HAL open science

Automated Security Analysis of IoT Software Updates

Nicolas Dejon, Davide Caputo, Luca Verderame, Alessandro Armando, Alessio Merlo

To cite this version:

Nicolas Dejon, Davide Caputo, Luca Verderame, Alessandro Armando, Alessio Merlo. Automated Security Analysis of IoT Software Updates. 13th IFIP International Conference on Information Security Theory and Practice (WISTP), Dec 2019, Paris, France. pp.223-239, 10.1007/978-3-030-41702-4_14 . hal-03173903

HAL Id: hal-03173903

https://inria.hal.science/hal-03173903v1

Submitted on 18 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Automated Security Analysis of IoT Software Updates*

Nicolas Dejon[1,2][0000−0002−2802−0922], Davide Caputo[1][0000−0002−5408−4735], Luca Verderame[1][0000−0001−7155−7429], Alessandro Armando[1][0000−0002−5246−2157], and Alessio Merlo[1][0000−0002−2272−2376]

[1] DIBRIS - University of Genova, Italy
{name.surname}@unige.it
[2] University of Technology of Compiègne, France
{name.surname}@etu.utc.fr

Abstract. IoT devices often operate unsupervised in ever-changing environments for several years. Therefore, they need to be updated on a regular basis. Current approaches for software updates on IoT, like the recent SUIT proposal, focus on granting integrity and confidentiality but do not analyze the content of the software update, especially the IoT application which is deployed to IoT devices. To this aim, in this paper, we present IoTAV, an automated software analysis framework for systematically verifying the security of the applications contained in software updates w.r.t. a given security policy. Our proposal can be adopted transparently by current IoT software updates workflows. We prove the viability of IoTAV by testing our methodology on a set of actual RIOT OS applications. Experimental results indicate that the approach is viable in terms of both reliability and performance, leading to the identification of 26 security policy violations in 31 real-world RIOT applications.

Keywords: IoT applications · Software Updates · SUIT · model checking · security policy.

1 Introduction

The Internet of Things (IoT) is spreading into diverse application domains at an unstoppable pace: homes, hospitals, means of transportation, manufacturing -just to cite some- are all being affected by the coming of the IoT, and will significantly benefit from its adoption. IoT devices collect, exchange, and process data to support the dynamic and possibly even autonomous adaptation to new and/or evolving contexts. Due to changing requirements, the functionalities required by a device at deployment time is very likely to change in the future. The software stack of IoT devices, consisting of bootloader, operating system, and application(s), will need frequent updates for a number of reasons: to offer

* This work was partially funded by the Horizon 2020 project "Strategic Programs for Advanced Research and Technology in Europe"(SPARTA).

additional functionalities, to support new communication protocols, and/or to patch software bugs (including security vulnerabilities).

Securing the IoT software update process is key to the security of the IoT. To this end, the IoT ecosystem must be provided with the means to ensure the integrity of the software updates, i.e., that the updated software has not been tampered with by a malicious agent. The IETF IoT group is addressing the problem through the development of a new standard, Software Updates for Internet of Things (SUIT) [19], for the software update process of IoT devices. In SUIT, an IoT Software Maintainer (ISM) creates an update bundle, i.e., the firmware image (composed of an operating system and an application) holding the core logic of the IoT device. Then, the ISM uploads the updates to a distribution server, the Update Server (US), that dispatches the update to the devices using over-the-air (OTA) or wire technologies. The SUIT workflow has been designed to enforce the integrity and the confidentiality of the software update, thus providing end-to-end security between the author of the update (i.e., the ISM) and the device, even if an untrusted US mediates the process. This ensures a form of end-to-end security between the (trusted) ISM and the devices.

Unfortunately, even when a mechanism such as SUIT is in place and ensures the integrity of the software updates, there are no guarantees on the content of the update. This shortfall implies that an ISM may introduce, wittingly or not, an insecure software component that can compromise the security of the updated device. For example, the Zigbee Worm [25], triggered using a malicious firmware update, allowed the attackers to get full control over Philips Hue Smart Lamps.

In this paper, we present the *IoT Application Verification (IoTAV) Framework*, a novel analysis methodology that supports the automatic verification of security properties in applications running on IoT devices. Given an IoT device application in an executable format and a set of security properties, the framework tries to determine if the app meets the expected security properties. This is done by automatically (i) extracting the IoT app from the firmware image (without the need of source code), (ii) building a formal (i.e., mathematical) model of the app, and (iii) automatically evaluating a set of security properties (i.e., a security policy) by leveraging state-of-the-art model checking techniques. The framework enables the definition of security policies directly by the ISM or by trusted third-party entities, e.g., the network operator or the IoT device manufacturer.

IoTAV can be applied to both new and previously deployed devices. Moreover, it does not require the source code and, therefore, can be applied to third party applications whose source code is not available. As we will see later (cf. Sect. 3.5) it is almost independent of the hardware that will host and run the application. Finally, the framework complements and leverages current firmware updates workflows, including the new SUIT solution.

To demonstrate the effectiveness of the proposed solution, we developed a prototype implementation of IoTAV for the SUIT update process in the RIOT

ecosystem [24]. Finally, we validated the prototype against a set of 31 real-world RIOT applications, thereby identifying 26 security policy violations.

