose SUIT-Tool[8] by ARM as the target for extension. This Python tool facilitates the generation and signing of manifests starting from a JSON input file structured similarly to the resulting manifest. Despite the limited documentation available, extending the tool to support the additional fields is relatively straightforward given the modular structure of the implementation.

For proof generation, we leveraged the capabilities of the cvc5 solver [6]. Obtaining proof certificates for the resource partitioning property described in Section 5.2.1 is straightforward: we only need to instruct cvc5 to produce a proof of the unsatisfiability of Equation (11). The control flow preservation properties of Section 5.2.2 are more involved since it is necessary to extract the CFG of a binary image in order to build the SMT problem described in Section 5.2.2. After a brief evaluation of the various tools available for this task, we selected ANGR,[9] a Python-based binary analysis framework which is widely used in the field and supports out of the box the most common ISAs (including x86_32, x86_64 and ARM). We thus developed a Python script, called *check_cfp.py*, that takes as input two binary files (interpreted as the original and the updated versions of a firmware component), establishes if the required property holds, and if the answer is affirmative generates a proof certificate for it. As motivated in Section 5.1.2, we instructed ANGR to avoid resolving calls to C standard library functions (i.e., we set the flag `auto_load_libs` to False). For the CFG reconstruction, we use the CFGEmulated procedure, which is based on a symbolic execution engine. In our experience, it yields better results than its CFGFast alternative, and the longer execution times were not a problem for the moderately sized binaries on which we applied

[7] https://github.com/CybersecurityUnitn/secureUpdate.
[8] https://gitlab.arm.com/research/ietf-suit/suit-tool.
[9] https://github.com/angr/angr.

Table 1. Size of Sample SBOMs

Id	# of libraries	Size
1	3	3.2 KB
2	25	24 KB
3	43	40 KB

Table 2. Size of Proofs for Non-Interference

Name	# of areas	Proof size
mempart	4	15 KB
pic32mz2	8	40 KB
newmap	16	111 KB

it. After obtaining the CFGs of both the original and the updated version of the binary, the two resulting graphs are converted into SMT-LIB format, as explained in Section 5.2.2, and bundled with the encoding of the relevant property. Then, the cvc5 solver is invoked on the resulting SMT file to obtain a model for the function f. Finally, the model is incorporated inside the SMT-LIB source together with the *negation* of the required property, in order to obtain an unsatisfiable formula. The corresponding proof certificate is then returned.

For the update installation phase, we extended ARM's reference implementation Suit-Parser,[10] written in C, to support the new fields added to the SUIT manifest. In addition to manifest parsing and verification, our implementation supports the extraction of the SBOM and of the proof certificates contained in the BCM. To perform proof checking, we used the ETHOS tool[11] by the same team that developed cvc5. A problem we encountered at this stage is that the proofs generated by cvc5 for the control flow-related properties are typically very large, of the order of 500 KB for small routines (CFG with ~10 vertices), 20–30 MB for medium-sized ones (CFG with ~40 vertices), and 100–200 MB for large ones (CFG with ~80 vertices). Compression mitigates this problem somewhat, at least while proofs are in transit, but the possibility of a local verification for proofs of this size is clearly limited to the most powerful device classes.

6.2 Test Manifests

To assess the performance of our implementation across different scenarios, we generated a set of 10 sample update manifests, each containing a varying number of updated components, an SBOM, and a BCM of various sizes. All the sample manifests use remotely fetched binary images, as the goal of our tests is only to analyze the impact of the new fields which are not already part of the SUIT standard.

For the SBOM, we adopted the CycloneDX v1.6 specification and prepared three different samples, whose characteristics are shown in Table 1. The first example covers the case of a small component with very few dependencies, as typically found in IIoT firmware. The other two examples are drawn from the official BOM Example repository[12] and are more representative of larger pieces of software linking many libraries from different sources.

The BCMs have been created by putting together elements drawn from a set of nine different proof certificates, obtained by verifying both classes of properties described in Section 5. For the memory partitioning case, we wrote three test configurations with 4–16 non-overlapping memory regions; the sizes of the resulting proof certificates are summarized in Table 2. For the control flow isomorphism property, we used the six Trusted Applications found in the distribution of the OpTEE Operating System[13] as typical examples of security-critical applications running on IIoT devices. We thus created a new version of each of these applications by altering the source code of a single function without altering the control flow of the application, and verified the corresponding

[10] https://gitlab.arm.com/research/ietf-suit/suit-parser.
[11] https://github.com/cvc5/ethos.
[12] https://github.com/CycloneDX/bom-examples.
[13] https://github.com/linaro-swg/optee_examples.

