# Comprehensive security analyses of a toys-to-life game and possible countermeasures

*Master thesis - July 2016*

---

Radboud University


---

*Author*
Kevin Valk
Radboud University
kevin@kevinvalk.nl

*Supervisor*
Robert Leyland

*Supervisor*
Lejla Batina
Radboud University
lejla@cs.ru.nl

*Second reader*
Eric Poll
Radboud University
e.poll@cs.ru.nl

**Abstract**

This thesis aims at modeling important attacks on a toys-to-life game using attack-defense trees. Using these trees, different practical attacks are executed to verify the current countermeasures and find possible new exploits. One critical exploit led to a binary dump of the firmware, which made it possible to reverse the key derivation algorithm. This led to breaking the security layer that protected the toys. With the key derivation algorithm known, toys could be forged for under a dollar and made it possible to search for unreleased toys and variants.

Given the possible attacks, numerous countermeasures are presented to protect games against these attacks and improve general security. The foremost countermeasure is the addition of digital signatures to the toys. This countermeasure makes it infeasible to forge toys. However, this does not stop 1-on-1 clones, but concepts are explored to protect against 1-on-1 clones in the future using Physical Unclonable Function (PUF).

# Contents


**1 Introduction** **4**

**2 Background** **5**
2.1 Attack Trees . . . 5
2.1.1 Basic attack-defense trees . . . 5
2.1.2 Quantitative analysis . . . 6
2.2 Public-key cryptography . . . 6
2.3 Near Field Communication . . . 7
2.3.1 MIFARE Classic . . . 7
2.3.2 MIFARE Classic knockoff tags . . . 8

**3 Threat model** **10**

**4 Attacks** **14**
4.1 Proxmark III . . . 14
4.1.1 Key A analyses . . . 14
4.1.2 Impact . . . 16
4.1.3 Conclusion . . . 16
4.2 Reverse engineering the reader . . . 17
4.2.1 Preliminary research . . . 17
4.2.2 Obtaining binary firmware from the reader . . . 17
4.2.3 Cryptanalysis . . . 18
4.2.4 Impact . . . 18
4.2.5 Recommendations . . . 19
4.2.6 Conclusion . . . 19
4.3 MaxLander . . . 20
4.3.1 Technical Specification . . . 20
4.3.2 USB protocol . . . 21
4.3.3 Similarities to PowerSaves for characters . . . 22
4.3.4 Impact . . . 22
4.3.5 Conclusion . . . 23

**5 Countermeasures** **25**
5.1 Counterfeiting toys . . . 25
5.1.1 Signature scheme concept . . . 25
5.1.2 Preventing cloning . . . 27
5.1.3 Production process . . . 28
5.1.4 Prototype one . . . 28
5.1.5 Prototype two . . . 31
5.1.6 Key Management . . . 34
5.1.7 Attack analysis . . . 36
5.1.8 Conclusion . . . 36
5.2 Hardening the reader . . . 40
5.2.1 Authentication of the readers . . . 40
5.2.2 Protocol verification through fuzzing and side-channel analysis . . . 42
5.3 Hardening game code . . . 47
5.3.1 Toy fuzzing . . . 47

**6 Conclusions & future work** **50**
6.1 Recommendation for future work . . . 51

**References** **52**

**Glossary** **55**

**Appendices** **56**

**A MaxLander USB Protocol** **56**

**B MaxLander and PowerSaves for characters PCB** **57**

**C Detect genuine reader script** **58**

**D Key derivation function** **59**
D.1 Get Key . . . . . 59
D.2 Accumulate . . . . . 60
D.3 Shift . . . . . 61

**E The reader printed circuit board** **62**

# 1 Introduction

There are many Radio-Frequency Identification (RFID) systems present in our lives, ranging from payment and public transport systems to door locks and (real life) gaming systems. These systems are convenient to the consumers or to their owners, but most of the time, the security of these systems have shortcomings [10, 11, 23, 44].

A new concept in the RFID field is a gaming genre called toys-to-life. The idea is to sell physical attractive smart toys that are also playable in some way in a computer game. Obviously different technologies could be leveraged for communication between the physical toy and the game but as of writing all toys-to-life concept use RFID. The genre generally uses RFID technology, and given the current limitations on the security of this technology, there may be risks of piracy.

Preliminary research into the genre shows that certain products use MIFARE Classic 1K and numerous attacks currently exists against MIFARE Classic systems [10, 11, 44]. Such system often consists out of RFID tags, RFID reader and a gaming system, which can be connected to the reader. The tags are required to play such games and more often than not, there is no way to play the game without tags. Moreover, often games are designed in such a way that the more toys one has the more areas are accessible. The toys containing these RFID tags generally cost around $12. This means that most of the revenue comes from selling these RFID tags. It should be hard to fool the game into accepting fake or cloned RFID tags.

More research is needed to uncover other possible security issues in toys-to-life genre games that use the MIFARE Classic RFID technology. The goal of this research is not just to disclose the security issues, but also to find possible countermeasures. This will not only be beneficially to the games itself, but can also be used as a guide to common mistakes and good practical countermeasures in toys-to-life games.

To that end the following research question has to be answered: *To what extent can attacks against the hardware of toys-to-life games be prevented?*, to fully answer this questions multiple subquestions are formulated.

- *What are the possible attack-defense trees?*
- *What attack trees can be turned into practical attacks?*
- *What is the impact of each attack?*
- *How can each attack be countered?*

The contribution of this paper is mainly a novel way of using a signature scheme on top of insecure storage to be able to detect digital forgeries of toys used in the toys-to-life games. The system does not protect against 1-on-1 clones and methods of increasing security against cloning are shown. Finally a novel system is presented using fuzzing together with side-channel analysis to do automated testing of hardware to increase the security and robustness of a device.

This thesis is structured in five main sections beginning with Section 2 which briefly clarifies some subjects that are required to be able to understand this thesis. Section 3 defines the threat model using attack-defense trees that model ways to obtain playable toys. These attack-defense trees are verified in practice in Section 4 to see if attacks can be mounted and how difficult these would be. Given the found weaknesses of the system, different countermeasures are presented in Section 5 that should prevent some attacks and improve the general stability and security of the total system. Section 6 concludes the thesis by giving an answer to the research question and possible future work is described in Section 6.1.

# 2 Background

To understand several concepts in this thesis, some background is required in Attack-Defense Trees, Near Field Communication (NFC) and asymmetric cryptography. This section gives brief explanations on these topics as a convenience for the reader to quickly consume the required background information. Each background section contains numerous references that can be consulted for more detailed explanations on the described topics.

## 2.1 Attack Trees

Attack Trees are informally introduced by Bruce Schneier and are more formally introduced by Mauw and Oostdijk [39, 27]. A practical guide to Threat Modeling using attack trees is written by Saini, Duan and Paruchuri [37].

Kordy, Kordy, Mauw and Schweitzer released a tool called ADTool to create attack-defense trees [22]. The attack-defense trees can be used to create tree-based security models integrating attack and defense components. Moreover, the system can do quantitative analysis of the attack-defense tree using different attribute domains from different point of views. When required, new domains can be added to model specific cases when the default models are insufficient. This makes it possible to thoroughly model attacks and defenses in order to find weak points in the system and to propose countermeasures to increase the overall level of security of the system.

### 2.1.1 Basic attack-defense trees

An example attack-defense tree from the ADTool paper can seen in Figure 1. The attack-defense tree models an oversimplified attack on a server. The root node names the attack and each subsequent node is a requirement to achieve the attack. The attack on the server can be achieved in three independent ways and is depicted with three sub-trees, “Insider Attack”, “Steal Server” or “Outsider Attack”. A node with independent outgoing lines is called an “or” node. There are no sub-trees to the “Steal Server” node, which means that it satisfies the requirement of the parent node. However, to pull off a insider attack, both the sub-trees have to be satisfied. A node that requires all sub-trees to be achieved (this is called an “and” node) is depicted with outgoing lines with a horizontal line connecting all outgoing lines together.

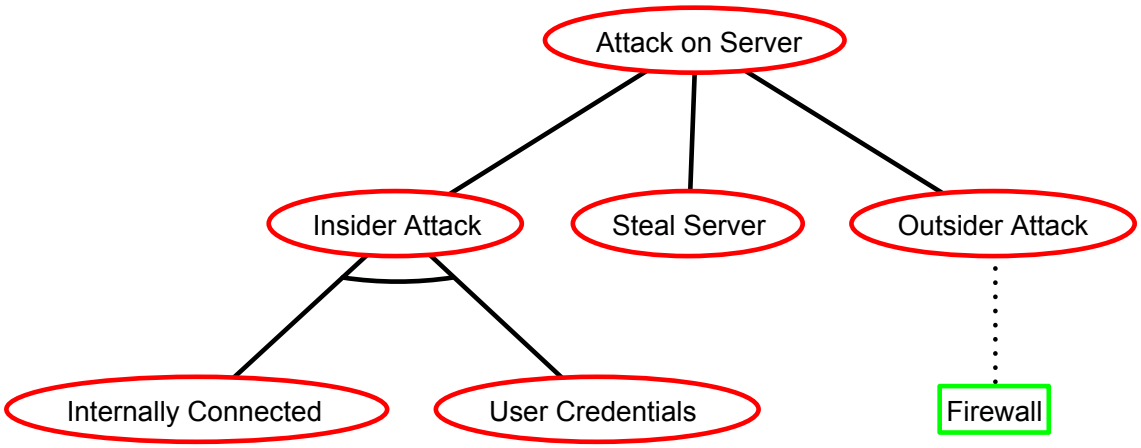


Figure 1: Sample attack-defense tree taken from ADTool [22]

The last option to attack a server is an “Outsider Attack”. However, this attack is thwarted by a firewall countermeasure shown in the green square. Green squares depict countermeasures and are called “counter” nodes. Obviously using the red attack ellipses, an attack on the firewall countermeasure could be mounted and this could also be modeled in the attack-defense tree.

### 2.1.2 Quantitative analysis

While the attack-defense tree shown in Figure 1 is a good visual assistant to pin-point weak spots in the system, it does not make it obvious where these points are. ADTools solves this by the possibility of adding attribute domains to an attack-defense tree. This is called an quantitative analysis and enables more detailed ways to find out exactly how easy or cheap an attack is. The attack-defense tree shown in Figure 2 shows the attribute domain "minimal cost for the proponent" using the same attack-defense tree as used in Figure 1.

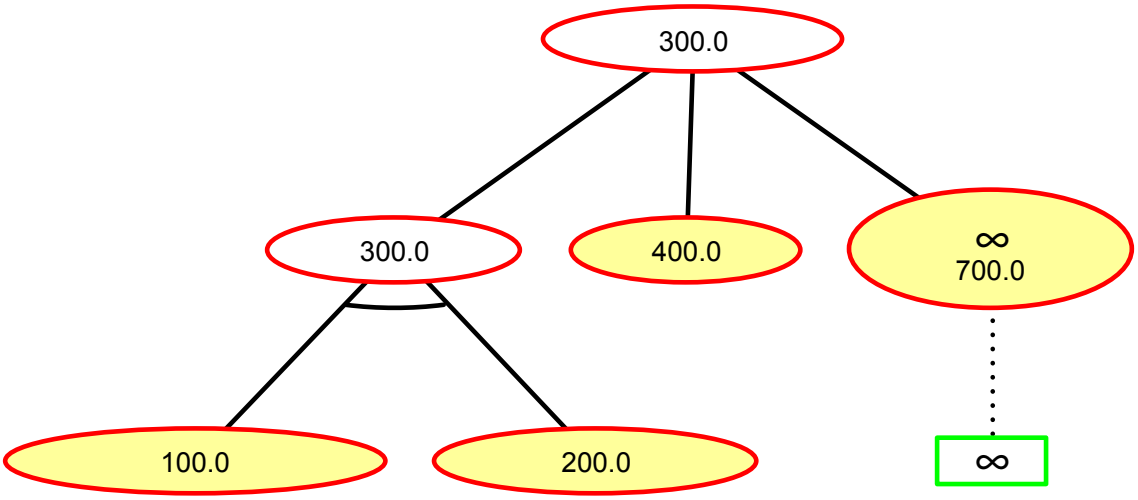


Figure 2: Sample attack-defense tree showing the minimum cost for the attacker based on the attack-defense tree shown in Figure 1, taken from ADTool [22]

The "minimal cost for the proponent" has a value domain of $\mathbb{R} + \cup\{\infty\}$ and all leaf nodes can be set to a real number or infinity. ADTool than resolves the value for all higher nodes through predefined rules specific to this domain which gives a minimum cost of 300.0 for the attack on a server in this model. In the rightmost node in Figure 2 two values are shown, infinity and 700.0 that depicts the presence of a countermeasure. The value of the root of the countermeasure is the final cost of the parent attack node and in this case is infinity because there is no attack possible against this firewall.

In larger attack-defense threes it is hard to mentally match the values and labels of the two different trees. In the rest of this paper, attack-defense trees will include both the labels and the attribute domain.

## 2.2 Public-key cryptography

Secret communication between two parties has been a long time necessity. However, with symmetric cryptography, this requires to share a common secret before hand. Sharing this secret is not trivial and in the information age, sharing this secret becomes increasingly difficult. In the eighties, multiple concepts were discovered that made it possible to communicate secretly without needing to share a common secret before hand. Nowadays this is called public-key cryptography or asymmetric cryptography [38].

While symmetric cryptography requires one secret to be known to two parties, asymmetric cryptography requires a secret only known to the owner of the key-pair and a public-key that should be publicly available. In the case of secure communication, users can use the public-key to encrypt the message and only the user holding the private key can decrypt the message. The public-key is closely related to the private-key, but for systems like RSA and Elliptic Curve Cryptography (ECC) it is computational infeasible to calculate the private key from a public-key. It is possible to calculate the public-key from the private-key and this is achieved by using one way problems like integer factorization and discrete logarithm. This makes the public-key harmless as an attack vector against the system and is the single biggest strength of asymmetric cryptography.

When decrypting an encrypted message in an asymmetric cryptographic system the private-key is required, when using a signature scheme the roles are reversed. The signature over a message is generated with the secret private-key and using the corresponding public-key the signature can be validated which can prove authenticity, non-repudiation and integrity [35].

## 2.3 Near Field Communication

NFC is a set of standards, protocols and products that encompass wireless communication using magnetic field modulation over approximately 10 centimeters between two devices. NFC uses point-to-point communication between two devices where each device can be either active or passive. Active devices like readers have their own power source and can generate the magnetic field that will carry information. Passive devices do not have their own power sources and instead harvest power from the active device through induction. Communication between two active devices is also possible and is called peer-to-peer communication. Most NFC products use one or more parts of the ISO 14443 standards to communicate contactless.

ISO 14443 is a standard that standardizes contactless communication between two devices in the 13.56 MHz range. The standard consists out of four parts and each part describes one layer, going from the transmission protocol to the physical characteristics of the communication. The parts are as follows:

1. Physical characteristics [21]
2. Radio frequency power and signal interface [19]
3. Initialization and anticollision [20]
4. Transmission protocol [18]

Moreover, parts 2 and 3 make a distinction between types A and B. These two types differ slightly in modulation, coding schemes and protocol initialization. This difference also makes type A and B incompatible with each other.

### 2.3.1 MIFARE Classic

The MIFARE Classic Integrated Circuit (IC) is a type A contactless passive memory storage device using the ISO 14443 standard. However, it replaces the transmission protocol from the ISO 14443 A-4 with a custom MIFARE protocol. The MIFARE Classic ICs comes in two variants, MIFARE Classic 1K (S50) and MIFARE Classic 4K (S70), respectively containing 1024 and 4096 bytes of memory. Not all of this memory can be used because one block is reserved for manufacturer information and each sector contains one block that defines the security of that specific sector. This brings the usable memory for the MIFARE Classic 1K and MIFARE Classic 4K respectively to 752 and 3440 bytes of usable memory. Table 1 shows the memory layout for MIFARE Classic 1K and Table 2 for MIFARE Classic 4K.

The security blocks contain space for up to two keys and an Access Control List (ACL). The ACL can be used to define the access to a specific blocks per key and can be read-only, write-only, read-write or no access. The second key can be used for a different ACL, making it possible to, for example, use key A as a master key and key B as a read-only key. If key B is not used, it can be defined as a data field resulting in the increase of the total usable size for MIFARE Classic 1K and MIFARE Classic 4K respectively to 848 and 3680.

Another feature is the always read-only manufacturer block zero. This block is initialized at manufacturing and contains an identifier, some extra type identification and some other fields. Tables 1 and 2 show a four bytes identifier but both 1K and 4K are available with a seven byte identifier. To fit the seven byte identifiers on the tag, the “ManufacturerData” field size is decreased by three bytes and all fields starting with “LRC” are moved three bytes to the right.

The identifier is often used to authorize specific tags access to resources making it paramount that others can not change the identifier of a tag after manufacturing. Disabling writes on block zero regardless of the ACL prevents this. However, by being able to copy an identifier, whole security systems can be defeated. This gave rise to knockoff versions of the MIFARE Classic that have a writable block zero after manufacturing.

#### 2.3.1.1 Attacks

A lot of research has gone into the security of RFID systems [36, 28, 46, 23, 10, 11, 44, 45]. However, in the case of MIFARE Classic there are two major attacks called the Darkside and NESTED attacks [10]. With these attacks and an exploitable MIFARE Classic tag, it is possible to do a tag only attack to find all the authentication keys in around 25 seconds. These attacks are thwarted by improved tags produced by NXP that do not contain flaws in the Pseudo-Random Number Generator (PRNG). However, by collecting enough nonces, it is possible to brute force the key space within minutes using an offline attack running on a modern computer [43]. This is possible and impossible to prevent because of the weaknesses in the CRYPTO1 cipher.

### 2.3.2 MIFARE Classic knockoff tags

When requiring a writable block zero MIFARE Classic tag, there are mainly two different variants available. The first variant has a build in backdoor, that when enabled, ignores the security for all subsequent commands until the device is power cycled (taken out of the Radio Frequency (RF) field). The backdoor is enabled by sending two standard MIFARE commands (section 9.1 in the datasheet [31, 32]) to the chip and one unknown command that is not defined in the MIFARE protocol.

- **Halt:** `0x50 0x00`
- **Personalize UID Usage:** `0x40`
- **Unknown:** `0x43`

All security is completely bypassed after sending this sequence, even making block zero writable. This makes it possible to make 1-on-1 clones. However, this unofficial tag only exists in 1K variant and does not come in the 4K variant.

