

UNIVERSITÉ DE LILLE
FACULTÉ DES SCIENCES ET TECHNOLOGIES

Embedded Systems Security, a posteriori update, a state of the art

Hugo FORRAZ

Master Informatique
Master mention Informatique



DÉPARTEMENT D'INFORMATIQUE
Faculté des Sciences et Technologies

April, 2024

Ce mémoire satisfait partiellement les pré-requis du module de Mémoire de Master, pour la 2^e année du Master mention Informatique.

Candidate: Hugo FORRAZ, No. 41909290,
hugo.forraz.etu@univ-lille.fr

Scientific Guidance: Michaël HAUSPIE, michael.hauspie@univ-lille.fr



DÉPARTEMENT D'INFORMATIQUE
Faculté des Sciences et Technologies
Campus Cité Scientifique, Bât. M3 extension, 59655 Villeneuve-d'Ascq

April, 2024

Abstract

Embedded systems are being increasingly utilized across a wide variety of domains, but the security aspect has been overlooked for a long time.

In embedded systems security, a topic that may have lacked the spotlight is that of **firmware updates** due to their *eclectic* nature. For example, if a connected light bulb starts malfunctioning because of its firmware, the seller might not push an update due to the *lack of adequate resources* or because it may *not be worthwhile monetarily*, as buying a new light bulb is inexpensive. In comparison, if the same issue occurred with a car, it could become significantly more dangerous if the update was not performed, and it would be costly to replace.

This **master thesis** attempts to address the question of firmware updating by conducting an *exhaustive study* on a wide variety of devices and update methods. Whether the devices are on the lower or higher end, if there is a means to update their firmware remotely, they will be considered in this paper.

Keywords: Embedded Systems, Security, Firmware update, LoRA, Updates through Blockchain, FPGA, SoC, OTA, State of the art, Static analysis, Abstract Interpretation

Abstract

Les systèmes embarqués sont de plus en plus utilisés dans une grande variété de domaines, mais l’aspect sécurité en a longtemps été négligé.

En terme de sécurité des systèmes embarqués, un sujet qui n’a peut-être pas bénéficié de l’attention qu’il mérite est celui des **mises à jour de firmware** en raison de la nature *éclectique* de ces systèmes. Par exemple, si une ampoule connectée commence à dysfonctionner à cause d’un problème de firmware, le vendeur pourrait ne pas faire de mise à jour en raison du *manque de ressources adéquates* ou parce qu’elle pourrait *ne pas être rentable financièrement*, étant donné que l’achat d’une nouvelle ampoule est peu coûteux. En comparaison, si le même problème survenait avec une voiture, cela pourrait devenir beaucoup plus dangereux si la mise à jour n’était pas effectuée, et remplacer une voiture n’est pas l’acte le moins coûteux.

Ce **mémoire de master** tente de répondre à la question de la mise à jour du firmware en menant une *étude exhaustive* sur une grande variété de dispositifs et de méthodes de mise à jour. Que les dispositifs soient de gamme inférieure ou supérieure, s’il existe un moyen de mettre à jour leur firmware à distance, ils seront abordés dans ce papier.

Keywords: Systèmes embarqués, Sécurité, Mise à jour de firmware, LoRA, Blockchain pour la mise à jour, FPGA, SoC, OTA, État de l’art, Analyse statique, Interprétation abstraite

Contents

List of Figures	vii
Listings	ix
1 Introduction	1
2 Working with Software	7
2.1 Over the Air (OTA) update	7
2.1.1 Concept	7
2.1.2 Assert trust	8
Cryptography	8
Using the blockchain for certification	10
2.2 Static analysis	14
2.2.1 Integrity Verification	15
Concept	15
Remote Attestation	16
Time-consistency	17
Capturing the control-flow	19
2.2.2 Abstract Interpretation	21
Concept	21
How it is being used in Embedded systems	26
2.2.3 Proof Carrying Code	28
Concept	29
Embedded scenarii	30
2.2.4 Summing up	32
3 Working with hardware	35
3.1 How different is it from software in this case study ?	35
3.1.1 Differenciation	35
3.1.2 Hardware / Software Partitioning	36
3.2 System on Chip (SoC)	36
3.2.1 Dedicated crypto-engine	37
3.2.2 Intra-communication safety verifications	39
3.2.3 Reconfiguration	40

3.2.4	Physically Unclonable Functions (PUFs)	41
3.3	Virtualisation-based Security	43
3.3.1	Protected Module Architectures (PMA)	43
	Different kinds	43
	Usage in updates	45
3.3.2	Hardware Sandboxing	46
4	Conclusion	49
	Références	51
A	Sheet n°1	59
A.1	Description of the article	59
A.2	Synthesis of the article	60
B	Sheet n°2	63
B.1	Description of the article	63
B.2	Synthesis of the article	64
C	Sheet n°3	69
C.1	Description of the article	69
C.2	Synthesis of the article	70
D	Sheet n°4	75
D.1	Description of the article	75
D.2	Synthesis of the article	76

List of Figures

1.1	Typical architecture of embedded systems by Alooseel <i>et al.</i> [1]	3
1.2	Security threats associated with their corresponding layer by Leloglu [2]	4
2.1	Securing data with both private and public key	9
2.2	Lee and Lee's overall procedure [3]	11
2.3	Son and Kim's blockchain organization [4]	12
2.4	By Castro <i>et al.</i> [5]	16
2.5	CFG based on listing 2.2 (generated with code2flow.com)	20
2.6	Suomela's explanation on the difference between invariant and inductive invariant [6]	23
2.7	Hasse diagram of a powerset of a set of 3 items (Source: [7, Wikipedia])	27
2.8	Model transformation schematic of Jarus <i>et al.</i> [8]	28
3.1	Apple's SoC schematic [9]	37
3.2	Performance comparison on cryptographic algorithms between software and hardware [10]	38
3.3	Difference between software and hardware based attestation [11, Figure 1]	42
3.4	Evolution of Embedded platform security architecture [12]	44
3.5	Integration of hardware sanboxes described by Mead <i>et al.</i> [13] . . .	47

Listings

2.1	Pseudo-code of CalcAlg [5]	18
2.2	Code example for CFG	19
2.3	Modulo function	22
2.4	Simple loop code	24
2.5	Infinite loop code	25

Chapter 1

Introduction

Nowadays, embedded systems usage has been more and more widespread. Their usage has been so ubiquitous that a new buzzword arose to talk about this field: **IoT** – or **Internet of Things** – which emphasizes on their interconnected nature. Whether it's a plaything you would offer to your children for Christmas, the machine handling your MRI scan or even inside a real airplane ; they are used everywhere for anything. By definition an embedded system is a microprocessor-based computer designed to perform a specific task and is not programmed by the end-user ; and this type of computer is usually meant to be a small piece of a larger system. Aloose *et al.* also add up the following to this definition : "It is noticeable that the main criterion in calling a system an embedded system is the embedding of a processing unit or the integration of computational functionality within a larger physical system to steer the functions of that Cyber Physical System" [1]. Those systems are often used in places that are quite critical in term of human lives, such as *e.g.* you wouldn't want the MRI scan to jeopardize your brain or your brand new Tesla to stop moving while you were driving. Nevertheless, a malicious person – an attacker – could hack into the MRI scanner or your car and make it wreak havoc. From this kind of scenarios arise some security concerns, these concerns have remained neglected for quite a long time in the embedded system field but have gained more and more spotlight throughout the years. In his taxonomy, Gollman [14] presented the core points of Computer security, those that can interest us in the context of embedded systems are : Access control, OS Security, Internet Security and, Software Security. For this state of the art, the two that might be the most important are the first (Access

control) and the last one (Software Security).

Firstly, **Access Control** is about regulating access for unauthorized users. This topic then became the CIA triad which is characterized as *Confidentiality*, *Integrity* and *Availability*. The first one – *Confidentiality* – is about preventing unauthorized disclosure of information. Here in our context it could be preventing a third party to access conversations in my own house through a hacked babyphone which I am using to monitor my baby sleeping in his bed. The second one – *Integrity* – describes the fact that unauthorized modification shouldn't occur (as it is unauthorized). In the context of embedded systems, it could be something as harmless as switching two buttons on a TV's remote, the information I get by clicking on the button "1" is no longer the channel assigned to 1 but let's say the one assigned to 6. Last but not least, *Availability* is about keeping authorized users to access the protected data. For example, putting black paint on a camera's lens is a risk against availability [14].

The second one, **Software security** is about having correct software without bugs or undefined behaviours. An undefined behaviour is a piece of code where the action cannot be predicted when the program was made, for example accessing memory on the go without having used some kind of allocator results to be an undefined behaviour because we don't know what was there either if something else could erase it ; Most undefined behaviours results in being bugs, an error in the flow of the program. Other kind of bugs could be – as stated earlier – a poor *Memory Management* or using dangerous patterns / functions that could lead to *Code Injection*, code that was not supposed to be executed when we first wrote it. To make sure a program is correct, there are multiple things that can be done such as: 1/ testing it empirically, ensure each small part of a code does what it needs to do ; 2/ or proving it, meaning that we made mathematic specifications of the program to ensure the correctness of each of these specifications and that they will eventually terminate [14].

One has to take note that the previous paragraphs stem from the general case not the embedded one. General-purpose computers are meant to be used by anyone for anything, do multiple things at once, display things, run a lot of programs, can get quite powerful, and so on, and so forth. The general purpose computer is a Jack-of-all-trades yet, a master of none. In some cases, you need a master of one and that's where an embedded system – or as literature often call them "Cyber-Physical Systems (CPS)" – can become something one needs, it is tailored to do a particular task which doesn't make it resource dependent and allow the manufacturer to focus the costs on enhancing other aspects such as fitting in its environment, e.g.: warmth, coldness, humidity, sand, etc ; or how well it can handle its task, e.g.: having multiple audio codecs for a headset, temperature control for a fridge, how grounded you desire your coffee beans before brewing it, etc. The security concerns

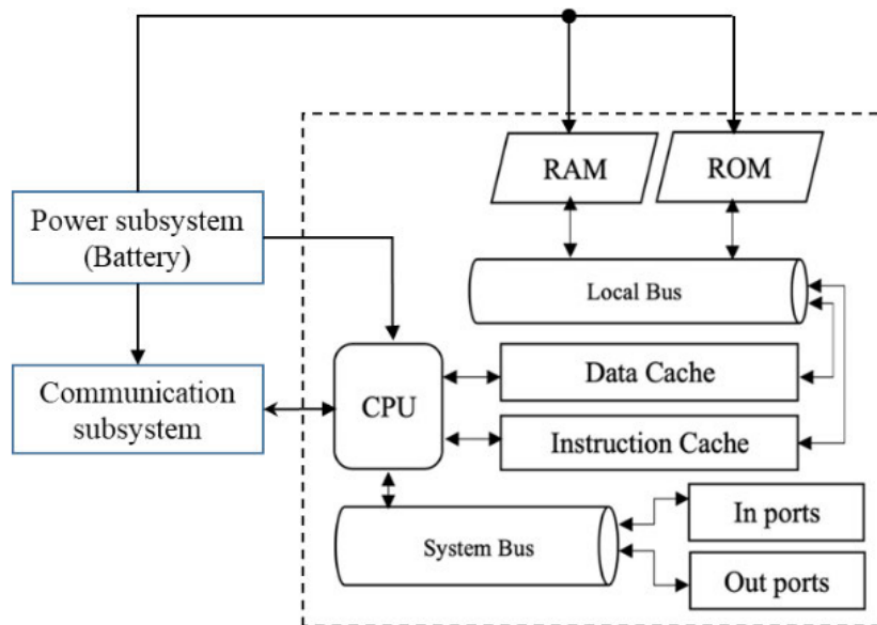


FIGURE 1. Typical architecture of embedded system.

Figure 1.1: Typical architecture of embedded systems by Alooseel *et al.* [1]

for both of these fields might differ a bit too. In their review, Alooseel *et al.* [1] described more extensively "what is an embedded system ?" by describing those system's architecture, their role and more importantly their limitations. The first point comes with Figure 1.1 which describes how manufacturers usually craft an embedded system to make it fit its purpose perfectly.

In this figure, you can notice that the CPU is more or less the central point, the nerve center of the architecture because 1/ it's less expensive to build this kind of system this way, and 2/ this allows easier communication between the core components of the system. Even though it is the nerve center of these systems, one must add components such as a memory or peripheral interfaces to communicate with the outside world [1]. Finally, they also introduce us to the concept of Systems-on-Chip (SoC) which is an integrated circuit that integrates all of the required components for the given system. They then describe their role as a way to make the computer interact with the real world through sensors and actuators.

Finally, they summarize the main limitations that exist in the embedded platform being: 1/ their processing capabilities, they cannot run advanced solution to defend

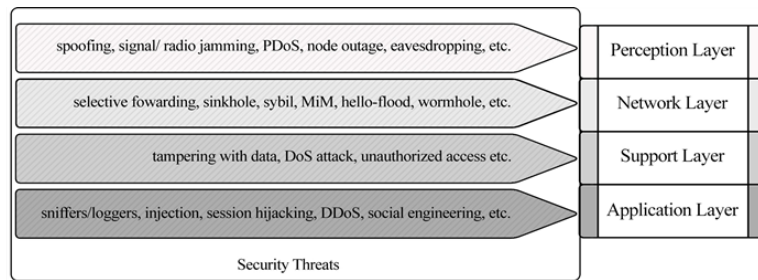


Figure 1.2: Security threats associated with their corresponding layer by Leloglu [2]

against attacks like a general purpose computer would with for example an anti-malware ; 2/ their power supply, which limits the resources that can be allocated to security ; 3/ operating in uncontrolled or harsh environments, which makes them vulnerable to physical attacks ; 4/ remote and unmanned operations, which impose difficulties in controlling physical access to the device or *updating it* ; and their network connectivity which makes them also vulnerable to remote attacks [1].

To understand a bit more about the security concerns in the world of IoT and embedded systems, Leloglu [2] reviewed and summarized research to find out that there are 4 layers of functionalities for IoT devices and networks :

- Perception layer, this one brings together every threats related to sensors. Example of sensors could be : RFID, heat, light, humidity or even acceleration.
- Network layer, to gather every wireless networks that exist in IoT.
- Support layer, regroups every "processing" tasks, when an input A becomes an output B. This layer is quite related to the next one (Application), therefore articles often treat those two together.
- Application layer, is for the final result of what the system / network produces. The author gives smart-traffic, precise agriculture and smart home as examples.

These 4 layers are what an attacker will target on an embedded system when they want to compromise it. Leloglu describes the following threats with the associated layers (Figure 1.2).

Here the most interesting to understand are the followings :

- Spoofing: Falsification of data to gain an illegitimate advantage.
- DoS: Denial of Service, when a legitimate user becomes unavailable temporarily or indefinitely to access data.
- Signal/Radio jamming: Type of DoS attack that consists on blocking the communication channel.

- Sinkhole attack: Redirection of every node to a given node to disrupt the network, steal information or cause nodes to fail.
- Wormhole attack: Type of DoS that consists to relocate data from its original source to another point that was not the one planned originally.
- Eavesdropping: Sniff out confidential information by listening somewhere (a sensor or a port) we should not.
- MitM: Man-in-the-Middle, a form of eavesdropping consisting in monitoring or control communications between two parties.
- Tampering: Modification of legitimate data.
- Session Hijacking: Exploiting security flaws in authentication and session management.

According to the layers that have been exposed, we can focus on some more of Gollman's taxonomy [14]. If we put the spotlight on the network layer, we can study Intrusion Detection Systems (IDS) which are pieces of software able to recognize attack patterns on the network and alert if one is detected. Two kind of mechanisms were described at that time: *Knowledge*-based and *anomaly*-based systems, the first one relies on detecting known attacks and the second one is more about detecting behaviours that deviate from the normal one. As IoT is specific due to all of the existing limitations and the protocols used, generic IDS's may not be suitable, this is why Roy *et al.* proposed a mechanism able to detect efficiently some of the major IoT-centric attacks using machine learning [15]. This approach consists of 4 main mechanisms : (A) removal of multicollinearity, they used the Variance Inflation Factor (VIF) to generate new features – points to look for in the model – and drop the original ones that had a VIF too high when compared with the rest of the dataset ; (B) sampling, as some attacks are less frequent than others, we have less data for these attacks, to avoid specialization on one kind of detection they undersampled the most frequent ones and oversampled the less ones ; (C) dimensionality reduction, to avoid overfitting – being able to detect only data we trained upon and not new data – because of the two previous points they utilized Principal Component Analysis (PCA) to turn the matrix resulted from training to a linear dataset ; and finally (D) an effective classification algorithm, here they proposed a novel algorithm called "B-Stacking" [15].

Another point that was discussed by Gollwan was Race conditions, a problem known as "TOCTOU" – Time of Check (to) Time of Use" – in the litterature [14]. A race condition is when two processes try to access the same data at the same time and the result changes depending on the order of execution. Carpent *et al.* point the

Inc-Lock-Ext and **Cpy-Lock & Writeback** mechanisms as able to protect against this kind of attack [16], we will explore what those mechanisms entail to later.

Now that we have discussed more general concepts of security in embedded systems, let's say that we evaluate that the system we built is 1/ weak against one of the threats or attacks that was discussed earlier and 2/ is critical enough or (inclusive) costs too much money to be replaced everywhere (e.g.: a car), you would want to fix the issue with an update but, how can an update be carried out safely on this kind of devices ? This is what this master thesis will try to answer.

To comprehensively address security concerns in IoT and embedded systems, it is important to explore strategies for implementing software and firmware updates effectively. These updates serve multiple purposes, including patching security vulnerabilities, improving system performance, and ensuring compliance with evolving standards and regulations. Moreover, they enable manufacturers to proactively respond to emerging threats, thereby enhancing the resilience of these systems. This master thesis will delve into the significance of software and firmware updates in IoT and embedded systems security. Its aim will be to investigate the challenges associated with updating these systems securely, considering their main characteristics such as limited processing capabilities, resource constraints, and operating environments. By understanding these challenges, we can develop tailored approaches for implementing updates that minimize security risks and ensure the integrity and availability of these systems. Furthermore, it will explore state-of-the-art methods and techniques for securely updating software and firmware in IoT and embedded systems.

We will delve into the practical aspects of implementing software and firmware updates in IoT and embedded systems. More specifically, we will explore key methodologies and techniques such as Over-the-Air (OTA) updates, static analysis for integrity verification in Chapter II ; and hardware-based protection mechanisms like Protected Module Architectures (PMA) and Sandboxing throughout Chapter III.

Chapter 2

Working with Software

2.1 Over the Air (OTA) update

To keep a system up to date in the IoT, firmwares and softwares need to be updated, and more than probably without a physical access to a computer, for example when one tries to update the software of their smart car [17, 18]. This is where Over-The-Air (OTA) updates come in the spotlight. This section will go through its concept, how trust can be asserted during an update process and existing methods in the IoT field to handle it.

2.1.1 Concept

The concept of OTA update is described by simply downloading a firmware update using the network interface of a device then performing the update on itself [19] (based upon [20]). According to Gündoğan *et al.*, regular updates are part of the common security life cycle nowadays and it is required that it becomes a norm in IoT too [21]. To make this process more regular, they describe their workflow for OTA firmware updates in IoT but it is general enough to describe a general updating workflow for IoT. It consists in the following steps : Firstly, **Version discovery**, devices *naturally* come out of the factory with an up to date firmware but depending on their network availability they may get one or more update late throughout time, therefore we need to know when to upgrade the firmware. Two main strategies exist for this : *proactively pushing notifications of an update* or *periodically polling*

the *firmware repository* (i.e.: where the update lies) [21]. Secondly, **Retrieval of firmware version** is the process of retrieving the fitting update version, here in this study it is down through a namespace management based upon Unix timestamp [21] but we could ensure this through versionning as it is done with SemVer within the world of Rust using Cargo [22, rust-lang.org] (as an example of a different method). Thirdly, **Implicit consumption of firmware version**, different devices of the same class can have their polling intervals drift apart over the time, to avoid this (we are still in the context of using Unix timestamps), the forwarding node will compare the current epoch with the received one and update the device if this comparison results in being true (i.e `forwarder_epoch > device_to_update_epoch`) [21]. Then comes the **Retrieval of the image**, which consists in providing the end-device the version it asked for. This step can be done using two strategies:

- Going in one shot, downloading and then processing the whole image at once.
- Going by chunks, downloading parts of the update concurrently or in cascade and handling it chunk by chunk.