Paper organization. The rest of the paper is structured as follows. Section 2 introduces the major concepts of the IoT software update process and then details the SUIT standard, along with its security limitations. Section 3 describes our novel IoT Application Verification Framework for the automatic analysis of the applications contained in the IoT updates. Furthermore, it provides the specifications of a prototype implementation for the RIOT ecosystem. Section 4 discusses an assessment of IoTAV against real-world RIOT applications and presents the collected results along with a discussion on the current limitations of the approach. Section 5 analyzes the state-of-the-art proposals for securing IoT software updates and for analyzing IoT apps, thereby underlying the differences w.r.t. our approach. Finally, Section 6 provides some concluding remarks.

2 Software Updates for IoT Devices

The IoT software update process is an essential operation for maintaining a suitable level of efficiency and security of IoT devices. Over the last few years, the research community has been working on the definition of several IoT update processes [20], among which the software update for resource-constrained devices is still an open research challenge [1]. Resource-constrained devices, as specified in RFC 7228 [7], use microcontrollers (like the Arm Cortex-M) on which they run a real-time operating system such as Contiki, FreeRTOS or RIOT [14], just to cite a few. To this aim, several firmware update solutions have been proposed in the last years, like FOSE [11], The Update Framework (TUF)[3], and Uptane [21]. However, most of the proposed mechanisms are tied to specific operating systems or hardware architectures, and thus, they are not general-purpose.

To overcome such limitations, the Internet Engineering Task Force (IETF) is defining a standard for firmware updates called Software Updates for Internet of Things (SUIT) [19]. The main goals of SUIT are interoperability (w.r.t. the platform and the firmware distribution technology) and end-to-end security.

The SUIT standard, currently in draft status, includes a definition of the firmware update architecture [17], an information model [18], and a manifest description [16]. Hereafter, we define the firmware image as a binary file that contains the complete software stack of an IoT device (i.e., the OS and the IoT application), according to the terminology adopted by IETF [17]. The update process involves the IoT devices to be updated, the IoT software maintainer, and the Firmware Update Server, as sketched in Fig. 1.

The typical firmware update procedure works as follows: an IoT software Maintainer compiles the OS and the IoT app and generates a new firmware image. In the SUIT specification, firmware images comprise a manifest file that embeds information such as the location of the firmware image for delivery,

[3] `https://github.com/theupdateframework/tuf`

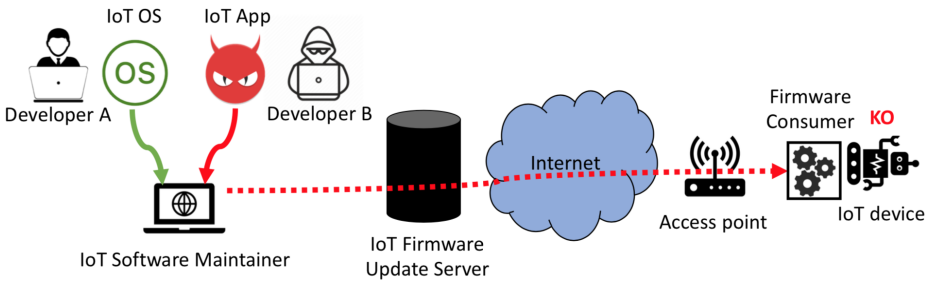

Fig. 1. A SUIT update scenario (inspired by [29]) where a developer is able to introduce a malicious app in the update pipeline.

dependencies, cryptographic information, and device data. Both the firmware and the manifest are then published onto the IoT Firmware Update Server, which is responsible for storing the update and notifying the IoT devices about the availability of a new update. On the device side, the firmware update is handled by a firmware update module named Firmware Consumer, which retrieves both the manifest and the firmware image.

Upon receiving a notification from the IoT Firmware Update Server, the Firmware Consumer retrieves the manifest, checks the digital signature and the firmware sequence number to ensure the integrity and the freshness of the update image. If the verification succeeds, the IoT device pulls the firmware from the URI provided in the SUIT manifest, and stores the firmware image on the flash memory. The flash memory is divided into several memory regions (slots) containing (i) the bootloader and (ii) two slots, one containing the current firmware and the other is reserved for the update firmware. After the writing process, the bootloader reads the metadata from the firmware slots and chooses to boot the newest valid firmware. Using such an approach, an interruption in the update process (e.g., due to power loss) cannot cause the system to boot an invalid, corrupted or incompletely received image [29].

2.1 Security Issues in SUIT

The SUIT information model [18] defines a collection of security threats for the update process. As discussed in [29], such threats can be categorized into: (i) tampered firmware, (ii) firmware replay, (iii) offline device attack, (iv) firmware mismatch, (v) flash memory location mismatch, (vi) unexpected precursor image, (vii) reverse engineering, and (viii) resource exhaustion. Although the SUIT model suggests a set of security requirements and countermeasures, it is worth noticing that all these threats are related to the integrity and the confidentiality of the update process only, while the content of the update is inherently assumed as trusted. Therefore, the SUIT workflow allows an ISM to upload a firmware image containing security vulnerabilities or malicious behaviors. Furthermore, SUIT allows the ISM to transfer its authority to another entity, e.g., a third-

party developer, that can deliver to the ISM some components of a software update (e.g., the executable of the application to be updated) or triggers the update process directly. In this case, the ISM has no mechanism to assess the content of the external software components, and must fully trust the external entity.