The other unofficial variant does not have a backdoor, but treats block zero just like any other block. If the ACL of sector zero defines block zero as writable, it enables the user to change the contents of block zero including the identifier of the tag. Just like the backdoor variant enables users to make 1-on-1 clones. However, this is significantly less powerful than the backdoor because after locking the ACL it can never be unlocked. Given that some products verify that the ACL is indeed locked, makes this variant often only usable once.

| | Block | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sector 0** | **0** | Non-UID | | | | LRC | SAK | ATQA | | Manufacturer data | | | | | | | |
| | **1** | | | | | | | | | | | | | | | | |
| | **2** | | | | | | | | | | | | | | | | |
| | **3** | Key A | | | | | | ACL | | | | Key B / User data | | | | | |
| **Sector 1** | **4** | | | | | | | | | | | | | | | | |
| | **5** | | | | | | | | | | | | | | | | |
| | **6** | | | | | | | | | | | | | | | | |
| | **7** | Key A | | | | | | ACL | | | | Key B / User data | | | | | |
| ⋮ | | ⋮ | | | | | | | | | | | | | | | |
| **Sector 15** | **60** | | | | | | | | | | | | | | | | |
| | **61** | | | | | | | | | | | | | | | | |
| | **62** | | | | | | | | | | | | | | | | |
| | **63** | Key A | | | | | | ACL | | | | Key B / User data | | | | | |

Table 1: MIFARE Classic 1K (S50) memory layout [31].

| | Block | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sector 0** | **0** | Non-UID | | | | LRC | SAK | ATQA | | Manufacturer data | | | | | | | |
| | **1** | | | | | | | | | | | | | | | | |
| | **2** | | | | | | | | | | | | | | | | |
| | **3** | Key A | | | | | | ACL | | | | Key B / User data | | | | | |
| ⋮ | | ⋮ | | | | | | | | | | | | | | | |
| **Sector 31** | **124** | | | | | | | | | | | | | | | | |
| | **125** | | | | | | | | | | | | | | | | |
| | **126** | | | | | | | | | | | | | | | | |
| | **127** | Key A | | | | | | ACL | | | | Key B / User data | | | | | |
| **Sector 32** | **128** | | | | | | | | | | | | | | | | |
| | **129** | | | | | | | | | | | | | | | | |
| | **130** | | | | | | | | | | | | | | | | |
| | ⋮ | ⋮ | | | | | | | | | | | | | | | |
| | **141** | | | | | | | | | | | | | | | | |
| | **142** | | | | | | | | | | | | | | | | |
| | **143** | Key A | | | | | | ACL | | | | Key B / User data | | | | | |
| ⋮ | | ⋮ | | | | | | | | | | | | | | | |
| **Sector 39** | **240** | | | | | | | | | | | | | | | | |
| | **241** | | | | | | | | | | | | | | | | |
| | **242** | | | | | | | | | | | | | | | | |
| | ⋮ | ⋮ | | | | | | | | | | | | | | | |
| | **253** | | | | | | | | | | | | | | | | |
| | **254** | | | | | | | | | | | | | | | | |
| | **255** | Key A | | | | | | ACL | | | | Key B / User data | | | | | |

Table 2: MIFARE Classic 4K (S70) memory layout [32].

# 3 Threat model

According to Juniper Research, smart toys revenues hit 2.8 billion at the end of 2015 [41]. This shows that games that fall in the toys-to-life category make a lot of revenue from selling smart toys. The same research also notes that smart toys have a rather limited level of security. Because this minimal level of security, it is currently possible to create pirated smart toys for all games in the toys-to-life genre. Moreover, it is possible to sell pirated smart toys to people and have them requiring no specific technical knowledge or hardware alterations making it completely plug and play. This makes it not only attractive from a commercial point of view by the illegal seller, but also for potential buyers as one can obtain playable smart toys for significantly less costs. To that end this research will mainly focus on the security of smart toys.

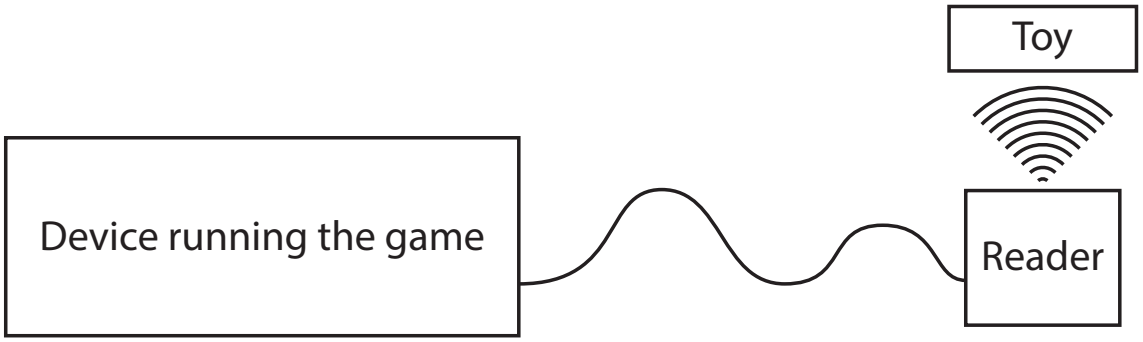


Figure 3: Architecture of current generation toys-to-life games. In some cases the reader is integrated in the device running the game.

The current generation toys-to-life games have a rather simple architecture and is shown in Figure 3. In the case of games, this is no different and the technology used for wireless communication between toy and reader is NFC, more specifically MIFARE Classic. Each toy has a MIFARE Classic tag that contains data that defines that specific toy. The reader can read the data from the tag so it can be send back to device running the game. The reader is called a reader and is connected through Universal Serial Bus (USB) with the console. Creating a playable toy that does not require changes to the console of the user can be achieved mainly in two ways. Creating an NFC tag that can be read by the reader and is accepted by games. Another way is to create a device that emulates a reader to send valid emulated toy data to the console.

To thoroughly explore the different possible attacks, requirements and countermeasures, the attack-defense tree modeling methodology is used. This makes it possible to visually model the different attacks so that numerous ways of achieving the attacks should be found. Moreover, using the quantitative analyses of the attack-defense trees, the final difficulty level of the attacks can be calculated.

Emulating a toy using reader emulation is significantly easier than cloning or forging tags as can be seen in the attack-defense tree given in Figure 6. However, some hardware platforms use an authentication scheme for all third party peripherals that will prevent this attack from working on these two consoles. This makes it significantly less attractive for an attackers as a hypothetical product will only work on $\frac{2}{3}$ of all the gaming platforms. However, if cloning and forging of tags appear to be to difficult, this is the next best attack an attacker can execute.

Cloning a MIFARE Classic tag sounds easier than forging one, but as can be seen in the attack-defense trees given in Figures 4 and 5 they both require the MIFARE Classic authentication keys. The attack-defense tree for learning the key derivation algorithm is given in Figure 7 and is set to extreme because of the usage of tags that can not be attacked with the Darkside and NESTED attacks [10]. Obtaining the algorithm or a set of keys is extreme given that all countermeasures are in place and working correctly. However, if a countermeasure is not present or not correctly implemented the difficultly level will drop significantly making the difficulty level medium for cloning and hard for forging.

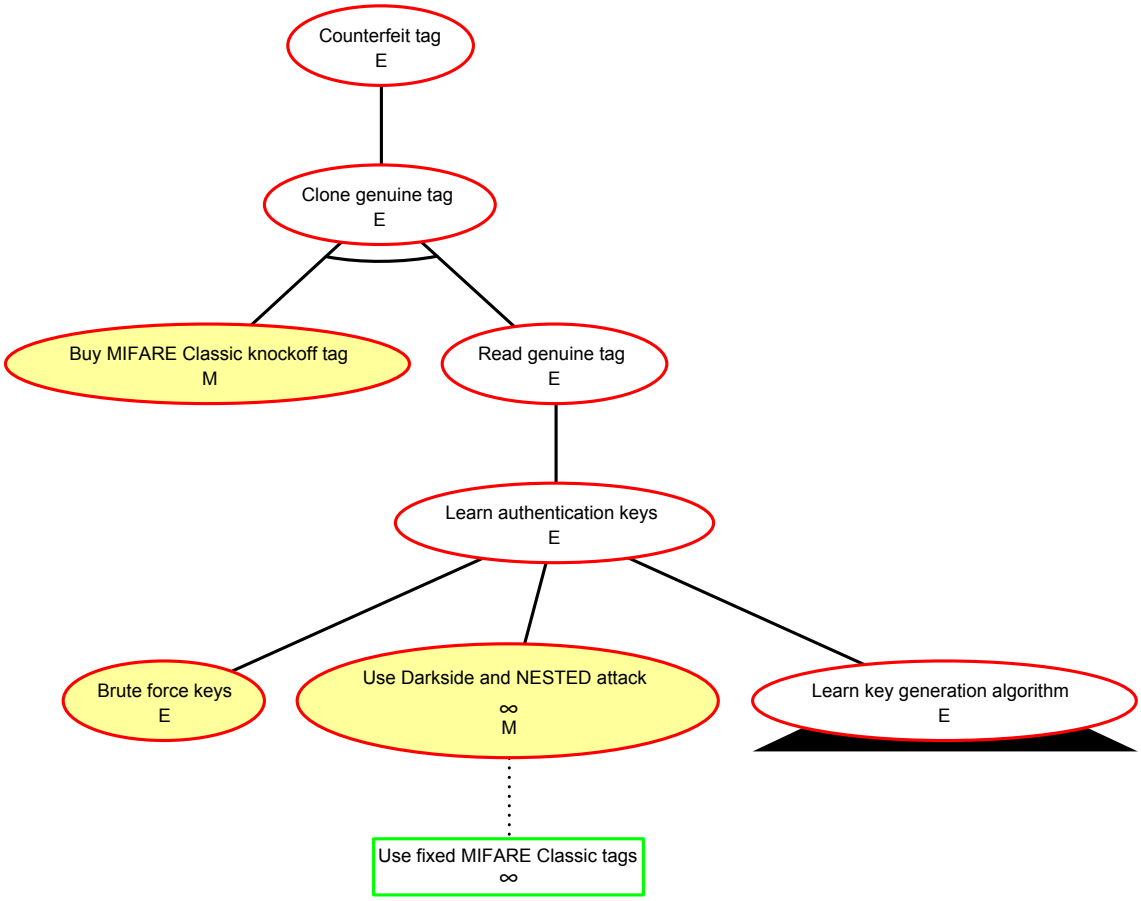


Figure 4: Attack-defense tree to clone a genuine tag using the minimal difficulty level attribute domain {**L**ow, **M**edium, **H**igh, **E**xtreme, $\infty$}, see Figure 7 for the Learn key generation algorithm tree.

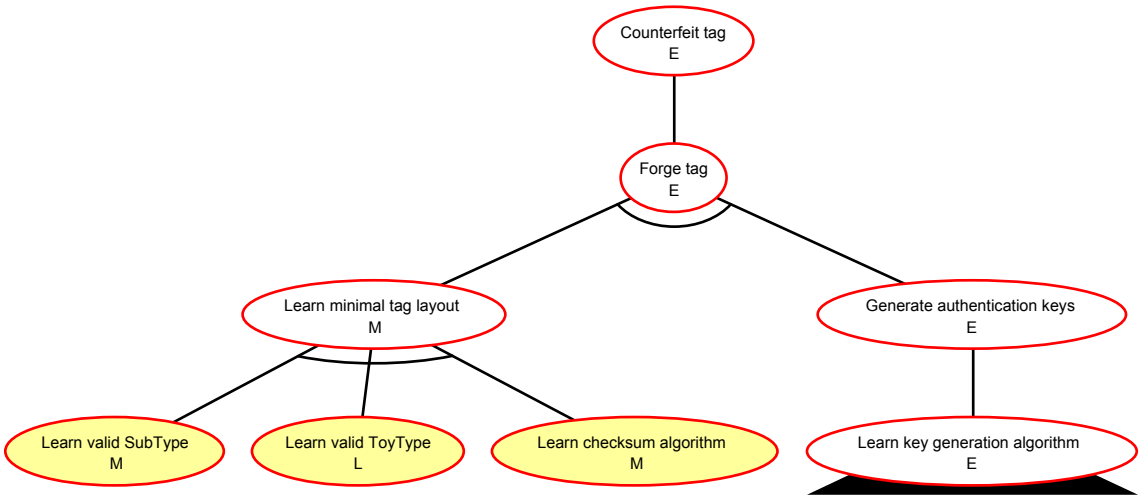


Figure 5: Attack-defense tree to forge a tag using the minimal difficulty level attribute domain {**L**ow, **M**edium, **H**igh, **E**xtreme, $\infty$}, see Figure 7 for the Learn key generation algorithm tree.

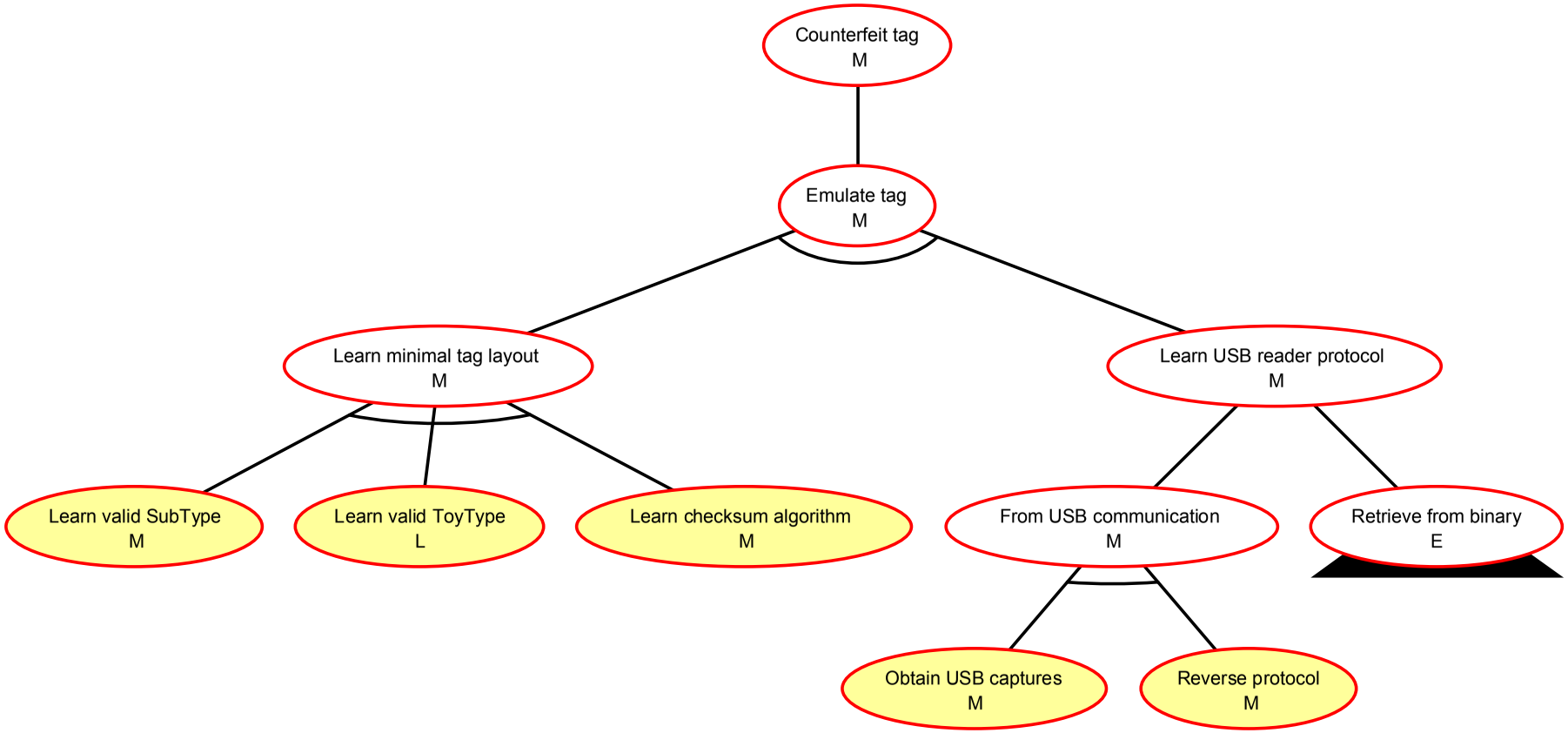


Figure 6: Attack-defense tree to emulate a tag through a custom USB device using the minimal difficulty level attribute domain {**L**ow, **M**edium, **H**igh, **E**xtreme, $\infty$}. The "Retrieve from binary" node is analog to the one given in Figure 7

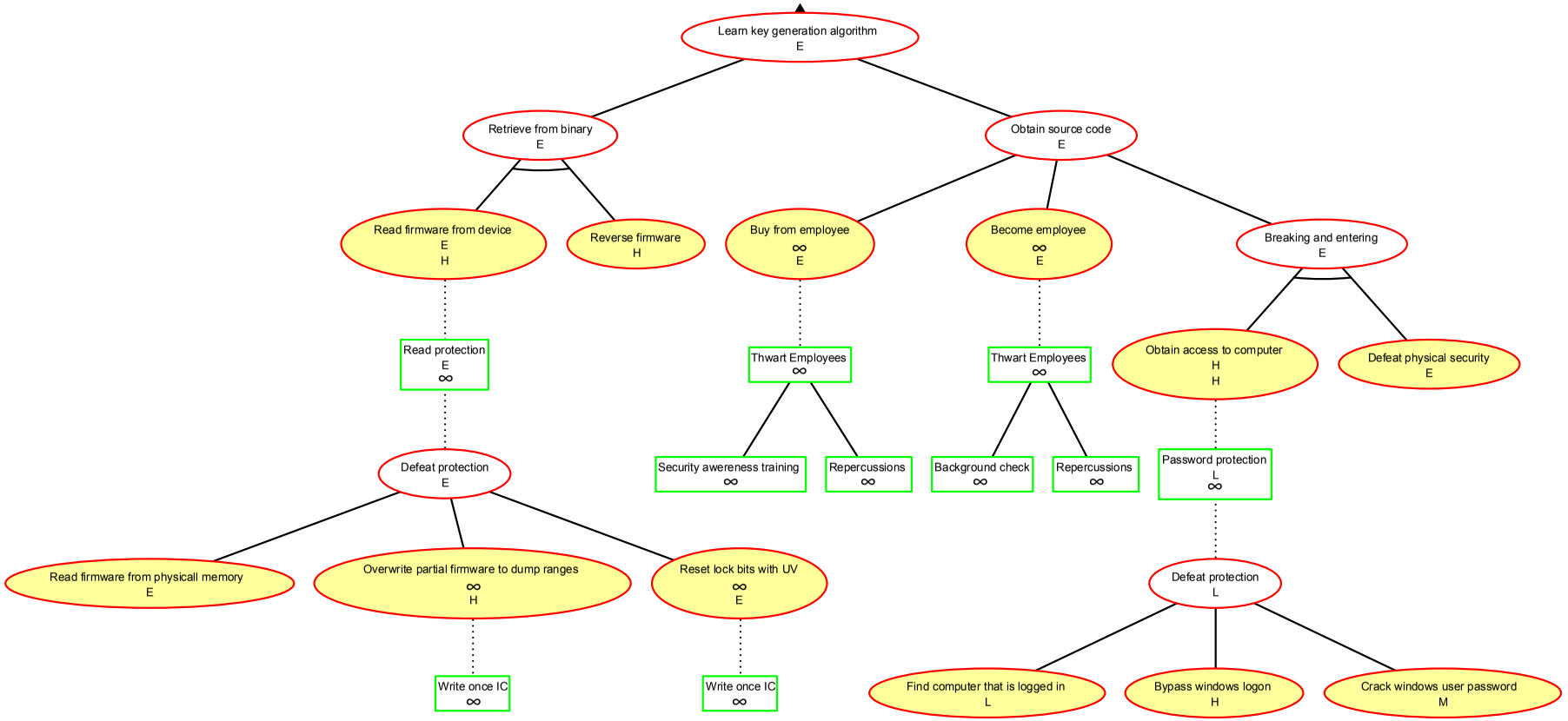


Figure 7: Subtree for the “Learn key generation algorithm” node in the attack-defense trees given in Figures 4 and 5

# 4 Attacks

The different attack-trees presented in Section 3 show countermeasures that normally are present and well designed. However, it is unknown if all games in this genre utilize all such countermeasures in their final products. This section will present three attacks on countermeasures presented in the attack-trees given in Figures 4, 5.