Finally, they describe the **Verification** which ensures the update is a correct one (we will dive onto this topic later) and the **Fault tolerance systems** against connectivity loss and DoS detection.

Kim *et al.* describe a similar process to make an OTA software update for vehicles, they specify that when the *version discovery* phase is successful, the server containing the firmware update will (in their case) send a path (e.g.: an URL) to reach the new update. They also describe their fault tolerance system as a "back to square one" one, if any step fails, we go back to the *version discovery* phase [18].

But, now that we know how to perform an OTA software update, what are the methods used to assert we have been provided the correct firmware or software update ?

2.1.2 Assert trust

As of today, to assess we have been provided the correct version of an update, multiple methods exist. This master thesis will describe cryptography and blockchain. Other methods such as remote attestation or abstract interpretation will be studied later on to assert a different kind of trust.

Cryptography

According to Aloseel *et al.*, cryptography is one of the two main defensive mechanisms to prevent disclosure attacks and other kind of cyber threats [1]. This is Kaspersky's definition of "Cryptography" :

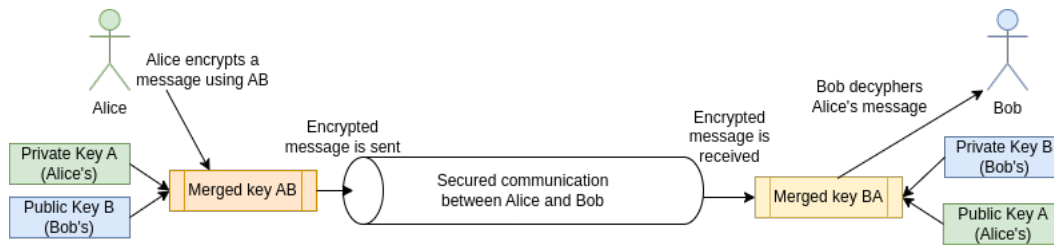


Figure 2.1: Securing data with both private and public key

> Cryptography is the technique of obfuscating or coding data, ensuring that only the person who is meant to see the information – and has the key to break the code – can read it. The word is a hybrid of two Greek words: “kryptós”, which means hidden, and “graphein”, which means to write. Literally, the word cryptography translates to hidden writing, but in reality, the practice involves the secure transmission of information. [23, kaspersky.com]

There are two ways to assert trust by using cryptography: 1/ Using symmetric cryptography and 2/ using asymmetric cryptography. Symmetric cryptography relies on ciphering and deciphering using the same process but reversed. For example, a simple symmetric cryptographic algorithm could be adding 1 to every character’s index in the alphabet in the chain such as “hello” becoming “ifmmn”, the reverse process would be subtracting the index by one. State-of-the-art methods rely on this symmetric pattern but rather than using a simple ± 1 will use more robust and meticulous algorithm such as AES, ChaCha20 or hashing using SHA-2 [24, ANSSI].

The other way to use cryptography is by using asymmetric algorithms. Asymmetric cryptography uses a similar mechanism encrypting data using a key A, but here it is a bit complexified because to decrypt this data you will be using another key B. Key A is known as the public key, the key used to share data with you and Key B is the private one, the one you use to read this encrypted data. This mechanism can also be used in the opposite direction to assert that you wrote a message because you are the only one to know the private key. Now that we have this basis, we can illustrate using cryptography’s most famous protagonists, Alice and Bob. If you merge both of the ideas that were explained earlier, asymmetric encryption allows Alice to speak with Bob and only Bob ; and it will allow Bob to verify that the message he received was sent by Alice and only Alice, see figure 2.1.

Now that this process has been explained, we can note that Zandberg *et al.* chose to use only asymmetric cryptography in their prototype of the SUIT model and did not use firmware encryption – a process that consists in concealing the firmware on the device using cryptography and deciphering it on demand of a given feature / function call – because it would have required too much time to develop a well made proof-of-concept using every refinement possible and it was more interesting to focus only on the essential [25]. On the other hand, Alooseel *et al* insist on the

fact that in constrained IoT devices, cryptography is not an easy task because of its performance cost and that keys are vulnerable to unauthorized access if not well protected and not made for data encryption [1]. This problem is so well known that Apple introduced the Secure Enclave Processor (SEP) since the A7 processors that is dedicated to handle all the cryptographic and authentication tasks [26].

In conclusion, cryptography in embedded systems presents both advantages and challenges. On the positive side, cryptography serves as a defensive mechanism against disclosure attacks and cyber threats, ensuring the security and privacy of sensitive information. It provides a means to obfuscate or encode data, allowing only authorized individuals with the appropriate keys to access it. However, there are drawbacks to implementing cryptography in embedded systems. One notable concern is the performance cost, especially in constrained IoT devices, where computational resources are limited. The use of cryptography can significantly impact the device's speed and efficiency, posing challenges for real-time applications. Moreover, ensuring the protection of cryptographic keys from unauthorized access is essential, as compromised keys can lead to security breaches. This necessitates robust key management practices to safeguard sensitive information effectively.

Using the blockchain for certification

We have explored how cryptography can be used to certify that data comes from Alice rather than Bob but other methods have been explored is Blockchain. The first question that may rise is "What is blockchain ?", He *et al.* proposed the following definition for a blockchain : "Blockchain is a widely popular, emerging technology that utilizes distributed immutable ledgers, consensus algorithms, and smart contracts to essentially provide an incorruptible digital ledger that can be used to record and validate any kind of transaction" [19]. If we break this definition down, the terms "ledger", "consensus algorithm" and "smart-contracts" can seem a bit abstract. The first one – the ledger – is a register where we will write down any transaction – a record – that will be done, e.g. "Alice pays 3 units to Bob". In the context of blockchain, the ledgers are distributed, meaning that each node of the chain – a computer – will hold the blockchain's ledger [27, ledger.com]. Finally those ledgers will hold immutable records that no one can update and are distributed across the whole chain of ledgers.

The next bullet point is about the "consensus algorithms". According to Binance, "A consensus algorithm is a mechanism that allows users or machines to coordinate in a distributed setting. It needs to ensure that all agents in the system can agree on a single source of truth, even if some agents fail. " [28, binance.com]. They denote two main families of consensus algorithms : "**Proof of Work**" (PoW) and "**Proof of Stake**" (PoS) but other exist. PoW was first introduced for bitcoins, the validators will hash the data repeatedly until they find a satisfactory solution. As

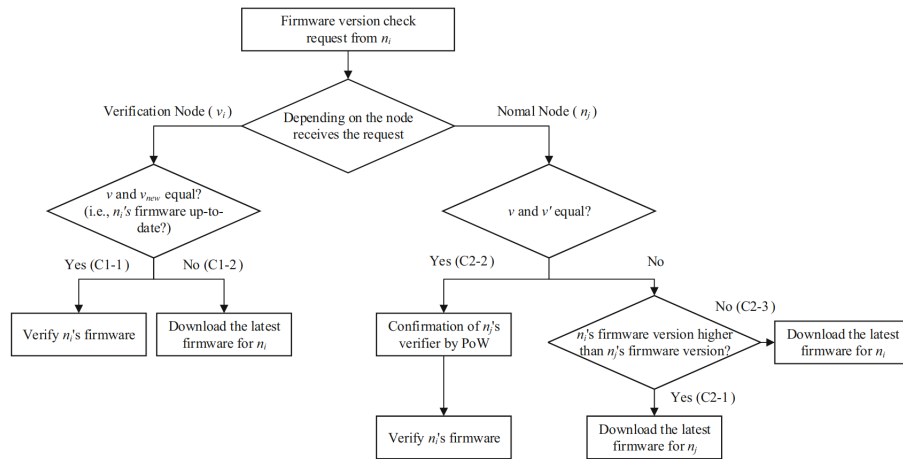


Figure 2.2: Lee and Lee's overall procedure [3]

a hashing function works like a mathematical function – i.e. an input A will always give an output B – validators will try to tweak the input until the conditions are met. An example of conditions could be "hash must start with 00 and end with Az". With this process in mind, it is easy to understand that if you provide an input A and the blockchain returns a valid output B or C, the data is correct whereas if the output was another solution D not matching the base predicate, the output would be wrong and therefore the input data not accepted as valid [28].

On the other hand, PoS is not about hashing data to validate it which costs in terms of power or hardware. It rather relies on an internal currency or for example reputation in the "Proof-of-Authority" variant. This kind of algorithm relies on voting for which node will be the next one using multiple peers which will hold different wallet value or reputation points [28].

Finally, the last point that was highlighted was "smart-contracts" which are – according to Wikipedia – "computer programs or transaction protocols that are intended to automatically execute, control or document events and actions according to the terms of a contract or an agreement. The objectives of smart contracts are the reduction of need for trusted intermediators, arbitration costs, and fraud losses, as well as the reduction of malicious and accidental exceptions." [29].

The articles studied proposed different approaches. Lee and Lee proposed a framework based on the PoW consensus where the IoT device trying to update is a node in the blockchain. Figure 2.2 describes their procedure. C1-1, C2-1 and C2-2 describe cases where our device (n_i)'s firmware is up to date. On the other hand, C1-2 and C2-3 depict the cases where our device's firmware is not up to date. C1 procedures are associated with verifying nodes, nodes that will be checking the integrity of the firmware. C2 procedures work by comparing the version of two given devices (supposed to have the same firmware) in the blockchain [3].

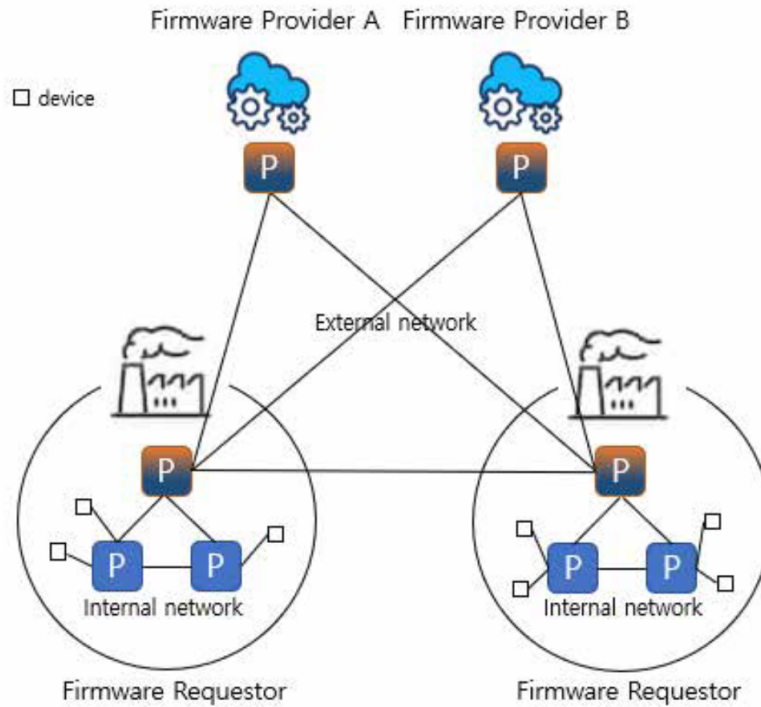


Figure 2.3: Son and Kim's blockchain organization [4]

Another proposition that has been explored is Son and Kim's 'HyperLedger Fabric combined with IPFS' approach. This approach works by first creating a private blockchain which will identify nodes using certificates and PKI (public key infrastructure). This blockchain is a bit special as it uses the HyperLedger Fabric project. This project does not use consensus algorithm but rather use two states, the "world state" and "blockchain state". The first one displays only the latest record available, whereas the second one holds every record that have been encountered. The second step is focused on the record, which will be written in the order of arrival. Finally, records can be assessed and trusted using IPFS to store the actual firmwares and their checksums [4]. They explain that their proposition cannot be based on a single blockchain because, if all nodes – i.e. all devices – on the blockchain were to hold at all time for each binary their firmware's version, the size of transactions would increase, which will cascade to increase the network traffic. Another issue is that, even though IPFS ensures the integrity of the files, we cannot verify the integrity of the URL that will be used to download the firmware's version. Because of those issues, their solution relies on using firmware providers and firmware requestors blockchains at the same time.

Figure 2.3 schematizes how these two kind of blockchains would work together. Firmware requestors are blockchains constructed to be an *internal network* – e.g. an organization's network – where targets for an update will be found. On the other hand, another blockchain is built on an *external network*, the latter is the

firmware providing blockchain, it will propagate the information about the firmware. Finally, to make those blockchains work together, some *brokers* exist, their role is to propagate firmware providers' information to the firmware requestors. Using this architecture coupled with IPFS – which guarantees integrity of the firmware – we can assert the data integrity by sharing the IPFS URL through the blockchain [4].

However, IoT devices do not solely rely on conventional internet and networking, for example a protocol that is widely used in IoT is LoRaWAN. LoRaWAN is a protocol based on LoRA – i.e. "**Long Range**". The base protocol stack aims to be one that can be usable on a low energy consumption device. It works using two main layers : the physical layer and the MAC layer. The first one relies on the fact that communications operate on particular bandwidth thanks to a Chirp Spread Spectrum modulation whereas the second one relies on other protocols, for example LoRaWAN [30]. This protocol describes a network composed of end-devices (i.e. an IoT device), gateways (forward packets between the former and the next point) and network servers [30]. End-devices are classified as follows [31]:

- *Class-A*: Send / ask information when they have / need it, mostly random (as we cannot predict it) and the time windows for their transmissions are really short.
- *Class-B*: Usually synchronized with the gateway using a Beacon. The server "knows" when a device is listening because the time windows are scheduled.
- *Class-C*: Almost continuously open to receive new information, which makes them more resource-hungry but offers a lower latency to the server.

Another important point of LoRaWAN is that it allows the configuration of Bandwidth, the width of the signal in hertz ; Spreading Factor, handles the size of the data sent ; Coding Rate, the error correction ; and Transmit Power, which also describes the power consumption [31]. Anastasiou *et al.* described a framework to perform the remote update using both the blockchain described in this chapter and the LoRa protocol we just described. According to them, even though LoRa/LoRaWAN are becoming more and more popular, updating firmware using these protocols is challenging because they do not offer the most efficient channel for something as big as a firmware image. To make the firmware OTA update more efficient, they proposed to add a security layer based on the blockchain [31].

This process works through the following steps : Create a new firmware that will be held by UpdateServer ; next, transmit this firmware's constraints to the Blockchain. At the same time, the firmware data will be sent to the application server which will negotiate with the network server the device's class (this study was based on class-C devices). The following step is the configuration of the node to perform an update (Class-C multicast, synchronization of the clock) ; The initialization of

the fragmentation session setup occurs at this moment, this is used by the end-device to know if it is concerned by the update. Transmission of the firmware update as fragment files. Verification of the sender's identity through a public key examination. Verification of the firmware compatibility and hardware compatibility with the new firmware, this is where we start to use the blockchain, since at this point the IoT end-device will send the blockchain the constraints of the received firmware. The next step is preparing the reboot to apply the firmware update on the end-device, this is done by marking this image as ready and asking the application to validate this process may it be either automatically or manually handled. Once reboot has occurred, the bootloader will verify if the update is available through the computation of the CRC and decompression process to install the new firmware, thus overwrite the previous one or if there is enough unused space available, write it there. The last step is performed by the blockchain which will validate or invalidate the end device thanks to the information it will send to it. If the information is not what was expected, by the blockchain, a roll-back ID shall be sent to the end-device to downgrade the process [31].

In conclusion, on the *brighter* side, blockchain offers an incorruptible digital ledger that ensures the integrity and security of firmware updates. By leveraging distributed immutable ledgers, consensus algorithms, and smart contracts, blockchain provides a transparent and tamper-resistant mechanism for recording and validating firmware transactions. This helps in certifying the authenticity of firmware updates, thereby mitigating the risk of unauthorized modifications or malicious attacks. Additionally, blockchain enables the efficient sharing and verification of firmware data across multiple nodes, enhancing trust and reliability in the update process. However, there are also some *darker* sides associated with using blockchain for firmware updates in embedded systems. One notable challenge is the complexity of integrating blockchain technology with existing communication protocols, such as LoRaWAN, which are commonly used in IoT devices. Furthermore, ensuring compatibility and security between blockchain-based firmware updates and hardware configurations requires careful consideration and robust validation mechanisms. Additionally, the need for manual intervention in certain stages of the update process, such as validation and rollback decisions, may introduce delays and human error.

2.2 Static analysis

The previous part of this chapter was more about securing the transfer of the firmware OTA update . . . But multiple times, the integrity was an important subject for this update's process. The second part of this chapter will describe how we can make sure the firmware is the correct one and how we can ensure it is not a harmful.

One of the possibilities to ensure a given software is not harmful is by using static analysis, i.e. analyzing code quality characteristics without program execution.

Static Analysis has been widely used in software development to assess the code quality using tools such as linters but it is more broader than this as it focuses on program analysis without executing them. This master thesis will study only three fields of static analysis but more exist and are used.

2.2.1 Integrity Verification

Concept

Earlier in this reading, the sentence "even though IPFS ensures the integrity of the files" rose but the term "integrity" was – purposefully – not explained. Integrity verification is a mechanism that checks if a given software is the one we expect, that it was not modified unintentionally [5]. As of today, multiple methods exist to ensure the integrity of a software, such as digesting it using MD5 or SHA to ensure not even the slightest modification has occurred. Another state of the art method to handle this process in the case of firmware updates is using a Prover/Verifier scheme [5, 32, 16, 33].

This scheme shares some similarities with some of the consensus algorithms that were studied earlier, Castro *et al.* describe this scheme using figure 2.4. On this figure, we have 3 entities, the user who does an action that requires being verified, the verifier that will request information to the Prover and the prover which will send the information it was asked for [5]. The two latter communicate with each other through three different tasks : *Challenge*, the verifier asks the prover information about its current state that will depend on a challenge (Castro *et al.* and Carpent *et al.* chose to send a random challenge [5, 16]), the verifier will also precompute an expected value ; *Calculation*, the Prover computes according to its current state a value that will be sent to the verifier ; and *Classification*, the verifier receives the evidence that the challenge was completed and will judge whether the Prover is a good or malicious peer or – in our case – software. According to different propositions, the challenges are usually based upon the running software's memory [5, 32, 16].

To improve this scheme, Beyer *et al.* suggested to add a witness to this method. The witness is used as a prover, will receive the same exact proposition as the prover. The only difference between the witness and the prover is that we trust the witness to be correct, whereas the prover needs to provide evidences on its correctness [33]. Now, let's explore solutions that fit the IoT case rather than more general ones.

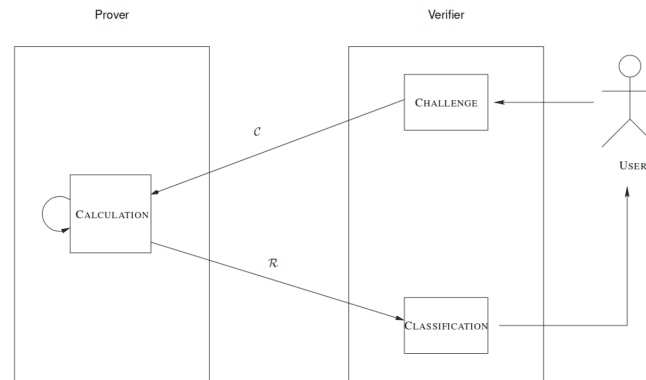


Fig. 1. Phases of the Integrity Verification Scheme.