For instance, consider the scenario depicted in Fig. 1. The ISM delegates two external developers (i.e., A and B) for updating the OS and the IoT application, respectively. Let us assume that developer A is honest (i.e., she dispatches a benign and reliable OS image), while developer B is malicious (i.e., she introduces a malware IoT application). According to the SUIT workflow, the ISM blindly composes the firmware image and dispatches the update to the Firmware Server. Then, the IoT device only verifies the authenticity and the integrity of the firmware image and installs the malware update. Such scenario depicts an actual and widespread attack vector, as the Philips Hue smart lamps security incident [25] and the Jeep Cherokee hack[4] have been carried out by injecting malicious software components inside the update process, without triggering any security enforcement mechanism.

To reduce the impact of unreliable updates, we argue that the SUIT update process needs to rely on a methodology to assess the security of the firmware image and in particular, of the IoT application. Such a methodology must be able to automatically evaluate the behavior of the firmware according to a set of security requirements, in order to allow the same ISM to deliver only validated and certified software updates. The security requirements can be defined directly by the same ISM, the IoT device manufacturer, or by a trusted third-party entity involved in the update process, like a Network Operator or a Device Operator, as defined in the SUIT standard.

We also argue that the methodology should work as a black box (i.e., without requiring the source code), in order to be systematically applied to any executable provided by third-parties. Finally, we argue that the analysis process must be carried out on the firmware image before it is submitted to the SUIT pipeline, in order to leverage the security mechanisms provided by SUIT to prevent any further modification of the image.

3 The IoT Application Verification Framework

In order to mitigate the aforementioned security concerns, we propose a novel verification solution called the **IoT Application Verification Framework** (IoTAV). IoTAV allows to automatically evaluate the security of the IoT applications included in firmware images in a black-box fashion. In details, IoTAV enables the definition of a set of security requirements, codified as a *security policy*, that are then automatically evaluated on the application executable using state-of-the-art model checking techniques. As depicted in Fig. 2, IoTAV can be seamlessly included in the existing update pipeline, like the one defined in SUIT.

[4] `https://www.kaspersky.com/blog/blackhat-jeep-cherokee-hack-explained/9493/`

IoTAV is able to detect malicious updates (dashed arrows), thereby discarding those that doe n comply with the security policy and notifying the ISM, without affecting the normal operation in case of secure updates (solid arrows).

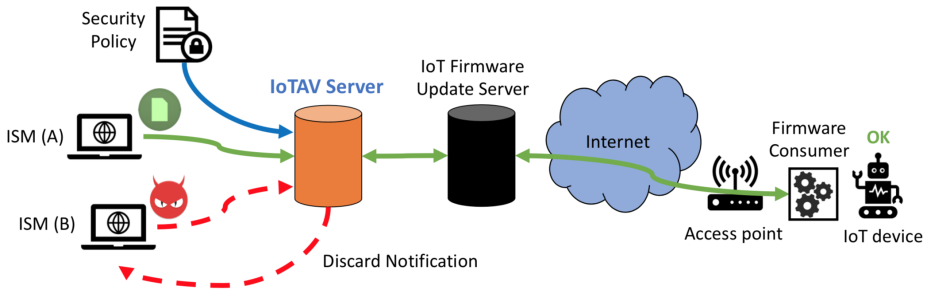

Fig. 2. SUIT Update process with IoTAV. The IoTAV Server ensures that the IoT app bundled in the software update is compliant with the security policy.

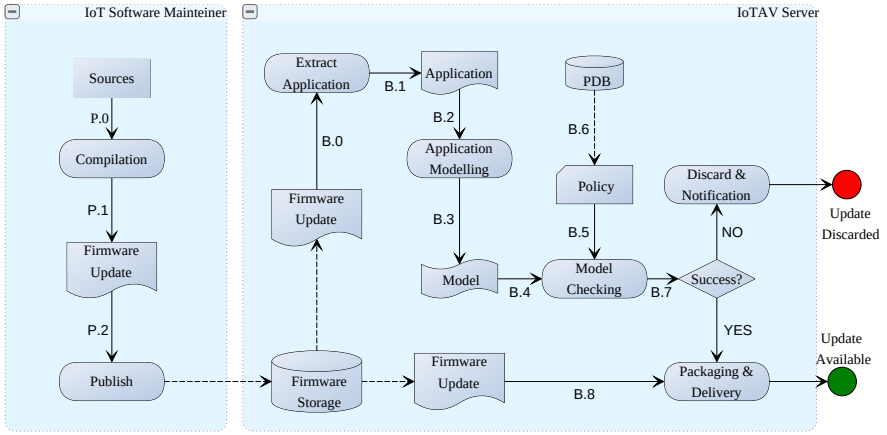

Fig. 3. The IoTAV verification workflow.

3.1 Formal Security Assessment Workflow

IoTAV features are granted by the workflow depicted in Fig. 3. Initially, the ISM compiles (P.0) and generates the firmware image update (P.1). Then, she publishes the executable (P.2) to the IoTAV Server, which stores the firmware update

in a database (Firmware Storage). Then, an application extraction procedure is applied to the firmware image (B.0) to extract only the application part (B.1). After that, the application executable goes through a modeling phase (B.2) that outputs the corresponding application model (B.3). Hence, the model is passed (B.4) to a verification process that checks its compliance against the security policy (B.5). Security policies are retrieved (B.6) from a policy database (PDB) handling policy instances that can be customized over the configuration of the single device. If the verification succeeds (B.7 → YES), IoTAV server executes a packaging procedure for the firmware (B.8). Finally, it bundles the system update following the regular SUIT publish procedure and notifies the IoT Firmware Update Server. Otherwise (B.7 → NO), IoTAV Server notifies the IoT Software Maintainer and discards the update. The notification contains the results of the verification process (i.e., which parts of the model violates the security policy and why).