Subsection 4.1 verifies if tags are using the upgraded MIFARE Classic tags that protect against MIFARE attacks to retrieve the authentication keys. Furthermore, attempts are made to break the key derivation algorithm by analyzing the authentication sector keys. Finally, in Subsections 4.2 and 4.3 the difficulty of retrieving the key derivation algorithm is explored and analyzed.

## 4.1 Proxmark III

Certain toys analyzed contain one or two MIFARE Classic 1K tags. In such toys, the security configuration may be setup in such a way, that the tag can only be accessed with the correct sector key. Moreover, all key B fields are configured to be usable as a data field, this means that only key A can be used to authenticate the tag. Sector zero is set to read-only after authentication and the rest of the sectors are set to read/write after authentication. Key A is unknown to the attacker, however, there are card-only attacks, reader only and sniffing attacks that can help with finding the correct keys. The Proxmark III is the swiss army knife for RFID communication and can mount most of these attacks [47].

"The Proxmark III is a device developed by Jonathan Westhues that enables sniffing, reading and cloning of RFID tags" [50]. Coupled with research on card-only attacks like the Darkside and NESTED attacks it can be used to find keys for a tags [10, 11]. The two attacks where executed on numerous toys and all keys could be found in under one minute. The collected keys showed that the first sector always has the same static key, but all other sector keys were seemingly random. NXP fixed the bug exploited by the two attacks and starting in 2014, all toys starting using these newer tags and in effect prevent obtaining keys through this method.

Changing bits in the first two blocks of data in cloned toys on a backdoor MIFARE Classic 1K tag, made it clear that the keys were derived in a specific way from the Non Unique Identifier (NUID). When the NUID was changed and the keys where left as is, the tag became completely unreadable, while other changes made the tag unplayable in the game but still detectable.

While this does gives us enough information to make clones using backdoor MIFARE Classic tags that have a writable block zero, it does not make it possible to forge new toys on genuine new MIFARE Classic tags. However, it could be that the key derivation algorithm leaks information about the inner workings of the algorithm through the keys and that could lead to forgeries.

### 4.1.1 Key A analyses

To see patterns more easily, all MIFARE Classic authentication A keys are shown in binary and hexadecimal representation and can help to reveal patterns. Table 3 shows all keys for a toy tag with NUID `0xCB142CF0`.

| Key # | Binary | Hexadecimal |
|---|---|---|
| 0 | 010010110000101100100000000100000111110011001011 | 4B0B20107CCB |
| 1 | 100101010001101000011100001101001100001011010111 | 951A1C34C2D7 |
| 2 | 101100110111011111001000011001110001010100010100 | B377C8671514 |
| 3 | 001000000100000100100010110011101111111011110101 | 204122CEFEF5 |
| 4 | 011011001001101110001010011010010101000001110010 | 6C9B8A695072 |
| 5 | 111111111010110101100000110000001011101110010011 | FFAD60C0BB93 |
| 6 | 110110011100000010110100100100110110110001010000 | D9C0B4936C50 |
| 7 | 010010101111011001011110001110101000011110110001 | 4AF65E3A87B1 |
| 8 | 110100100100001000001111011101011101101010111110 | D2420F75DABE |
| 9 | 010000010111010011100101110111000011000101011111 | 4174E5DC315F |
| 10 | 011001110001100100110001100011111111001101001100 | 6719318FE69C |
| 11 | 111101000010111111011011001001100000110101111101 | F42FDB260D7D |
| 12 | 101110001111010101110011100000011010001111111010 | B8F57381A3FA |
| 13 | 001010111100001110011001001010000100100000011011 | 2BC39928481B |
| 14 | 000011011010111001001101011110111001111111011000 | 0DAE4D7B9FD8 |
| 15 | 100111101001100010100111110100100111010000111001 | 9E98A7D27439 |

Table 3: All keys for tag with NUID 0xCB142CF0

While certainly not obvious, it feels as if there is some sort of shifting to the left happening. Moreover, when doing an exclusive or operation between a key and the key that follows ($K[i] \oplus K[i+1]$), it becomes apparent that there is indeed some pattern, this can be seen in Table 4.

| Key # | Binary | Hexadecimal |
|---|---|---|
| $0 \oplus 1$ | 110111100001000100111100001001001011111000011100 | DE113C24BE1C |
| $1 \oplus 2$ | 001001100110110111010100010100111101011111000011 | 266DD453D7C3 |
| $2 \oplus 3$ | 100100110011011011101010101010011111010111100001 | 9336EAA9EBE1 |
| $3 \oplus 4$ | 010011001101101010101000101001111010111010000111 | 4CDAA8A7AE87 |
| $4 \oplus 5$ | 100100110011011011101010101010011111010111100001 | 9336EAA9EBE1 |
| $5 \oplus 6$ | 001001100110110111010100010100111101011111000011 | 266DD453D7C3 |
| $6 \oplus 7$ | 100100110011011011101010101010011111010111100001 | 9336EAA9EBE1 |
| $7 \oplus 8$ | 100110001011010001010001010011110101110100001111 | 98B4514F5D0F |
| $8 \oplus 9$ | 100100110011011011101010101010011111010111100001 | 9336EAA9EBE1 |
| $9 \oplus 10$ | 001001100110110111010100010100111101011111000011 | 266DD453D7C3 |
| $10 \oplus 11$ | 100100110011011011101010101010011111010111100001 | 9336EAA9EBE1 |
| $11 \oplus 12$ | 010011001101101010101000101001111010111010000111 | 4CDAA8A7AE87 |
| $12 \oplus 13$ | 100100110011011011101010101010011111010111100001 | 9336EAA9EBE1 |
| $13 \oplus 14$ | 001001100110110111010100010100111101011111000011 | 266DD453D7C3 |
| $14 \oplus 15$ | 100100110011011011101010101010011111010111100001 | 9336EAA9EBE1 |

Table 4: Difference between a key and its subsequent key ($K[i] \oplus K[i+1]$) for tag with NUID 0xCB142CF0

It turns out that $K[i] \oplus K[i+1]$ for $i$ in range $[2, 16)$ has the same outcome for every NUID. Sector zero key is static and by using the sector one key all other sector keys can be derived. Verification of this hypothesis is shown in Table 5 and holds for all tested NUIDs. However, the sector one key is the only exception as $K[1] = K[0] \oplus (K[0] \oplus K[1])$ does not hold across different NUIDs.

| $\mathbf{i}$ | $\mathbf{K}[\mathbf{i}-\mathbf{1}]$ | $\mathbf{K}[\mathbf{i}-\mathbf{1}] \oplus \mathbf{K}[\mathbf{i}]$ | $\mathbf{K}[\mathbf{i}] = \mathbf{K}[\mathbf{i}-\mathbf{1}] \oplus (\mathbf{K}[\mathbf{i}-\mathbf{1}] \oplus \mathbf{K}[\mathbf{i}])$ |
|---|---|---|---|
| 2 | 951A1C34C2D7 | 266DD453D7C3 | B377C8671514 |
| 3 | B377C8671514 | 9336EAA9EBE1 | 204122CEFEF5 |
| 4 | 204122CEFEF5 | 4CDAA8A7AE87 | 6C9B8A695072 |
| 5 | 6C9B8A695072 | 9336EAA9EBE1 | FFAD60C0BB93 |
| 6 | FFAD60C0BB93 | 266DD453D7C3 | D9C0B4936C50 |
| 7 | D9C0B4936C50 | 9336EAA9EBE1 | 4AF65E3A87B1 |
| 8 | 4AF65E3A87B1 | 98B4514F5D0F | D2420F75DABE |
| 9 | D2420F75DABE | 9336EAA9EBE1 | 4174E5DC315F |
| 10 | 4174E5DC315F | 266DD453D7C3 | 6719318FE69C |
| 11 | 6719318FE69C | 9336EAA9EBE1 | F42FDB260D7D |
| 12 | F42FDB260D7D | 4CDAA8A7AE87 | B8F57381A3FA |
| 13 | B8F57381A3FA | 9336EAA9EBE1 | 2BC39928481B |
| 14 | 2BC39928481B | 266DD453D7C3 | 0DAE4D7B9FD8 |
| 15 | 0DAE4D7B9FD8 | 9336EAA9EBE1 | 9E98A7D27439 |

Table 5: Verification of the key derivation hypothesis for tag with NUID 0xCB142CF0

Given this, it is not apparent how sector one key is generated. An exhaustive key search over the single unknown $2^{48}$ bit key is infeasible as the counterfeited MIFARE Classic 1K tag has to have sector one key replaced and then tested by the hardware reader that comes with the game. This increases the time per try significantly. More importantly, the MIFARE Classic S50 datasheet gives a write endurance of 100.000 cycles which means that most likely block seven will give out before sector one key can be found [31]. This makes the exhaustive key search near to impossible in this setup.

#### 4.1.2 Impact

Even while only one from all the keys is unknown, the impact of this attack is minimal, simply because the unknown key is hard to find.

#### 4.1.3 Conclusion

At first sight the results are significant, however for forging tags one crucial piece of information is missing, how to calculate the second sector key. Cloning tags is relatively expensive and one needs special hardware and tags. Moreover, NXP has solved critical bugs to stop the Darkside and NESTED attacks work against MIFARE Classic tags. However, certain older games in the toys-to-life genre that were analyzed used tags that did contain the critical bugs and were susceptible to attacks. Nowadays, no games are using these exploitable tags in their newer products.

## 4.2 Reverse engineering the reader

Readers that are typically bundled with games in the toys-to-life genre are in essence just an NFC reader with some extra functionality that can be controlled over the USB. The protocol is not encrypted and fairly straightforward. With an USB sniffer one can reverse engineer the protocol by capturing a few traces while playing games on the console. From that, it becomes clear that the key derivation algorithm is inside the reader, nothing resembling the key or setting the key is sent over the USB.

This means that by retrieving the firmware from the reader and reversing this, any unknowns in the USB protocol and the key derivation algorithm could be figured out. If this can be achieved, counterfeiting of tags is possible because the key derivation will be known.

### 4.2.1 Preliminary research

The best candidate Printed Circuit Board (PCB) for trying to retrieving the firmware from, is one with most information visible and accessible. A well known strategy for manufactures is to put a blob of epoxy on the ICs covering the version information and access to all the pins. Another simple strategy is to mill off model information from the ICs. Table 6 shows what methods, if any, are being used for different official readers. An important observation is that all PCBs had their debug header clearly visible and each pin marked with a label.

| # | **Microcontroller Unit (MCU)** | **Programming header** | **Labels** |
|---|---|---|---|
| 1 | Hidden using epoxy | Present | No labels |
| 2 | Visible | Present | Clearly labeled |
| 3 | Hidden using epoxy | Present | Minimal labels |
| 4 | Hidden using epoxy | Present | Minimal labels |

Table 6: Preliminary research on some readers

The second reader was a good candidate to see if the firmware could be dumped because everything was clearly labeled and accessible. If it turned out to be very hard to read the firmware from this reader, more readers could be investigated or other strategies explored to retrieve the firmware from the MCU.

### 4.2.2 Obtaining binary firmware from the reader

Appendix E Figure 38 on page 62 shows the PCB of the used reader. Every element of the PCB is clearly labeled and visible. The different ICs had all the model numbers visible, hence finding the datasheets was possible. Table 7 shows the used integrated circuits in this particular reader. The header in the left top corner for the CLK, DAT, GND, VDD and VPP was added later to make it easier to access the MCU.

| | |
|---|---|
| MCP2120 | Only present in this specific reader, used for infrared communication with the console. |
| PIC18F14K50 | Microcontroller that is USB 2.0 compliant, used for controlling the MFRC53001T and communication over the USB and infrared. |
| MFRC53001T | Used for contactless communication at 13.56 MHz, it supports ISO/IEC 14443A and the CRYPTO1 security algorithm for authenticating MIFARE products. |

Table 7: Integrated circuits within the reader

Not only was the chip clearly visible and all information publicly known. There is also research available on dumping protected firmware from this particular PIC18F microcontroller [28]. To verify what security was in place, the official in-circuit debugger for the PIC microcontroller was

connected. This specific reader had limited security in place and therefore it was possible to read the firmware from the MCU with an In Circuit Serial Programmer (ICSP).

Machine code is rather hard to read for humans, so IDA Pro was used to disassemble the byte code in to assembly. While the PIC18F is a rather simple MCU, Static Random-Access Memory (RAM) usage is done by moving a set of pointers forward and backwards over the RAM, just like a tape machine. This works great for computers but is much harder for humans. Because of this, the firmware was rather difficult to reverse but certainly not impossible and with sufficient time the key derivation algorithm was found and reversed.

The disassembled key derivation algorithm exists out of three important functions.

- **get_key:** returns either the static key or calculates the requested sector key.
- **key_accumulate:** shifts the key eight times to the left and mixes in a constant with an exclusive or operation if a specific equation holds.
- **key_shift:** shifts the whole key 1 bit to the left.

The appendix contains a flow chart for the three functions. Get key function can be found in Appendix D.1. Appendix D.2 shows the key accumulate function. The shift function is presented in Appendix D.3.

### 4.2.3 Cryptanalysis

As explained in the previous section, the key derivation algorithm uses three subfunctions. This section will give a thorough analysis on the three functions.

**4.2.3.1 Key Shift** The key shift function as given in Appendix D.3 shifts each byte in the working key one bit to the left. As this function is called a total of 40 times (five rounds times eight sub rounds), 40 bits of the 48 original bits get shifted out of the end result. This looks like an implementation bug where the algorithm should have been a rotate function. Just adding the rotate to the current algorithm yields a functioning algorithm that still works perfectly fine but is also still susceptible to the previous attack given in Section 4.1.

**4.2.3.2 Get Key** Retrieving the key for a specific sector is achieved by calling the get key function. Its corresponding flow diagram can be found in Appendix D.1. Calling the function with the sector number argument set to zero will return the master key. Using a static key for sector zero shows a misunderstanding of the NFC technology. The key derivation algorithm only uses the UID and sector number as an input. The UID of a MIFARE Classic tag can be retrieved without any authentications and thus the static master key could have been removed and always using the real key derivation algorithm for all sectors.

When calling the function for sector greater than zero, it will call the key accumulate function five times (four times with one byte of the UID and once with the sector number as the argument).

**4.2.3.3 Key Accumulate** The heavy lifting of the key derivation algorithm is done in this function as can be seen in the flow chart in Appendix D.2. The biggest conceptual problems with this algorithm is that the only difference between running the algorithm for sector one and sector two is one exclusive or operation (one bit difference). This is because of the sector number being used as a state variable that defines if it shift and does an exclusive or, or only shift. This fact alone is responsible for the exploit shown in Section 4.1.

### 4.2.4 Impact

Having the key algorithm known makes it possible to generate valid toy tags with cheap off the shelf MIFARE Classic 1K tags. This opens up paths for counterfeiters to make money.

### 4.2.5 Recommendations

While the algorithm is flawed in multiple ways, it is still next to impossible to find a way to generate sector one keys without reversing the algorithm. Even using a well defined and accepted key derivation algorithm does not really add more protection as all algorithms will fail against reverse engineering in this specific case. All parameters for the algorithm need to be known to the reader, hence reversing the reader gives full understanding of the key derivation algorithm. Offloading the algorithm to a secure access module that can be used as a black box cryptographic device could help but will increase the production costs significantly.

Because there is only one bit difference between sector key $i$ and $i + 1$, the algorithm could be improved by increasing the difference between sector keys. Moreover, something as predictable as a sector number should not be used plainly as an input for the key derivation algorithm. Using a secure cryptographic hashing function over the NUID together with a secret and the sector number would at least prevent the dependency between the keys.

The fact that that the firmware was readable from a reader significantly decreased the security. Setting the lock bits to prevent reading the chip would make it much harder to get the firmware from the device [28]. Therefore, this protection mechanism should be enabled as it significantly increases the effort required by attackers.

### 4.2.6 Conclusion

Hardware should never be released without the default security enabled, it is straightforward to use and adds a significant barrier for the attacker. Extra hardware and software protection can be added like covering the IC with epoxy, making the MCU write-only or obfuscating the code. The amount of security to add to a product will always be a cost risk analysis. However, no matter how good the security is, against a determined attacker with knowledge and money, it will simply not hold and in time, all secrets contained within will be revealed.

Instead of trying to make a Fort Knox, not hiding secrets in the first place can have better results. For example, signing toys with asymmetric cryptography makes it possible to detect tags that are not signed by companies while not storing any secrets like private keys in software or hardware given to the user. This is a completely different approach and makes the product much more secure as it can removes attack vectors altogether.

## 4.3 MaxLander

MaxLander is a hardware and software commercial product that can backup, restore and copy toys. The system can make a backup of a real toy and restore it to the same genuine toy. It can also be used to restore a backup to a special knockoff MIFARE Classic 1K tag (see Section 2.3.2) to make an 1-on-1 copy of a genuine backed up toy. MaxLander tries to make more money on the special knockoff tags by making them only usable for 99 times. After restoring backups 99 times to the knockoff NFC tag, they become unusable by the MaxLander system and new knockoff tags have to be bought. The check also prevents generic blank knockoff MIFARE Classic 1K tags working with MaxLander. However, it is a purely artificial limit and trivial to work around. On the key B field in sector 16, there are two bytes that defines how many writes are left, Figure 8 shows how this is implemented. Setting `block63[14]` to `0x00` and `block64[15]` to `0xFF`, allocates 255 writes for that particular tag. The artificial limitation was removed in version 1.4[1]

```
block63[14] = ~numberOfWritesLeft; // Inverse of the number of writes left
block63[15] = numberOfWritesLeft;  // Number of writes left
```

Figure 8: The implementation of the artificial limit build into MaxLander up to version 1.4

To see exactly what kind of attack MaxLander used to achieve this, further analyses of the product was required. This section will give an in depth analysis of the hardware, firmware and client software application.