Figure 2.4: By Castro *et al.* [5]

Remote Attestation

The first solution we can explore to ensure integrity will be "Remote Attestation". According to Sun *et al.* this is the process we described earlier, proving a device's or software's integrity to a remote verifier [32]. Carpent *et al.* dive a bit deeper in this definition by explaining that "Remote Attestation involves verification of current internal state (i.e., RAM or flash) of an untrusted remote hardware platform (prover) by a trusted entity (verifier). RA can help the latter establish a static or dynamic root of trust in prover" [16]. They also spotlight the fact that many methods have been tested for the case where we try to prove the integrity of a single device and they emphasize on the three main methods for implementing remote attestation. The first one is a hardware-only solution that relies on Trusted Protected Modules (see section 3.2). Their main advantage is that they are the most resistant to attacks, while their main disadvantage is their cost, whether it is in complexity or money. The second solution they highlight is a software-only solution which is very low-cost compared to their hardware counterpart but this kind of solution faces a huge drawback, we have to make a lot of assumptions against the kind of attacks we can receive and "adversarial capabilities". De Castro *et al.* provide insights into various attack models and considerations to mitigate them, including the risk associated with hardware and communication channel vulnerabilities [5].

Finally, they make us pay attention to hybrid solutions that are based on both hardware and software [16]. This hybrid approach is the one that Sun *et al.* took for OAT [32] and that Carpent *et al.* chose for their framework [16] by both relying on the TPM / PMA. This approach has the advantage to limit some of the assumptions that software-only approaches would make by having trusted functions verify the internal state (the notion of trust / protection will then depend on the hardware solution for its main limitations).

Time-consistency

Carpent *et al.* also described an approach based on time-consistency [16]. They describe multiple approaches, the two most trivial ones would be : 1/ something as simple as disabling interruptions – actions that when triggered stop every other one to be registered such as e.g.: a key press on the keyboard or a button press on a controller – which is done in the *SMART primitive* [34] ; 2/ copying to a reserved and protected memory space [16]. Through these methods, we can make sure that no other program will intervene while we are attesting the software / firmware. But, if we try to look at this situation through the scope of safety-critical systems, Carpent *et al.* warn us about the fact that we could miss some critical deadlines and therefore those trivial methods aren't the best to use [16]. In their study, they evaluated multiple methods based on a few keypoints. The first keypoint is based on malwares, they describe two kind of them : migratory malwares, they will move themselves in the memory (copying then erasing their older version) to remain invisible to an attestation algorithm ; and transient malwares, that will erase themselves to "escape detection" [16]. Having those two "species" in mind, to attest an host is corrupted, we have to detect if at a time T one or more blocks of memory were holding on the unexpected process. This detection is ensured through a function F secured through TPM and taking an input M at time t (M^t) will always return the same output R . This results as the following mathematical function : $F(M^t) = R$ [16]. While the previous definition holds, if R is considered *benign* – or expected to be present – we can conclude no malware is present, and the opposite is also true, if we consider R as harmful – or unexpected –, the malware cannot escape detection.

We discussed earlier about TOCTOU preventing mechanisms : **Inc-Lock-Ext** and **Cpy-Lock & Writeback**. The first one is based on a progressive **All-Lock** (named **Inc-Lock**) which will lock every bit of memory until all memory has been locked, by allowing the lock to be released if the prover explicitly asks for this to happen. The other one is based on a **Cpy-Lock** mechanism which will run the F function over a locked M' memory, a copied version of M which will be locked while computing F . The writeback version will – once the memory has been trusted – write it back to its original place instead of erasing it [16].

A similar approach is De Castro *et al.*'s integrity verification scheme called EVINCED. This scheme allows integrity verification and remote attestation through time and clock cycles [5]. Its operation works the same as the prover / verifier method explained earlier with a few little tweaks. The first one is to compute the cycle counts at the start (c_{start}) and at the end (c_{end}) of the calculation phase and to return with the computed answer the total computation time ($c = c_{end} - c_{start}$ in cycle counts) to solve the challenge. As the algorithm uses a while loop (see Listing 2.1), the computation time of two different input can be different. The authors also note that this loop also acts as the main part of the algorithm in term of "elapsed

time" [5].

```

1  Input: 1) Challenge C =(s, p, bs); // seed s, stop
          criterion p and block size bs
2          2)TheProver instruction memory size ms.
3  Output: Response R
4  c0 <- CycleCount() // also c_start ;
5  h  <- Sha1(s)       // the current answer ;
6  Prng.Seed(s)        // pseudo random number
7                      // generator ;
8  while not IsPrefix(p, h) do
9      bl <- Prng.Get(max = ms/bs)
10     v  <- Mp[(bl * bs)...[(bl +1) * bs]]
11                      // Mp describes the
12                      // memory of the prover ;
13     h  <- Sha1(h||v)
14                      // || is the
15                      // concatenation
16                      // operation ;
17 end while
18 cf <- CycleCount()
19 c  <- cf - c0
20 R = (h, c)

```

Listing 2.1: Pseudo-code of CalcAlg [5]

On the verifier side, we have to take this while loop in account when creating the challenge there choosing a p that would account for a computation time that isn't too small. To avoid this, the authors choose to create the same algorithm on the verifier's side but to make it use a for-range loop rather going from 1 to N rather than a while loop to produce this p .

Finally, the classification phase will compare a verified prover's (P^v having $R^v = (h^v, c^v)$) results on a given challenge with a potentially tampered prover's (P) result (R) where both h and c must match with h^v and c^v respectively. These two verifications allow to detect if we have changed any byte in the memory ($h == h^v$) or if the computing algorithm has been changed ($c == c^v$) therefore, prove if P has been tampered or not [5].

Time-based remote attestation for embedded systems offers a robust method to ensure the integrity of software/firmware by evaluating the internal state of devices at specific time intervals, thereby detecting and mitigating various types of malware. Additionally, approaches such as EVINCED proposed by De Castro *et al.* provide a detailed scheme for integrity verification and remote attestation through time and clock cycles, enhancing security measures [5]. However, the reliance on time-based methods may introduce computational overhead and timing constraints,

particularly in safety-critical systems where missing critical deadlines can have severe consequences, as cautioned by Carpent *et al.* [16]. Consequently, while effective for many applications, time-based approaches may not be suitable for scenarios where real-time execution is essential.

Capturing the control-flow

While timed-based methods focus on evaluating the internal state of devices at specific intervals, another interesting approach is one based on the control-flow approach which assesses the sequence and logic of operations performed within the software. This is an approach that Sun *et al.* took for their attestation method called OEI (which stands for "Operation Execution Integrity"). The authors describe their method as the following : "An operation satisfies OEI if and only if the operation was performed without its control flow altered or its critical data corrupted during the execution" [32]. This operation works with the prover sending an unforgeable token and the other output data to the verifier which will then be used to tell whether the prover is tampered or not. They *informally define* an operation to be a task with independent logic which must not contain another operation. Every operation must be declared by explicitly tagging the entry and exit points of this given operation in order to construct a control-flow-graph (CFG) of the operation [32]. A CFG is a way to describe a program by using a schematic, for example the C code in Listing 2.2 could be translated to the graph in Figure 2.5.

```

1  int i = 0;
2  int j = 12;
3  if(i < 12) {
4      j = 16;
5  } else {
6      j = 0;
7  }
8
9  for (i = 0; i < 0 ; ++i) {
10     j = j + j * i;
11 }
12
13 operation_on(j);
14
15 while (j >= 0)
16     j = j - 1;
17
18 print("%d\n", j);

```

Listing 2.2: Code example for CFG

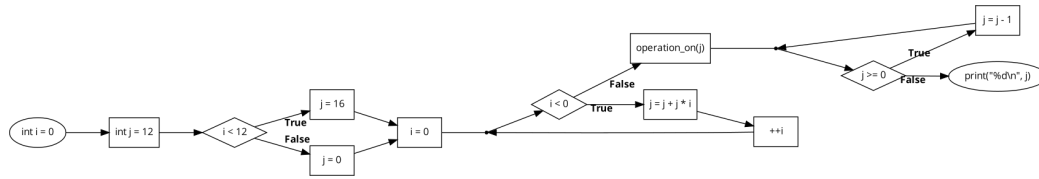


Figure 2.5: CFG based on listing 2.2 (generated with code2flow.com)

This approach based on CFGs has been thought to solve two points :

- Enable remote detection of attacks such as control-flow hijack or data corruption.
- Demonstrate how feasible an operation-oriented approach is to detect these kind of attacks.

But one reading through this section might be stuck to a question : "What is control flow ?" according to Wikipedia, it is "the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated" [35, Wikipedia]. If we try to popularize what OEI does, it evaluates the code / logic in a given task and intends to check if a device has been tampered or not through how its control flow will react to a given input.

Now that the gist has been brought, we can see in more details what this control-flow capture entails. According to the authors, the operation oriented nature of embedded devices is ideal for this kind of method as it allows to avoid whole - program instrumentation – having a program monitor at whole time what we are doing – or always-on measurement collection – having a program monitoring at all time every data related to our code – which makes this treatment lightweight and more importantly suitable for embedded devices [32]. This lightweight approach also allows a per-operation tracking which makes this approach modular, we choose which part of the code needs to be instrumented and only this / these one(s) [32]. When going to the assembly code of a program, the control-flow graph of a program is determined by three types of directives [32] : direct call/jump, usually `call` and `jmp` instructions using a fixed address ; conditional branches, usually `CMP` and "jump if X" instructions; and indirect transfers which work like the direct instructions described earlier but with a register as an argument rather than a fixed address in the memory. When generating the CFGs, the authors chose not to include direct calls because "their destinations are unique and statically determinate" [32]. When the CFG is built and the actual execution trace intended generated, this approach goes back to the remote attestation explained earlier.

This method offers a comprehensive way for detecting attacks such as control-flow hijacks or data corruption in embedded systems. By evaluating the sequence and logic of operations within the software, this approach enables remote detection of

tampering attempts, enhancing the security posture of embedded devices. However, constructing and maintaining CFGs for complex software can be resource-intensive, potentially impacting the performance of resource-constrained embedded systems.

2.2.2 Abstract Interpretation

In the previous part, the study of OEI has been covered, this framework is a part of OAT (**OEI AT**tester) which performs a part of its remote attestation through abstract interpretation using the CFGs discussed earlier. This section will explain what is "Abstract Interpretation" and how it can be used in the context of IoT.

Concept

When talking about Abstract Interpretation, one of the most important concept to understand and reach is **soundness**. Wikipedia defines it through the fact that it must be valid – if and only if the premises are true, the conclusion can not be false – and the premises are true. Another implicit idea when defining soundness in a system means that we can logically prove any member of the given system [36, Wikipedia]. Miné also add up by stating that "whatever the properties inferred by an analysis, they can be trusted to hold on actual program executions" [37].

Rather than a platform, abstract interpretation and static analysers are usually built on or for a programming language. Keidel *et al.* define the three following challenges when building abstract interpretation for a transformation language, a language that will usually assess formal components then be transpiled to another language such as Gallina and the CoqProof platform will with C / Haskell. They identify *Domain Specific Features*, features that cannot be found in generic-purpose languages such as *rich* pattern-matching ; *Term abstraction*, how a semantic will be translated to perform the best analysis [38] ; and *Soundness* that we defined earlier. Here their approach is tailored for Stratego but this concept could be applied to more generic compilers like CompCert for C, which guarantees that the program obtained through the compilation will have the same semantic that were provided in the base C code [38].

Now, let's dive into what "Abstract Interpretation" is really about. To explain what it is, Miné explains that to make a sign analysis, we could try out every integer passed to a function and look at its sign through the program execution ; this method will naïvely work but will also be quite inefficient on the time basis (it won't be fast). A solution to avoid testing every single possibility would be to abstract the computation. To exemplify this behaviour, he took a simple modulo function that takes inputs A and B and will return the remainder R of the operation, nothing is done with Q. This program would look like this in C [37]:

```

1  int modulo(int A, int B) {
2      int Q = 0;
3      int R = A;
4      while (R >= B) {
5          R = R - B;
6          Q = Q + 1;
7      }
8      return R;
9  }

```

Listing 2.3: Modulo function

An execution trace of this function could be the following (with $A = 13$ and $B = 5$ as input values) :

- 1 : $A = 13, B = 5$
- 2 : $A = 13, B = 5, Q = 0$
- 3 : $A = 13, B = 5, Q = 0, R = 13$
- 4 : $13 \geq 5$
 - 5 : $A = 13, B = 5, Q = 0, R = 8$
 - 6 : $A = 13, B = 5, Q = 1, R = 8$
- 4 : $8 \geq 5$
 - 5 : $A = 13, B = 5, Q = 1, R = 3$
 - 6 : $A = 13, B = 5, Q = 2, R = 3$
- 8 : $R = 3$

Now, if we go back to the modulo function, we want to use it only on positive values. If we change the input values with abstract terms, we would have $A = (\geq 0)$ and $B = (\geq 0)$. As $(\geq 0) + (\geq 0) = (\geq 0)$, we can easily abstract this trace as the following :

- 1 : $A = (\geq 0), B = (\geq 0)$
- 2 : $A = (\geq 0), B = (\geq 0), Q = 0$
- 3 : $A = (\geq 0), B = (\geq 0), Q = 0, R = (\geq 0)$
- 4 : $(\geq 0) \geq (\geq 0)$
 - 5 : $A = (\geq 0), B = (\geq 0), Q = 0, R = ?$

▲ I don't know anything about the topic, so let me try to give a very simple example and let's see what others say.

19

▼ Program:

- Initialise: $x \leftarrow 2$.
- Iterate: $x \leftarrow 2x - 1$.

🔖

✓ Now " $x > 0$ " is an invariant. It certainly holds initially and after each iteration.

🔄 However, it is not an *inductive* invariant: merely knowing that $x > 0$ before an iteration is not sufficient to guarantee that $x > 0$ after the iteration. After all, if $x > 0$ is *all* that we know, then we might have $x = 0.1$ before an iteration and thus $x < 0$ after the iteration.

On the other hand, something like " $x > 1$ " is an invariant and also an inductive invariant. It is easy to check that if $x > 1$ before an iteration, then also $x > 1$ after the iteration.

Share Cite Improve this answer Follow

answered Nov 25, 2010 at 17:18


 **Jukka Suomela**
11.5k ● 2 ● 53 ● 116

Figure 2.6: Suomela's explanation on the difference between invariant and inductive invariant [6]

- 6 : $A = (\geq 0), B = (\geq 0), Q = (\geq 0), R = ?$
- 4 : $? \geq (\geq 0)$
 - 5 : $A = (\geq 0), B = (\geq 0), Q = (\geq 0), R = ?$
 - 6 : $A = (\geq 0), B = (\geq 0), Q = (\geq 0), R = ?$
- 8 : $R = ?$

You can notice that $0 + 1 = (\geq 0)$ which changes the abstract form of Q and that $(\geq 0) - 1 = ?$ because after this point we don't know the type of R anymore. To keep the idea that $R = (\geq 0)$ we would need to keep the idea that $R \geq B \geq 0$ [37]. This allows Miné to bring the concept of "invariants" and "inductive invariants" on the table. The first one is a mathematical property that remains unchanged after operations, the second one is stronger in the fact that by definition, if it used to hold at state S , it must hold at states S' or S'' . A good example of their difference is Suomela's one [6] provided in Figure 2.6.

Bringing the concept of inductive invariants allows me to put the spotlight on another really important part of abstract interpretation, being able to precisely know when an iteration will stop and more importantly, when it won't stop [37]. If we keep talking about integers, we can start by considering listing 2.4 :

```

1  int loop_index = 0;
2  int counter = 0;
3  while (loop_index < 3) {
4      loop_index = loop_index + 1;
5      counter = counter + 1;
6  }

```

Listing 2.4: Simple loop code

If we do an execution trace like we did with the modulo function(2.3), we will obtain a trace that will look like the following one :

- 1 : $loop_index = 0$
- 2 : $loop_index = 0, counter = 0$
- 3 : $0 < 3$
 - 4 : $loop_index = 1, counter = 0$
 - 5 : $loop_index = 1, counter = 1$
- 3 : $1 < 3$
 - 4 : $loop_index = 2, counter = 1$
 - 5 : $loop_index = 2, counter = 2$
- 3 : $2 < 3$
 - 4 : $loop_index = 3, counter = 2$
 - 5 : $loop_index = 3, counter = 3$
- 3 : $3 < 3 \leftarrow$ we don't go back in the loop

But now comes the interesting question, what will the abstraction look like ? Here's an answer :

- 1 : $loop_index \in [0, 0]$
- 2 : $loop_index \in [0, 0], counter \in [0, 0]$
- 3 : Should iterate over $loop_index \in [0, 2]$
 - 4 : $loop_index \in [1, 3], counter \in [0, +\infty]$
 - 5 : $loop_index \in [1, 3], counter \in [1, +\infty]$
- 6 : $loop_index \in [3, 3], counter \in [0, +\infty]$

Here rather than abstracting terms over a sign (positive: (≥ 0), null: (0), negative: (≤ 0) and unknown: (?)), we abstract terms using intervals, the `loop_index` being constrained by its starting point (0) and 3 (exclusive) and the `counter` that is in the realm of \mathbb{N} . `loop_index` is called the "loop invariant" because it is the invariant that characterizes the loop we wrote. Another thing one can notice is that in the loop, once we incremented `loop_index`, both bounds were incremented because, from this point, we are not sure whether we have "overflowed" the iteration index or not [37]. Now, what if we changed a single character in the source code provided earlier ? Rather than incrementing the `loop_index` we will decrement it, the code is provided in listing 2.5.

```

1  int loop_index = 0;
2  int counter = 0;
3  while (loop_index < 3) {
4      loop_index = loop_index - 1;
5      counter = counter + 1;
6  }
```

Listing 2.5: Infinite loop code

Now if we do the abstract analysis of this code, we get the following :

- 1 : $loop_index \in [0, 0]$
- 2 : $loop_index \in [0, 0], counter \in [0, 0]$
- 3 : Should iterate over $loop_index \in [0, 2]$
 - 4 : $loop_index \in [-\infty, -1], counter \in [0, +\infty]$
 - 5 : $loop_index \in [-\infty, -1], counter \in [1, +\infty]$
- 3 : Should iterate over $loop_index \in [-\infty, 2]$
 - 4 : $loop_index \in [-\infty, -2], counter \in [1, +\infty]$
 - 5 : $loop_index \in [-\infty, -2], counter \in [2, +\infty]$
- 6 : $loop_index \in [3, 3], counter \in [0, +\infty]$

Here, for a better understanding, the first iteration has been unrolled from the loop. As the `loop_index` is decremented inside the loop, the condition `loop_index < 3` will always be true, thus the loop will never conclude. The three kinds of invariants that we explored are not the only things that can be done in abstract interpretation but they are some keystones to understand how everything works. For example, if we take an array, the loop invariant can be used to see whether an out-of-bound

access occurs or not. Another example is numeric invariant that can be used with pointers or any kind of numeric data ; other forms could be shape analysis to analyse dynamic memory allocations and recursive data structures ; cost analysis that refers on the program execution rather than the semantics ; or backward analysis, a way to analyse a program from a point of interest (rather than the start) and goes back from that point to see if a given state can be reached or not [37].

Now that we own common knowledge of this topic, we can explore the embedded case.