Table 3. Size of Proofs for Control Flow Isomorphism

Name of TA	Analyzed function	Size of CFG (nodes, edges)	Proof size	Proof size (gzip+b64)
acipher	cmd_enc	(32, 49)	15 MB	2.4 MB
aes	cipher_buffer	(12, 17)	1.2 MB	212 KB
hello_world	inc_value	(8, 10)	418 KB	84 KB
hotp	get_hotp	(64, 101)	104 MB	16 MB
random	random_number_generate	(17, 23)	2.6 MB	477 KB
sec_storage	read_raw_object	(45, 71)	38 MB	6.1 MB

Table 4. Structure of the Example Manifests

Name	# of comp.	SBOM	Proofs in the BCM	Total size
small-1	1	1	mempart, hello_world, aes	304 KB
small-2	2	1	pic32mz2, hello_world, aes, random	787 KB
medium-1	1	2	mempart, hello_world, aes	331 KB
medium-2	2	2	pic32mz2, hello_world, aes, random	814 KB
medium-3	4	2	newmap, aes, aes, random, random	1.4 MB
medium-4	4	2	pic32mz2, aes, random, acipher	3.2 MB
large-1	2	3	pic32mz2, hello_world, aes, random	835 KB
large-2	4	3	pic32mz2, aes, random, acipher	3.2 MB
large-3	7	3	newmap, aes, random, acipher, acipher	5.6 MB
large-4	10	3	newmap, aes, random, acipher, sec_storage	9.2 MB

invariance property by running the *check_cfp.py* script on them. The characteristics of the function analyzed, as well as the sizes of the resulting proof certificates, are shown in Table 3. The generated proof certificates were compressed using gzip and encoded with Base64 to enable inclusion inside the JSON file used by the manifest generator.

The contents of the sample manifests are summarized in Table 4. The Components column refers to the size of the list of updated components. The identifier in the SBOM column refers to Table 1, and the Total size column reports the size of the signed manifest (the overhead added by the signature is irrelevant).

6.3 Experimental Evaluation

Once the update packages were prepared, we proceeded to set up the test environment using the implemented tools. For the target device, we selected a Raspberry Pi 4B equipped with a quad-core ARM Cortex-A72 processor running at 1.5 GHz and 2 GB of RAM, as it is a widely used representative of high-end IIoT devices. As operating system we used Linux 6.6.28, in a minimal installation typical for embedded devices provided by the OpTEE project.[14]

The test results and the structure of each example are presented in Table 5. We note that one of the proof certificates that we generated (hotp) is too large to be successfully verified on the chosen device within a time limit of 10 minutes, and for this reason, it has not been used in the sample manifests. This failure is caused by the exhaustion of the available memory on the target platform, which leads to thrashing. Thus, the control flow invariance for the hotp application is an example of a property for which local verification is impossible on the selected platform. The remedy, as explained in Section 4, is to delegate this step to a more capable remote verification server. We also

[14]https://github.com/OP-TEE/optee_os.

Table 5. Tests Execution Time

Name	Manifest parsing time (s)	Proof checking time (s)	Total time (s)	Maximum used memory (MB)
small-1	0.4	5.9	6.3	126
small-2	1.2	16.1	17.3	246
medium-1	0.5	5.9	6.4	126
medium-2	1.2	16.1	17.3	246
medium-3	2.0	29.7	31.7	247
medium-4	5.6	89.6	95.2	1,371
large-1	1.3	16.1	17.4	247
large-2	5.4	89.6	95.0	1,371
large-3	10	165	175	1,371
large-4	15	341	356	3,970

note that our toolchain does not exploit parallelism at the moment: the proofs in the BCM are verified sequentially, and each proof-checking task is bound to a single core of the four available on the Raspberry Pi, which has obvious consequences for running times.

By comparing the results for the manifests small-{1,2}, medium-{1,2} and large-1, we can see that the size of the SBOM has a very limited impact on the manifest parsing time. As our tests are purely local, this conclusion does not take into account the time that may be spent in a more sophisticated vulnerability detection process (e.g., if an external server must be queried for every dependency). We also observe that the manifest parsing time scales linearly with the size of the manifest and does not create a bottleneck, even when processing the largest examples. In contrast, the proof verification time scales linearly for small instances, but seems to grow quadratically for the larger examples. This highlights the heavy demand placed on the device by the proof verification process, in terms of both CPU and memory usage. This holds especially for the control flow-related properties, which tend to generate large proof certificates. In contrast, BCMs containing many smaller proofs with a similar cumulative size are processed much more efficiently.