### 4.3.1 Technical Specification

The hardware of the MaxLander is a simple, well made USB NFC reader. It contains an integrated antenna and three integrated circuits. One of them is a MIFARE IC that can drive the integrated antenna and communicate with different MIFARE products. The MCU is an STM32F0 and handles directly the USB communication. Lastly, an unknown "SAM12" IC is present that is connected over Inter-Integrated Circuit ($I^2C$) to the STM32F0. This unknown IC looks to be a Secure Access Module (SAM) and contains the cryptographic algorithm required to generate sector keys, Section 4.2.2 describes one possibility on how this algorithm could have been obtained.

| **Microcontroller Unit** | |
|---|---|
| MCU | STM32F042K6T6 |
| Package | LQFP32 |
| Type | Cortex-M0 |
| Architecture | ARMv6-M (Little Endian) |
| ROM offset | 0x08000000 |
| ROM size | 0x8000 |
| SRAM offset | 0x20000000 |
| SRAM size | 0x1800 |

| **Near Field Communication controller** | |
|---|---|
| IC | MFRC63002HN |
| Package | HVQFN32 |
| Connection | Serial Peripheral Interface (SPI) to main MCU |
| Usage | NFC communication |

| **Secure Access Module** | |
|---|---|
| IC | SAM12 (Unknown) |
| Package | QFN12 |
| Connection | $I^2C$ to main MCU |
| Usage | Generating sector keys for authentication of the MIFARE Classic tags |

Table 8: Technical specifications for the three ICs

[1] http://www.maxlander.net/maxlander-v1-4-software-released-removed-99-times-limit/

### 4.3.2 USB protocol

To get a grasp on what kind of USB communication was done by the device, USB descriptors where analyzed and can been seen in Table 9. The USB descriptors are purely informative as they do not have to be correct and should be used by the software as guidance. The descriptors the device returned showed that it was a Human Interface Device (HID) based device with two endpoints and the mandatory endpoint zero. However, there are several errors in the USB descriptors. For example, endpoint one is both and input and an output which is impossible as USB endpoint are one way only. Something that should be pointed out is that the USB descriptor almost exactly matches that of genuine readers. Moreover, some tested genuine readers an error in their USB descriptors. Endpoints one and two are respectively input and output while in practice this is swapped. It is clear that MaxLander tried to copy the USB descriptors from the genuine reader but failed to do so perfectly. Even if they did, their usage of endpoint one is also as an output and they would have copied the mistake from the genuine reader]. Lastly, a small difference can be found in the device release number field (bcdDevice), which MaxLander set to 0x0101 instead of 0x0100.

| **Device descriptor** | |
|---|---|
| Vendor Identifier | 0x5C60 |
| Product Identifier | 0xDEAD |
| Manufacturer | Zippy |
| Product | Portal |
| Serial number | 0xA036333C5342 |
| USB version | 0x0200 |
| Device release number | 0x0101 |
| Max packet size EP0 | 64 |
| Configurations | 1 |

| **Configuration descriptor** | |
|---|---|
| Interfaces | 1 |

| **Interface descriptor** | |
|---|---|
| Endpoints | 2 |
| Interface class | 0x03 (HID) |

| **HID descriptor** | |
|---|---|
| HID version | 0x0111 |
| Descriptor type | 0x22 |
| Descriptor length | 0x19 |

| **Endpoint descriptor** | |
|---|---|
| Address | 1 |
| Direction | Input |
| Transfer type | Interrupt |
| Max packet size | 64 |

| **Endpoint descriptor** | |
|---|---|
| Address | 1 |
| Direction | Output |
| Transfer type | Interrupt |
| Max packet size | 64 |

Table 9: Simplified MaxLander USB descriptors

To further facilitate the understanding of the USB protocol, the client software and the firmware of the STM32F0 IC had to be reversed and the USB communication captured. The MaxLander PCB contains five contacts that seem to be the receivers for pogo pins to interface with the MCU. It also contains five contacts in the exact same fashion as the MCU for the SAM12 IC. Creating a simple device that had pogo pins in the same layout as the contacts on the MaxLander PCB on one side and normal male pins on other side was created to more easily test devices to establish any communication with the MaxLander PCB. Open On-Chip Debugger (OpenOCD) is software that specializes in connecting to devices and already had support for STM32F0. Because the pin out of the STM32F0 was known, it was possible to correctly label the five contact points. Using OpenOCD and the custom build pogo pin device to establish a connection to the STM32F0 showed that the chip did not had any protection in place and the firmware could be read. Sadly, this did not turn out to be true for the SAM12 IC. No matter what was tried, no communication was possible to the SAM12 IC and it can be assumed that the access to this IC over those contacts is completely disabled. Not only that, it was impossible to find anything about this mysterious IC.

Reversing the firmware of the STM32F0 IC and the client PC software, gave insight into the USB protocol and how the SAM12 IC was being used. The USB protocol can be found in Appendix A.

While searching for information about this elusive SAM12 IC, a blog[2] post was found that describe another device that looked exactly the same and also contained this mysterious IC. The blog post described the PowerSaves for characters[3] device.

### 4.3.3 Similarities to PowerSaves for characters

Datel is an electronics and game console peripherals manufacturer. The company has been making hardware since the 1980s and spans a broad range of hardware. Datel released a device before the MaxLander that can apply cheats to characters. Strangely enough, while not advertised, that device is also able to read toys and that is only possible if the device has a copy of the key derivation algorithm for official toys. After closer inspection of the hardware, the MaxLander PCB is identical to that of the Datel PowerSaves for characters as can be seen in Figures 31, 32, 33 and 34.

Not only is the hardware the same, but the PC client software and the firmware share a lot of commonalities. After analyzing library usage and code style, many similarities can be found between the two products. It is however unknown how these two products are connected and why they share so many similarities, but this falls outside the scope of this thesis.

### 4.3.4 Impact

Making an analysis of the impact of MaxLander is difficult. There are obviously no metrics available on how many people create their own toys with MaxLander and play with them. Not only that, if it would be possible to track how many toys are made through MaxLander, that does not directly translate to lost sales. Nevertheless, having some insight into MaxLander is better then nothing.

The people behind MaxLander have created a social sharing website called NGmore[4] to share toy data files with others. Consumers of the MaxLander device can download these data files and create usable toys using MaxLander. Luckily, the website keeps a download counter per toy upload. Hence if it would be possible to get these download counts on timed intervals it would be possible to see usages trends.

Retrieving the data is achieved by running a scraper every six hours. NGMore had multiple layers of protection and to get around this the scraper was operated on an automated system could create new prebuild virtual machines. The scraper creates 1 to N prebuild proxy servers and waits until the virtual machines are reachable. Then the scraper uses the proxy servers to multiplex access to the NGMore website. NGMore also used obfuscated JavaScript to verify if the visitor is able to run JavaScript, as often bots can not run JavaScript and thus is used to detect non human visitors. Using a JavaScript virtual machine, the scraper ran the JavaScript in a controlled environment to pass the checks given by NGMore. This system started collecting data on 2015/10/21 and was able to continuously collect data except for some downtime due downtime of NGMore, changes in NGMore and issues on the control server.

Plots 9 and 10 show global members and posts on NGMore. Important to note is that NGMore also contains sections about characters and other non related game subjects. A sharp drop can be seen in plot 11 on 2015/11/19. This drop corresponds with lawyers sending an NGMore user a cease and desist order to stop infringing copyrights. The user publicly announced this fact on the NGMore discussion board and he removed all his uploaded toys, and thus causing a sharp drop in toys, downloads and posts totals. However, other members that already downloaded the uploads, re-uploaded the original uploads and that explains the steep increase right after the drop. The download distribution can been seen in plot 13 and shows the difference between the download distribution when the system began to collect data and stopped collecting data. Lastly, plot 14 shows the minimum, maximum and average downloads for the released toys. The average shows an increasing trend which makes sense given the fact that their is an finite number of toys to upload and users can theoretically download endlessly. Every drop in the minimum corresponds to a new toy being added.

[2] Accessed on 2016/02/03 http://ghettohaxxx-blog.azurewebsites.net/reversing-powersaves-for-amiibo/
[3] Accessed on 2016/02/03 http://codejunkies.com/powersaves-for-amiibo/
[4] Accessed on 2015/10/19 http://www.ngmore.com/

### 4.3.5 Conclusion

It is clear from the given facts that MaxLander and PowerSaves for characters contain a copy of the key derivation algorithm inside the SAM12 IC. This means that people behind these products successfully reversed the key derivation algorithm and can potentially create any toy with any tag. MaxLander can only restore backups of toys to the special MIFARE Classic tags or the original toy, but that is a purely artificial limit probably imposed to make more profit from this system.

Another issue that NGMore caused is the fact that users of MaxLander tried to search for unreleased toys by changing fields in the genuine backup. This let the users of MaxLander find almost all of the unreleased toys before they where officially announced by the respective companies. Once again, the impact of this is not clear and certainly falls outside the scope of this thesis.

While the impact is not significant compared to the global scale on which these companies distributes toys, the growing trend of the toys-to-life games and the clear linear growth in the user base of NGMore does show a market for pirated playable toys-to-life.

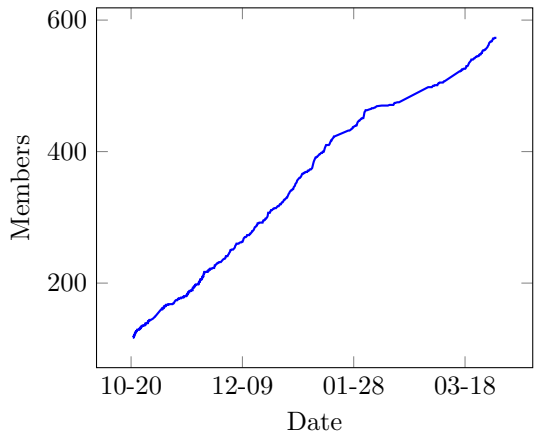


Figure 9: Members over time

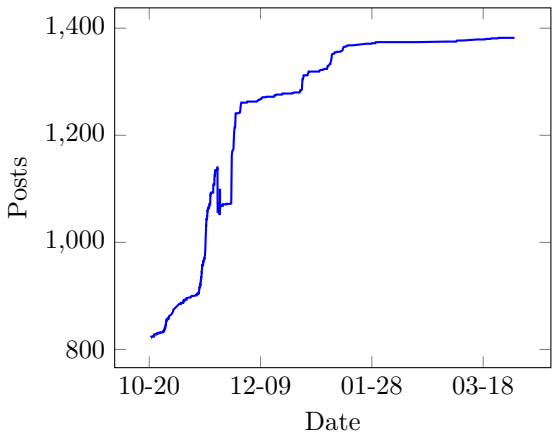


Figure 10: All forum posts over time

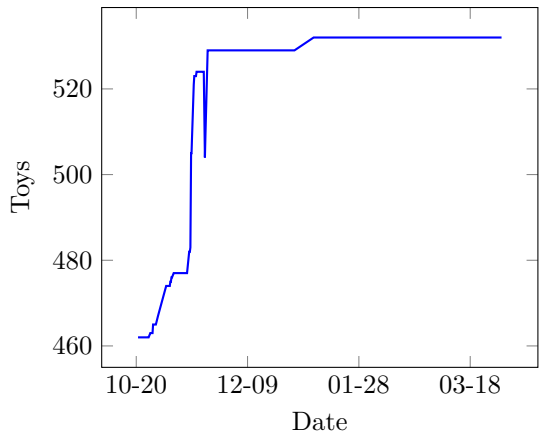


Figure 11: Total number of toys over time

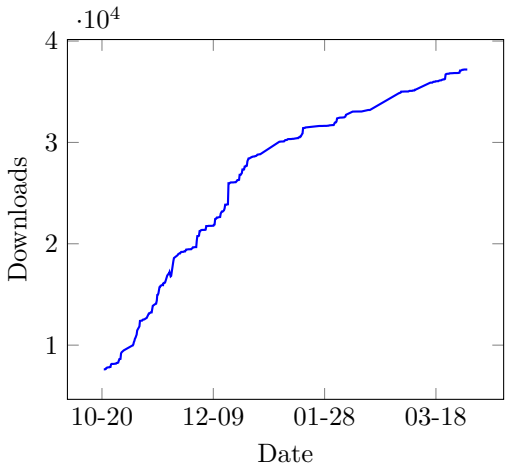


Figure 12: Total downloads over time

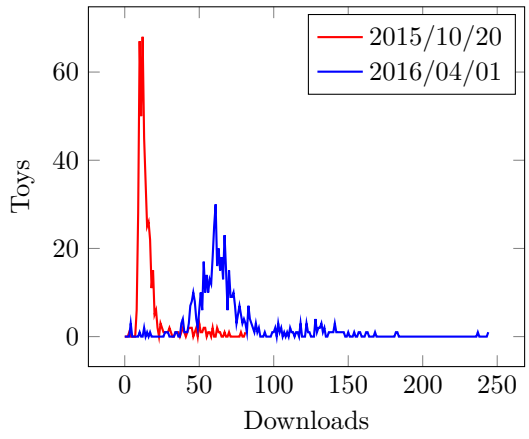


Figure 13: Download distribution over toys

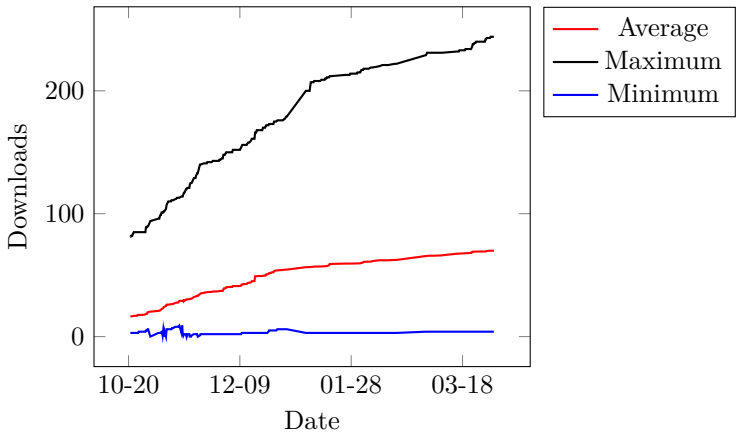


Figure 14: Downloads min/max/avg over time

# 5 Countermeasures

As shown in the previous section, the countermeasures that should have made learning the key derivation algorithm extremely hard (as shown in attack-defense tree Figure 7) were not present in the tested readers. This led to at least two independent parties learning the key derivation algorithm and making it possible to make forgeries and clones. This section will explore what countermeasures can be added that do not break backwards or forwards compatibility.

Subsection 5.1 will explain how a signature over data that defines the toy protects against forgeries but not against clones.Lastly, Subsection 5.3 presents a way of automatic fuzzing the game by emulating the hardware reader. This helps to find bugs in software.

## 5.1 Counterfeiting toys

The chosen MIFARE Classic RFID technology that is used for games in the toys-to-life genre is inherently not very secure [10, 11, 10]. Moreover, the backwards compatibility requirements make it currently impossible to change technologies. This means that the countermeasure has to work on all readers and toys or at least, not make them unusable in games with the countermeasure present. Given this, the countermeasure has to work under the assumption that all data on the tag is read and writable, including block zero that normally is not. Moreover, because of the size of the production process and in turn the difficulties of making significant changes to this process, it was decided that the security should last for around 10 years. More precisely, it has to last 11 years and the reasoning behind this is that all toys have a year field that goes from 2011 up to 2027. Hence it would make a lot of sense to have the system work until the end of this field.

Preventing attackers from forging tags (i.e. creating tags that are not officially produced by companies), can be achieved by the de facto method of verifying the genuineness of data, a signature using public-key cryptography. Adding a signature to the toy, makes it possible to prove that companies produced this particular data on the tag and in turn, if such a signature is not present or invalid, it is not a genuine toy. This scheme only works if assumed that the underlying signature technology is secure.

Implementing the described signature scheme would prevent attackers from forging playable toys inexpensively as special tags are required to make clones. Moreover, it would protect games against attackers enumerating identifiers to find unreleased toys.

### 5.1.1 Signature scheme concept

Table 10 gives that layout of sector zero for certain toys that are currently produced, it is apparent that only block one is used to define what type of toy this particular tag is. It is important to note that only signing block one would not be enough, as the data and signature could be written to a genuine NXP tag with a different block zero and only a collision between the checksum and the two different zero blocks have to be found which is less difficult in comparison against breaking the signature technology. However, by generating the signature over block zero and one, only the tags that are bought by companies and initialized to a specific toy will be valid. This is the basis of the described signature scheme in this section. Only one significant attack on this scheme remains and that is cloning (i.e. copying all bytes from tag A to tag B), ways to make this harder with the current architecture described in Section 5.1.2.

| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Block 0** | Non-UID | | | | LRC | SAK | ATQA | | Manufacturer data | | | | | | | |
| **Block 1** | ToyType | | | | TradingCardId | | | | | | | | SubType | | Checksum | |
| **Block 2** | Unused | | | | | | | | | | | | | | | |
| **Block 3** | Key A | | | | | | ACL | | | | Key B / User data | | | | | |

Table 10: The first sector of a toy MIFARE Classic 1K tag. Block zero is initialized at manufacturing by NXP and is read-only, block three is a section trailer defined in the MIFARE standard

#### 5.1.1.1 RSA versus ECC signing algorithm

The signature has to be stored on the tag in a way that is accessible by any reader currently available. This means that storing the signature in "hidden" memory that is only accessible by special MIFARE commands can not be used as it will not be backwards compatible. Moreover, if possible, the signature should be placed in read-only memory, this will prevent any accidental corruption of the signature.

The smallest tag used in games is the MIFARE Classic 1K [31] which has 768 bytes of memory. However, almost all memory is in use and as of 2015, there are 16 bytes of free continues memory available and fifteen six byte fragmented blocks available. Using all free memory would amount to 106 bytes (848 bits) and is the upper limit of the signature size.

RSA signatures are always exactly the size of RSA key size and this would mean that the more traditional key sizes of 512 or 768 bits could be used. However, current research shows that 512-bit moduli can hypothetically be broken in less then ten minutes and 768-bit moduli can be broken in 70 days [24, 40]. Moreover, even if a 848 bit key would be used, it would not even provide enough security to last for four years according to ECRYPT II [5].

ECC can give the same amount of security as RSA, but with a smaller key size. According to Arjen K. Lenstra and Eric R. Verheul around 2000, it was accepted that 163-bit ECC gives about the same security as a 1024-bit RSA [25, 8]. However, a paper released by ECRYPT II in 2012 adjusts some numbers to represent a more recent view of the security of different key size. ECRYPT also gave recommendations for symmetric key sizes and the expected time to life, Table 11 shows this and also shows there respective RSA and ECC key sizes. Moreover, research by Erich Wenger and Paul Wolfger in 2015 confirmed the numbers given by ECRYPT and state that the $\approx 30$ years for 256 ECC is a conservative prediction [49]. History shows that it is hard to predict how much security is needed to survive a specific amount of time, so it is important to take this into account and if resources allow it, use more bits of security then is strictly needed according to the current predictions.

| Security (bits) | RSA | ECC | Protection |
|---|---|---|---|
| 80 | 1248 | 160 | Smallest general-purpose level, $\leq$ 4 years protection |
| 96 | 1776 | 192 | Around 10 years protection |
| 112 | 2432 | 224 | Approximately 20 years protection |
| 128 | 3248 | 256 | Recommend protection level, approximately 30 years protection |
| 256 | 15424 | 512 | Foreseeable future unless Shor's algorithm applies |

Table 11: Level of protection given by key sizes in bits for different algorithms [5]

Given the memory constrains, ECC is a better candidate then RSA because of the smaller key size. ECC with key size of 192 give around 10 years of protection and should fit in the current available space. In the next section, two different ECC signature schemes are evaluated to see what better fits the requirements of the system.