How it is being used in Embedded systems

Jarus *et al.* describe the main method to perform abstract interpretation in IoT as "model transformation" [8]. A model is a way to describe a program or a part of a program. They identify two concerns. The first one is the integration of the different parts of a given system into a complete one, this is what they describe as *heterogeneous model composition* ; the other one is the transformation of a model for a system to a different one for the same system or a related one. In the context of IoT, model transformation researches are focused on building hierarchical models, *i.e.* models that allow different model types to be combined into more complex model, as if we were playing with some *Legos* and building a car. Another option could be the one described by Trippel *et al.*, who go as far as transforming physical parts of the system into software in order to fuzz test properties, try out a lot of invalid, unexpected or random data as an input to a program or part of it [39]. This transformation from hardware to software is a kind of model transformation. Others like Vittorini *et al.* (with OsMoSys) and Barbierato *et al.* (with SIMTHESys) also chose to use methods based upon software to verify whether a given embedded system is correct or not [8, 40, 41].

Jarus *et al.* use this model transformation through abstract interpretation, model are considered as abstractions of this system's semantic. They note that a semantic or model will hold properties (also called P_x or \mathbb{P}) – information about a part of the given model – that concern only specific parts of this system, thus generating the properties of a system then deriving those properties in the generated model, which is – according to the authors – an effect that is required of abstract interpretation. They organize their system semantics through **lattices**, a partially ordered set that allows to know for any pair what is the *supremum* (the upper bound / join) and the *infimum* (the lower bound / meet) ; here the *meet* constrains both elements on a system whereas the *join* will display that either the constraint P_0 or the constraint P_1 in the pair P is met [8]. The example they give is by choosing a property where two elements can overlap themselves, for example a property where P_0 implies a date to be january (1) and june (6) – which makes an interval of $[1 : 6]$ – and P_1 to be between april (4) and september (9) – $[4 : 9]$ – the meet between P_0 and P_1 will

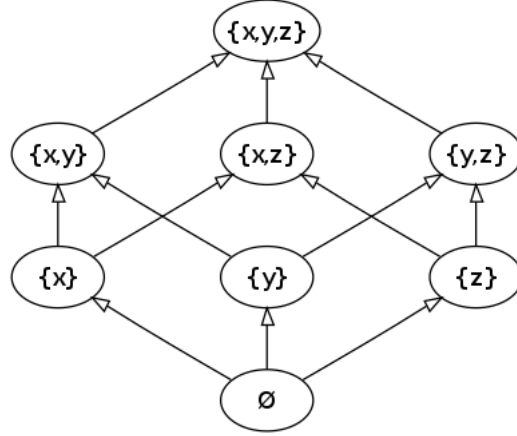


Figure 2.7: Hasse diagram of a powerset of a set of 3 items (Source: [7, Wikipedia])

be $[4 : 6]$ whereas its join will imply $[1 : 9]$. They also emphasize that this method shows elements that are incompatible, for example, if $P_0 = [2 : 4]$ and $P_1 = [6 : 8]$, both constraints on this property cannot be met neither joined, therefore it is an *"impossible" constraint* [8]. To formalize a model (also called M or \mathbb{M}), they used a *powerset lattice* which will more or less directly describe a reliable model formalism to describe the model [8]. The powerset of a set S is the set containing all of its subsets, see Fig 2.7 for a visual example.

This approach of describing a model through lattices also allows us to see whether in a system S , if a model M_1 and a model M_2 are compatible or not, therefore if a system contains coherent or contradictory properties [8]. They later introduce the notion of a Galois connection, which involves monotone functions between two complete lattices representing concrete and abstract domains. The abstraction operator ($\alpha : \mathbb{P} \rightarrow \mathbb{M}$) abstracts a model from system constraints, while the concretization operator ($\gamma : \mathbb{M} \rightarrow \mathbb{P}$) derives system constraints from a model. We can establish relationships between these operators and properties with the following formulae :

$$(\gamma \circ \alpha)(p) \sqsupseteq p \quad (2.1)$$

$$(\alpha \circ \gamma)(m) \sqsubseteq m \quad (2.2)$$

Here, the complete lattice \mathbb{P} represents the concrete domain (properties) while \mathbb{M} represents the abstract domain (models). The \sqsubseteq and \sqsupseteq are quite similar to their round counter part, the only difference being that the one used here (and by Jarus *et al.* [8, page 4]) denotes that we are only talking about partial ordering rather than only sets [42, math.stackexchange.com]. The previous relationship ensures that abstraction relaxes irrelevant constraints but does not contradict the original constraints, and concretization introduces additional constraints while maintaining

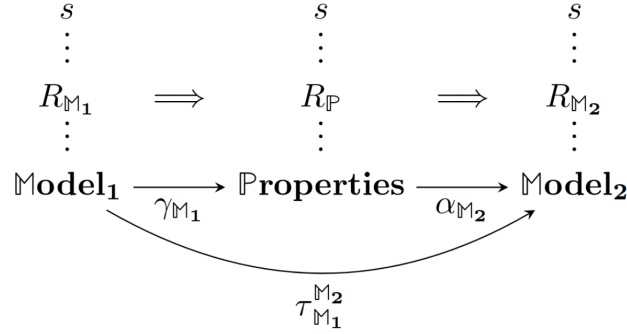


Fig. 1. Sound model transformation.

Figure 2.8: Model transformation schematic of Jarus *et al.* [8]

specificity. They then prove that each Galois connection induces a correctness relation on the abstract domain, providing a sound definition of model abstraction where correct system constraints map to correct models and vice versa [8]. Having explained this, we can now explain in an easier way "how they make their model transformation"; they make the following *definition*: "A model transformation from a model M_1 to a model M_2 is a semantically sound mapping $\tau_{M_1}^{M_2}(m_1) : M_1 \rightarrow M_2$. That is, if $m_1 \in M_1$ is correct, then $\tau_{M_1}^{M_2}(m_1)$ is also correct" which more or less implies the following *theorem* if we also take in account what was *said* earlier: "The mapping $\tau_{M_1}^{M_2}(m_1) = (\alpha_{M_2} \circ \gamma_{M_1})(m_1)$ is sound." [8]. The whole proof is provided in [8, page 4, part "E. Model Transformation"] and the transformation can be visualized in Fig 2.8.

Such an approach can be used in our context to verify if a given device M_1 is a part of the system S or rather if we can trust the properties provided by M_1 to make sure we can update it safely.

2.2.3 Proof Carrying Code

The current need for authentication methods tailored to Embedded Systems relies heavily on cryptographic techniques (which were explored shortly in 2.1.2), but they have limitations in providing holistic authentication covering user, content, and time/location aspects, especially on resource-constrained devices [43]. While source and data authentication are primary classes of authentication, they alone cannot fulfill all security requirements of embedded systems. Lightweight cryptography like Elliptic Curve Cryptography can help secure some resource-poor systems, but many devices, like light bulbs, remain vulnerable due to their inability to run cryptography operations efficiently [43]. Simply adding processing power or hardware-based security to devices may increase cost or energy consumption. Hence, there is a

need for authentication mechanisms feasible for resource-constrained hardware while maintaining low-cost and dependability. Additionally, the long-term security of current authentication schemes is uncertain due to ongoing cryptanalytic advances [43]. Therefore, to try solving this problem, the last concept we will explore in this "static analysis" part is the concept of "Proof Carrying Code", according to Wikipedia it is "a software mechanism that allows a host system to verify properties about an application via a formal proof that accompanies the application's executable code" [44, Wikipedia] and it was first introduced by Necula in [45] (which is not the first paper he wrote about it but the most known).

Necula outlined the necessity for Proof Carrying Code (PCC) due to the challenges posed by mixing components written in different languages within software systems. Traditional mechanisms such as sockets and processes are expensive and inadequate for ensuring safety when incorporating foreign code. Essentially in distributed and web computing, ensuring that code from one agent respects the invariants of another is crucial. Proof-carrying code addresses these issues by requiring code producers to create safety proofs. Consumers can efficiently validate these proofs to ensure code safety without relying on external authentication or cryptography [45]. It was initially employed in network packet filters and ML's runtime systems (a programming language, short of "Meta-Language"), providing technical details and theoretical foundations for its soundness and adequacy [45]. Since Necula's introduction the concept of PCC has evolved and nowadays its trustworthiness does not have to be proven anymore. Subsequent subsections will explain how it works and how it has been used in an embedded context.

Concept

As explained earlier, Proof-Carrying Code (PCC) is a technique that addresses the challenge of ensuring the safety and trustworthiness of executable code, especially when incorporating components written in different languages or originating from untrusted sources. Necula described PCC as a mechanism in which code producers generate a safety proof alongside their executable code [45]. This safety proof attests that the code adheres to a formally defined safety policy, which includes safety rules describing authorized operations and associated preconditions, as well as an interface specifying calling conventions between the code consumer and the foreign program [45]. He also described the lifecycle of a PCC binary which involves three stages: certification, validation, and execution [45]. During certification, the code producer verifies that the source program conforms to the safety policy, generates a proof of successful verification, and encodes it along with the native code to form the PCC binary. In the validation stage, the code consumer quickly validates the proof part of the PCC binary using a straightforward algorithm, ensuring that the code obeys the safety policy. Finally, in the execution stage, the code consumer can execute

the machine-code program multiple times without additional runtime checks, as the validation stage ensures compliance with the safety policy [45].

While PCC ensures safety and trustworthiness, Beyer *et al.* extend the concept by introducing flexibility in the form of correctness witnesses which we explored in section 2.2.1 [33]. Instead of strictly requiring a full proof, their approach allows for less detailed witnesses, reducing the burden on the validator while guiding it towards the proof. They emphasize the separation of concerns, treating the witness as a first-class object separate from the program, which enhances flexibility and maintainability [33]. Another approach inspired by PCC could be Wu *et al.*'s, they propose the idea of Assurance-Carrying Code (ACS) as a means to ensure the trustworthiness of software components in a supply chain [43]. In ACS, each software component carries its own assurance case, enabling system developers to verify the safety or security of the code before integrating it into the supply chain [43]. This approach mirrors the concept of PCC in providing assurances about code integrity but extends it to the physical world, enabling trust in software components throughout their lifecycle.

Similarly, Matsuno *et al.* drew inspiration from PCC to develop an Assurance-Carrying Code system that integrates with a software supply chain [46]. Each software component is accompanied by an assurance case, represented as a Goal Structuring Notation (GSN) diagram. Before integration, the assurance carrying code system checks the assurance case to determine the component's suitability for inclusion in the supply chain, ensuring overall system security [46].

Overall, PCC and its derivatives offer mechanisms for ensuring the safety, trustworthiness, and integrity of executable code, whether in isolated programs, distributed systems, or software supply chains. These techniques address the challenges of code verification and trust establishment in diverse computing environments, contributing to the development of secure and reliable software systems.

Embedded scenarios

The **Correctness witnesses** introduced by Beyer *et al.* are evidences provided by verification processes to demonstrate that a given program satisfies a specified set of requirements or specifications. Unlike *violation witnesses* – they have been introduced by Beyer *et al.* in [47] – which aim to identify errors, correctness witnesses focus on confirming the program's adherence to specified invariants at each control state [33]. The process of analyzing correctness witnesses involves checking the invariants annotated on each abstract program state. If these invariants hold true for their corresponding states, the correctness witness is accepted ; otherwise, it is rejected. Notably, correctness witnesses differ from violation witnesses in that they do not contain assumptions that restrict the state space but instead include state

invariants at each control state [33]. Overall, correctness witnesses play a role in verifying program correctness and ensuring adherence to specified requirements [33].

To construct and verify correctness witnesses, two main approaches have been employed: one based on k-induction in CPAchecker and the other based on automata in UltimateAutomizer [33]. In the CPAchecker-based verifier, the k-induction technique is utilized to obtain unbounded safety proofs. This technique combines bounded model checking with induction to verify candidate invariants for a verification task, particularly those containing unbounded loops. The process involves iteratively asserting invariants for consecutive predecessors, with the aim of proving their inductiveness. An auxiliary-invariant generator runs concurrently to strengthen the induction hypothesis, gradually producing stronger invariants until the induction proof succeeds. These auxiliary invariants, along with the main invariant, are attached to the respective locations in the correctness witness [33].

On the other hand, Automizer follows an automata-based verification approach, where correctness proofs are represented as sequences of automata. The program and correctness specifications are transformed into a Control Flow Automaton (CFA) with error locations, treating the CFA as an acceptor of a formal language. The verification process constructs automata iteratively over the program's operations, each accepting only infeasible sequences of operations. Once the union of these automata covers the language accepted by the CFA, the verification process is complete, and the constructed automata constitute a correctness proof for the program [33]. These automata are then transformed into correctness witnesses [33].

During validation, both CPAchecker and Automizer consider each invariant as an additional specification that must be proven [33]. CPAchecker confirms a witness if it can validate all specifications, while Automizer confirms the witness if all specifications, including the original one, hold true. If validation succeeds, the validators produce another correctness witness containing all confirmed invariants, making them not only consumers but also producers of correctness witnesses. This feature, known as testification, is essential for cases where the validator's trustworthiness is in question, allowing for a chain of validators to enhance credibility [33].

Another vision is proposed by Wu *et al.*. They designed a Proof-Carrying Sensing (PCS) framework for authentication in Embedded Systems [43]. The PCS mechanism relies on leveraging physical data available locally between closely located devices to establish mutual trust. These data can either be intrinsic to the physical environment, such as temperature, luminosity, or noise, or extrinsic, actively injected by devices into the physical world. Authentication protocols within the PCS framework require the coordination of various expertise areas including signal processing, statistical detection, cryptography, software engineering, and electronics.

The framework encompasses two main aspects: intrinsic and extrinsic signatures

[43]. Intrinsic signatures refer to inherent signatures in devices, channels, and sensing environments. These signatures can be exploited to authenticate devices and channels, such as the Electric Network Frequency (ENF) in power grids, which can be captured intrinsically and used for authentication purposes [43]. Additionally, sensors may carry unique intrinsic traces like physical or electronic randomness, known as Physical Unclonable Functions (PUFs) (we will explore them in more details in 3.2.4), which can be utilized for authentication. Extrinsic signatures, on the other hand, are signals and data injected and monitored by a system. These can include physical-layer watermarks, pilot sequences, or challenges injected into an embedded system. By monitoring deviations from expected outcomes of these signatures, potential abnormalities or malicious behavior can be detected [43]. For example, Satchidanandan and Kumar developed a notion of watermarking in embedded systems to detect malicious activity [48].

The vision of the PCS framework revolves around strategically combining digital, extrinsic, and intrinsic signatures for authentication purposes. Digital signatures provide access control and protection against malicious actors, while extrinsic signatures offer alternatives to cryptographic primitives, especially for resource-constrained nodes. Intrinsic signatures authenticate not only source and data but also time and location of data collection [43]. Unlike traditional authentication mechanisms, PCS embeds authentication proofs within sensor data, allowing continuous validation without resource-intensive cryptographic operations [43].

In conclusion, the PCS framework aims to address authentication challenges in embedded systems by utilizing physical data for mutual trust establishment, incorporating intrinsic and extrinsic signatures, and strategically combining digital, extrinsic, and intrinsic signatures for authentication purposes [43].

2.2.4 Summing up

In summary, the reviewed literature presents a comprehensive overview of methods aimed at ensuring the integrity of firmware updates in IoT devices. Various approaches have been discussed based upon static analysis such as integrity verification mechanisms ; the Prover/Verifier scheme for software authenticity assurance ; Remote Attestation is explored as a mean of verifying the internal state of remote hardware platforms ; while time-consistency approaches are investigated to prevent malware interference during updates. Moreover, integrity verification schemes like EVINCED and OEI are examined, focusing on factors such as time, clock cycles, and control-flow to detect tampering.

These methods offer valuable insights into enhancing the security and reliability of firmware updates in the context of IoT devices. Furthermore, we discuss the application of Abstract Interpretation in IoT, highlighting key concepts such as

soundness, specific features, term abstraction, and inductive invariants. The exploration extends to model transformation and proof-carrying code mechanisms, as well as correctness witnesses and the Proof-Carrying Sensing (PCS) framework for authentication in embedded systems. Overall, these concepts aim to ensure the safety, trustworthiness, and integrity of executable code and data across diverse computing environments, including resource-constrained hardware, thereby contributing to the advancement of secure IoT ecosystems.

Chapter 3

Working with hardware

3.1 How different is it from software in this case study ?

3.1.1 Differentiation

Software and **Hardware** are two sides of the same coin, they work together to make one piece of *something*. But as names differ, their purpose do too. **Software** is all the *high level* code that will be written by a developer or engineer. The term "*high level*" refers to an ordered sequence of instructions for changing the state of a computer *hardware* part. On the other hand, **hardware** is the physical part of a computer, may it be a hard-drive, a display monitor, the microprocessor or anything that we can touch, it is hardware [49, diffen.com].

In IoT and embedded systems, there are a lot of different CPU based systems such as microprocessors, microcontrollers and digital signal processors (DSP), reconfigurable devices and application specific integrated circuits – shortened ASIC – [50]. Most developpers will always work on the first kind of devices (microprocessors / microcontrollers), programming them using C or equivalent. But as Salewski *et al.* do point out, it is not the only option that exist. Reconfigurable devices – which is the superset that contains both complex programmable logic devices (CPLD) and field programmable logic arrays (FPGA) – blur out the difference between software and hardware which makes them *programmable* in a sense by using the right tools.

For example in the case of their lab courses they use VHDL (a hardware description language) with CPLD [50].

The rest of this thesis will focus on the main hardware mechanisms that exist to provide security during a software or firmware update in embedded systems such as PMAs, reconfiguration, PUF or even a more recent technique that involves sandboxing directly on the hardware (every term will be explained in due time).

3.1.2 Hardware / Software Partitioning

The first point we will explore is how we can "partition" hardware and software. According to Apvrille and Li, it "intends to split the functions of a system between software components (Operating Systems, application code) and hardware components (processors, FPGA, hardware accelerators, buses, memories, ...)" [51]. In the literature, another term used for it is "co-design" (*i.e.* Hardware/Software co-design) [52, 53, 50].

To explain a more thoroughly, partitioning is about determining how core components of the software and core components of the hardware interact between each other for each functionality of the system [51]. Under the term "*functionality*" comes every kind of functions (in term of code), task or communication the system can do. This approach is still quite close to the "software approaches" we studied earlier, for example Apvrille and Li used abstract interpretation (2.2.2) with SysML-Sec (which has been described in [54]) and TTool which is a FOSS "that offers modeling, verification and code generation capabilities" [51, 55].

Apvrille and Li's approach is guided by three pinacles : **Safety**, the avoidance of system states that can cause personnel or property damage [51] ; **Security**, which has been studied at length in 1 ; and **Performance**, the time needed for a system to *perform* its task. They modeled every part of their partitioned system – namely : abstract behaviour of tasks, algorithms, communications, architectures and hardware components – so that they could simulate and formally verify any component. This approach allowed them to evaluate and iterate over safety / security aspects easily [51].