7 Conclusions

In this article, we introduced a new scheme for remote updating of trusted, security- and safety-critical components in IIoT devices. Our proposal is based on an extension of the SUIT standard that integrates an SBOM to quickly identify and reject problematic components and a BCM to formally guarantee the preservation of the updated software's relevant safety and security properties.

As future work, we plan to extend the scope of our approach to new kinds of behavioral properties in addition to the ones considered in Section 5, such as safety or security properties that model checking techniques can prove; investigate alternative SAT/SMT encodings for the control flow properties of Section 5.2.2, to minimize the size of proof certificates; and consider the use of other SMT solvers and proof checkers. Furthermore, we would like to develop our current tools into a complete implementation of the SUIT standard, which can run in realistic environments.

Acknowledgment

The authors would like to thank the anonymous referees for their valuable feedback.

Disclosure

The authors declare that they have no competing interests.

References

[1] Xz: Malicious Code in Distributed Source. 2024. Retrieved November 14, 2024 from https://www.cve.org/CVERecord?id=CVE-2024-3094

[2] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-flat: Control-flow attestation for embedded systems software. In *CCC '16*. ACM, New York, NY, 743–754.

[3] Ron Amadeo. 2022. Samsung's App-Signing Key Leaked to Sign Malware. Retrieved from https://arstechnica.com/gadgets/2022/12/samsungs-android-app-signing-key-has-leaked-is-being-used-to-sign-malware/

[4] N. Asokan, T. Nyman, N. Rattanavipanon, A. Sadeghi, and G. Tsudik. 2018. ASSURED: Architecture for secure software update of realistic embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (Nov. 2018), 2290–2300.

[5] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *TACAS (1)*. Dana Fisman and Grigore Rosu (Eds.), Lecture Notes in Computer Science, Vol. 13243, Springer, 415–442.

[6] Haniel Barbosa, Clark W. Barrett, Byron Cook, Bruno Dutertre, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, et al. 2023. Generating and exploiting automated reasoning proof certificates. *Communications of the ACM* 66, 10 (2023), 86–95.

[7] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2021. Satisfiability modulo theories. In *Handbook of Satisfiability*. Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (Eds.), IOS Press, 1267–1329.

[8] Jan Olaf Blech and Michaël Périn. 2012. Generating invariant-based certificates for embedded systems. *ACM Transactions on Embedded Computing Systems* 11, 2 (Jul. 2012), Article 34, 1–22.

[9] Conner Bradley and David Barrera. 2023. Towards characterizing IoT software update practices. In *Foundations and Practice of Security*. Guy-Vincent Jourdan, Laurent Mounier, Carlisle Adams, Florence Sèdes, and Joaquin Garcia-Alfaro (Eds.), Vol. 13877, Lecture Notes in Computer Science, Cham, 406–422.

[10] Hans Chandra, Erwin Anggadjaja, Pranata Setya Wijaya, and Edy Gunawan. 2016. Internet of things: Over-the-air (OTA) firmware update in lightweight mesh network protocol for smart urban development. In *2016 22nd Asia-Pacific Conference on Communications (APCC)*, 115–118.

[11] E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 8, 2 (Apr. 1986), 244–263.

[12] Gareth Corfield. 2022. Leaked stolen Nvidia key can sign Windows malware. Retrieved from https://www.theregister.com/2022/03/05/nvidia_stolen_certificate/

[13] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*. ACM, 238–252.

[14] Markus Dahlmanns, Constantin Sander, Robin Decker, and Klaus Wehrle. 2023. Secrets revealed in container images: An internet-wide study on occurrence and impact. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIA CCS '23)*. ACM, 797–811.

[15] Nicolas Dejon, Davide Caputo, Luca Verderame, Alessandro Armando, and Alessio Merlo. 2019. Automated security analysis of IoT software updates. In *Proceedings of the 13th IFIP WG 11.2 International Conference on Information Security Theory and Practice (WISTP '19)*. Maryline Laurent and Thanassis Giannetsos (Eds.), Lecture Notes in Computer Science, Vol. 12024, Springer, 223–239.

[16] Krishna Doddapaneni, Ravi Lakkundi, Suhas Rao, Sujay Gururaj Kulkarni, and Bhargav Bhat. 2017. Secure FoTA object for IoT. In *2017 IEEE 42nd Conference on Local Computer Networks Workshops (LCN Workshops)*, 154–159.

[17] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (Jul. 2008), 1165–1178.

[18] F. Ebbers. 2023. A large-scale analysis of IoT firmware version distribution in the wild. *IEEE Transactions on Software Engineering* 49, 2 (2023), 816–830.