#### 5.1.1.2 Most suitable ECC signature scheme

Within ECC there are currently two major signature schemes, Elliptic Curve Digital Signature Algorithm (ECDSA) and Edwards-curve Digital Signature Algorithm (EdDSA). The major difference between the two schemes is that EdDSA uses Twisted Edwards curves instead of Weierstrasse curves and other standard curves. The major advantage of Twisted Edwards curves is the fact that it provides faster arithmetics and does not contain any significant disadvantages over other standard curves [3, 15]. On top of that, some EdDSA implementations are released in the public domain and makes EdDSA the obvious choice.

#### 5.1.1.3 Most suitable EdDSA implementation

A well known implementation of EdDSA is Ed25519 [6]. This implementation is being used in numerous high profile open source products and libraries which contributes to the robustness and quality of Ed25519 [5]. Another major benefit to Ed25519 is that it does not require an Cryptographic Pseudo-Random Number Generator (CPRNG). Not having a proper CPRNG can lead to security issues such as revealed on one of the

[5] Accessed on 01/25/2016 https://ianix.com/pub/ed25519-deployment.html

consoles [33]. Moreover, CPRNG are generally hard to get right in microcontrollers, hence it is beneficial that Ed25519 does not require this.

There are two prominent libraries in C that contain an implementation of Ed25519 and are released in the public domain.

- **TweetNaCl:** library written in C that fits in 100 tweets or 14000 characters and contains an Ed25519 implementation [7]. Because of the small size of the library, it is possible to verify properties. The paper verifies two memory-safety properties of TweetNaCl.
- **$\mu$NaCl:** a speed centered library that is tailored for 8-bit microcontrollers [17].

Two other notable libraries are original NaCl library and SUPERCOP. NaCl does not yet contain an Ed25519 implementation but, it is scheduled to be added in a future release. SUPERCOP is a toolkit for measuring the performance of cryptographic software and currently contains an NaCl compatible Ed25519 implementation. This implementation could be used on a non 8-bit platform where TweetNaCl does not provide enough speed [1].

#### 5.1.1.4 Storing the signature on a toy

An EdDSA signature is twice the key bit size big, so in the case of Ed25519 it would be 64 bytes or 512 bits. There is not a continuous block of 64 bytes available (even when ignoring sector trailers) so the signature has to be split up into pieces. All tags are setup in such a way that key B fields are defined as read-only data fields and the fields can be used to store the signature while honoring the read-only and compatible constraints. Every key B field has a size of 6 bytes, so a minimum of 11 key B fields are required. Two bytes will be unused and could be used for meta information about the signature, for example the index of the public private key pair. It is important that this meta information can not be critical to the system. When changed, it should never make an invalid tag valid and vice versa. The signature will be stored in key B fields starting from sector one up to sector 11. Sector zero is skipped because the key B data field is reserved for a possible feature.

### 5.1.2 Preventing cloning

Genuine NXP manufactured tags can not be used to make 1-on-1 clones simple because block zero can not be written to after it leaves the NXP factory. However, as explained in Section 2.3.2 page 8 about the MIFARE Classic knockoff chips, it is possible to create 1-on-1 copies with those tags and could bypass the signature security for making clones. There are currently three different flavors of these MIFARE Classic knockoffs available.

- **1K backdoor:** Contains a backdoor that disables the complete ACL and makes every block writable regardless the security state of the tag.
- **1K writable:** Behaves exactly as a genuine NXP MIFARE Classic 1K, except block zero is writable if ACL allows it.
- **4K writable:** Behaves exactly as a genuine NXP MIFARE Classic 4K, except block zero is writable if ACL allows it.

There is currently no MIFARE Classic 4K backdoor tag available and this means that it is currently impossible to reuse a MIFARE Classic 4K writable tag multiple times. This is due to the fact that the ACL for sector zero that is required to have a valid playable toy will lock block zero. Moreover, the MIFARE Classic 4K writable tags are expansive as they go for around $20.00 per tag[6]. Hence, it is not a viable piracy strategy as MIFARE Classic 4K writable tags are more expansive than a retail toy. The MIFARE Classic 1K backdoor tags however go for around $5.00 per tag[7] and can be reused many times because of the backdoor that completely bypasses the security. Buying a few of these tags to make 3 or 4 playable toys at a time would be a good strategy and is exactly what the MaxLander product did (see Section 4.3 page 20).

The constraints given prevents switching technologies altogether. However, the first generation readers can read MIFARE Classic 4K tags and as they are a less viable way for producing pirated

[6] Prices compared in March 2016 on eBay and AliExpress for small quantity orders
[7] Prices compared in January 2016 on eBay and AliExpress for small quantity orders

toys, it can be leveraged as an effective counter measure. If it would be possible to start using the 4K tags across all toys, it would make it very unappealing for attackers to create copies or tools that could create copies as the tags required would be very expansive and only usable once.

Switching to 4K tags may not be feasible for all companies in the genre because logistical costs, and risks associated with switching inventory in the total production process; regardless of any price change from switching to the 4K tags. However, it would have made good deterrent when coupled with the described signature scheme. Not only would it offer more security, it would also future proof games as more storage would be available in subsequent years.

**5.1.2.1 Physical Unclonable Function** One major obstacle in the security of smartcards is the possibility of making clones as it is currently hard to protect against. However, PUF could be the answer to this attack vector. In short, PUF leverage the minuscule differences in the physical end product to make an physical unclonable function [34, 12, 26]. There are different ways to approach PUFs and it is not yet clear what the best way is.

The Bitline PUF is a Challenge-Response PUF using Static Random-Access Memory (SRAM) and using this technology, cloning toys can be made tremendously more expansive than buying the toys themselves, hence making cloning a non issue [16]. The Bitline PUF leverages existing SRAM technology by only adding a few extra logical components to read out multiple bit and wordlines at the same time based on the challenge. The minute physical differences in the silicon will in turn make sure that the resulting read operation is different for each IC and challenge. Generating and storing a set of Challenge-Response Pairs (CRPs) in the controlled factory environment can later be used to verify the tag while in the field. Using PUF is certainly stronger then adding a signature to a tag as it also protects against clones. Moreover, using a designated block in the MIFARE memory range to send and read the challenge-responses it could even be possible to use PUF with older readers keeping the backwards compatibility.

NXP released on February 2013 a public document describing PUF in smartcard ICs using an SRAM based PUF [29]. While it is clear that NXP wants to use PUF in their products, it has been relatively silent after the initial announcement. It is unclear when PUF will be ready for large scale deployment but considering the security requirements nowadays it will probably be available before the end of the describe signature scheme.

### 5.1.3 Production process

The production lines that produce toys use so called programming stations to program a specific toy into a MIFARE Classic tag. The programming stations are using an ATmega128 microcontroller running at 8 MHz and are equipped with a CH375 IC for USB communicate with a readers [4, 48]. Using a reader makes a lot of sense from a business point of view as these devices can do everything that is required in the production process and are produced by companies anyways. There are thousands of programming stations produced that are being used over different factories in different production lines, hence the hardware used in the programming stations can not be changed. Because time is money, the extra signing step can not add a significant extra amount of time. For one of the companies that was reviewed, the third party in charge of the production process gave the magic number of two seconds, so what ever needs to be done has to be done under the two second limit.

### 5.1.4 Prototype one

For the first prototype it was not clear how to production process worked, only later precise information about the production process was available. However, it was hinted that the programming stations where custom build devices containing a Cortex MCU and a MIFARE reader IC including an integrated antenna. Moreover, it was stated that the security addition could not cost more than one second instead of the later given two seconds.

To get a general idea on the performance, a prototype had to be fabricated. Because it was still unclear what the precise specs where of the programming stations, a Teensy 3.1 was used. The device was readily available and running on a Cortex-M4 clocked at 72 MHz. Moreover, the Teensy was easy to hook up through $I^2C$ to a MIFARE PN532 reader IC that also also was readily

available [2, 30]. The prototype had two important parts that had to be tested for performance, the signature generation using the Ed25519 algorithm and the MIFARE communication.

**5.1.4.1 MIFARE communication** As explained in Section 5.1.1.4, the signature will be stored over 11 key B fields starting from sector one and will require a number of MIFARE actions. The signature has to be generated over the first two block of data and will also require some MIFARE actions. Splitting all actions up into there respective operations gives the following list of 26 operations.

- Reading the first two blocks from the chip
  - 1 Authentications
  - 2 Data read exchanges
- Writing 11, six byte key B fields
  - 11 Authentications (every key B field is in a new sector and requires a new authentication)
  - 11 Data write exchanges

To get precise timings for each operation, a logic analyzer was used that could gather 500 Mega-Samples per second (MS/s). With the logic analyzer the write, read and authenticate operation where timed. From Figure 15 one can see that most of the time is spend waiting on an answer from the PN532 IC. While the PN532 is a diverse and powerful NFC IC solution, it is targeted towards phones and is rather generic. Switching the PN532 to a different NFC reader IC that is more specialized towards industrial reading and writing could increase the performance.

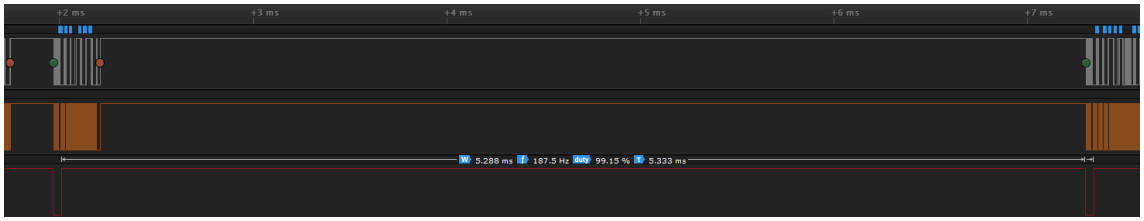


Figure 15: Partial $I^2C$ trace of a single read command of the signing process

Table 12 show the breakdown of the timing information for the three different types of operations while communication with the PN532 IC. It takes approximately 229.8066 *ms* to execute and process all the 26 commands, this means that there is only 770 *ms* left to generate the signature and run any other code that is required.

| | **Authenticate** | **Read exchange** | **Write exchange** |
|---|---|---|---|
| **Write to PN532** | 0.7275 | 0.4623 | 0.8345 |
| **Wait acknowledge** | | 0.4648 | |
| **Wait response ready** | 5.2854 | 5.2912 | 9.3587 |
| **Read response** | 0.4155 | 0.7966 | 0.8005 |
| **Time per operation** | 6.8932 | 7.0149 | 11.4585 |
| **Number of operations** | 12 | 3 | 11 |
| **Total time** | 82.7184 | 21.0447 | 126.0435 |
| **Total** | | 229.8066 | |

Table 12: Timing information for the PN532 in milliseconds

The difference between read and write exchange time per operation is obviously due to the time it requires to write into the flash memory.

**5.1.4.2 Signature generation** Teensy 3.1 has a C/C++ compiler and as TweetNaCl is written in C, no modification is needed to get Ed25519 working on the Teensy. One thing to take into account is the fact that the signature generation algorithm expects the private key and public key to be both available. However, the public key can be generated from the private key but is an expensive operation. Over 100 iterations the Teensy took 692.36 *ms* on average for the algorithm to calculate the public key from the private key. However, this operation only has to be done once on startup of the Teensy and would thus not impact the time per toy.

To generate a signature only one function call to `crypto_sign_ed25519` is needed in the TweetNaCl library. It took on average over 100 times, 696.25 ms. This means that when adding the time it takes for all MIFARE operations, it takes a total of 926.0566 *ms* to apply the proposed signature scheme to a tag which is awfully close to the one second limit that was initially given.

**Optimizing** Ed25519 is clearly aimed to have fast secure cryptography. However, it turns out that very small trade offs could be made that would perhaps make sense in this particular setting. Figure 16 gives an overview of the signature generation of Ed25519 and Figure 17 gives an overview of the signing part of Ed25519. The SHA512, Low 256b and Bit flips steps in Figure 17 are done every time when calling the sign function and could be precomputed at key generation.

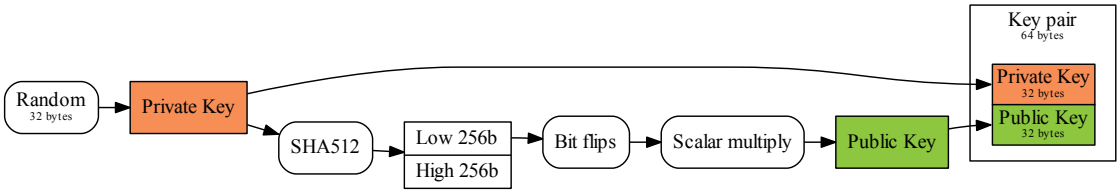


Figure 16: Key generation for the Ed25519 algorithm in the implementations for TweetNaCl, SUPERCOP and $\mu$NaCl.

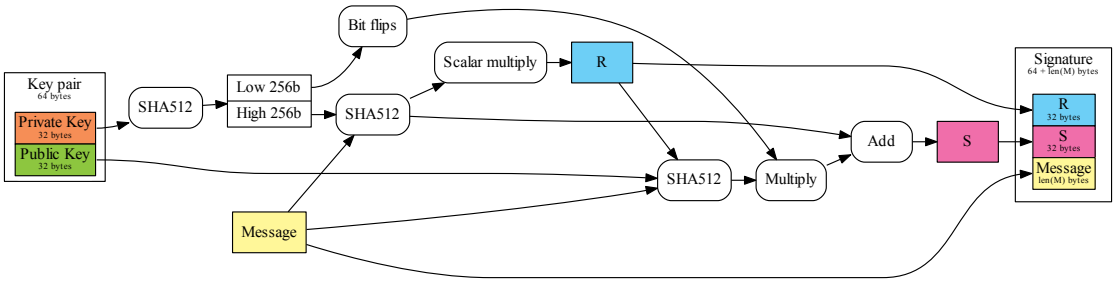


Figure 17: Signature generation for the Ed25519 algorithm in the implementations for TweetNaCl, SUPERCOP and $\mu$NaCl.

As a consequence this would remove one SHA-512 operation but increase the key-pair with 32 bytes of initialization vector. The reason why this is not done normally is that the speed improvement is very limited compared to storing 32 more bytes as only 10 *ms* per sign operation where saved on the Teensy 3.1. However, when producing one million toys, 10 milliseconds translate to almost three hours. Seemingly insignificant improvements could have an impact in the bigger scale. Figures 18 and 19 show the possible new structure of the Ed25519 sign and key generation functions.

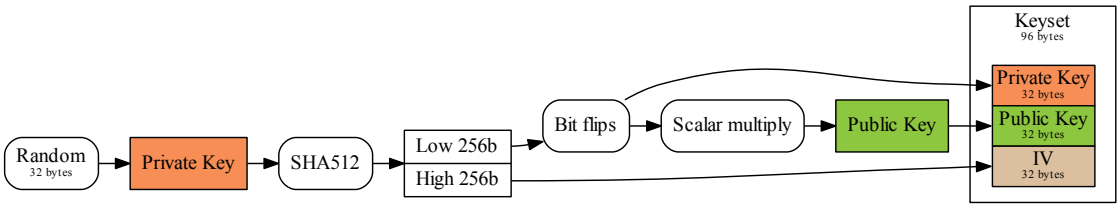


Figure 18: Key generation for the slightly modified Ed25519 algorithm in the implementations for TweetNaCl, SUPERCOP and $\mu$NaCl.

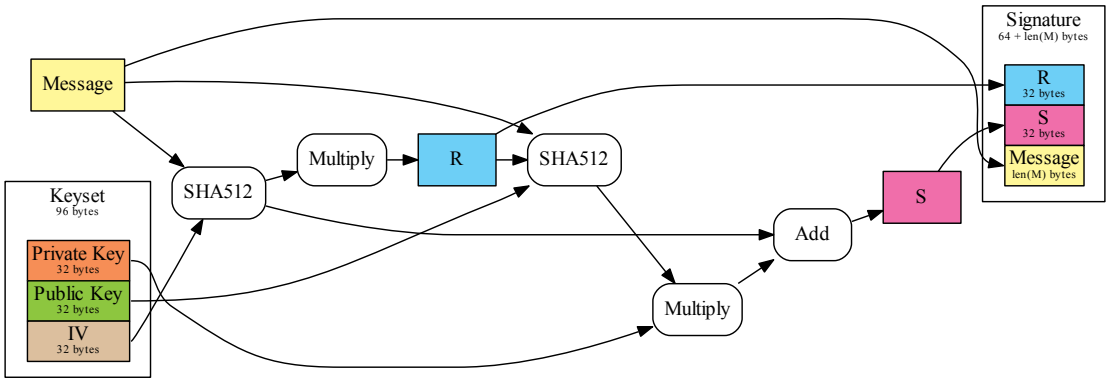


Figure 19: Signature generation for the slightly modified Ed25519 algorithm in the implementations for TweetNaCl, SUPERCOP and $\mu$NaCl.

Depending on how much focus there is on optimizing the programming step in the production process, these optimization could be applied.

**5.1.4.3 Results** Without changing TweetNaCl and given the fact that it is clearly optimized for size brings the total time for reading, signing and writing under one second. If the need ever arises to make this step in the production faster, it can be achieved by switching to a more speed oriented Ed25519 implementation and perhaps even use the small optimization described. Moreover, another MIFARE reader IC can be used that is designed for industrial reading and writing of tags instead of the PN532.

Later in the project the full specifications of the programming station where available and it turned out that the programming stations are severely less powerful. The production process is explained in Section 5.1.3 and shows that an 8 bit MCU clocked at 8 MHz is used in combination with the official reader.

### 5.1.5 Prototype two

After getting information about the specifications and setup of the programming station, it was clear that the 8 MHz clocked ATmega128 certainly would not cut it. Normally, the ATmega128 runs at 16 MHz and the production company agreed to the request to replace the 8 MHz crystal to a 16 MHz crystal of all programming stations if the described signature scheme would be used. There is some research available on the feasibility of ECC cryptography on 8-bit architectures and from this the ATmega128 could be powerful enough to do everything it has to do in the acceptable time frame of two seconds [17, 9, 14].

The 8-bit centered $\mu$NaCl library is especially suited for the task, as the paper even gives an ATmega128 as an example [17]. The paper did not provide timing information in seconds, only in cycle and the Ed25519 sign operation takes 23216241 cycles. While one could divide this number by the 16 MHz to get the time in seconds, it is not clear if this time would be representative of the

time it takes in the real world. In the hypothetical case where the MCU could execute one cycle per clock tick, it would take 1.45 seconds and that would be certainly promising.