3.2 System on Chip (SoC)

Partitioning was still a lot about software. Starting from this point, we will study mechanisms that are deeper and deeper into the hardware. The first one will be on the brink between software and hardware, System-on-(a-)Chip – shortened as SoC. According to Wikipedia, a SoC is an "integrated circuit that integrates most or all components of a computer or other electronic system" [56, Wikipedia]. We talked about FPGAs and ASICs earlier, those are used quite extensively to create SoCs. The former is expensive to produce therefore it is used more often to prototype than

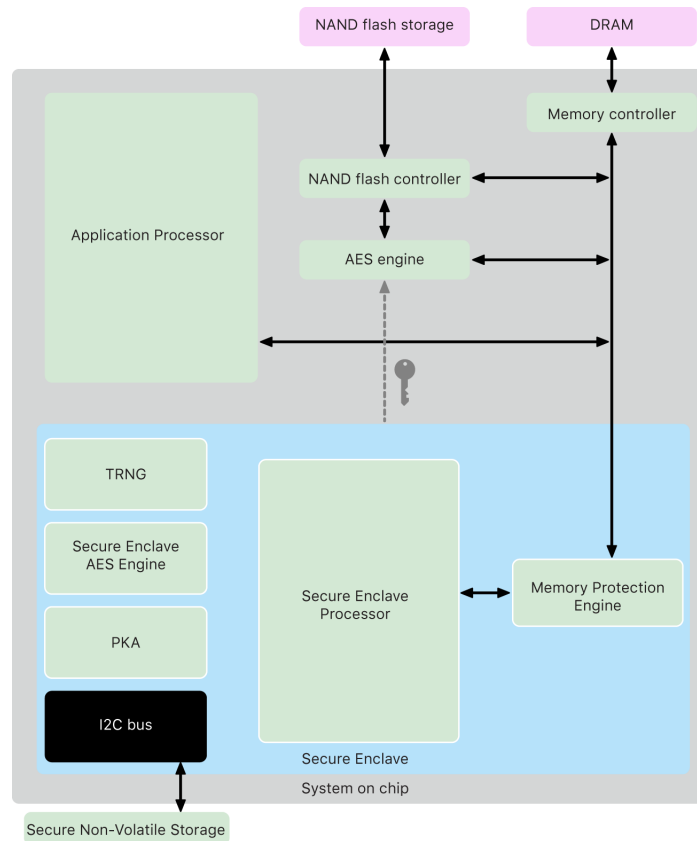


Figure 3.1: Apple's SoC schematic [9]

as the final piece of a project (there are some special cases, one is reviewed in 3.3.2) whereas the latter will be used in production because it would have been specifically designed for it [57]. Of course, SoCs are not restricted only to FPGAs and ASICs, some CPU-based architectures include SoCs. An everyday life example could be the Apple-designed silicon [58] (we will come back to them soon enough).

This section will study how dedicated crypto-engine, safety verification of hardware components such as the AXI bus, reconfiguration and Physically Unclonable Functions are made and performed onto a system.

3.2.1 Dedicated crypto-engine

We have seen in section 2.1.2 the usage of cryptography in order to establish trust between two peers. This section will rather explore why some systems will bring their own cryptographic engine that only does cryptography, such as Apple's SoC having a dedicated AES engine in both the *general* part of the SoC and the Secure Enclave (see Fig 3.1) [26, 58, 9].

Having a dedicated cryptographic engine allows to run cryptographic operations (for given algorithms) more efficiently and more safely [9]. Behind the term efficiency, studies show that computing cryptographic operations on the hardware directly

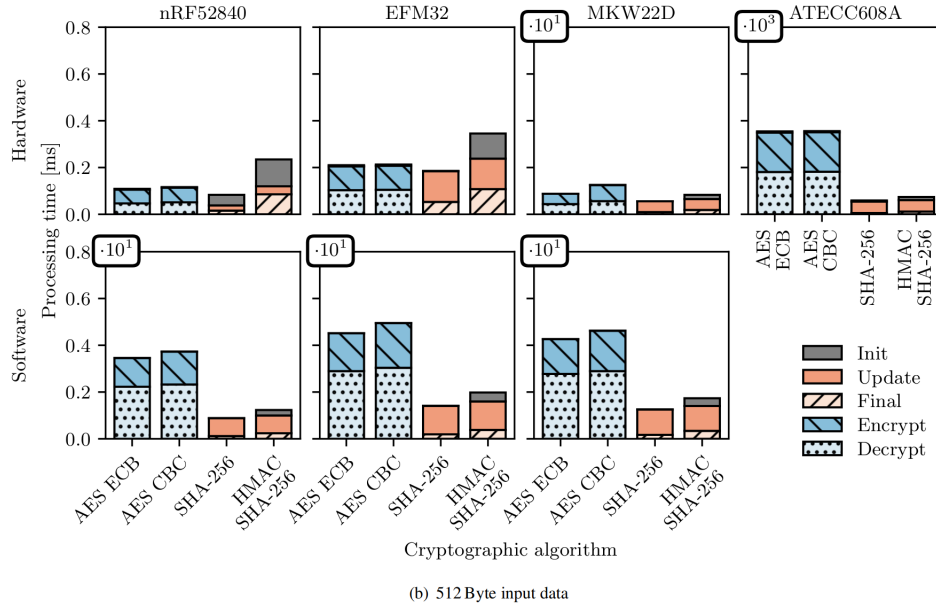


Figure 3.2: Performance comparison on cryptographic algorithms between software and hardware [10]

consumes less energy, Kietzman *et al.* are quite extensive on it. For example, they show that the processing time for classic cryptographic algorithms such as AES or SHA and on *state-of-the-art devices* (*nRF52840* / *EFM32*) the hardware version operating 5 to 10 times faster than the software one and having a ciphering speed enhanced by a factor of 20 to 30[10].

The fourth column displayed on Fig 3.2 shows the usage of an external cryptographic chip (ATECC608A connected via nRF52840) [10]. This chip was also a contestant in this study because it is one protected against *side-channel attacks* [10]. A side channel attack is an "exploit that aims to gather information from or influence the program execution of a system by measuring or exploiting indirect effects of the system or its hardware" [59, techtarget.com]. Once again, we see the trade-off between security and performance thanks to those two devices.

Even though this is a huge trade-off, we have the example of Apple that still chose to propose (starting from A9 SoC) a crypto-engine in the Secure Enclave that is also protected against side channel attacks such as power analysis [9]. The purpose behind this idea is that this crypto-engine is dedicated to the tasks that require the highest level of authentication / privacy – such as passwords or the FaceID features – therefore, if the information this works with are the most sensitive we will find on this phone (if one gets the PIN code of my phone, they can basically access most information available on it), we can value the overhead of protecting this dedicated crypto-engine to a maximum. For operations that require encryption but are less sensitive and requires more performance than password encryption (such as file or disk encryption / decryption), they use the dedicated crypto-engine [9].

Vliegen *et al.* designed reconfigurable hardware (we will go back to the concept of reconfiguration in 3.2.3) that uses cryptography as one of its main components. They note that a cryptographic protocol should take three principles in account : *data confidentiality*, prevent someone to read the communication ; *data authentication*, prevent impersonation ; and *mutual entity authentication*, ensure that both sides are communicating with the desired peer [60]. The protocol they designed establishes session keys dynamically, enhancing security against attacks. Utilizing public key cryptography reduces the need for centralized key storage, enhancing scalability and security [60]. Their approach is pushed by the "cryptocore" – or to stick with the terms we used previously, a crypto-engine – they realised that implements SHA256, AES128, a Random Number Generator and an Elliptic Curve Processor [60] that will then be used for reconfiguration.

If we go back to our context of firmware updates, Gündoğan *et al.* remind us that it is quite infeasible to perform asymmetric cryptography on low-end embedded devices without hardware acceleration, as we described it throughout this whole section [21, 10]. We can conclude that cryptography is an important and even mandatory element in embedded devices security. The question is rather, "do we want safer or faster cryptography ?".

3.2.2 Intra-communication safety verifications

If we take a deeper look at what Apple did or even Vliegen *et al.* to link their crypto-engines to the outside world, we see that there are output channels [60, 9]. Those output channels have to be verified, that's what Meza *et al.* propose through their safety verification method [61].

The first thing we have to note is that, this approach focuses on data buses. To be even more accurate, on AXI bus following the AMBA AXI Standard. A bus serves as a communication pathway between controller devices and peripheral devices [61]. The AXI bus stall problem arises within this context, particularly when multiple controllers contend for access to a shared peripheral. In scenarios where one controller delays providing data after issuing a write request, it monopolizes the bus, hindering other controllers' access to the peripheral [61]. Despite the AXI standard's lack of specification regarding time limits for data provision, controllers can exploit this ambiguity to unfairly affect the availability of the shared peripheral to other components, potentially leading to system performance degradation [61].

Safety verification involves using tools to specify security properties and ensure that a system complies with them, either through formal methods or simulation-based approaches. Due to scalability concerns with formal methods, simulation-based tools are often employed for safety verification, as exemplified by the methodology introduced by Meza *et al.*, which utilizes a simulation-based Information Flow Tracking – shortened as IFT, which is a "verification technique that enables the

tracking of information as it propagates through the hardware" [61, 62] – tool along with custom *trigger modules* to detect issues like the AXI bus stall problem [61]. To address the AXI bus stall problem, Meza *et al.* propose verifying that each controller provisions data within a specific time frame after booking the bus with a write request. This entails determining the acceptable delay for each controller based on system constraints and then ensuring that controllers adhere to these limits [61].

In order to solve this issue, their methodology involves utilizing a custom trigger module to track the state of write transactions for individual controllers. This module receives inputs indicating the maximum delay limit for each controller and monitors the incoming and outgoing AXI signals related to write transactions. It outputs the controller's state regarding write transactions, categorizing it as idle, within the delay limit, or exceeding the limit [61]. By employing this module, system integrators can assess controller safety and verify compliance with specified delay requirements, thereby mitigating the AXI bus stall problem [61].

If we try to apply this concept to firmware updates, the trigger modules can be used to monitor the state of write transactions during the update process. This could allow us to monitor that the firmware update proceeds smoothly and efficiently without encountering delays or stalls, thereby reducing the risk of update failures or system instability. Additionally, the modules can enforce time constraints on write transactions, ensuring that firmware updates are completed within a specified timeframe, which enhances overall system reliability and security.

3.2.3 Reconfiguration

Now that communication between components of a system has been explored, we can take a look at reconfiguration. This concept is a special one among firmware updates in the sense that, when reconfiguring the *system*, you don't change only the software but also how the hardware will react. Due to this need to reconfigure the hardware, we need to use a board that can be designed and redesigned *on the fly*, we will explore Vliegen *et al.*'s approach using FPGAs.

Their approach is called *Dynamic Partial Reconfiguration* (DPR). The word "partial" involves dividing the FPGA into a single static partition containing essential communication interfaces – such as the "cryptocore" discussed in 3.2.1 – and one or more reconfigurable partitions housing the main application [60]. By segmenting the FPGA in this manner, maximum reconfigurable resources are allocated to the main application, enhancing its flexibility and adaptability. Furthermore, a soft-core processor, MicroBlaze, orchestrates the overall system within the static partition, managing the communication between IP cores and handling incoming network packets. The reconfigurable partition operates independently and accommodates the main application, allowing for dynamic updates while minimizing downtime [60].

The methodology employed by Vliegen *et al.* for DPR is a regular updating process, a server sends the *bitstream* – instead of the firmware – divided in ciphered chunks to the device. Those chunks are then deciphered by using the crypto-engine that lies in the static partition [60]. Only upon successful validation, the partial bitstream is written through the Internal Configuration Access Port (ICAP) for reconfiguration, updating the behavior of the reconfigurable partition without disrupting the main application’s functionality [60]. This approach ensures secure and efficient FPGA reconfiguration, with downtime limited to the time required for transferring and updating the partial bitstream [60].

In summary, thanks to the mutable nature of the FPGAs, this approach allows to remotely change *physical* parts of a system that could have been deployed. The main downside to this approach is the monetary cost of FPGAs, as described earlier, their flexibility is needed mostly to prototype systems that will later be provided as ASICs, that cost less to produce and consume less power [57].

3.2.4 Physically Unclonable Functions (PUFs)

The last SoC topic we will take on a hot topic of hardware security, Physically Unclonable Functions – shortened as PUF [63]. Schulz *et al.* describe them as noisy functions that are part of a physical object [64]. A "physical object" refers to the hardware as those functions lie directly onto the electronic parts of the system [58, 63]. The term "noisy" means that those functions are susceptible to noise – such as [65, thermal (also called Johnson-Nyquist)] or [66, atmospheric] noises – their return value can change because of the environment if we ask the function’s value multiple times [64]. Schulz *et al.* add up on this noise that those functions can be used deterministically through *fuzzy extractors* on embedded devices (see [67, 68]) [64].

Of course, we can use this concept in the field of firmware updates. We could for example take a deeper look at remote attestation where Castro *et al.* suggested to take a look at propositions from : Schulz *et al.*, that chose to extend existing software attestation protocols by introducing a PUF-based hardware checksum to include device-specific properties of the prover’s hardware, leveraging limited external interface throughput to prevent computation outsourcing and assure the verifier of the original hardware’s involvement in the attestation process [64] ; Kocabas *et al.* – whose attestation method derive from the one described by [64] – describe an iterative method that integrates the outputs of the prover’s PUF into the software attestation process, exploiting the inter-dependency between the software checksum algorithm and PUF outputs to ensure both device identity and software integrity, while addressing challenges such as efficient prediction of PUF outputs and the need for PUF designs with high throughput and large challenge/response space (see Figure 3.3) [11] ; and later Kong *et al.* – their approach is based on [64] and [11] – approach

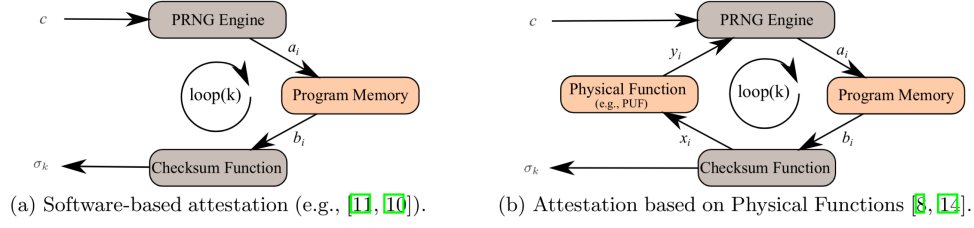


Figure 3.3: Difference between software and hardware based attestation [11, Figure 1]

introduces the integration of the attestation checksum computation with a comprehensive PUF system, which includes features like error correction and obfuscation networks, providing a more sophisticated and comprehensive solution compared to traditional PUF-based attestation methods [69]. But Castro *et al.* also emitted a warning about the fact that it might require additional hardware to implement the usage of PUFs, which may have an unexpected overhead cost [5].

Another method we can look at is Prada-Delgado *et al.* which is directly made for firmware updates over BLE. Their communication use are obfuscated through a PUF-Based approach. This obfuscation method involves a registration phase during which symmetric cryptography is utilized to establish the initial communication between IoT devices and the server, ensuring secure transmission of information during firmware updates. During this phase, the devices make use of the intrinsic randomness of the SRAM start-up values generated with PUF, to generate unique identifiers and nonces [63]. These start-up values are classified based on their characteristics, with certain cells chosen to obfuscate information and others to provide nonces. Through a defined operation involving these values and a unique random key set generated by the manufacturer, termed Helper Data (HD), the devices generate and store encrypted keys for future firmware updates [63].

Following the registration, in the reconstruction phase, when a device is activated by a client, it utilizes the SRAM start-up values to reconstruct the original key set. Although the values used for obfuscation may not exactly match those recorded during registration due to potential bit flipping, they are manipulated using error correction codes to obtain the original key set [63]. Subsequently, a Key Derivation Function (KDF) is employed, incorporating nonces from both the server and the device, to derive a new key set for future updates, ensuring continual security [63].

Their communication protocol is trustworthy due to several factors. Firstly, the encryption and authentication keys are utilized only once, ensuring that firmware updates are released by a trusted source and mitigating the risk of compromise [63]. Secondly, despite potential errors such as transmission and reception errors or response delays, the update process is designed to proceed correctly, thereby maintaining the integrity of the firmware. Lastly, the protocol includes mechanisms

to detect interruptions in the update process, such as power loss during the update, allowing the device to recover and resume the process seamlessly, thereby ensuring the security and reliability of the update mechanism [63].

The integration of PUFs into firmware updates offers several benefits and drawbacks. On the positive side, PUFs provide a robust mechanism for generating unique identifiers and nonces, enhancing the security of communication between IoT devices and servers during firmware updates. Additionally, PUF-based obfuscation can be used to ensure that encryption and authentication keys are used only once, minimizing the risk of being compromised and ensuring the integrity of the update process. However, as it was pointed out by Carpent *et al.*, implementing PUFs may introduce additional hardware requirements, potentially leading to unexpected overhead costs.

3.3 Virtualisation-based Security

3.3.1 Protected Module Architectures (PMA)

The next abyss we reach during our dive is the one containing Protected Module Architectures – shortened as PMAs. PMA's are "hardware extensions that enable the secure and isolated execution of software modules by means of fine-grained memory access control" [53]. They are also the concept behind "GlobalPlatform's Device Architecture Trust" [70] described by Siddiqui and Sezer [12]. The latter also describe the fact that PMA become more and more used to "harness, build and maintain robust security". In Fig 3.2, they display how embedded systems have evolved to gain security. The (4) *Virtualisation-based Security* corresponds to the step we are describing here (and in the next section). On the other hand, you can notice (3) *Dedicated Crypto-engines* and PUFs that were discussed earlier come into the (5) *Hardware-based Trusted Computing*, which should be the approach we should aim for [12].

This section will further detail the different PMA that exist nowadays and how they are used security-wise.

Different kinds

Vendors like Intel, ARM and Risc-V have developed their own solutions, namely Intel SGX, ARM-TrustZone and Hex-FIVE Multizone Security. Each of these three have their own specificities but some base foundation remain within GlobalPlatform's Device Trust Architecture framework.

First, they all share a *virtualization-based security* approach to create *isolated* environments within the processor. These isolated domains, often referred to as "secure worlds," are designed to protect sensitive data and code from unauthorized access

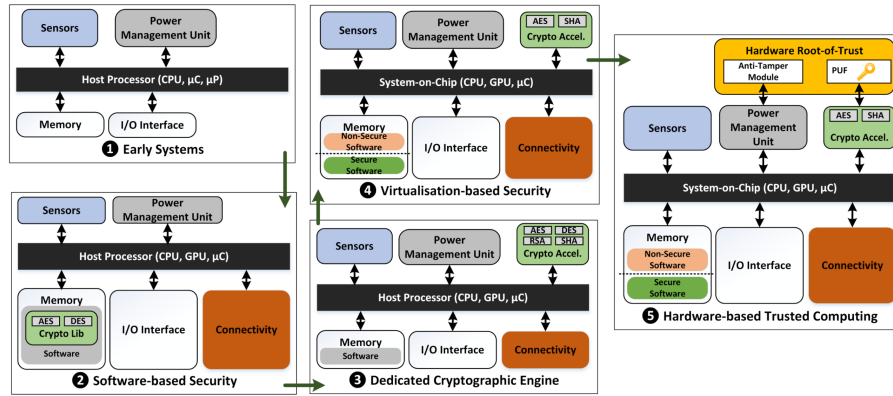


Fig. 1. Evolution of Embedded Platform Security architectures to harness, build and maintain robust security to meet diverse security requirements and technological challenges posed by dynamic threat landscape.

Figure 3.4: Evolution of Embedded platform security architecture [12]

or modification [12]. Secondly, each implementation employs a form of hardware-enforced memory and peripheral isolation to maintain separation between secure and non-secure domains. Finally, all three technologies aim to minimize the risk of human error and software vulnerabilities by implementing lightweight and responsive security solutions, such as bare-metal kernels or policy-driven security environments [12]. By prioritizing hardware-based security mechanisms and minimizing the reliance on complex software stacks, these implementations strive to enhance the robustness and resilience of the overall system architecture against potential security threats and attacks [12].

As for distinct features, Intel SGX employs enclaves to partition sensitive information, accompanied by a system call mechanism for transitioning between trusted and non-trusted execution environments, albeit vulnerable to exploitation via synchronous and asynchronous entry/exit points [12]. ARM's TrustZone, on the other hand, introduces a dual-domain system architecture, comprising secure and non-secure worlds, governed by a security monitor and enforced by address space controllers [12]. Finally, Hex-FIVE MultiZone Security offers a fine-grained compartmentalization approach through a lightweight bare-metal kernel, facilitating policy-driven hardware-enforced separation of resources and utilizing physical memory protection controllers to filter data communication requests [12].