3.2 Application Extraction and Modeling

The first step of the IoT app verification process is the *model generation*. To do that, IoTAV parses the firmware image and identifies the part of the executable related to the application logic. After that, the Application Modeling lifts the application machine code to a higher-level language, usually an Intermediate Representation (IR), by relying on a disassembler. From the IR, the service can then deduce the structure of the application program. The IoT Application Verification Framework builds a complete application model through a fruitful combination of Control Flow Graphs (CFGs), Call Graphs (CGs), and Interprocedural Control Flow Graphs (ICFGs). A CFG is a directed graph made by nodes representing basic blocks, e.g., pieces of branch-less code, chained through edges to represent the control flow transfer. Although CFGs are widely used to model all the possible execution paths of a function call [28], they are able to represent the control-flow of a single procedure only. To overcome such limitation, IoTAV model generation procedure combines the CFGs of each procedure with the calling relationship between them, through CGs, thereby obtaining the ICFG of the whole application.

Algorithm 1 shows the pseudo-code of the ICFG construction algorithm. Each call site node is a root node of its own (local procedure) CFG. It is referenced in other computed CFGs since its procedure is called by the other ones. As a consequence, the app_icfg contains a list of the CFGs rooted at the $start_node$ and at each $callee_node$, each related to another one by reference. Hence, by constructing the ICFG from an executable, IoTAV gets the structure of the entire application.

Nevertheless, since the IoT Application Verification Framework aims to describe the behavior of a system from a security standpoint, any operation which is not security-relevant is abstracted away. Therefore, the Application Modeling block records only security-relevant operations defined in the security policies, notably, file operations, cryptographic primitives, and network procedures. All irrelevant API (Application Programming Interface) calls that are invoked in a

Algorithm 1 Compute the ICFG from local CFGs

```
procedure ICFG CONSTRUCTION
    app_icfg ← []
    start_node ← get the entry node
    callees_nodes ← get callees and callees of callees
    add local CFG of start_node to app_icfg
    for callee_node ∈ callees_nodes do
        add local CFG of callee_node to app_icfg
    end for
    return app_icfg
end procedure
```

sequential way or in a conditional way (branches) are grouped and then pruned. This way, IoTAV optimizes the application model for the model checking phase.

3.3 Policy specification

A policy describes the properties that must hold in the model, while properties mirror a system description that can be formally expressed. In detail, the IoT Application Verification Framework allows the definition of security properties that need to be enforced in the IoT application once it is encapsulated inside the firmware update. Following the same approach of [3] and [2], IoTAV enables the definition of security policies on the interaction between the IoT application and the underlying OS in terms of API calls. Since the ICFG extracted from the application can be interpreted as a state graph, IoTAV uses Temporal Logic formulas, namely Linear Temporal Logic (LTL) [12] and Computational Tree Logic (CTL) [13]. An LTL formula describes a pattern for a sequence of events. Any actual sequence of events may match or not the pattern. Hence, one can express properties about the sequence of events with temporal operators. For example, from a given state, $\mathbf{F}p$ ("eventually") means the property p will *eventually* hold at some point in the future, while $\mathbf{G}p$ ("globally") means that the property p *always* holds in the future.

Instead, the Computation Tree Logic is based on a branching notion of time, meaning that its model of time is a tree-like structure in which the future is not determined. CTL considers different paths in the future, any one of which might be an actual path that occurs. Indeed, such a notion of time can represent the possible execution of a software program. In order to express if a property holds for all paths or some of them, two quantifier operators are introduced: the **A** operator ("for all paths") and the **E** operator ("there exists a path").

For example, the formula "$\mathbf{AG}p$" states that the property p should hold at each state of any path, whereas the formula "$\mathbf{EG}p$" states that there exists a path where the property p always holds (and eventually some paths that never hold property p).

Finally, IoTAV policies enable the definition of security properties in terms of a logical expression to be evaluated, as in the following example.

```
"never_fread" : A [G FRD=0];
```

Here, the sample policy bans any use of the *fread* C function. It states that along any path from the initial state, no state should set the *FRD* variable. In other words, the variable representing the *fread* function should never be part of the application model.

3.4 Model checking

In order to verify the policy compliance of the IoT application, the IoT Application Verification Framework leverages model checking techniques that have been successfully applied to numerous real-world problems. Model checking can be mapped to a reachability problem, i.e., checking whether the model of the application cannot reach an undesirable state. Applying Temporal Logic policies to the model allows to verify some properties at any time (or state of the system). The model checking process ends up with a compliance result, that states whether the security policy is satisfied by the application model.

In order to prevent unbounded computations that are unacceptable in IoTAV workflow, the Model Checking module includes a timeout mechanism. Thus, the model checker produces three possible results: (i) YES — the model complies with the policy; (ii) NO - the model violates the security policy; or (iii) TIMEOUT (TO) — the time threshold has been reached.