Another team of researchers implemented Twisted Edwards curves on 8-bit AVR-based sensor nodes [9]. These sensor nodes use the exact same MCU as the programming stations and make the research very applicable. The given implementation was able to do a 158-bit scalar multiplication in 0.79 seconds on average. However, this is using an 158-bit key and according to ECRYPT II is two bits under the minimum general purpose protection. Obviously the key size could be increased to give enough security while still being under the two second boundary. A major difference between the fast implementation by Dalin Chu et al. and the Ed25519 implementation in $\mu$NaCl is the absence of any protection against side-channel attacks. However, there is no immediate danger in having side-channel attacks because all the signature generating devices are in a controlled environment with no outside access. This faster implementation is not released to the public, but if it turns out that $\mu$NaCl does not provide enough speed, this alternative could be explored.

To help prototype the signature scheme, two programming station with 16 MHz crystals installed where sent to the studio. The original code together with the hardware schematics of the programming station where also provided.

**5.1.5.1 Porting the firmware to 16 MHz** The original firmware was written with an 8 MHz clock speed in mind and did contain communication over SPI, Universal Asynchronous Receiver/-Transmitter (UART) and USB. When turning on the programming station, it obviously did not work because all communication timings where off, but more importantly, the CKOPT fuse has to be programmed to let the ATmega128 work correctly with an external 16 MHz crystal. After fixing timing issues, small compiler differences and the fuse settings, the programming station was producing valid playable tags and it was ready to be extended with the $\mu$NaCl library. However, static analyses showed that the programming station was using 2441 bytes from the total 4196 bytes available SRAM. Considering that the $\mu$NaCl high-speed implementation of the Ed25519 sign operations uses 1642 byte of stack space, somehow the memory usage of the programming station had to be decreased.

**5.1.5.2 Flash strings** Upon closer inspection of the code, there where 84 strings present that where initialized in SRAM. The total length of all the strings was 1282 characters and by moving all strings to flash and only loading them into SRAM on runtime, around 1400 bytes of SRAM was saved. To verify that reading in all strings from flash would not be adding to much time, timing information was collected. Reading one string from flash took around 30 ns and over the span of one tag programming, the print function is called around 25 times. This would mean that moving the string to flash added approximately 750 ns. However, as can be seen in Table 13, moving strings to flash showed an increase of around 0.25 seconds. This can be explained by the rather large differences in time between each run and is probably not due to the moving of strings but just inherit to the device and communication with the peripherals. Later in the process, a timing issue in the USB communication was found which could have caused the fluctuation between runs.

**5.1.5.3 Ed25519** The next step was to add in signature generation. Implementing the $\mu$NaCl library together with some glue code to chain everything together was trivial. Using the same timing gathering technique as before, one `crypt_sign` call took 1.413 seconds and this was well within reason as our new time limit was two seconds. However, the total time per programming pass added 2.30 seconds over just the 16 MHz port as can be seen in Table 13. This is due the fact that writing those 11 key B fields took on average 0.84465 seconds. Taking the total time over the two second limit. Moreover, after generating tags using this schema and starting to plan out how signed tags would work in logistics, it became apparent that the key B field can not be used for storing the signature. The blank MIFARE Classic tags from NXP will be formatted to have all the sector trailers correctly set and stored for further usage. However, the formatting steps makes the key B field read-only. Seeing as there was a significant amount of inventory of these formatted tags, the key B fields could not be used and another solution had to be found.

**5.1.5.4 Moving the signature location** To free-up 64 bytes of storage outside the key B fields for the signature scheme, a system was made to support multiple tag layouts for different toys. This meant that fields could be moved and with the help of this technology, blocks 2, 4, 34 and 62 became available for use for storing the signature. One extra byte was also reserved from the key B field in sector zero to be able to define some meta information. This was possible because only the key B field in sector zero was writable after the formatting step. Placing the signature in these blocks would not require any logistic changes and the signature of a tag could even be updated in development so that tags could be reused.

Storing the signature in blocks 2, 4, 34 and 62 the signature was now in a writable area. This means that the preferable constraint of storing the signature in a read-only location is violated. This is not a big problem as the software and tools produced by the studio will never overwrite or change the signature if not explicitly ordered to do so. However, there is always a chance that unforeseeable circumstances could potential corrupt valid signatures making genuine toys unusable. The potential risks of moving the signature to writable memory may be outweighed by the benefit of implementing increased security.

**5.1.5.5 Results** To accurately collect timing information, a logic analyzer was connected to the programming station. A specific pin was kept high as long as the programming station was creating a tag. With this, precise timing information could be collected to see the exact impact of the changes. Table 13 shows timing information where each column is the state of the firmware where "16 MHz port" is the initial port to 16 MHz without any changes, "Flash strings" is the firmware after moving all strings to flash, "Ed25519" includes the $\mu$NaCl library to generate a signature and writes it to the 11 key B fields on the tag and "New location" is the final state of the prototype after moving the signature location to blocks 2, 4, 34 and 62.

| | **Programming a single tag** | | | |
|---|---|---|---|---|
| | **16 MHz port** | **Flash strings** | **Ed25519** | **New location** |
| | 6.724 | 6.563 | 8.895 | 8.501 |
| | 6.594 | 6.604 | 8.834 | 8.380 |
| | 6.674 | 7.218 | 8.845 | 8.541 |
| | 6.312 | 7.007 | 8.865 | 8.582 |
| | 6.785 | 6.906 | 8.774 | 8.269 |
| | 6.473 | 6.896 | 8.956 | 8.299 |
| | 6.614 | 6.694 | 8.865 | 8.369 |
| | 6.624 | 6.775 | 8.966 | 8.400 |
| | 6.503 | 6.856 | 8.834 | 8.561 |
| | 6.372 | 6.564 | 8.865 | 8.591 |
| **Average** | 6.5675 | 6.8083 | 8.8699 | 8.4493 |
| **Increase over 16 MHz port** | | 0.2408<br>3.67 % | 2.3024<br>35.06 % | 1.8818<br>28.65 % |

Table 13: Total time of one complete programming pass for the four different states of the prototype over 10 experiments. All numbers are given in seconds except the three percentages

The final state of the prototype took on average 8.4493 seconds in total which is an increase of 1.8818 or 28.65 % compared to the 16 MHz port and is under the 2 second limit. While working with the firmware, it became apparent that the programming stations are not optimized for speed at all and improvements could be made to reduce the time. It would probable be possible to shave off around 1 second of the total time by removing redundant code and unused wait times and depending what the production company decides can be done or not.

### 5.1.6 Key Management

Storing private keys is an extremely important part of any cryptographic system and is often done using an Hardware Security Module (HSM). Such a device has a tamper resistant storage unit to store the private keys and can sign or encrypt data send to the device. This makes sure that the (private) key never has to leave the HSM. The secure storage makes it hard for attackers to retrieve the key from an HSM even when the attacker has access to the physical device. However, because of the thousands of programming stations it would be way to complex to manage all private and public keys, let alone the costs associated to buying thousands HSMs.

Using an Local Area Network (LAN) to let the programming stations use a single HSM per factory would reduce the complexity of the key management and costs significantly. However, there are no networking capabilities on the factory floors where the production process takes place. Moreover, the programming stations do not have any networking capabilities making network attached HSM impossible as well.

The unavailability of a HSM means that somehow the keys have to be entered into each programming station. The company in charge of the production process will annually flash a new version of the firmware into the programming stations and this annual enrollment could be used to program a private key into the programming stations. However, this also introduces many attack vectors on the private keys and may also introduces human error. To this end, a key management strategy is presented that balances security and usability.

**5.1.6.1 Annual renewal** Instead of using a single key, the annual firmware upgrade of the programming stations can be leveraged to roll out new private keys each year. This reduces the impact of a stolen or broken private key to a single year. Annual renewal of the private keys does not significantly impact the production process as the firmware would be flashed each year to all programming stations anyways.

**5.1.6.2 Number of private keys** Giving every programming station an unique key makes it possible to blacklist a specific programming station if it ever gets stolen or lost. However, that means that thousands of public keys need to be installed in the game. Not to mention the secure storing of all private keys.

The number of private keys could be reduced to an $X$ amount of keys that is equal to the number of factories. With such a setup, the keys could be linked to specific factories and this could help forensics and tracing down issues. However, this only adds minimal benefits against significant overhead of key distribution. To further reduce to complexity of key management, only two annual keys are used, the production key and a debug key. This debug key will be available to the studio so that valid toys can be produced for debugging purposes while not requiring the presence of the private production keys.

**5.1.6.3 Storing keys** All key pairs have to be known from the beginning to support forwards compatible toys (i.e. a toy can get a repose or another model in later years and still has to work in the year when the toy was first released). Storing these keys securely and reliable presents a challenge. Placing the keys on the internal network (encrypted or not) means that the keys are easily accessible and prone to malware. Not only that, it is not guaranteed that the network will be the same for the next eleven years and that in a possible migrate such files could be lost. Storing the files on removable media like USB flash storage, CD, DVD or a portable hard disk has a chance of becoming unreadable or corrupted over time making the private keys unreadable. To this end, the safest storage is arguably plain paper backups. The paper sheets containing the private keys should be stored in fire proof containers over at least two different locations to minimize the chance of losing the private keys.

**5.1.6.4 Paper key-pair format** Each key-pair is stored on a single sheet of paper and contains a legal disclaimer, a small description on how to use it and the public and private key in the following forms.

- **Private key in hex**, example `59 CB 51 02 59 3A 54 6E 88 A0 10 CF 0D 32 FC 54 9D 1E 27 51 40 99 4A 32 66 15 79 C5 18 91 D1 65 17 75 BE`

- **Private key in C++**, example

```
// 32 bytes reserved for runtime public key extension
unsigned char eccKey[67] = {
    0x59, 0xCB, 0x51, 0x02, 0x59, 0x3A, 0x54, 0x6E, 0x88,
    0xA0, 0x10, 0xCF, 0x0D, 0x32, 0xFC, 0x54, 0x9D, 0x1E,
    0x27, 0x51, 0x40, 0x99, 0x4A, 0x32, 0x66, 0x15, 0x79,
    0xC5, 0x18, 0x91, 0xD1, 0x65, 0x17, 0x75, 0xBE,
};
```

- **Public key in hex**, example `1F C5 E2 5D 44 B0 61 66 F6 52 DF E1 31 72 F5 42 4C 3C 19 31 A8 63 00 07 3F 8A 19 C9 B5 2C DE C0`

The reason why the private key is not 32 bytes but is actually 35 bytes is because a checksum and some meta information for the key is included and the structure of the key is presented in Figure 20. This is mainly done to eliminate the chance of human error. When software processes the private key, it has to verify the stored checksum to the CRC16-CCITT over the 33 other bytes. If those match one can be relatively sure that the key is correct. The KeyIndex is meta information to quickly pick the correct annual key instead of having to try different keys as it not known if a debug or production key was used to sign the tag.

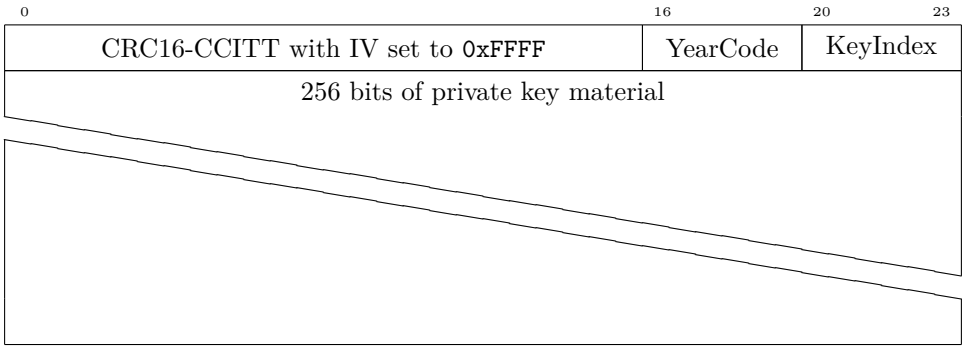


Figure 20: Structure of the printed private keys, the total size is 280 bits (35 bytes).

All debug private keys are available to internal tools through an user based ACL because simple usability outweighs the risks. Because the debug keys are available in electronic form, each debug key is not printed on a single sheet but instead all debug keys are printed on one sheet for backup purposes.

**5.1.6.5 Annual production key enrollment** To keep the risks to a minimum, the private production keys should be hand carried to the company responsible for building and flashing the firmware. It can than be entered into the programming station firmware which does a self check on the checksum to make sure the key has been entered correctly. The YearCode is in there to prevent accidental production of toys with the wrong key. If the YearCode on a toy is not equal to the YearCode of the private key, the programming station shows an error and refuses to program tags.

**5.1.6.6 Producing valid tags while working on the game** To facilitate users who work on the game, internal tools use a set of debug private keys to produce valid toys. The debug keys are exactly the same as the production keys but have the key index set to zero. The tools will use the correct debug key based on the YearCode of the produced toy. This ensures that the tool will automatically produce valid and working toys for the upcoming eleven years.

Every year, Quality Assurance (QA) should test tags signed with the production key. To facilitate this, tools should include a way to override the automatic debug key with a given production key. The users responsible for the test can request the paper production key backup and use it to produce valid production toys. The production key should only be stored in volatile memory to prevent storing the production key electronically. To minimize human errors, the same checksum

strategy is used when entering the production key in the software as with the annual production key enrollment.

### 5.1.7 Attack analysis

The next sections will present attacks to bypass the security and the implications of executing such an attack.

**5.1.7.1 Downgrade attack** Generally each toy has a year code field that describes what year this toy has been produced. However, this code may not be checked due to forward compatibility requirements. Moreover, this means that by downgrading the year code of a toy to before 2016 and by just not including a signature, the security system could be bypassed. To prevent this, the data definitions for toys inside the game should also contain when this toy was produced. With this information the game can verify that the year code is at least not smaller than the first time the toy is produced. This would prevent new toys from being downgraded to the time where there was no security support. However, this does not prevent toys released before 2016 that are being reissued or released with a different model in or after 2016 to be downgraded. To also protect these toys, the game data should also contain release dates for these alternative releases of toys. Then for each toy it can be checked if that specific model has been released with or without a signature and if it contains a signature, from what year to use the public key to verify the signature instead of having to rely on the tag data which can contain crafted data by an attacker.

**5.1.7.2 Bypassing the security check in the game** The signature verification comparison in the binary of the game was designed to make it hard for the normal users of the game to bypass such security measure. The goal of this countermeasure is to make it hard for attackers to sell plug and play products. If the buyer also has to somehow bypass the security on the console, it makes the consuming market small enough to make piracy unprofitable.

**5.1.7.3 Using debug keys** If toys signed with the debug keys would work in the final retail game, it would make the debug keys as valuable as the production keys. However, the development process could require the debug keys to be present in the final game. This could mean that having the debug keys enabled in the final game could outweigh the potential security risks. If the development process allows for disabling the debug keys in the final game, this would add extra security.

**5.1.7.4 Cloning toys** As described in Section 5.1.2 bypassing the protection by making clones is still possible and is hard to protect against.

### 5.1.8 Conclusion

Security by design is sometimes not even done in mission critical systems, so it is understandable that games are often not designed with security in mind. However, that means that when security becomes important, it is much harder to implement security at all and near impossible to do it well. The described countermeasure balances the added security, costs and usability by trying to keep changes to the current processes to a minimum while adding enough to security to make it significant harder for attackers to get any commercial gain of piracy.

While the signature scheme does not prevent against 1-on-1 clones, it does make it extremely difficult for attackers to generate new tags to find unreleased toys as can be seen in the attack-defense tree shown in Figure 22.

It looks that the signature scheme protect against emulating toys according to the attack-defense tree presented in Figure 23. However, this is not truly the case according the attack-defense tree given in Figure 21. Using this attack, the complete toys can be read using a genuine reader and simply replayed to fool the game into thinking one owns such a toy. This is similar to cloning and without PUF challenge-response mechanisms there is no real way to protect against.

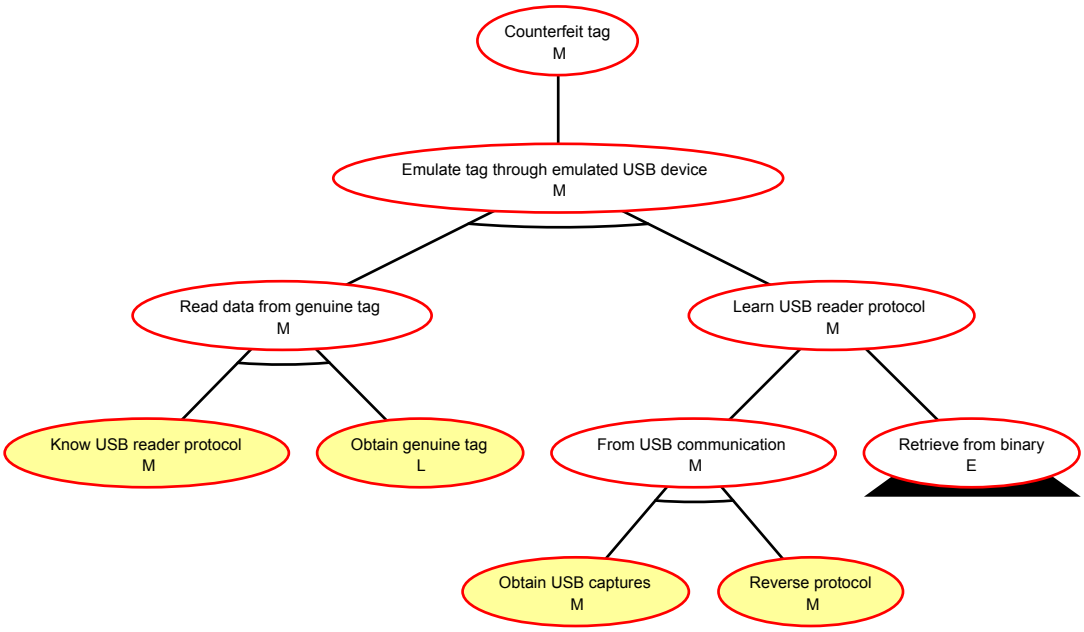


Figure 21: A different attack-defense tree to emulate a tag through a custom USB device which effectively bypasses the added signature countermeasure in contrast to the the attack-defense tree given in Figure 23. The “Retrieve from binary” node is analog to the one given in Figure 7