On the other hand, Van Bulck *et al.* present lighter solutions such as the Sanctus architecture which extends the concept of Self-Protecting Modules (SPMs) to low-end platforms, such as TI MSP430 microcontrollers, facilitating hardware-level module protection [53]. SPMs represent a safeguarding mechanism within shared address spaces, delineated by a public code segment and a private data segment. They are accessible only through predefined entry points, maintaining an independent call stack to insulate control flow from external influence. Memory access within an SPM

is regulated by Protected Memory Accesses, where the program counter must align with the corresponding code section for access [53].

Through remote attestation (2.2.1), Sancus assures software deployment integrity and confidentiality, employing symmetric cryptographic keys derived from module contents. Unique module identifiers expedite authentication and access control enforcement. Sancus automates SPM creation via a dedicated C compiler, embedding secure entry and exit code stubs for integrity verification and context management. TrustLite and TyTAN architectures offer alternative approaches, featuring configurable hardware memory protection units and trusted software layers for dynamic loading and attestation. These architectures collectively enhance embedded system security through hardware-enforced module protection and runtime integrity verification [53].

Usage in updates

Knowing those virtualization-based approaches, one reading this master thesis would like to understand why they would be useful for firmware updates. Van Bulck *et al.* point out multiple mechanisms that would still guarantee Real-Time availability such as protecting the interruptions through a trusted software layer rather than totally disabling them as it was done in [34, SMART] or having trusted threads that could handle this process rather than untrusted one [53].

Other approaches based on Siddiqui and Sezer's could be to implement active defense mechanisms that integrate hardware and software layers to detect and respond to malicious activities in real-time. Furthermore, trust-based security enhancements should incorporate runtime response and recovery functions to complement passive defense measures, particularly focusing on vulnerabilities targeting runtime Trusted Execution Environments (TEEs) and microarchitecture [12]. Moreover, access control mechanisms must be enhanced to provide robust runtime security, utilizing distributed access control primitives for fine-grained software compartmentalization and detection of policy violations, thereby facilitating efficient response and recovery actions against malicious activities [12].

In conclusion, PMAs can be of use for firmware updates, providing robust security measures to safeguard sensitive data and code from unauthorized access or modification. Virtualization-based security approaches create isolated environments within processors, minimizing the risk of human error and software vulnerabilities. However, drawbacks include potential exploitation vulnerabilities, as seen with Intel SGX's enclave entry/exit mechanisms. Additionally, while PMAs enhance security, they may introduce complexities in system architecture and require careful consideration to ensure seamless integration with firmware update processes.

3.3.2 Hardware Sandboxing

Finally, the last concept this master thesis will explore is **sandboxing** and more appropriately "Hardware Sandboxing". For now this iteration of the concept has been seen rather on higher end FPGAs used in edge computing [71, 72, 52] but more conventional sandboxing options exist for IoT devices and networks, for example, Jin Kang *et al.* presented IoTBox for smarthouses monitoring [73]. Here we will go on with the **Virtualisation- based security** that was studied earlier 3.3.

In software security, **Sandboxing** is a mechanism used to run untrusted softwares in an isolated environment. This is done to avoid malicious code that can spread on the whole machine [74]. Hardware sandboxing is a concept that derives from it and was introduced by Mead *et al.* in [13]. As it is presented by Bobda *et al.*, hardware sandboxes are the opposite of *ARM's TrustZone* we studied earlier. In "opposite", we have to understand that the TrustZone isolates trusted components and marks them as secured thus making them hard to be accessed by other components that can act *freely* anywhere else, while Hardware Sandboxing is more about letting trusted components *live as they want* and restricting the access to untrusted components that will be *quarantined* in their sandboxes [72].

This approach integrates *checker components* and *virtual resources*, along with controller and configuration registers, to monitor and enforce security rules defined by the system integrator [13]. **Checker components** are designed to inspect the properties of signals associated with IP components within the sandbox, utilizing techniques such as the *Open Verification Library* (OVL) to define and enforce signal properties. **Virtual resources** within the sandbox emulate shared resources between trusted systems and potentially compromised components, ensuring secure communication protocols to prevent denial-of-service attacks [13]. Additionally, status and configuration registers facilitate communication between the sandbox manager and the system, enabling monitoring of IP behavior and facilitating data exchange within the sandbox. The sandbox manager oversees data exchange, handles results from checkers, configures the sandbox, and manages virtual and physical resource interactions, ensuring a secure environment for hardware operations [13].

Mead *et al.* took this approach to harden drones anti-jamming systems and integrated at the boundary of the external radio-frequency (RF) receiver. Fig 3.5 shows the *wiring* of this system. This setup ensures that only legitimate signals, compliant with the protocol, are forwarded to the flight control system via a virtual receiver, thereby mitigating the impact of jamming attacks on the overall system performance [13].

Bobda *et al.* took this approach a bit farther and developed a framework named *Component Authentication Process for Sandboxed Layouts* (CAPSL) [72]. It aims at automating the generation of sandboxes from hardware IPs, represented as components with associated resource access rules in *Property Specification Language* (PSL).

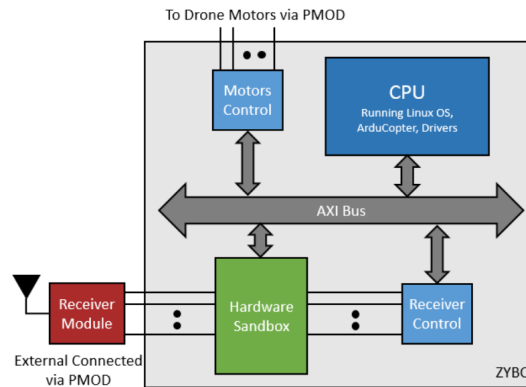


Fig. 2: High-Level Drone System Design with Hardware Sandboxing

Figure 3.5: Integration of hardware sandboxes described by Mead *et al.* [13]

This framework takes inputs of components and PSL rules and produces complete sandboxes suitable for integration into System on Chips (SoCs) [72]. The generation process involves formal modeling of interfaces using *Interface Automata* – shortened IA, they are a shared boundary that acts as a communication channel between components – and extracting *Linear Time Logic* (LTL) representations of PSL rules. These representations are combined to create behavioral standards for sandbox monitoring [72]. Leveraging efficient composition and refinement techniques, sandboxes are optimized to reduce resource requirements, such as by consolidating interfaces of interacting IPs. The generation process also includes *Labeled Transition Systems* (LTS) generation, where IA are mapped to corresponding LTS, and optimizations are applied using a forward-matching algorithm and one-hot coding technique to minimize flip-flop usage and improve design compactness and speed. This automated approach streamlines the design flow from specification to implementation, facilitating the generation of VHDL code and drivers for FPGA integration and enabling compatibility with Xilinx Vivado, Altera Quartus, and SystemC/TLM for high-level simulation [72].

Hardware sandboxing could be used in firmware updates to ensure the security and integrity of the update process. By isolating the firmware update process within a sandbox, potentially malicious firmwares or updates can be contained and prevented from affecting the rest of the system. Sandboxing also allows for monitoring and enforcing security rules during the update process, ensuring that only legitimate updates are applied, thereby reducing the risk of unauthorized access or tampering. Additionally, automated generation frameworks like CAPSL facilitate the creation of sandboxed environments for firmware updates, streamlining the development and integration of secure update mechanisms into hardware systems.

To conclude hardware sandboxing’s largest drawback is that even though their

overhead in performances is not tremendous, drones are considered as *higher end* systems, they often bring in an OS, which is not doable on *lower end* systems that may lack the adequate resources for such a process. On the other hand, sandboxing data received before passing them to the system allows to certify that data will not be harmful for the system.

Chapter 4

Conclusion

To conclude, throughout this master thesis we studied how to perform firmware updates onto resource constrained devices more securely. Out of this, it becomes really apparent that cryptography acts as a keystone to ensure confidentiality, authentication and integrity of an update.

As embedded systems are often viewed from two aspects, the software and the hardware, we reviewed methods that would come from both sides of this coin.

Out of the software methods, one of the limitations that rose was about the communication protocols used in IoT. They often differ from the usual network stack we encounter for the better and the worst, an example being how integrating blockchain communications into LoRA-based protocols has been difficult. On the other hand, those methods allow to efficiently discover which hosts might be tampered or not, thus helping us to decide whether we should update them or not. Further researches could be led on how remote attestation might benefit from Jarus *et al.*'s model transformation method.

As for the hardware side, we noticed how from one type of board to another architecture might differ, thereby how to work with them might too. For example, when Working with FPGAs, they can be reconfigured whereas trying to modify the circuits of an ASIC is impossible. Going through the existing hardware methods, it became clear that software should rely as much on hardware as possible to ensure security of its system, whether it is with a dedicated crypto-engine or through virtualization-based techniques. Finally, using safety verifications such as those presented by Meza *et al.*, as well as sandboxing the firmware update upon reception

before installing might need to be further looked into to understand how they could help in the context of firmware updates.

Bibliography

- [1] A. Alooseel, H. He, C. Shaw, and M. A. Khan, “Analytical Review of Cybersecurity for Embedded Systems,” *IEEE Access*, vol. 9, pp. 961–982, 2021. Conference Name: IEEE Access. [Cited on pages vii, 1, 3, 4, 8, and 10]
- [2] E. Leloglu, “A Review of Security Concerns in Internet of Things,” *Journal of Computer and Communications*, vol. 5, pp. 121–136, Dec. 2016. Number: 1 Publisher: Scientific Research Publishing. [Cited on pages vii and 4]
- [3] B. Lee and J.-H. Lee, “Blockchain-based secure firmware update for embedded devices in an Internet of Things environment,” *The Journal of Supercomputing*, vol. 73, pp. 1152–1167, Mar. 2017. [Cited on pages vii and 11]
- [4] M. Son and H. Kim, “Blockchain-based secure firmware management system in IoT environment,” in *2019 21st International Conference on Advanced Communication Technology (ICACT)*, pp. 142–146, Feb. 2019. ISSN: 1738-9445. [Cited on pages vii, 12, and 13]
- [5] C. G. d. Castro, S. d. M. Câmara, L. F. R. d. C. Carmo, and D. R. Boccardo, “EVINCED: Integrity Verification Scheme for Embedded Systems Based on Time and Clock Cycles,” in *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pp. 788–795, Nov. 2017. [Cited on pages vii, ix, 15, 16, 17, 18, and 42]
- [6] J. Suomela, “Answer to "What’s the difference between an invariant and an "inductive" invariant?,"” Nov. 2010. [Cited on pages vii and 23]
- [7] Wikipedia, “Power set,” Mar. 2024. Page Version ID: 1214547690. [Cited on pages vii and 27]
- [8] N. Jarus, S. S. Sarvestani, and A. Hurson, “Formalizing Cyber–Physical System Model Transformation Via Abstract Interpretation,” in *2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*, (Hangzhou, China), pp. 107–114, IEEE, Jan. 2019. [Cited on pages vii, 26, 27, and 28]

- [9] A. Support, “Secure Enclave.” [Cited on pages vii, 37, 38, and 39]
- [10] P. Kietzmann, L. Boeckmann, L. Lanzieri, T. C. Schmidt, and M. Wählisch, “A Performance Study of Crypto-Hardware in the Low-end IoT,” 2021. Publication info: Published elsewhere. EWSN’21: Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks. [Cited on pages vii, 38, and 39]
- [11] U. Kocabas, A. R. Sadeghi, C. Wachsmann, and S. Schulz, “Poster: practical embedded remote attestation using physically unclonable functions,” in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS ’11, (New York, NY, USA), pp. 797–800, Association for Computing Machinery, Oct. 2011. [Cited on pages vii, 41, and 42]
- [12] F. Siddiqui and S. Sezer, “Evolution of Embedded Platform Security Technologies: Past, Present & Future Challenges,” in *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, pp. 13–18, Sept. 2020. ISSN: 2164-1706. [Cited on pages vii, 43, 44, and 45]
- [13] J. Mead, C. Bobda, and T. J. Whitaker, “Defeating drone jamming with hardware sandboxing,” in *2016 IEEE Asian Hardware-Oriented Security and Trust (AsianHOST)*, pp. 1–6, Dec. 2016. [Cited on pages vii, 46, and 47]
- [14] D. Gollmann, “Computer security,” *WIREs Computational Statistics*, vol. 2, no. 5, pp. 544–554, 2010. [_eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.106](https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.106). [Cited on pages 1, 2, and 5]
- [15] S. Roy, J. Li, B.-J. Choi, and Y. Bai, “A lightweight supervised intrusion detection mechanism for IoT networks,” *Future Generation Computer Systems*, vol. 127, pp. 276–285, Feb. 2022. [Cited on page 5]
- [16] X. Carpent, K. Eldefrawy, N. Rattanaivanon, and G. Tsudik, “Temporal Consistency of Integrity-Ensuring Computations and Applications to Embedded Systems Security,” in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS ’18, (New York, NY, USA), pp. 313–327, Association for Computing Machinery, May 2018. [Cited on pages 6, 15, 16, 17, and 19]
- [17] M. Steger, C. A. Boano, T. Niedermayr, M. Karner, J. Hillebrand, K. Roemer, and W. Rom, “An Efficient and Secure Automotive Wireless Software Update Framework,” *IEEE Transactions on Industrial Informatics*, vol. 14, pp. 2181–2193, May 2018. Conference Name: IEEE Transactions on Industrial Informatics. [Cited on page 7]

-
- [18] G. Kim and I. Y. Jung, “Integrity Assurance of OTA Software Update in Smart Vehicles,” *International Journal on Smart Sensing and Intelligent Systems*, vol. 12, pp. 1–8, Jan. 2019. [Cited on pages 7 and 8]
- [19] X. He, S. Alqahtani, R. Gamble, and M. Papa, “Securing Over-The-Air IoT Firmware Updates using Blockchain,” in *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, COINS ’19, (New York, NY, USA), pp. 164–171, Association for Computing Machinery, May 2019. [Cited on pages 7 and 10]
- [20] C. E. Andrade, S. D. Byers, V. Gopalakrishnan, E. Halepovic, M. Majmundar, D. J. Poole, L. K. Tran, and C. T. Volinsky, “Managing Massive Firmware-Over-The-Air Updates for Connected Cars in Cellular Networks,” in *Proceedings of the 2nd ACM International Workshop on Smart, Autonomous, and Connected Vehicular Systems and Services*, CarSys ’17, (New York, NY, USA), pp. 65–72, Association for Computing Machinery, Oct. 2017. [Cited on page 7]
- [21] C. Gündoğan, C. Amsüss, T. C. Schmidt, and M. Wählisch, “Reliable firmware updates for the information-centric internet of things,” in *Proceedings of the 8th ACM Conference on Information-Centric Networking*, ICN ’21, (New York, NY, USA), pp. 59–70, Association for Computing Machinery, Sept. 2021. [Cited on pages 7, 8, and 39]
- [22] “SemVer Compatibility - The Cargo Book.” [Cited on page 8]
- [23] Kaspersky, “What is Cryptography?,” Sept. 2023. Section: Resource Center. [Cited on page 9]
- [24] ANSSI, “Mécanismes cryptographiques | ANSSI.” [Cited on page 9]
- [25] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, “Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check,” *IEEE Access*, vol. 7, pp. 71907–71920, 2019. Conference Name: IEEE Access. [Cited on page 9]
- [26] T. Mandt, M. Solnik, and D. Wang, “Demystifying the Secure Enclave Processor,” [Cited on pages 10 and 37]
- [27] Ledger, “What Is Blockchain?,” [Cited on page 10]
- [28] Binance, “What Is a Blockchain Consensus Algorithm?,” [Cited on pages 10 and 11]
- [29] Wikipedia, “Smart contract,” Dec. 2023. Page Version ID: 1188555829. [Cited on page 11]

-
- [30] A. Augustin, J. Yi, T. Clausen, and W. M. Townsley, “A Study of LoRa: Long Range & Low Power Networks for the Internet of Things,” *Sensors*, vol. 16, p. 1466, Sept. 2016. Number: 9 Publisher: Multidisciplinary Digital Publishing Institute. [Cited on page 13]
- [31] A. Anastasiou, P. Christodoulou, K. Christodoulou, V. Vassiliou, and Z. Zinonos, “IoT Device Firmware Update over LoRa: The Blockchain Solution,” in *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pp. 404–411, May 2020. ISSN: 2325-2944. [Cited on pages 13 and 14]
- [32] Z. Sun, B. Feng, L. Lu, and S. Jha, “OAT: Attesting Operation Integrity of Embedded Devices,” in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1433–1449, May 2020. ISSN: 2375-1207. [Cited on pages 15, 16, 19, and 20]
- [33] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann, “Correctness witnesses: exchanging verification results between verifiers,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, (New York, NY, USA)*, pp. 326–337, Association for Computing Machinery, Nov. 2016. [Cited on pages 15, 30, and 31]
- [34] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, “Smart: secure and minimal architecture for (establishing dynamic) root of trust.,” in *Ndss*, vol. 12, pp. 1–15, 2012. [Cited on pages 17 and 45]
- [35] Wikipedia, “Control flow,” Feb. 2024. Page Version ID: 1209097766. [Cited on page 20]
- [36] Wikipedia, “Soundness,” Dec. 2023. Page Version ID: 1191952391. [Cited on page 21]
- [37] A. Miné, “Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation,” *Foundations and Trends® in Programming Languages*, vol. 4, pp. 120–372, Dec. 2017. Publisher: Now Publishers, Inc. [Cited on pages 21, 23, 25, and 26]
- [38] S. Keidel and S. Erdweg, “A Systematic Approach to Abstract Interpretation of Program Transformations,” in *Verification, Model Checking, and Abstract Interpretation* (D. Beyer and D. Zufferey, eds.), Lecture Notes in Computer Science, (Cham), pp. 136–157, Springer International Publishing, 2020. [Cited on page 21]
- [39] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, “Fuzzing Hardware Like Software,” pp. 3237–3254, 2022. [Cited on page 26]

-
- [40] V. Vittorini, M. Iacono, N. Mazzocca, and G. Franceschinis, “The OsMoSys approach to multi-formalism modeling of systems,” *Software & Systems Modeling*, vol. 3, pp. 68–81, Mar. 2004. [Cited on page 26]
 - [41] E. Barbierato, M. Gribaudo, and M. Iacono, “SIMTHESysER: a tool generator for the performance evaluation of multiformalism models,” [Cited on page 26]
 - [42] kballou, “Answer to "What does square subset and square union symbol mean?,"” May 2021. [Cited on page 27]
 - [43] M. Wu, F. M. Q. Pereira, J. Liu, H. S. Ramos, M. S. Alvim, and L. B. Oliveira, “Proof-Carrying Sensing: Towards Real-World Authentication in Cyber-Physical Systems,” in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys ’17, (New York, NY, USA), pp. 1–6, Association for Computing Machinery, Nov. 2017. [Cited on pages 28, 29, 30, 31, and 32]
 - [44] Wikipedia, “Proof-carrying code - Wikipedia.” [Cited on page 29]
 - [45] G. C. Necula, “Proof-carrying code,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’97, (New York, NY, USA), pp. 106–119, Association for Computing Machinery, Jan. 1997. [Cited on pages 29 and 30]
 - [46] Y. Matsuno, Y. Yamagata, H. Nishihara, and Y. Hosokawa, “Assurance Carrying Code for Software Supply Chain,” in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 276–277, Oct. 2021. [Cited on page 30]
 - [47] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer, “Witness validation and stepwise testification across software verifiers,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ES-EC/FSE 2015, (New York, NY, USA), pp. 721–733, Association for Computing Machinery, Aug. 2015. [Cited on page 30]
 - [48] B. Satchidanandan and P. R. Kumar, “Dynamic Watermarking: Active Defense of Networked Cyber-Physical Systems,” *Proceedings of the IEEE*, vol. 105, pp. 219–240, Feb. 2017. Conference Name: Proceedings of the IEEE. [Cited on page 32]
 - [49] “Hardware vs Software - Difference and Comparison | Diffen.” [Cited on page 35]
 - [50] F. Salewski, D. Wilking, and S. Kowalewski, “Diverse hardware platforms in embedded systems lab courses: a way to teach the differences,” *ACM SIGBED Review*, vol. 2, pp. 70–74, Oct. 2005. [Cited on pages 35 and 36]