One of the most critical issues in model checking is the so-called *state explosion problem* [10]. In order to check some properties, the model checker needs to explore the entire state space, which increases the complexity as the number of states grows large. Our security model addresses this problem by reducing the analysis to the sole security-sensitive operations, thus limiting the size of the corresponding model.

3.5 IoTAV Implementation

In order to evaluate the feasibility and effectiveness of the IoT Application Verification Framework, we developed a prototype implementation of IoTAV as a server appliance compatible with the SUIT update process for a RIOT ecosystem. It is worth noticing that, although the IoT Application Verification Framework is compatible with a generic SUIT update process, the focus of this prototype is the compliance with the current RIOT implementation.

SUIT in the RIOT Ecosystem. RIOT [5] is an open-source OS, based on a modular architecture built around a soft real-time micro-kernel. RIOT is structured in software modules that are aggregated at compile-time, around a kernel providing core functionality like process scheduling, inter-process communication, and threading. This approach allows building the complete system in a modular manner, including only modules that are required by the application at stake. One of these modules is the `application` module, which contains the IoT application.

RIOT implements the SUIT update described in Section 2. The RIOT update firmware is a bundle that contains both the OS and the IoT application in a single Executable and Link Format (ELF) file. The IoT Software Maintainer can (i) build the update, (ii) generate the corresponding manifest file, and (iii) push them to the IoT Firmware Update Server, by using a suit/publish command[5]. On the device side, after the board boots the new firmware, RIOT starts two threads: the idle thread and the main thread. The main thread is the first thread that runs and calls the `main` function. This function needs to be defined by the user application.

Application Modeling. Concerning static code analysis, the angr framework[6] is one of the most popular frameworks used in top-ranked teams of the DARPA Cyber Grand Challenge. It is a python-based binary analysis framework that currently supports the most common architectures, including x86, ARM, MIPS, and AMD, and it allows to retrieve the CFG of a program from its executable. IoTAV uses this tool to extract the CFG of each program procedure and to compute the overall ICFG. Since the entry point of the user application is the `main` function, the IoTAV computes the ICFG from there, which lets the analysis focus only on the application. For this, angr can be configured to begin the ICFG recovery directly from the `main` symbol in the program.

Policy Specification and Model Checking. PRISM (Probabilistic Symbolic Model Checker) [22] is one of the many existing model checkers. It is free, open-source, and it analyzes complex systems according to probabilistic behaviors. It also supports the model checking of non-probabilistic properties using LTL and CTL. The latter capability supports the definition of the IoTAV security policies. Indeed, the properties we would like to check with PRISM are the use of APIs and the call ordering in all execution paths. We mainly focus on APIs with a security meaning, because they are the only ones relevant in a security policy, notably any file operation or crypto primitive.

With non-probabilistic expressions, PRISM can also generate counterexamples and witnesses for further investigation on a failed property verification. Such a feature allows manual investigation as a post-process to determine the reasons that caused the policy check to fail.

Hence, the IoTAV Server embeds PRISM for the model checking phase, thereby adopting security policies in LTL and CTL. However, since the PRISM model checker needs to be fed with a model in its own PRISM language, we added a conversion block from the recovered ICFG to the PRISM language. PRISM can then compare these policies to the application model.

4 Experimental Evaluation

We carried out an experimental evaluation of IoTAV to prove the viability of our proposal and evaluate the impact on the SUIT update process. The expe-

[5] `https://github.com/RIOT-OS/RIOT/tree/master/examples/suit_update`
[6] `https://github.com/angr/angr`

rimental scenario is composed by an ISM which deploys updates verified by an IoTAV in an IoT ecosystem made of RIOT-based devices. More specifically, the experimental setup is defined as follows.

IoT applications. We took into consideration two sets of RIOT applications. The first one is composed of 21 RIOT sample applications available on the official repository[7], while the latter is made by 10 RIOT applications used for a demo dashboard[8] use case by the RIOT Development Team. In particular, the latter set contains a series of collecting nodes of environmental data (e.g., temperature, humidity, and pressure) that rely on CoAP [26] and MQTT [15] protocols to send their data to a real-time visualization dashboard.

SUIT Setup. We setup a standard SUIT environment composed by a Firmware Update Server connected over-the-air to a SAMR21 Xplained Pro evaluation board[9] equipped with RIOT OS Release 2019.07. Then, we deployed an IoTAV Server, according to the scenario depicted in Fig. 2. Finally, we simulated the ISM, thereby producing a set of firmware images for the update that are then pushed to the IoTAV Server to trigger the SUIT update process. In detail, each of the application under test is bundled with the OS on the evaluation board (i.e. RIOT OS Release 2019.07) to build the corresponding firmware image. Both the Firmware Update Server and IoTAV Server executes on two entry-level PCs equipped with Ubuntu 18.04.2 LTS, Intel Pentium (R) P6200 @2.13GHz * 2, 4 GB of RAM and 50GB HDD.

Security Policies. We defined a set of security policies describing three of the OWASP IoT Top 10 Vulnerabilities 2018[10]. The first two enforce the confidentiality of (i) the data transfer using the MQTT protocol and of (ii) the local file storage, as recommended in the *"Insecure Data Transfer and Storage"* - #7 OWASP Risk. The third policy enforces the exclusion of insecure or deprecated C functions in IoT apps, as suggested in the *"Use of Insecure or Outdated Components"* - #5 OWASP Risk.