[19] S. El Jaouhari and E. Bouvet. 2022. Secure firmware over-the-air updates for IoT: Survey, challenges, and discussions. *Internet of Things* 18 (2022), 100508.

[20] Alberto Giaretta, Nicola Dragoni, and Fabio Massacci. 2022. S×C4IoT: A security-by-contract framework for dynamic evolving IoT devices. *ACM Transactions on Sensor Networks* 18, 1 (2022), Article 12, 1–51.

[21] Dan Goodin. 2023. Leak of MSI UEFI signing keys stokes fears of "doomsday" supply chain attack. Retrieved from http://arstechnica.com/information-technology/2023/05/leak-of-msi-uefi-signing-keys-stokes-concerns-of-doomsday-supply-chain-attack/

[22] U.S. Federal Government. 2021. Improving the Nation's Cybersecurity. Retrieved November 14, 2024 from https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity

[23] Michele Grisafi, Mahmoud Ammar, Marco Roveri, and Bruno Crispo. 2024. Flashadow: A flash-based shadow stack for low-end embedded systems. *ACM Transactions on Internet of Things* 5, 3 (Aug. 2024), 19:1–19:29.
[24] Andrew Healy, Rosemary Monahan, and James F. Power. 2016. Predicting SMT solver performance for software verification. In *F-IDE@FM*, 20–37.
[25] Trishank Karthik, Akan Brown, Sebastien Awwad, Damon McCoy, Russ Bielawski, Cameron Mott, Sam Lauzon, André Weimerskirch, and Justin Cappos. 2016. Uptane: Securing software updates for automobiles. In *International Conference on Embedded Security in Car*, 1–11.
[26] James C. King. Symbolic execution and program testing. *Communications of the ACM* 19, 7 (Jul. 1976), 385–394.
[27] Antonio Langiu, Carlo Alberto Boano, Markus Schuß, and Kay Römer. 2019. UpKit: An open-source, portable, and lightweight update framework for constrained IoT devices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2101–2112.
[28] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. 2020. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES '20)*, 1–14.
[29] Dimitris Mbakoyiannis, Othon Tomoutzoglou, and George Kornaros. 2019. Secure over-the-air firmware updating for automotive electronic control units. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM, Limassol, Cyprus, 174–181.
[30] Brendan Moran, Hannes Tschofenig, and Henk Birkholz. 2022. A manifest information model for firmware updates in internet of things (IoT) devices. *RFC* 9124.
[31] Brendan Moran, Hannes Tschofenig, Henk Birkholz, Koen Zandberg, and Øyvind Rønningstad. 2024. A concise binary object representation (CBOR)-based serialization format for the software updates for internet of things (SUIT) manifest. *Internet-Draft Draft-Ietf-Suit-Manifest-34, Internet Engineering Task Force*. Retrieved from https://datatracker.ietf.org/doc/draft-ietf-suit-manifest/
[32] Brendan Moran, Hannes Tschofenig, David Brown, and Milosch Meriac. 2021. A firmware update architecture for internet of things. *RFC* 9019.
[33] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, 83–94.
[34] George C. Necula and Peter Lee. 1998. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices* 33, 5 (May 1998), 333–344.
[35] European Parliament. 2024. Cyber Resilience Act. Retrieved November 14, 2024 from https://www.europarl.europa.eu/doceo/document/TA-9-2024-0130_EN.html
[36] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, 151–166.
[37] A. Poetzsch-Heffter and M. Gawkowski. 2005. Towards proof generating compilers. *Electronic Notes in Theoretical Computer Science* 132, 1 (2005), 37–51.
[38] Rustam Sadykov, Azat Abdullin, and Marat Akhin. 2025. Cache-a-lot: Pushing the limits of unsatisfiable core reuse in SMT-based program analysis. DOI: 10.48550/ARXIV.2504.07642
[39] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. 2010. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, New York, NY, 61–72.
[40] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. 2006. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the 2006 ACM Workshop on Wireless Security*. ACM, 85–94.
[41] Qiao Yan, Wenyao Huang, Xupeng Luo, Qingxiang Gong, and F. Richard Yu. 2018. A multi-level ddos mitigation framework for the industrial internet of things. *IEEE Communications Magazine* 56, 2 (2018), 30–36.
[42] Koen Zandberg, Kaspar Schleiser, Francisco Acosta, Hannes Tschofenig, and Emmanuel Baccelli. Secure firmware updates for constrained IoT devices using open standards: A reality check. *IEEE Access* 7 (2019), 71907–71920.

Received 16 November 2024; revised 18 May 2025; accepted 17 July 2025