- [51] L. Apvrille and L. W. Li, “Harmonizing Safety, Security and Performance Requirements in Embedded Systems,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1631–1636, Mar. 2019. ISSN: 1558-1101. [Cited on page 36]
- [52] F. Hategekimana, J. Mandebi Mbongue, M. J. H. Pantho, and C. Bobda, “Secure Hardware Kernels Execution in CPU+FPGA Heterogeneous Cloud,” in *2018 International Conference on Field-Programmable Technology (FPT)*, pp. 182–189, Dec. 2018. [Cited on pages 36 and 46]
- [53] J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens, “Towards availability and real-time guarantees for protected module architectures,” in *Companion Proceedings of the 15th International Conference on Modularity*, (Málaga Spain), pp. 146–151, ACM, Mar. 2016. [Cited on pages 36, 43, 44, and 45]
- [54] L. Apvrille and Y. Roudier, “SysML-Sec: A Model Driven Approach for Designing Safe and Secure Systems,” in *3rd International Conference on Model-Driven Engineering and Software Development, Special session on Security and Privacy in Model Based Engineering*, (Angers, France), SCITEPRESS Digital Library, Feb. 2015. [Cited on page 36]
- [55] L. Apvrille, “TTool for DIPLODOCUS: an environment for design space exploration,” in *Proceedings of the 8th international conference on New technologies in distributed systems*, NOTERE ’08, (New York, NY, USA), pp. 1–4, Association for Computing Machinery, June 2008. [Cited on page 36]
- [56] Wikipedia, “System on a chip,” Mar. 2024. Page Version ID: 1213565384. [Cited on page 36]
- [57] “FPGA Prototyping to Structured ASIC Production to Reduce Cost, Risk & TTM.” [Cited on pages 37 and 41]
- [58] A. Support, “Apple SoC security.” [Cited on pages 37 and 41]
- [59] “What is a side-channel attack?.” [Cited on page 38]
- [60] J. Vliegen, N. Mentens, and I. Verbauwhede, “Secure, Remote, Dynamic Reconfiguration of FPGAs,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 7, pp. 35:1–35:19, Dec. 2014. [Cited on pages 39, 40, and 41]
- [61] A. Meza, F. Restuccia, R. Kastner, and J. Oberg, “Safety Verification of Third-Party Hardware Modules via Information Flow Tracking,” [Cited on pages 39 and 40]

-
- [62] W. Hu, A. Ardeshtiricham, and R. Kastner, “Hardware Information Flow Tracking,” *ACM Computing Surveys*, vol. 54, pp. 83:1–83:39, May 2021. [Cited on page 40]
- [63] M. A. Prada-Delgado, A. Vázquez-Reyes, and I. Baturone, “Trustworthy firmware update for Internet-of-Thing Devices using physical unclonable functions,” in *2017 Global Internet of Things Summit (GloTS)*, pp. 1–5, June 2017. [Cited on pages 41, 42, and 43]
- [64] S. Schulz, A.-R. Sadeghi, and C. Wachsmann, “Short paper: lightweight remote attestation using physical functions,” in *Proceedings of the fourth ACM conference on Wireless network security*, (Hamburg Germany), pp. 109–114, ACM, June 2011. [Cited on page 41]
- [65] Wikipedia, “Johnson–Nyquist noise,” Mar. 2024. Page Version ID: 1214036160. [Cited on page 41]
- [66] Wikipedia, “Atmospheric noise,” Jan. 2024. Page Version ID: 1195674544. [Cited on page 41]
- [67] Y. Dodis, L. Reyzin, and A. Smith, “Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data,” in *Advances in Cryptology - EUROCRYPT 2004* (C. Cachin and J. L. Camenisch, eds.), (Berlin, Heidelberg), pp. 523–540, Springer, 2004. [Cited on page 41]
- [68] P. Tuyls and L. Batina, “RFID-Tags for Anti-counterfeiting,” in *Topics in Cryptology – CT-RSA 2006* (D. Pointcheval, ed.), (Berlin, Heidelberg), pp. 115–131, Springer, 2006. [Cited on page 41]
- [69] J. Kong, F. Koushanfar, P. K. Pendyala, A.-R. Sadeghi, and C. Wachsmann, “PUFatt: Embedded Platform Attestation Based on Novel Processor-Based PUFs,” in *Proceedings of the 51st Annual Design Automation Conference, DAC ’14*, (New York, NY, USA), pp. 1–6, Association for Computing Machinery, June 2014. [Cited on page 42]
- [70] “Introduction to Device Trust Architecture.” [Cited on page 43]
- [71] C. Wulf, M. Willig, and D. Göhringer, “A Survey on Hypervisor-based Virtualization of Embedded Reconfigurable Systems,” in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 249–256, Aug. 2021. ISSN: 1946-1488. [Cited on page 46]
- [72] C. Bobda, T. Whitaker, J. M. Mbongue, and S. K. Saha, “Synthesis of Hardware Sandboxes for Trojan Mitigation in Systems on Chip,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, Sept. 2019. ISSN: 2643-1971. [Cited on pages 46 and 47]

- [73] H. Jin Kang, S. Qin Sim, and D. Lo, “IoTBox: Sandbox Mining to Prevent Interaction Threats in IoT Systems,” in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 182–193, Apr. 2021. ISSN: 2159-4848. [Cited on page 46]
- [74] Wikipedia, “Sandbox (computer security),” Mar. 2024. Page Version ID: 1216002953. [Cited on page 46]

□

Appendix A

Sheet nº1

A.1 Description of the article

Title : Computer Security

Link : <https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.106>

Authors list :

Dieter Gollmann

Authors affiliation :

Hamburg University of Technology, 21071 Hamburg, Germany

Name of the conference / journal : WIREs Computational Statistics

Classification of the conference / journal :

Q4 when the article was released (2010), as of today Q1

Number of article citations (which source?) :

1731 (Google Scholar)

A.2 Synthesis of the article

Question

What are the evolving mechanisms and policies for enforcing computer security in IT architectures and software systems?

Computer security involves protecting sensitive resources in computer systems through policies that regulate access and mechanisms for enforcement. This article provides context to enforcement mechanisms by examining their original design for specific IT architectures. It also touches on network security and concludes with remarks on security evaluation.

Computer security encompasses four essential aspects. First, *controlling system access* which involves regulating the entry points to computer systems, ensuring that only authorized users can gain entry. Second, *managing resource access* focuses on controlling the permissions and privileges granted to users for accessing and utilizing system resources. Third, *securing data in transit* i.e. protecting sensitive information while it is being transmitted between systems, preventing unauthorized interception or tampering. Finally, *protecting applications from malicious inputs* which is about implementing measures to prevent and mitigate the risks posed by malicious code or inputs that can exploit vulnerabilities in software. These four fields collectively form the foundation of computer security, safeguarding systems, resources, data, and applications from potential threats and unauthorized access.

Possible trails (pointed by the authors)

As this is a taxonomy, we have a lot of available trails that are showcased. Some of which would be : Regulating **Access Control** which can be described by the CIA triad, *confidentiality*, *integrity* and *availability*. While reading through this article, we can also learn about the **Commercial security** that is based upon the data we hold rather than how the software is built. This could be described by *Database Security* or *Role-based Access control (RBAC)*, the author also puts emphasis on the following :

Few operating systems support RBAC directly, but native access control features might be adapted for that purpose. RBAC is more often found in database management systems and workflow management systems.

This quotation allows me to introduce about the **OS security** which has also been discussed, some of the examples that have been provided could be *Discretionary Access Control (DAC)* or *Mandatory Access Control (MAC)*. The next point coming is the **Internet Security**, here the keystones will be *Secure Channels* such as SSL or TLS, *Firewalls* for access control and *Intrusion Detection System* to identify when an attack occurs. As for the **Software Security**, a few mechanisms can make it

easier to create bugs in a program that could be correct – *without a malicious hand* – such as a poor *Memory Management* that could allow e.g. a buffer overflowing or usage of dangerous patterns like *Code Injections* which result in executing code that wasn't supposed to be executed when it was first written. Sometimes, the issue can come from concurrent access which weren't initially planned, this is a *Race Condition*. But some of these issues can be partly solved using *Language-based Security*, an example of this could be the Garbage Collector in languages such as Java and C#. Speaking of languages, the following noteworthy topic is **Code-Based Access Control**. This is a user-centric approach, therefore more suitable for cases where users can be trusted. Here the two main topics explored are *Trusted Computing* and *Digital Right Management (DRM)*. Now that we explored a bit of every topic, we can delve into **Web Security** which relates a bit to every previous ones. Firstly, there's the *Code Origin Policies* which states that applets or cookies are restricted to connecting back to the domain they originated from. Secondly, *Cross-Site Scripting (XSS)* that can be described as a kind of attacks – there are subkinds discussed here, e.g. stored XSS, and DOM-based XSS – aiming to exploit the client's trust in a web server to execute malicious code. Thirdly comes *Cross-Site Request Forgery (XSRF)*, this one is about using the trust given by the session, the difference with the XSS attack is that this one goes from the client to the server whereas the previous one goes from the server to the client. Lastly some countermeasures to this kind of attacks have been explored. These include changing the execution model of web applications, implementing input sanitization techniques on servers and clients, and improving authentication mechanisms. The role of temporary secrets, message authentication codes, and client-side defenses using proxies are discussed as potential solutions. The last topic explored is the **Security Evaluation**. It focuses on assessing the security level of an IT system to provide customers with assurance. Security evaluation, certification, and accreditation are distinguished, with the former analyzing a product against generic security requirements, while the latter two involve specific customer requirements and deployment decisions. The influential Trusted Computer Systems Evaluation Criteria (Orange Book) aimed at evaluating operating systems for defense applications. However, evaluating operating systems for commercial security has faced challenges due to diverse user requirements and the evolving nature of functionality. Smart cards, on the other hand, have seen success in security evaluation, benefiting from market demand and their inherent role as security devices. Evaluating application software poses further difficulties, with customization and usability considerations for end users playing a crucial role.

Research Question

How has computer security evolved in terms of policy enforcement mechanisms and

architectural layers, considering the historical foundations, technological advancements, and emerging challenges, such as distributed systems, web-based policies, digital rights management, and security evaluation of extensible software systems?

Approach

The approach taken in this research article is to examine the evolving mechanisms and policies for enforcing computer security in IT architectures and software systems, specifically focusing on access control, resource protection, data security, and application security.

Implementation of the approach

Here there is not much to add up as it is a taxonomy. The trails are more interesting to investigate.

Results

The results highlights a few keystones. Firstly, security policies are written in specific languages, often using lists in operating systems and firewalls. However, the limited processing capability of lists restricts the sophistication of policies. Turing complete policy languages can present undecidable questions about policies. To manage complexity, layers of indirection are introduced. The goal of research on policy languages is to strike a balance between language expressiveness and formal foundations. Additionally, the concept of noninterference provides theoretical foundations for studying separation properties between components. Virtualization allows for separate virtual machines for different applications, enhancing security. Different options exist for implementing a reference monitor within a software architecture, such as execution monitors and in-line reference monitors. Application writers need awareness and tools to write secure code, as all code accepting external input can be security relevant. Technological solutions should be complemented with organizational procedures and compliance with security best practices. End users now play a crucial role in security management, including software installation, patching, and policy decisions.

Appendix B

Sheet n°2

B.1 Description of the article

Title : Analytical Review of Cybersecurity for Embedded Systems

Link : <https://ieeexplore.ieee.org/abstract/document/9300139>

Authors list :

ABDULMOHSAN ALOSEEL, HONGMEI HE, CARL SHAW, MUHAMMAD ALI KHAN

Authors affiliation :

School of Aerospace, Transport and Manufacturing (SATM), Cranfield University, Bedford MK43 0AL, U.K.;Cerberus Security Laboratories Ltd., Bristol BS34 8RB, U.K.

Name of the conference / journal : IEEE Access

Classification of the conference / journal :

Q1

Number of article citations (which source?) :

6 (Elicit), 8 (Google Scholar)

B.2 Synthesis of the article

Question

What are the key factors influencing cybersecurity for embedded systems (CSES)? And can we find a solution to globally enhance CSES ?

In this research paper, the aim is to identify the key factors influencing cybersecurity for embedded systems (CSES) and propose a Multiple Layers Feedback Framework of Embedded System Cybersecurity (MuLFESC) to address these factors. The introduction provides an overview of embedded systems, their evolution, and their increasing vulnerability to cyber-attacks. The factors influencing CSES are discussed, including components, characteristics, implementation, technical domain, security requirements, problems, connectivity protocols, attack surfaces, impact of cyber-attacks, security challenges, security solutions, and the role of manufacturers, legislators, operators, and users. The security challenges faced by embedded systems due to their limited computing capabilities and connectivity are explored, with real-world examples highlighting potential physical damage. The paper also addresses cybersecurity objectives, countermeasures, and risk management techniques for CSES. Security risk metrics and the roles of involved parties are discussed, followed by the presentation of MuLFESC, which proposes nine layers of protection and new risk assessment metrics. The conclusion summarizes the findings, emphasizes the importance of understanding the CSES landscape, and highlights the potential impact of implementing MuLFESC in addressing the identified factors influencing CSES.

Possible trails (pointed by the authors)

The authors discuss several trails related to embedded systems. They highlight the integration of computing operations into physical systems, which gave rise to embedded systems and led to significant advancements in various fields. Understanding the architecture of embedded systems is emphasized as crucial for identifying potential entry points and vulnerabilities to cyberattacks. Embedded systems consist of components such as CPU, RAM, ROM, and input/output ports, with different designs and configurations based on their specific purpose. These systems play a pivotal role within cyber-physical systems (CPS), acting as the co-design of hardware and software to steer the physical parts and perform predefined functions. However, embedded systems also have their characteristics and limitations, including low power consumption, small size, and limited computing resources, which pose challenges in implementing advanced security solutions. These insights provide a comprehensive understanding of embedded systems, their architecture, their role within CPS, and the associated challenges.

The following section describes the security risks associated with embedded systems. These systems are susceptible to various attacks that can compromise their security. Examples include exhaustion attacks that drain power resources, physical intrusion, tampering with system integrity, snooping attacks, and damage to sensors or peripherals. Embedded systems, like conventional systems, have security objectives of confidentiality, integrity, and availability. They face threats such as malware attacks, unauthorized access to stored data or cryptographic keys, and authenticity issues. The limitations of processors in implementing advanced security techniques and their vulnerability to logical and physical attacks are highlighted. The sections also mentions the increase in attack surface due to hardware and software components, the availability of low-cost attack tools, and the challenge of balancing system flexibility with security restrictions. Various security problems related to connectivity, data privacy, resource constraints, and the lack of a unified theoretical framework are discussed. The paragraph further addresses classic attacks on embedded systems, such as physical attacks, reconnaissance attacks, denial-of-service attacks, privacy breaches, cyber-crimes, and destructive attacks. The challenges of implementing comprehensive security measures in embedded systems and the need for security-by-design approaches are emphasized. This section also touches upon the diverse applications of embedded systems in IoT-enabled cyber-physical systems and the challenges posed by non-controlled environments. Finally, it discusses the importance of considering the weakest link in system security, understanding the terminology of security risks, and the challenges faced during the design process, including cost, energy optimization, and fast development cycles.

Research Question

What are the key factors influencing CSES, and how do these factors interact and shape the security landscape of CSES? Furthermore, how can a comprehensive understanding of these factors contribute to identifying security gaps and finding more effective and robust solutions for enhancing cybersecurity in CSES?

Approach

This article being an analytical review, the approach taken was to go through a lot of articles and compile their results and observations into one consequent but concise article. Of this compilation results a new framework proposition, namely, Multiple Layers Feedback Framework of Embedded System Cybersecurity (MuLFESC).

Implementation of the approach

The paper outlines 7 perspectives of risk metrics based on the security triangle

(CIA). These perspectives include considering the Multiple Layers Feedback Framework, analyzing attack methods (physical, logical/software-based, side-channel attacks), classifying security risks into different types (vulnerabilities, exposure, threat, attacks), evaluating the attack surface (hardware, software, network components), assessing security risks based on the TCP/IP model, considering the limitations of embedded systems, and taking into account the capabilities of attackers and the value of assets and the risk X based on the twelve influencing factors of MuLFESC.

These factors play a crucial role in identifying gaps and weaknesses in current countermeasures. The architecture and characteristics of embedded systems, such as their limited computing capabilities and flexibility, pose challenges for implementing advanced security solutions. Furthermore, the implementations of embedded systems across diverse technical domains, including Distributed Control Systems (DCS), Industrial Automation and Control Systems (IACS), and Cyber-Physical Systems (CPS), necessitate making the implementation market-appropriate security solutions tailored to specific applications like healthcare, communications, military, and other sectors. The connectivity of embedded systems to the Internet expands their functionality but also exposes them to remote cyber-attacks through various attack surfaces and channels. Attack surfaces, such as Wi-Fi, Bluetooth, sensors, or USB, serve as potential entry points for attackers. Understanding and addressing these attack surfaces and channels are essential for implementing secure embedded systems. Moreover, the impact of cyber-attacks extends beyond the targeted system, affecting the entire system and connected systems. This underscores the critical importance of considering the potential impacts and planning effective responses to cyber incidents, emphasizing the need for comprehensive cybersecurity measures from the design stage to response stages. Balancing the characteristics of embedded systems with cybersecurity requirements presents a significant challenge in achieving robust and secure embedded systems. These solutions, implemented across the different levels of the embedded system, should align with its nature and embrace a "Security by Design" approach. Various actors, including manufacturers, suppliers, developers, operators, and legislators, along with user behavior and awareness of social engineering attacks, all contribute to securing embedded systems. By considering and addressing these interrelated factors, the MuLFESC framework aims to establish secure Cyber-Physical Embedded Systems in interconnected domains, ensuring their protection and resilience.

The MuLFESC framework provides a comprehensive and systematic approach to building robust and secure embedded systems. It consists of nine layers, each representing different components and entities involved in the system. By considering the twelve key factors cited described earlier and addressing vulnerabilities at each layer, organizations can improve the design and implementation of their embedded systems. The framework emphasizes the importance of iterative improvements

and "Security by Design" concept, with the initial design stage being critical and influenced by feedback from the other layers. It also highlights the significance of adopting secure communication protocols, securing physical and computational components, and complying with relevant legislation and regulations. The operational phase serves as a real test for the effectiveness of security measures, with feedback used to refine the system's design and mitigate cyber risks. The MuLFESC framework offers guidance for industry 4.0 and serves as a comprehensive reference for security assessment and solutions. It helps organizations integrate security countermeasures throughout the design stage and across different technical domains, ultimately enhancing the overall security of embedded systems.