Hence, we defined the following three PRISM policy expressions:

```
"mqtt_enc" : A [G MQPB=1 => (CPH_ENC=1 | AES_ENC=1 | CHA_ENC
    =1 | CHA_POLY_ENC=1)];
"st_enc" : A [G (FPRNT=1 | FWRT=1 | FPTS=1 | FPTC=1) => (
    CPH_ENC=1 | AES_ENC=1 | CHA_ENC=1 | CHA_POLY_ENC=1)];
"uns_c" : A [G SCPY=0 & SNCPY=0 & SCT=0 & SNCT=0 & SPRNT=0 &
    VSPRNT=0 & GTS=0 & MKPTH=0 & SPTH=0 & SCF=0 & SSCF=0 &
    SNSCF=0 & ATI=0 & ATF=0 & ATL=0];
```

[7] https://github.com/RIOT-OS/RIOT/
[8] http://riot-demo.inria.fr
[9] https://www.microchip.com/DevelopmentTools/ProductDetails/ATSAMR21-XPRO
[10] https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project

The first expression (`mqtt_enc`) ensures that the data are encrypted when sent through MQTT using one of the following cypher algorithms `cipher_encrypt`, `aes_encrypt`, `chacha_encrypt_bytes` or `chacha20poly1305_encrypt`. Instead, the second expression (`st_enc`) grants that data are encrypted whenever they are written on a file through `fprintf`, `fwrite`, `fputs` or `fputc` C functions.

To represent both `mqtt_enc` and `st_enc`, we relied on the "=>" (implication) operator in PRISM which states that when the left side condition is satisfied, the right side should be satisfied as well.

Finally, the `uns_C` policy verifies that none of the insecure C functions (`strcpy`, `strncpy`, `strcat`, `strncat`, `sprintf`, `vsprintf`, `gets`, `makepath`, `_splitpath`, `scanf`, `scanf`, `sscanf`, `snscanf`, `atoi`, `atof`, `atol`) is used in the application.

4.1 Experimental Results

Table 1 summarizes the analysis results on the entire dataset. For each of the analyzed applications, we provide general details (name and **S**ize of the executable), the execution times (**M**odeling time, **V**erification time, and **T**otal time), and the results of the verification on the three policies. Since our analysis unveiled some vulnerabilities in the RIOT applications, we reported our findings to the RIOT Development Team.

Table 1. Execution times and policy verification results.

		Time			***Policies***		
Applications	*S [kB]*	*M [s]*	*V [s]*	*Tot [s]*	*uns_c*	*mqtt_enc*	*st_enc*
`asymcute_mqttsn`	249.9	165.0	76.0	243.0	✗	✓	✓
`bindist`	33.9	31.0	7.0	38.0	✗	✓	✓
`ccn-lite-relay`	335.2	169.0	138.0	308.0	✗	✓	✓
`cord_ep`	255.0	161.0	84.0	247.0	✗	✓	✓
`cord_epsim`	195.9	337.0	140.0	479.0	✗	✓	✓
`dashboard_riot_a8_m3`	2400.0	1517.0	133.0	1652.0	✗	✓	✓
`default`	72.7	53.0	13.0	66.0	✗	✓	✓
`emcute_mqttsn`	235.8	476.0	133.0	611.0	✗	✗	✓
`filesystem`	107.1	100.0	18.0	118.0	✗	✓	✓
`gcoap_example`	249.9	427.0	171.0	600.0	✗	✓	✓
`gnrc_minimal`	157.6	195.0	59.0	255.0	✗	✓	✓
`gnrc_networking`	282.3	606.0	185.0	792.0	✗	✓	✓
`gnrc_tftp_example`	286.6	638.0	208.0	848.0	✗	✓	✓
`hello_world`	33.9	29.0	6.0	35.0	✗	✓	✓
`ipc_pingpong`	38.4	31.0	8.0	39.0	✗	✓	✓
`lua_basic`	335.3	5003.0	1658.0	6668.0	✗	✓	✗
`lua_repl`	339.6	6099.0	1688.0	7794.0	✗	✓	✗
`nanocoap_server`	184.1	312.0	123.0	436.0	✗	✓	✓
`ndn_ping`	181.5	147.0	136.0	284.0	✗	✓	✓
`node_bmp180`	3500.0	N.A.	N.A.	N.A.	N.A	N.A	N.A.
`node_bmx280`	3500.0	201.0	43.0	245.0	✓	✓	✓
`node_ccs811`	3500.0	N.A.	N.A.	N.A.	N.A	N.A	N.A.
`node_empty`	3400.0	137.0	29.0	167.0	✓	✓	✓
`node_imu`	2600.0	N.A.	N.A.	N.A.	N.A	N.A	N.A.
`node_io1_xplained`	3400.0	117.0	27.0	144.0	✓	✓	✓
`node_leds`	3400.0	112.0	27.0	140.0	✓	✓	✓
`node_mqtt_bmx280`	3400.0	110.0	34.0	144.0	✓	✗	✓
`node_tsl2561`	3500.0	179.0	35.0	214.0	✓	✓	✓
`posix_sockets_example`	240.3	664.0	195.0	861.0	✗	✓	✓
`saul_example`	49.3	39.0	10.0	49.0	✗	✓	✓
`timer_periodic_wu`	43.8	37.0	11.0	48.0	✗	✓	✓

Policy Verification Results. IoTAV was able to successfully analyze 28 out of the 31 IoT applications. Angr failed when trying to analyze the remaining three applications; as a consequence, IoTAV was not able to extract the model.