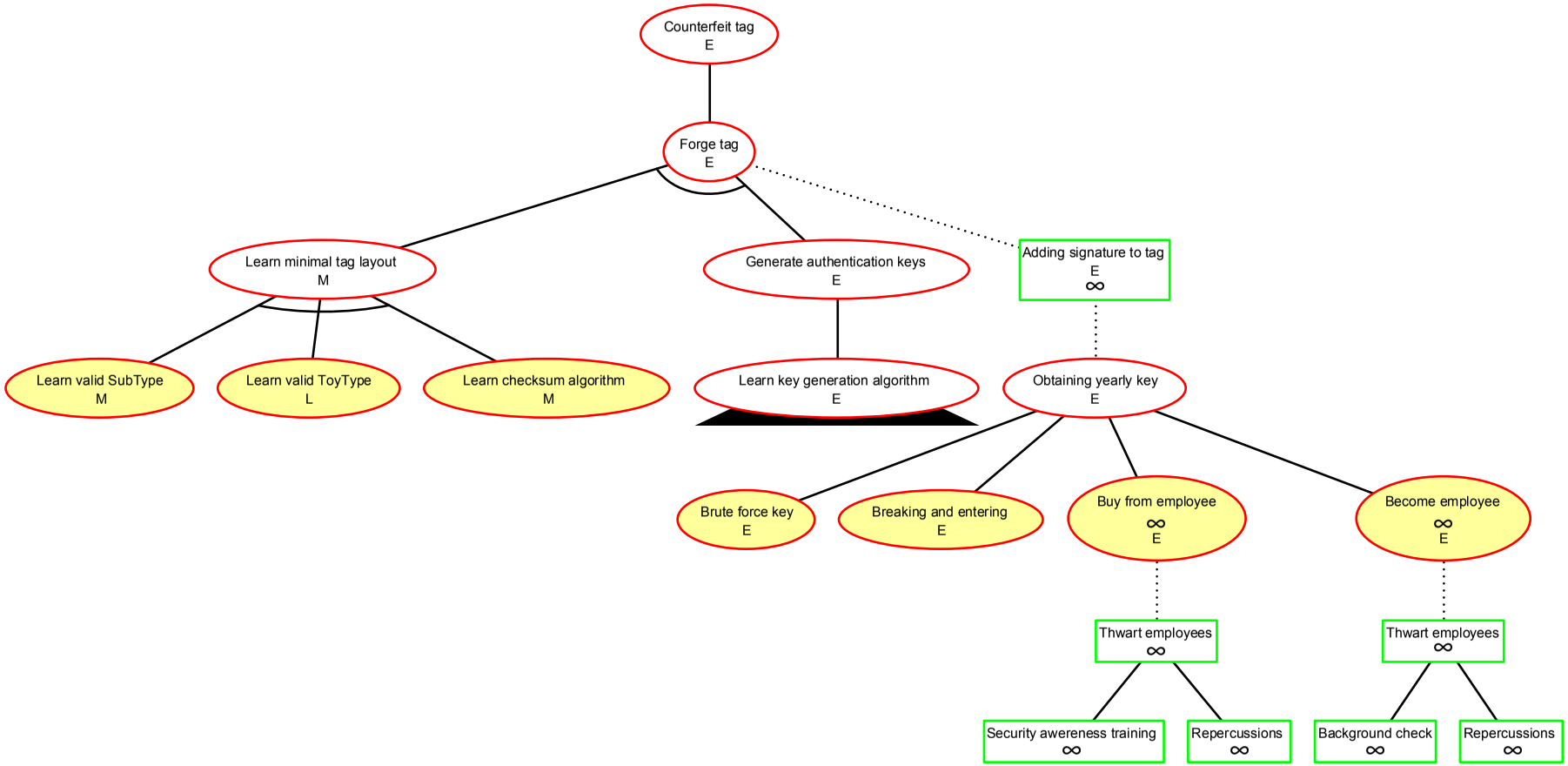


Figure 22: Attack-defense tree to forge a tag after the added countermeasure described in Section 5.1, see Figure 7 for the Learn key generation algorithm tree

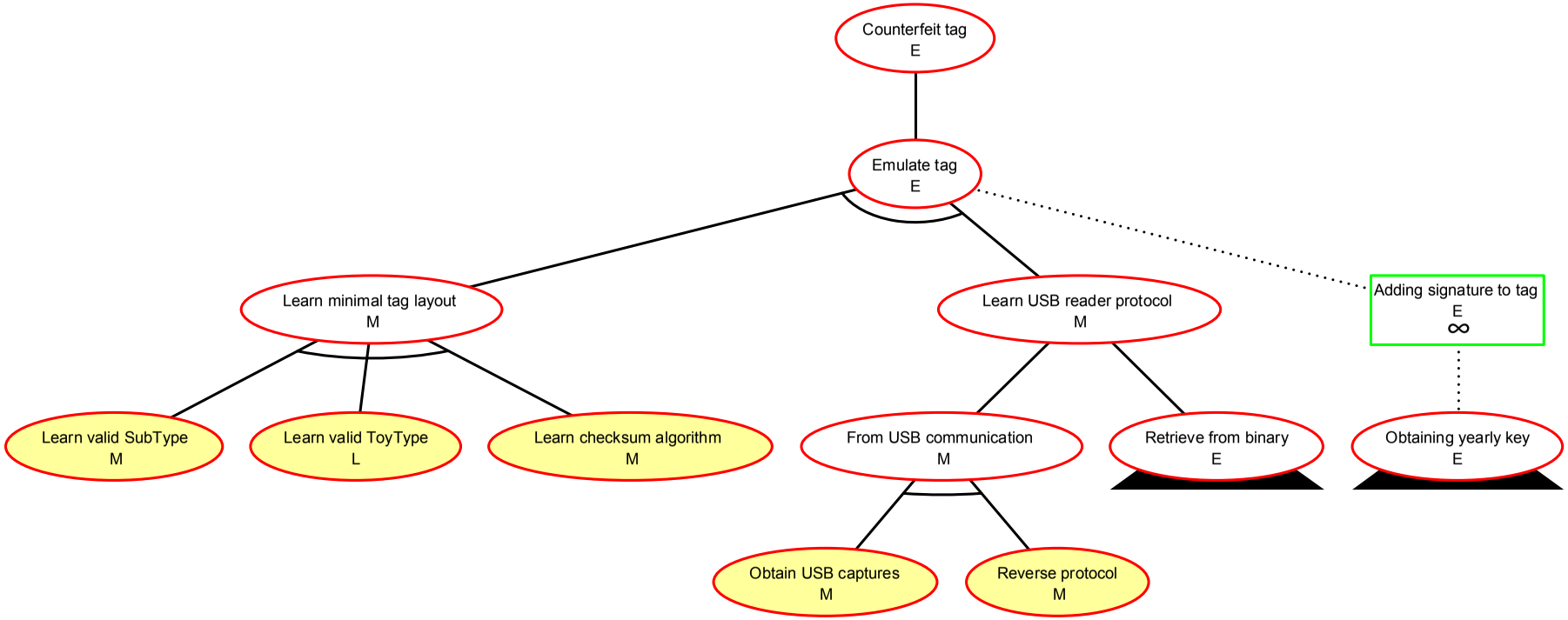


Figure 23: Attack-defense tree to emulate a tag through a custom USB device after the added countermeasure described in Section 5.1. The "Retrieve from binary" node is analog to the one given in Figure 7

## 5.2 Hardening the reader

### 5.2.1 Authentication of the readers

Some vendors obligate third party hardware developers to included an Infineon security IC inside peripherals for their consoles. The console can verify the genuineness of the peripheral by sending a challenge to the device over the USB. The USB device will then have to forward the challenge to the Infineon IC so it can calculate a response and return it to the console. If the response is invalid, the device is not accepted by the console else it can be used by the software running on the console. This mechanism protects such consoles from the described attack of creating a device that can emulate a reader. However, an older console uses the Infineon SLE 66PE and Christopher Tarnovsky was able to break into this IC to retrieve its secrets and hence defeating the security [42]. Luckily, Chritopher has no intention of releasing any secrets to the public and repeating the attack is tremendously time consuming and expensive making the protection not totally broken for the general public. The newer console uses a better Infineon IC and according public knowledge, is up to this day not broken.

The reason why hardware security is significantly harder to break than software security is the fact that the secrets resides in a secure element that has many layers of protection against attackers. To get to through these layers, expensive and complex tools are required like a Focused Ion Beam (FIB), where in software, strictly speaking no extra tools are required to defeat the security.

When peripherals are used by the game as with games, a security IC could be added to the device to be used for authentication. Using public-key cryptography the secret only has to reside in the peripheral making it rather difficult for attackers to bypass. However, adding a dedicate security IC to any device adds a significant production cost.

A better scenario would be where the console manufactures would have a hardware/software security system in place that would prevent unauthorized devices from working on there respective consoles.

**5.2.1.1 Software only** It would be possible for game developers to implement security mechanism to protect against unauthorized game specific peripherals. However, if there is no hardware support, it would need to be done in software, either in the game or the peripheral or both. Because the software only gives so many options to protect against attackers, it is no where near as secure as a hardware based solution. This makes it only a relatively small barrier for attackers to get through.

Considered that when an attacker takes the time to reverse the software to get secrets, it really does not matter if there are a more layers of software protection. It becomes a totally different ballgame when using state of the art advanced obfuscation techniques and virtualization to hide the keys in software from prying eyes, but this is not feasible on current microcontrollers used throughout cost effective embedded hardware like readers as they are simple not powerful enough. This makes adding symmetrical, asymmetrical cryptography or Hash-based Message Authentication Codes (HMACs) more a nuisance to both the attackers and developers than anything else. It will also break backwards compatibility when using newer readers with old games. Moreover, it will also break forwards compatibility when using older readers with new games.

Adding purely software protection adds a little bit of protection but compatibility is lost as well as the extra maintenance involved keeping the security from working well. However, perhaps a genuine reader can be detected by specific behavior.

**5.2.1.2 Detecting genuine readers** Most companies currently do not have a protocol implemented in their readers to detect if a reader is genuine or not. However, inherited behavior from the software hardware setup could be leveraged as a detection mechanism to test if the device behaves like a genuine reader. However, even if such deterministic and reliable behavior would be present, it could always be emulated by an attacker. Nevertheless, it could be used as an indicator that the reader is an imitation.

Imitation readers will almost always emulate the NFC system in software. As that enables the emulated device to return the tag data from memory instead of requiring a physical NFC tag.

Hence, if there is specific behavior present due to the NFC communication, it could be leveraged as a detection mechanism.

One can issue a read command to the reader to read a specific block from the tag. If the block is readable, the reader will return the data. However, if somehow the reading fails because the block is out of range or the tag has the wrong security settings, the reader will always retry exactly three times. Moreover, the reader operates on a command loop and every time the command loop runs, a command is processed and the retry mechanism is called. This means that while the reader is retrying, another command can be send to the reader. If the second command can be completed within that one command cycle a response is returned immediately while the reader is still trying to read the tag. This has the following behavior from the perspective of the reader.

- Receive read command and read the tag
- Reading fails, set the retry count to 2
- Receive another command, handle the command and send back response
- Retry count $> 0$, reading fails, set retry count to 1
- Retry count $> 0$, reading fails, set retry count to 0 and send back failed read response

A reader that uses real NFC communication with retries will flip the order of the responses. While if the reader would be emulated, it could immediately return the data for the read command even if it would fail and that would cause the response order to not get flipped. This behavior is present in the first reader throughout the last generation. To thoroughly test the reliability, a Python script was created that implemented this test and is given in Appendix C. The test works for all generation readers and only caused false positives once in a blue moon because of the timing requirements. Running this test a few times while playing the game should give definitive result if the device connected has this behavior or not. To reiterate, this does not mean that the device is indeed genuine as this behavior can be synthesized in an imitation reader.

**5.2.1.3 Conclusion** As long as all calculations and devices are done at the user, it is impossible to completely prevent attacks. Moving calculations to the cloud where the attacker can not access the device could help. However, care have to be taken to side channel attacks like timing attacks. The detection method works well but is also possible to bypass by simulating the same behavior. However, this method could be added to a supervisor system that checks for malicious or weird behavior constantly and if with enough evidence of cheating or piracy, penalizing actions could be executed.

### 5.2.2 Protocol verification through fuzzing and side-channel analysis

While not directly a pressing attack vector, making sure that the reader is working as intended and adheres to the protocol specification is tremendously important considering the number of readers being made.

The protocol the readers use to communicate with the host is a relatively simple command response protocol. However, the readers also has a few commands that will trigger certain behavior like changing the brightness or color of Light-Emitting Diodes (LEDs) or turning the RFID antenna on and off that does not generate a response. Verification of such commands can only be done by having a human watch and or test the reader for specific behavior. Currently, all commands that do not generate a response, either turn on or off a peripheral and that in turn will change the power draw from the USB power line. This change in power can be detected and is more commonly known as Simple Power Analysis (SPA). It is normally used to mount side-channel attacks on cryptographic devices but in this case could be used for automatic testing.

SPA can be used to check if commands change the device even if there is no response, fuzzing techniques will be used to fuzz the protocol. While fuzzing and SPA could be done in great detail using state of the art technologies, it currently is sufficient to have a fully automated process to quickly test readers and list all the commands the device supports even it is not according to specifications.

**5.2.2.1 Setup** With specialized hardware for side-channel attacks, the sensitivity can be high enough to analyze individual instructions. However, such setups are expansive and more often than not, intrusive. For the purposes of this setup, making any modification to the reader or the USB cable would be undesirable. Moreover, the current draw of driving peripherals will probably be high enough to be detectable from the total USB voltage. Figure 24 shows the setup of the hardware. A Saleae Logic Pro 8 was used as a voltage meter, the benefit of using this device to capture the traces was that the device has an Application Program Interface (API) to fully automate the process. Different resistor values where tested to balance the signal amplification versus the stability and 10Ω did not impact the reader in a negative way while causing more than enough amplification.

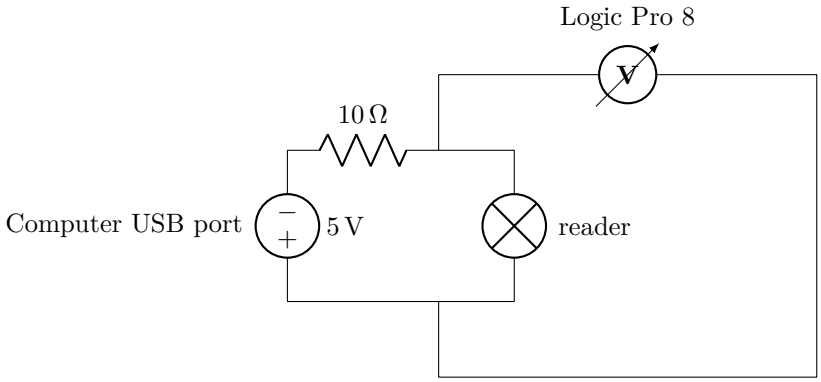


Figure 24: Circuit diagram to show the setup of the logic analyzer in respect to the USB port and the reader. Only the USB ground and power lines are shown.

**5.2.2.2 Fuzzing strategy** The only goal for this automated process was to find the supported commands by the reader under test regardless if there was an USB response or not. To that end, the fuzzing strategy was made as simples as possible. This meant that enumerating over the command byte would be sufficient and doable given the range of 0 to 255. However, some commands change the state of the device that prevent subsequent commands from working. While other commands needs arguments to work. The following strategy encompassed all requirements and had great coverage.

- **0 to 255:** Set rest of buffer to `0x00`

- **0 to 255:** Same test to catch any command that got enabled by a higher command
- **0 to 255:** Set rest of buffer to `0xFF`
- **0 to 255:** Same test to catch any command that got enabled by a higher command
- **255 to 0:** Set rest of buffer to `0x00`
- **255 to 0:** Same test to catch any command that got enabled by a lower command
- **255 to 0:** Set rest of buffer to `0xFF`
- **255 to 0:** Same test to catch any command that got enabled by a lower command

Going twice over the same range and also in reverse is required to catch commands that require a state that can be turned off or on in multiple ways. To send all these command, 2040 commands have to be send to the reader which only takes 40.8 seconds given the internal processing delay of 20 ms.

**5.2.2.3 Signal processing** Finding out if a command changed the state of a peripherals could be achieved by comparing the power trace before the command and after the command. The upper capture limits for the logic analyzer is a sample rate of 50 MS/s with the duration limited by the total RAM available to the computer. To keep the processing speed high, the sample rate and capture duration should be as low as possible while having enough fidelity to detect changes in the power trace. Using some preliminary tests it was clear that 40 *ms* at 125 kS/s had enough fidelity while keeping the required processing time of the trace reasonable. It turns out that while specifying a precise time span to sample, more samples will be captured by the logic analyzer and so the traces were effectively 68 *ms* long. All graphs that depict power traces in this section are down-sampled to 25 kS/s.

Figures 25 and 26 show two power traces after sending a particular command. A clear Pulse Width Modulation (PWM) signal can be seen in the power trace shown in Figure 26, the trace even makes the intensity of the different RGB components visible. What immediately becomes apparent by Figure 26, is that traces will not be aligned in-between different captures. A way to align different traces is required to keep false positives to a minimum.

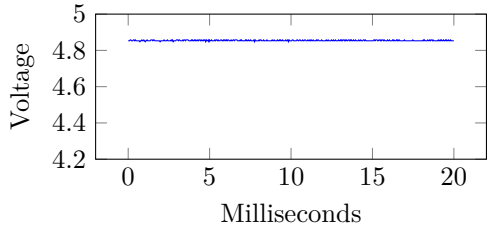


Figure 25: Power trace when idle, all peripherals turned off

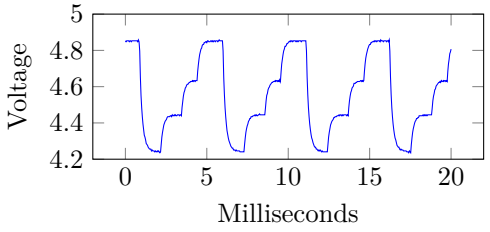


Figure 26: Power trace when setting one RGB LED to `E0`, `80`, `40` intensity

**Aligning traces** There will be two traces per command, namely $T_{pre}$ and $T_{post}$ where $T_{pre}$ is the power trace collected before sending the command and $T_{post}$ after. Each trace is a one dimensional array containing voltage measurements. Aligning $T_{post}$ with $T_{pre}$ is achieved by by calculating the cross-correlation between the the two traces and rotating $T_{post}$ by the lag with the highest correlation. This is far from perfect but sufficient enough for the purpose of the described system.

**Similarity** The similarity percentage between the two traces is defined by equation 1 and from here on will be named as Perfect Correlation Distance (PCD). Observations showed that 1 is

a good threshold for PCD. If it is equal or higher than 1 there was a significant difference in power consumption.

$$\frac{|T_{post} \cdot T_{post} - T_{post} \cdot T_{pre}|}{\frac{T_{post} \cdot T_{post}}{100}} \quad (1)$$

Using this formula was convenient as there already was logic to calculate correlation between two traces so nothing new had to be added to the system. Moreover, it turned out to work surprisingly well for this system. A more generic and better approach would be using something like the Pearson product-moment correlation coefficient to find the similarity between traces.

**Speed** While the USB communication was rather quick, the signal processing was not. This was due the fact that the software used (Saleae Logic 1.2.5) was still in beta and the public API could only automate the Graphical User Interface (GUI). Moreover, the API had a beta state as well, so multiple bugs where uncovered that had to be worked around to get an automated system working. All these factors contributed to the total capture and processing time of approximately 2 seconds, bringing the total test time to around 68 minutes. This was still reasonable as no human interaction was necessarily and only a few reader prototypes are produced over the development cycle. If more speed is required, other power trace capturing methods and native signal processing systems could be explored.