Results

In conclusion, this paper presents the results of an analytical study focused on cybersecurity for embedded systems (ESs). The research identified key deficiencies and gaps that need to be addressed to enhance the cybersecurity of ESs. It was found that the limited resources of embedded systems pose challenges in implementing compatible security solutions that align with their capabilities. However, the study highlights the importance of finding effective and efficient security measures that do not drain system resources. By analyzing the architecture of ESs and examining previous research in the field, critical factors influencing cybersecurity in ESs were identified, shaping the landscape of the cybersecurity industry for embedded systems. Additionally, a novel Multiple Layers Feedback Framework of Embedded System Cybersecurity (MuLFESC) was proposed, which can contribute to the implementation of comprehensive and effective "Security by Design" solutions by providing feedback to the design stage of ESs. By considering these identified factors, leveraging the MuLFESC framework, utilizing risk assessment metrics, and involving all relevant stakeholders, security practitioners can conduct comprehensive assessments and design application-specific security solutions, thereby enhancing the cybersecurity of Cyber-Physical Systems (CPS) and embedded systems as a whole.

Appendix C

Sheet n°3

C.1 Description of the article

Title : A lightweight supervised intrusion detection mechanism for IoT networks

Link : <https://doi.org/10.1016/j.future.2021.09.027>

Authors list : Souradip Roy, Juan Li, Bong-Jin Choi, Yan Bai

Authors affiliation :

Department of Computer Science, North Dakota State University, United States of America; Department of Statistics, North Dakota State University, United States of America; School Engineering and Technology, University of Washington Tacoma, United States of America

Name of the conference / journal :

Future Generation Computer Systems - Elsevier

Classification of the conference / journal : Q1

Number of article citations (which source?) :

38 (Google Scholar)

C.2 Synthesis of the article

Question

What are the advantages of the proposed machine learning-based intrusion detection mechanism for IoT networks?

The article focuses on the increasing security breaches associated with vulnerable IoT devices and the need for effective intrusion detection techniques in IoT environments. Traditional intrusion detection mechanisms may not be suitable for IoT due to limited device capabilities and specific protocols. To address this issue, the authors propose a novel intrusion detection model that utilizes machine learning. The model incorporates optimizations such as multicollinearity removal, sampling, and dimensionality reduction to identify crucial features for intrusion detection with fewer training data and reduced training time.

Possible trails (pointed by the authors)

The authors highlight several trails and approaches discussed in related work for intrusion detection in IoT networks. They categorize intrusion detection mechanisms into four types: signature-based, anomaly-based, specification-based, and hybrid.

Signature-based Intrusion Detection Systems (IDS) relies on predefined patterns or signatures to detect intrusions, while **anomaly-based IDS** compares activities with normal behavior profiles. The authors mention a lightweight signature-based IDS proposed by Sheikh et al. that utilizes a signature generator, pattern generator, intrusion detection engine, and output engine. Liu et al. propose an Artificial Immune System-based signature-based IDS, although its application in resource-limited IoT environments is not explained. Rebbah et al. present a signature-based IDS called IoTSecurity for IoT systems using the Cloud.

On the other hand, anomaly-based IDS, which are effective in identifying new intrusions, have also been explored. Larijani et al. propose a *random neural network-based IDS*, while Yin et al. propose a deep learning approach using a *recurrent neural network (RNN)*. Diro and Chilamkurti apply deep learning for attack detection in social IoT networks. Alom et al. use deep belief neural (DBN) networks, and Ahsan and Nygard propose a *hybrid algorithm of Convolutional Neural Network (CNN) and Long Short Term Memory (LSTM)*. Ieracitano et al. propose a *statistical analysis-driven* optimized deep learning system for intrusion detection.

To improve accuracy and reduce false predictions, different strategies have been proposed. Song et al. propose a multiple decision-based classification method, while Khan et al. propose a *hybrid-multilevel anomaly prediction* approach to handle unbalanced intrusion data.

The paper also mentions the application of various machine learning algorithms for intrusion detection using different datasets, such as the CICIDS2017 and NSL-KDD datasets. Deep neural networks (DNNs) have shown promising results compared to classical machine learning classifiers. *Reinforcement learning-based IDS* using Deep Q-Network logic and attention mechanisms has also been explored. Mechanisms like oversampling, Principal Component Analysis (PCA), and Ensemble Feature Selection (EFS) have been applied to improve the performance of IDS.

Additionally, other machine learning algorithms, such as K-Nearest Neighbor (KNN), K-MEANS clustering, Decision Tree, and Random Forest, have been investigated for intrusion detection based on the NSL-KDD dataset.

Research Question

How can a novel machine learning-based intrusion detection mechanism be developed to effectively detect cyber-attacks and anomalies in resource-constrained IoT networks, while addressing the limitations of traditional intrusion detection systems and optimizing for lightweight deployment?

Approach

The approach taken by the authors was to propose a novel intrusion detection model that utilizes machine learning to effectively detect cyber-attacks and anomalies in resource- constrained IoT networks.

Implementation of the approach

The approach described in the methodology consists of four main mechanisms: (A) removal of multicollinearity, (B) sampling, (C) dimensionality reduction, and (D) effective classification algorithm. The first mechanism involves identifying and removing multicollinearity, which is done using the Variance Inflation Factor (VIF) to measure the dependency between independent variables. The second mechanism is sampling, where undersampling and oversampling techniques are applied to address class imbalance in the dataset. The third mechanism is dimensionality reduction, which is achieved using Principal Component Analysis (PCA) to project the independent variables to a lower-dimensional space. Finally, an ensemble learning approach called B-Stacking, combining boosting and stacking algorithms, is proposed for effective classification. This approach involves using multiple base models as level-0 classifiers and a meta-model as a level-1 classifier to combine their predictions. Overall, this methodology aims to construct an efficient classifier model for detecting network attacks accurately using limited information and resources.

The evaluation of the proposed methodology involved testing it on two popular and publicly available datasets: CICIDS2017 and NSL-KDD. The CICIDS2017

dataset contains network traffic flows with both benign and common attack instances, while the NSL-KDD dataset is a revised version of the KDD'99 dataset. Various preprocessing techniques were applied to create balanced datasets for both datasets, including combining certain classes and removing records with missing values.

To evaluate the performance of the intrusion detection mechanism, several performance metrics were used, such as accuracy, precision, recall, F1 score, and area under the curve (AUC) for the receiver operating characteristic (ROC) and precision-recall (PRC) curves. The confusion matrix was utilized to calculate these metrics for each class in the datasets. The One-vs-Rest strategy was applied to extend the binary classification metrics to the multi-class classification problem.

For the CICIDS2017 dataset, the proposed B-Stacking algorithm achieved a high detection rate, correctly classifying the majority of instances across the different classes. The ROC and PRC curves covered a significant area, indicating excellent performance. Comparative analysis with other state-of-the-art techniques demonstrated the competitiveness of the B-Stacking algorithm in terms of accuracy, precision, recall, and F1 score. Additionally, the B-Stacking model showed significantly lower memory and CPU overhead compared to other algorithms, making it lightweight and suitable for resource-limited environments.

Similar experiments were conducted on the NSL-KDD dataset, and the B-Stacking algorithm exhibited strong performance. The ROC and PRC curves covered a substantial area, indicating high accuracy in classifying the different attack types. Comparative analysis with other intrusion detection systems showed that the B-Stacking algorithm achieved superior accuracy compared to most approaches. Furthermore, the B-Stacking algorithm demonstrated consistency in classifying instances across all attack types, outperforming models that combined certain classes or showed inconsistencies in detection. The B-Stacking model was lightweight and required minimal training and prediction time compared to deep learning-based approaches.

Results

This paper presented a novel IoT intrusion detection model called B-Stacking, which leverages optimized machine learning approaches to detect cyber-attacks in an IoT network. The evaluation of the model was conducted on two popular datasets: CICIDS2017 and NSL-KDD. The results showed that B-Stacking achieved a high detection rate and a low false alarm rate, outperforming most state-of-the-art techniques. The model exhibited excellent performance in classifying different attack types and demonstrated consistency across all classes. Additionally, B-Stacking was found to be lightweight, requiring less computational resources compared to other deep learning-based approaches. The study concluded that improving the security of IoT infrastructure is crucial, and the proposed B-Stacking model offers an effective

solution for IoT intrusion detection. Future work involves testing the model with different IoT datasets and applying it to real-world IoT networks.

Appendix D

Sheet n°4

D.1 Description of the article

Title : Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check

Link : <https://ieeexplore.ieee.org/abstract/document/8725488>

Authors list :

Koen Zandberg; Kaspar Schleiser; Francisco Acosta; Hannes Tschofenig; Emmanuel Baccelli

Authors affiliation :

INRIA, INFINE Team, 91120 Palaiseau Cedex, France, Freie Universität Berlin, 14195 Berlin, Germany, Arm Ltd., Cambridge CB1 9NJ, U.K.

Name of the conference / journal : IEEE Access

Classification of the conference / journal :

Q1

Number of article citations (which source?) :

52 (Elicit), 105 (Google Scholar)

D.2 Synthesis of the article

Question

What are the challenges and potential solutions for implementing secure firmware updates in constrained IoT devices?

This paper tries to answer this question by exploring the challenges and solutions for secure firmware updates in constrained IoT devices. It surveys open standards and libraries, designs a prototype, evaluates performance and security, and experiments with the IETF SUIF specification. The findings indicate the feasibility of creating a secure, standards-compliant firmware update mechanism for IoT devices, with recommendations for future work.

This paper explores the issues related to firmware updates in IoT devices and presents a structured approach to address them. At first, the paper *surveys available open standards and open-source libraries* that offer generic building blocks for enabling secure firmware updates in IoT devices. Building upon the surveyed building blocks, it will then *focus on the design and implementation of a prototype* that facilitates secure firmware updates on a wide range of constrained IoT devices, emphasizing the avoidance of proprietary mechanisms and code. The paper then moves to another important point, where the *performance of various cryptographic libraries* relevant to the context is measured and compared. Additionally, the *assessment the security properties of the implemented prototype*, providing insights into its effectiveness. Furthermore, the following section highlights the *measurement and comparison of the performance of different deployment configurations* using the prototype, along with the presentation of the first experimental evaluation of the IETF SUIF specification. Finally, the *limitations of the prototype are discussed*, culminating in the conclusion that a secure and standards-compliant firmware update mechanism can be developed for IoT devices. The paper concludes by providing recommendations for future work in this field.

Possible trails (pointed by the authors)

The possible trails pointed were as stated above the available open-source libraries (prior work) and open standards. The section II points out some of the prior work such as **Embedded Software Design on Low-End IoT Devices** with possibilities being e.g. : Exploring the use of bootloaders and memory layout considerations; Investigating different approaches for firmware updates ; Or examining the use of scripts or miniature virtual machines for software updates. Another prior kind of work explored is **Backend Framework** where discussions and highlightments about backend architecture and securing the supply chain of IoT software, Highlighting authentication, integrity protection, firmware encryption, and signing mechanisms. They also discussed existing frameworks and standards like SUIF,

FOSE, TUF, Uptane, ASSURED, and CHAINIAC. Lastly, the **Network Transport** topic was brought down. Here, different approaches for disseminating software updates through the network have been explored such as protocols optimized for multi-hop, low-power wireless networks. The transport mechanism for updating trusted applications running in trusted execution environments (TEEs) like Arm TrustZone and Intel SGX were also discussed.

As for the section III, some open standards were discussed. The first topic that comes up is **Cryptographic Algorithms** where the use of state-of-the-art cryptographic algorithms, such as Elliptic Curve Cryptography (ECC) and Ed25519, for securing firmware updates has been explored. Authors also point that there are some research currently going on the standardization efforts in the post-quantum crypto area. The second topic getting a highlight is **Firmware Metadata** and the work of the IETF SUIF working group in standardizing a format for describing firmware updates, including the use of manifest and metadata. We are then walked through some **Standards for IoT Firmware Transport** where different transport protocols, including Device Firmware Update (DFU) over specific low-power MAC layer technologies and bus technologies, the use of CoAP over UDP and CoAP over TCP/TLS for firmware updates over multiple hops or heterogeneous low-power networks have been examined. There are some **Standards for Remote IoT Device Management** that also got examined such as the Lightweight Machine-to-Machine (LwM2M) protocol and its use of CoAP and DTLS for data transfer and remote management of IoT devices. The CoAP Management Interface (CoMI) and the Open Connectivity Foundation (OCF) as alternative standards for IoT device management also found some highlight. Finally, here are some **De Facto Standard IoT Operating Systems**, as identified by the authors. The discussion focuses on the development of specialized IoT operating systems, including open source options such as RIOT, Zephyr, Mbed OS, MyNewt, and Tock, along with commercial options like µC/OS and FreeRTOS. It also addresses the limitations of using off-the-shelf open source operating systems like Linux for low-end IoT devices.

Research Question

What are *as of 2019* the available open standards and open source libraries that can be utilized to enable secure firmware updates for constrained IoT devices, and how can these building blocks be leveraged to design a prototype that ensures secure firmware updates for a variety of constrained IoT devices while adhering to open standards?

Approach

The approach focuses on exploring available options, designing a prototype, evaluating its security and performance, and experimenting with relevant standards. It

is described and discussed more extensively in sections IV and V (discussing the prototype and some cryptographic matters) and some assessments on this approach are made in sections VI and VII (security and performance wise). The section VIII also tries to draw some results out of the approach taken.

Implementation of the approach

The approach has been implemented by designing a prototype that addresses the functionality required for IoT device firmware updates. The prototype allows an authorized IoT software maintainer to perform various tasks, including producing integrity-protected and authenticated firmware updates, triggering the device to fetch and verify firmware images, delegating authorization to another maintainer, and reconfiguring cryptographic algorithms if needed. The prototype focuses on simplicity, open-source software, and compatibility with different IoT hardware vendors. It utilizes standard building blocks such as the IETF SUIF manifest for firmware metadata format, 6LoWPAN, IPv6, and CoAP transport stack, LwM2M IoT device management solution, and digital signature algorithms based on ed25519 and ECD-SA/P256r1. The RIOT operating system is used, but the prototype can be adapted to other real-time operating systems. It consists of components and provides a functional overview, starting with IoT device commissioning, where a minimalistic bootloader, firmware image slots, and a firmware update module are implemented. The RIOT build system is enhanced to allow building and flashing of the bootloader and initial firmware. It also enables producing and uploading authorized firmware updates. The build system allows a maintainer to build a new firmware image and produce signed metadata. The firmware and metadata are then uploaded to the IoT software update server through an HTTP-based API. The firmware update module retrieves the firmware image and manifest from the update server, verifies them, and stores the firmware image in flash memory. It will also perform necessary checks and launch a reboot if the update is valid. Firmware updates can be scheduled periodically or on demand, and they can be pushed or pulled from the device. The prototype supports multi-threading to prevent blocking during signature verification. However, advanced scheduling for firmware updates is not a focus, and the emphasis is on embedded system characteristics and constraints of constrained IoT devices. Lifecycle management is facilitated through trust anchor updates, allowing delegation of firmware update authorization and crypto agility. Different configurations are possible based on the prototype, including baseline (without firmware update functionality), Basic-OTA (over-the-air updates), IPv6-OTA (using an IPv6-compliant network stack), SUIF-OTA (following the IETF SUIF manifest), and LwM2M-OTA (using LwM2M v1.0).

The impact of cryptography on memory and power budgets is significant in such a system. Measurements show that cryptography accounts for 50% of the

memory budget and 68% of the total time spent on the firmware update process. Choosing appropriate cryptographic algorithms and libraries is crucial to strike a balance between size and speed. The authors have considered algorithms such as ed25519 and NIST P256r1 curve signatures. Several libraries are reviewed, including HACLS*, TweetNaCl, NaCl, C25519, Monocypher, WolfSSL, TinyCrypt, and Mbed TLS. They assess the libraries based on factors like code size, memory consumption, speed, and configurability. Among the libraries, the C25519 library with ed25519 signatures is chosen for the prototype firmware update due to its favorable speed/size compromise.

Overall, the implemented approach focuses on simplicity, open standards, and compatibility, providing a foundation for secure and efficient firmware updates in IoT devices.

Results

This article provides extensive documentation on the assessments of the prototype and how it could be enhanced.

Firstly, the **Security Assessment**. This assessment comes from eight different points. The first one being *Tampered Firmware*, every configuration had an integrity check of the firmware, its metadata and the author to avoid an update from an attacker. We then encounter *Firmware Replay* which is about avoiding the reflashing of an older firmware by using a sequence number. The next point that got assessed was the *Offline Device Attack* where only the LwM2M-OTA configuration had the possibility to mitigate this threat using time information. Furthermore comes *Firmware Mismatch*, this is about enforcing a firmware to a device and a given configuration, only the IPv6-OTA didn't provide a way to mitigate this threat. Another aspect that was taken in account was *Flash Memory Location Mismatch* which was successfully passed by every configuration by specifying the intended memory location of the firmware update. Moving forward we want to avoid *Unexpected Precursor Image* which has been passed only by SUIT-OTA and LwM2M-OTA configurations because they enable specifying the precursor software that must be installed before the update, allowing modular/incremental updates. Additionally, it was noted that there was no configuration that could *by default* mitigate the threat of *Reverse Engineering* through a vulnerability analysis because of the fact that none was protected against eavesdropping end-to-end, however, using (D)TLS in the SUIT-OTA or LwM2M-OTA configurations can protect the firmware image during transmission over the network. Finally, the *Resource Exhaustion* was brought up, overservations are that IPv6-OTA does not mitigate this threat, SUIT-OTA lowers the impact by verifying the manifest before downloading the firmware image and LwM2M-OTA adds an additional layer of defense by processing manifests conveyed via the device management infrastructure.

Secondly, the **Experimental Performance Evaluation**. This evaluation starts off with a description. Three different kinds of microcontrollers from different vendors were used : Atmel SAMR21, STM32F103REY, and Nordic nrf52840. All of these are Arm Cortex M microcontrollers. As of the metrics evaluated, memory measurements (RAM and flash size) and CPU performance measurements were the ones considered relevant. Here, two costs got evaluated, the *OTA update functionality's* and the *criptography's* ones. The first one is divided in two subparts, the cost of OTA (The flash memory overhead for OTA functionality was compared between the Baseline configuration and IPv6-OTA.) where the relative overhead in flash memory footprint was 137%, while the RAM requirements increased by 3 kB. The second supbart was the cost of Standard-Compliance for OTA. The use of standard-compliant specifications increased memory footprint, but the overhead per image was small (around 10% compared to the Baseline scenario). Lastly, the cost of Cryptography evaluation took different cryptographic libraries in account, evaluating their flash memory, RAM usage, and speed. C25519 was found to have a good compromise in terms of performance for ed25519 signatures. TinyCrypt's P256r1-based ECDSA outperformed most ed25519 implementations.

Lastly but most importantly, the paper suggests us the different keypoints that came out of this experiment in the "**Discussion: Going Forward**" section. It results in 8 keypoints which are (I quote) : *State-of-the-art crypto is doable on IoT devices, but it takes a toll, Making the firmware update reliable is key., Use delegation capabilities with care, Shielding against resource exhaustion and bestbefore vulnerabilities, Real-world requirements make firmware updates complex, IoT software updates are not just for critical infrastructure, Firmware update security is more than network security* and finally that *Something is better than nothing*.

Summary table

Category	Computer Security	Analytical Review of Cybersecurity for Embedded Systems	A lightweight supervised intrusion detection mechanism for IoT networks	Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check
Article type	State-of-the-art	Review	Research article	Research article
Goal described by the question	Describe the evolving mechanisms and policies for enforcing computer security	Globally enhance Cybersecurity for Embedded Systems	Find what's the worth of Machine Learning in intrusion detection	Update a firmware without risks
Target	General purpose computers	Any kind of Embedded System	Embedded systems (network)	Embedded systems
Outcome	State-of-the-art knowledge on Conventional systems security	State-of-the-art knowledge on Embedded Systems security + MuLFESC framework	B-Stacking model	A state-of-the-art method to update an embedded system's firmware
Limitations	"Technology on its own is rarely sufficient for solving security problems"	characteristics of embedded systems	Limited capacities of embedded systems	There are still big struggles against Reverse Engineering and Resource Exhaustion. The cost of cryptography was also brought up.