The outcome of IoTAV verification process showed that the *node_mqtt_-bmx280* application (belonging to the demo dashboard use case) does not comply with the `mqtt_enc` property, thereby indicating that the MQTT communication is unencrypted, and thus, the data are transmitted insecurely through the network. Since the source code of the application is available on Github[11], we both inspected the source code and tested the application to validate our findings. The manual analysis confirmed that data are published to an MQTT broker unencrypted. Besides, we were able to execute the node on the evaluation board. We successfully intercepted the plaintext data traffic sent by the application through the *tcpdump* tool. Also, the *emcute_mqttsn* application failed the `mqtt_enc` property as well.
Furthermore, IoTAV discovered that *lua_basic* and *lua_repl* applications do not comply with the `st_enc` policy, since they include some file storage operations without the adoption of any encryption support in place.
Finally, the experimental results show that the 71% of the dataset (22 out of 31) violate the `uns_c`, and thus adopting insecure or deprecated C primitives.

Notes on Performance. IoTAV successfully evaluated the applications of the dataset with a mean processing time of 191.2 seconds. The modeling generation phase takes on average 80% of the total processing time, while the model checking phase takes, on average, 20% of that time. The simplification and conversion phases have negligible impact on the global performance.

4.2 Limitations

The experimental results show both the effectiveness and the applicability of IoTAV in the SUIT update workflow, although its adoption comes with some restrictions. First, the evaluation techniques applied by IoTAV on the firmware image work with unstripped executables only, i.e., binaries containing symbols. Nevertheless, to the best of our knowledge, no tool is still able to extract CFGs without any available symbols that could otherwise be used in combination with our model extractor. Therefore, to overcome this limitation, we propose to add the possibility to strip the firmware only after the policy verification step. To this aim, the SUIT process for RIOT applications is still under active discussion and could eventually include this feature. In addition to that, the static evaluation of security policies may not cover all possible use cases for an IoT application. For example, if one security property requires to detect whether a file has been closed after being opened, the variable monitoring this property will still be set even if the file is later reopened, thereby potentially affecting the results of the analysis. To mitigate such issues, we are investigating the introduction of a runtime monitoring technique, by extending the approach in [4].

[11] `https://github.com/future-proof-iot/riot-firmwares/tree/master/apps`

Finally, applications have been manually verified afterwards, with no false positives. However, we noticed that some applications are not expected to verify the policies, albeit the verification step succeeds. For example, this is the case of *asymcute_mqttsn*, an asynchronous MQTT-SN implementation, marked as meeting the `mqtt_enc` policy even if no encryption is used for the published data. This is due to the fact that the API *asymcute_pub* is not listed among the relevant APIs in the security policy. Such a result underlines how crucial is the definition of appropriate security policies to be used in the IoTAV to avoid false negatives.

5 Related Work

The increasing number of vulnerabilities found in IoT devices have raised the need for reliable methodologies for securing firmware updates. To this aim, the scientific and industrial communities have proposed different solutions. In [29], Zanberg et al. survey open standards and open source libraries that provide useful building blocks for secure firmware updates for resource-constrained IoT devices. The authors propose the design and the implementation of a prototype that leverages these building blocks. Bettayeb et al. [6] discuss security threats against firmware update for IoT devices and all available secure firmware update methods for IoT devices in the literature, like [20]. However, all of these works are focused only on providing end-to-end security between the IoT Firmware Update Server and the device, but they do not deal with the analysis of the IoT application.

On this topic, some proposals for static and dynamic analysis of IoT applications have been already put forward. Soteria [8] and IotSan [23] are static analysis systems that automatically extract a model of an IoT application and use a model checker to validate application-specific properties. However, they require the source code of the application. On the dynamic side, IoTGuard [9] is a policy-based enforcement system that monitors the execution of IoT applications. IoTGuard requires to instrument the source code to collect application data at runtime and build up a dynamic model that represents the runtime behavior of the application. The limitation of this approach is its invasiveness as well as the need to modify the business logic of the application. Previous solutions focus only on a single application, while SIoT [27] is the first tool that analyzes distributed IoT applications to detect buffer overflow attacks. The authors' idea is to look at a distributed IoT system as a single monolithic application.

Our proposal extends the current state of the art by allowing us to systematically verify the compliance of the binary code of IoT applications w.r.t. user-defined security policies without the need to modify applications.

6 Conclusion

In this paper, we introduced a novel methodology, called *IoT Application Verification Framework* (IoTAV), for the systematic assessment of IoT applications

w.r.t. a set of given security properties. We applied such a methodology to the assessment of software updates in the IoT ecosystem. We proved the viability of our proposal experimentally by carrying out automatic analyses of RIOT applications on an actual deployment based on the SUIT update pipeline. The results yielded the identification of 26 security policy violations in 31 real-world RIOT applications.

As future work, we will deal with the limitations described in Section 4, at first. Then, the next step of our research will be to test the methodology on other IoT architectures, OSes and firmware update workflows. Finally, although we defined a set of policies based on the OWASP IoT Top 10 security risks, we argue that novel and more comprehensive security policies should be investigated and defined. To this aim, the interaction among IoT developers, network operators, and device manufacturers could lead to the definition of more sophisticated and widely-accepted security policies.

References

1. Acosta Padilla, F.J., Baccelli, E., Eichinger, T., K., S.: The Future of IoT Software Must be Updated. IAB Workshop on Internet of Things Software Update (IoTSU) (2016)
2. Armando, A., Costa, G., Merlo, A., Verderame, L.: Enabling byod through secure meta-market. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless Mobile Networks. pp. 219–230. WiSec '14, ACM, New York, NY, USA (2014)
3. Armando, A., Costa, G., Merlo, A., Verderame, L.: Formal modeling and automatic enforcement of Bring Your Own Device policies. International Journal of Information Security (2015)
4. Armando, A., Costa, G., Verderame, L., Merlo, A.: Securing the" bring your own device" paradigm. Computer (2014)
5. Baccelli, E., Gundogan, C., Hahm, O., Kietzmann, P., Lenders, M.S., Petersen, H., Schleiser, K., Schmidt, T.C., Wahlisch, M.: RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. IEEE Internet of Things Journal (2018)
6. Bettayeb, M., Nasir, Q., Talib, M.A.: Firmware Update Attacks and Security for IoT Devices. In: Proceedings of the ArabWIC 6th Annual International Conference Research Track. ArabWIC 2019, ACM (2019)
7. Bormann, C., Ersue, M., Keranen, A.: Terminology for constrained-node networks. Internet Engineering Task Force (IETF): Fremont, CA, USA (2014)
8. Celik, Z.B., McDaniel, P., Tan, G.: Soteria: Automated IoT Safety and Security Analysis. Proceedings of the 2018 USENIX Annual Technical Conference (2018)
9. Celik, Z.B., Tan, G., McDaniel, P.: IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. Network and Distributed Systems Security (NDSS) Symposium 2019 (2019)
10. Clarke, E.M., Klieber, W.: Model Checking and the State Explosion Problem. Tech. rep. (2011)
11. Doddapaneni, K., Lakkundi, R., Rao, S., Kulkarni, S., Bhat, B.: Secure fota object for iot. In: 2017 IEEE 42nd Conference on Local Computer Networks Workshops (LCN Workshops) (2017). https://doi.org/10.1109/LCN.Workshops.2017.78

12. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple On-the-fly Automatic Verification of Linear Temporal Logic. Springer US (1996)
13. Goldblatt, R.: Logics of time and computation. Center for the Study of Language and Information Stanford (1992)
14. Hahm, O., Baccelli, E., Petersen, H., Tsiftes, N.: Operating Systems for Low-End Devices in the Internet of Things: A Survey. IEEE Internet of Things Journal
15. Hunkeler, U., Truong, H.L., Stanford-Clark, A.: Mqtt-s — a publish/subscribe protocol for wireless sensor networks. In: 2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08) (2008)
16. IETF: Firmware manifest format. `https://tools.ietf.org/html/draft-moran-suit-manifest-01` (2019), [Online; accessed September 11, 2019]
17. IETF: Website of : A firmware update architecture for internet of things devices draft-ietf-suit-architecture-06. `https://tools.ietf.org/pdf/draft-ietf-suit-architecture-06.pdf` (2019), [Online; accessed September 11, 2019]
18. IETF: Website of : Firmware updates for internet of things devices - an information model for manifests draft-ietf-suit-information-model-03. `https://tools.ietf.org/html/draft-ietf-suit-information-model-03` (2019), [Online; accessed September 11, 2019]
19. IETF: Website of : Ietf suit draft architecture. `https://tools.ietf.org/html/draft-ietf-suit-architecture` (2019), [Online; accessed August 7, 2019]
20. Kolehmainen, A.: Secure firmware updates for iot: A survey. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) (2018)
21. Kuppusamy, T.K., DeLong, L.A., Cappos, J.: Uptane: Security and Customizability of Software Updates for Vehicles. IEEE Vehicular Technology Magazine (2018). https://doi.org/10.1109/MVT.2017.2778751
22. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic realtime systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806. Springer (2011)
23. Nguyen, D.T., Song, C., Qian, Z., Krishnamurthy, S.V., Colbert, E.J., McDaniel, P.: IotSan: Fortifying the safety of IoT systems. CoNEXT 2018 - Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (2018)
24. RIOT: Riot-os. `https://www.riot-os.org` (2019), [Online; accessed August 7, 2019]
25. Ronen, E., Shamir, A., Weingarten, A.O., Oflynn, C.: IoT Goes Nuclear: Creating a Zigbee Chain Reaction. IEEE Security and Privacy (2018)
26. Shelby, Z., Castellani, A.P., Bormann, C.: Coap: An application protocol for billions of tiny internet nodes. IEEE Internet Computing (2012)
27. Teixeira, F.A., Pereira, F.M., Wong, H.C., Nogueira, J.M., Oliveira, L.B.: SIoT: Securing Internet of Things through distributed systems analysis. Future Generation Computer Systems (2019)
28. Xu, L., Sun, F., Su, Z.: Constructing Precise Control Flow Graphs from Binaries. Tech. rep., University of California (2009)
29. Zandberg, K., Schleiser, K., Acosta, F., Tschofenig, H., Baccelli, E.: Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check. IEEE Access (2019)