**5.2.2.4 Results** The described system was used to run automatic tests on different readers and the results can be seen in Table 14.

**False positives** The results for 2011 and 2014 show false positives around the `A` command. This is due to the fact that the reader processes that specific command rather slow and the change in the power consumption can be mid trace. This effect can be seen in Figure 27 and makes the subsequent command trigger a false positive. This could be solved by increasing the wait time between sending a command and triggering the capture. Delaying each capture by 13 *ms* increases the total capture time only by approximately 27 seconds and is neglectable in contrast to the 68 minutes.

The first 2016 prototype has three false positives that are just barely over the threshold PCD value of 1. This is due the fact that the RFID field stays on for approximately 39 *ms* and then turns off for approximately 26 *ms* as can be seen in Figure 28. Moreover, the traces are only approximately 68 *ms* long and will trigger false positives when the two traces are not well enough aligned.

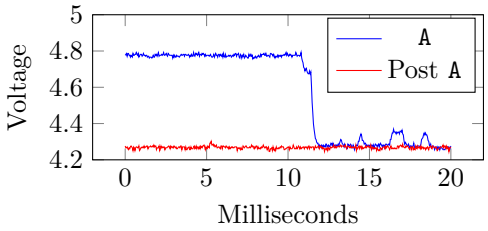


Figure 27: The `A` command changed the power draw mid trace making the subsequent trace trigger a false positive

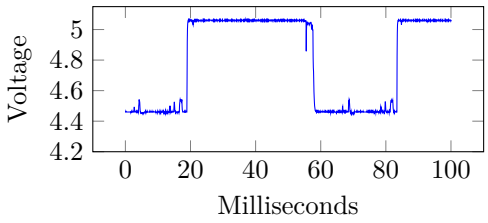


Figure 28: Power trace when the antenna is turned on in the first 2016 prototype

The second 2016 prototype fixes the weird behavior on the antenna signal and the traces looks similar to all other readers. Moreover, all false positives are gone in the second prototype as can be seen in Table 14.

**Bugs** Because the official specifications are known, it can be used to verify the results. Each result table contains a column called “Fuzzing Results” and “Specifications”. In each of these column there is a column called “Impl” which stands for implemented and “Resp” which stands for response. The results show that there are no commands implemented in the reader that according

to specification should not exists. However, in 2014 the commands `B` and `J` have responses while according to the specifications should not. This turned out to be a genuine bug and was fixed in future firmware versions. The 2014 reader had another bug, after a test run, the device behaved very slow and it is currently unknown why the device behaves like this. Only power cycling the reader made the device behave correctly again.

| 2011 | | | | |
|---|---|---|---|---|
| Fuzzing Results | | | Specifications | |
| Cmd | PCD | Resp | Impl | Resp |
| @ (40) | 7, 18 | ✗ | ✗ | ✗ |
| A (41) | 3 – 15 | ✓ | ✓ | ✓ |
| B (42) | 18 | ✗ | ✗ | ✗ |
| C (43) | 12 – 30 | ✗ | ✓ | ✗ |
| Q (51) | < 1 | ✓ | ✓ | ✓ |
| R (52) | 13, 17 | ✓ | ✓ | ✓ |
| S (53) | < 1 | ✓ | ✓ | ✓ |
| W (57) | < 1 | ✓ | ✓ | ✓ |

| 2014 | | | | |
|---|---|---|---|---|
| Fuzzing Results | | | Specifications | |
| Cmd | PCD | Resp | Impl | Resp |
| @ (40) | 2, 12 | ✗ | ✗ | ✗ |
| A (41) | 1 – 8 | ✓ | ✓ | ✓ |
| B (42) | 6 – 20 | ✓ | ✓ | ✗ |
| C (43) | 4 – 26 | ✗ | ✓ | ✗ |
| J (4A) | < 1 | ✓ | ✓ | ✗ |
| M (4D) | 1.5 | ✓ | ✓ | ✓ |
| Q (51) | < 1 | ✓ | ✓ | ✓ |
| R (52) | 7, 11 | ✓ | ✓ | ✓ |
| S (53) | < 1 | ✓ | ✓ | ✓ |
| W (57) | < 1 | ✓ | ✓ | ✓ |

| 2016 Prototype One | | | | |
|---|---|---|---|---|
| Fuzzing Results | | | Specifications | |
| Cmd | PCD | Resp | Impl | Resp |
| A (41) | 3 – 5 | ✓ | ✓ | ✓ |
| C (43) | 9 – 18 | ✗ | ✓ | ✗ |
| J (4A) | 1 | ✗ | ✗ | ✗ |
| Q (51) | < 1 | ✓ | ✓ | ✓ |
| R (52) | 3 – 5 | ✓ | ✓ | ✓ |
| S (53) | < 1 | ✓ | ✓ | ✓ |
| W (57) | < 1 | ✓ | ✓ | ✓ |
| b (62) | 1 | ✗ | ✗ | ✗ |
| } (7D) | 1.3 | ✗ | ✗ | ✗ |

| 2012 & 2013 | | | | |
|---|---|---|---|---|
| Fuzzing Results | | | Specifications | |
| Cmd | PCD | Resp | Impl | Resp |
| A (41) | 14 | ✓ | ✓ | ✓ |
| C (43) | 13 – 25 | ✗ | ✓ | ✗ |
| Q (51) | < 1 | ✓ | ✓ | ✓ |
| R (52) | 9, 12 | ✓ | ✓ | ✓ |
| S (53) | < 1 | ✓ | ✓ | ✓ |
| W (57) | < 1 | ✓ | ✓ | ✓ |

| 2015 | | | | |
|---|---|---|---|---|
| Fuzzing Results | | | Specifications | |
| Cmd | PCD | Resp | Impl | Resp |
| A (41) | 15 | ✓ | ✓ | ✓ |
| Q (51) | < 1 | ✓ | ✓ | ✓ |
| R (52) | 13 | ✓ | ✓ | ✓ |
| S (53) | < 1 | ✓ | ✓ | ✓ |
| W (57) | < 1 | ✓ | ✓ | ✓ |

| 2016 - Prototype Two | | | | |
|---|---|---|---|---|
| Fuzzing Results | | | Specifications | |
| Cmd | PCD | Resp | Impl | Resp |
| A (41) | 13 | ✓ | ✓ | ✓ |
| C (43) | 13 – 25 | ✗ | ✓ | ✗ |
| Q (51) | < 1 | ✓ | ✓ | ✓ |
| R (52) | 8, 12 | ✓ | ✓ | ✓ |
| S (53) | < 1 | ✓ | ✓ | ✓ |
| W (57) | < 1 | ✓ | ✓ | ✓ |

Table 14: Fuzzing results per reader year, 2012 and 2013 used the same hardware so only one year is shown

**5.2.2.5 Conclusion** Testing software is a good practice but automated tests are often difficult to do well on firmware for devices. Results show that non invasive SPA can be used in automated tests to check for change in behavior without the device communicating this change back to the host. The described testing method could be improved by better hardware to capture power traces and a more advanced way of processing and comparing power traces. Nevertheless, even without the improvements the tests where easy to execute and helped to make sure the readers where behaving according to specifications.

## 5.3 Hardening game code

The game implicitly trusts the responses by the reader and thus also in turn the data on the tags. However, as shown in previous sections, bot the reader and tags can return data that is crafted by an attacker. Making sure that no data returned by the reader or on the tag can be used as an attack vector is important. This section will explore ways of making the game code more secure against this data.

### 5.3.1 Toy fuzzing

Internal security tests revealed that the low level reader communication code in the game to be very robust. However, no automated or extensive fuzzing tests were done and the researchers noted that this would be a good addition. The researchers also noted that it would be hard to do automated tests of placing tags on the reader and having those tags change data like experience, progress, toy type, etc. It would be way to difficult to make an automatic machine to physically place tags on a reader, but instead a device could be build to emulate the reader and the toys. Figure 29 shows the hardware and software components that needs to be virtualized to have complete control over all data send to the game.

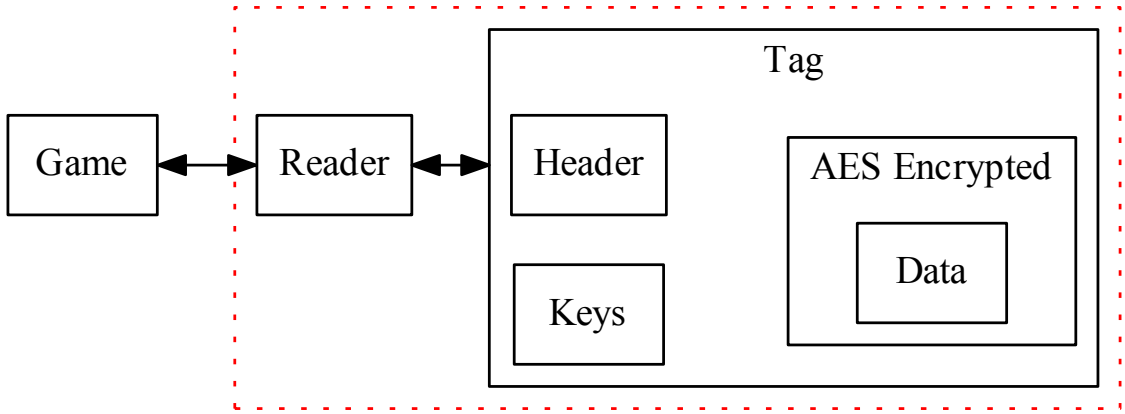


Figure 29: Game to RFID tag communication and important intermediate components

Facedancer is a device developed by Travis Goodspeed to make it easy to do research on USB [13]. This is achieved by using Python to emulate the full USB stack of a device making it possible to emulate all the aspects of any USB device from Python running on any computer. Travis released his code online[8] to emulate the full USB stack and simple devices. Porting the reader to Python was trivial, the only thing not supported by the original code was USB control transfers and this was required for emulating the reader.

**5.3.1.1 Requirements** The internal security test advised to add a way to disable security for a more accessible and workable automatic fuzzing system. However, it would make more sense to leave all security in place while fuzzing to have a more accurate real life scenario. That did mean that all parts of the tag including the security had to be generated or fixed on the fly, but this also made it possible to test complete game including the security layers. This section will explain each component in Figure 29 in more detail starting from the deepest component.

**Data and header definitions** While fuzzing without any data definitions can work fine, in this case it is more difficult because hardware is involved and it is harder to incorporate feedback from the game into the hardware fuzzing. Moreover, if one would like to test if a data field works correctly for all values a definition on where this field is would be required. Having a formalization

[8] https://github.com/travisgoodspeed/goodfet

of all different fields can provide this and also can be used for pointers to make fuzzing smarter. Figure 30 gives a simplified Backus-Naur Form for the structure of the JSON data. Using this schema, it was possible to define all the different fields on a RFID tag.

$$
\begin{aligned}
\langle \text{nodes} \rangle &\models \langle \text{node} \rangle \mid \langle \text{node} \rangle \langle \text{JSON seperator} \rangle \langle \text{nodes} \rangle \mid \lambda \\
\langle \text{node} \rangle &\models \langle \text{JSON name} \rangle \langle \text{JSON assign} \rangle \langle \text{elements} \rangle \mid \lambda \\
\langle \text{elements} \rangle &\models \langle \text{element} \rangle \mid \langle \text{element} \rangle \langle \text{JSON seperator} \rangle \langle \text{elements} \rangle \mid \lambda \\
\langle \text{element} \rangle &\models \langle \text{block} \rangle \mid \langle \text{offset} \rangle \mid \langle \text{size} \rangle \mid \langle \text{type} \rangle \mid \langle \text{childs} \rangle \mid \langle \text{value} \rangle \mid \lambda \\
\langle \text{block} \rangle &\models \textit{block} \langle \text{JSON assign} \rangle \langle \text{JSON number} \rangle \\
\langle \text{offset} \rangle &\models \textit{offset} \langle \text{JSON assign} \rangle \langle \text{JSON number} \rangle \\
\langle \text{size} \rangle &\models \textit{size} \langle \text{JSON assign} \rangle \langle \text{JSON number} \rangle \\
\langle \text{type} \rangle &\models \textit{type} \langle \text{JSON assign} \rangle \langle \text{types} \rangle \\
\langle \text{childs} \rangle &\models \textit{childs} \langle \text{JSON assign} \rangle \langle \text{nodes} \rangle \\
\langle \text{value} \rangle &\models \textit{value} \langle \text{JSON assign} \rangle \langle \text{JSON value} \rangle \\
\langle \text{types} \rangle &\models \textit{uchar} \mid \textit{ushort} \mid \textit{uint} \mid \textit{ulong} \mid \textit{char} \mid \textit{short} \mid \textit{int} \mid \textit{long} \mid \textit{array} \mid \textit{bitmask} \mid \textit{null}
\end{aligned}
$$

Figure 30: Abstract Backus-Naur Form for the JSON schema to describe all the data fields on tags (JSON syntax is omitted)

**AES Encryption** MIFARE Classic blocks that contain data fields are encrypted with Advanced Encryption Standard (AES) in Electronic Codebook (ECB) mode. The key is based on a constant, the block number and the header. Making sure no changes to the game are necessarily, all blocks that requires to be encrypted are encrypted just before they are sent to the game. Moreover, blocks received from the game that are encrypted are decrypted before processing the block. This can all be done in real time as the overhead of encryption is relatively small and no encryption chaining is used and prevents from omitting this security layer from the game when executing tests.

**NFC authentication** Because the tags are also emulated, no authentication keys are required to authenticate to a physical MIFARE Classic tag as the data from the emulated tags can just be used as is. However, for completion, the key derivation algorithm was implemented so that in the future, the generated tags could even be written to physical MIFARE Classic tags.

**Reader** The reader had to behave just like a real reader. This meant that all command responses had to be implemented correct but was trivial by following the internal specifications for readers. Furthermore an interface had to be created to control the reader and the fuzzing of the game. This was achieved by a text based User Interface (UI) that supports numerous commands.

- **arrive <slot><toytype>:** Generates a valid toy with a specific toy type and places it on the reader at a specific slot.
- **depart <slot>:** Remove a toy from the reader at a specific slot.
- **save <slot><filename>:** Saves the binary tag memory of a toy from a specific slot.
- **load <slot><filename>:** Loads the binary tag memory from a toy and then places it on the reader at a specific slot.
- **set <slot>[<field>=<value>]:** Sets one or more fields of a toy to the given value. An example would be `set 0 Header.ToyType=3 DataA.Region1.Experience=6000`, that would change the toy type to 3 and give it 6000 experience points. It would also force an departure and arrive event to force the game to reload the data.

- **reset <slot>:** Resets a toy data fields to default values.
- **fuzz <test type>:** Starts fuzzing according to a specific test, every test will have to reload the character with a depart and arrive between cycles.
  - **limits:** Sets all fields to the limit derived from the type of that specific field.
  - **bytes:** Increases each field by one.
  - **cyclet:** Cycle through all toy types, specific ranges or sets of toy types.
  - **cycles:** Cycle through all sub types, specific ranges or sets of sub types for a specific toy type.
  - **cycle:** Cycle through all known toy types and all known sub types.

A major problem was that the reader does not know when the game is done loading a toy. However, the game always writes a specific field in the toy if it is zero. By setting this field to zero before emulating placing the toy on the reader, the field can be watched until it becomes non-zero meaning that the game is done loading the toy. This is used extensively for the fuzz tests to go through the test cases as quickly as possible.

**5.3.1.2 Results** No bugs were discovered in the low level code, but that is not really surprising given that this code has been used through the games since the beginning.

The higher level code was remarkable resilient and this probably due to the generalizations and abstractions in the game engine. Only one minor bug was uncovered in an UI when fields where set to the theoretical. Nevertheless the bug got fixed and the UI can now handle these artificial enormous numbers.

**5.3.1.3 Conclusion** While only one bug was discovered, the system did add confidence to the stability of the game. Moreover, it turns out that this system could be helpful for QA. The QA department requires large amounts of tags and readers to test all the different toys while playing the game. This system can condense all the different tags to a single device and a computer or a laptop. This makes it easier to test the game and can improve the speed of testing. Moreover it can assist the tester by automating parts of the tests in a way no human could. Finding new ways to test the security and other parts of the system either automatically or semi-automatically can add to the total security and reliability of the product.

# 6 Conclusions & future work

To conclude the research, multiple sub-questions where stated and those will be handled first before the research questions will be answered.

- The question of "*What are the possible attack-defense trees?*" was answered in Section 3 by presenting three attack-defense trees (Figures 5, 4 and 6) that makes it possible to play the game with forged toys. The attacks-defense trees were limited to these three attacks because of the importance of the playable toys in the toys-to-life genre games for revenue.
- In Section 4 different attacks were mounted to find the MIFARE Classic authentication key derivation algorithm. This will answer the question "*What attack trees can be turned into practical attacks?*". The USB protocol was not attacked simply because it being trivial and already done. It became clear that the algorithm itself was not strong enough to prevent the calculation of all authentication keys from the third key. Moreover, there was no simple way of obtaining the second key which makes this attack unusable. However, it was possible to retrieve the firmware from a reader making it possible to reverse the key derivation algorithm.
- The impact analyze for the question "*What is the impact of each attack?*" was difficult because it is generally hard to put numbers on piracy. However, NGMore made it possible to collect statistics from their website over the period of 2015/10/20 to 2016/04/01 as can be seen in Section 4.3. In this time period the total members of NGMore grew from 117 to 573 and a total of 29582 binary toy dumps were downloaded over 532 available toys.
- Section 5.1 describes a strong countermeasure against forgeries involving signing tags to answer "*How can each attack be countered?*". Moreover, numerous other countermeasures are explored to protect against clones, emulation and other small attacks. However, all the presented countermeasures have drawbacks and are difficult or expensive to implement. Especially because the forward and backwards compatibility could not be broken. The addition of signatures to tags struck a perfect balance between added security and difficulties for the production process and development while adding a significant barrier for attackers. Furthermore, it made it infeasible for attackers to find unreleased toys before they are official released and this was deemed the most important aspect to defend against.

The answer to the research question "*To what extent can attacks against the hardware of toys-to-life games be prevented?*" is that it is difficult for companies in the toys-to-life genre to properly protect against clones, forgeries and emulation without changing the design of the hardware. If that would be possible, PUF together with strong authentication of the hardware and an online Digital Rights Management (DRM) system could make it infeasible to produce plug and play commercial solution for playable toy piracy. Obviously, there is no perfect way to protect against a capable attacker, but protecting against a buyable plug and play piracy solution seems feasible.

Implementing strong security is difficult and tedious work and does not make the game more fun to play. From that perspective it is difficult to justify the spend time and costs on security. This becomes apparent from the existence of known attacks in this genre and the limited number of countermeasures implemented. A shift is required in Information Technology (IT) to understand this dilemma and accept that secure software is important, especially given the fact that more and more devices are always connected to the internet. History shows that this is a slow and lengthy process and this thesis has shown common risks and ways to protect against these risks. This could be used by developers to bridge this gap. Moreover, this thesis has shown different ways of adding layers of security to RFID systems even after being released in the field for a multitude of years that protects the system against some attacks.

## 6.1 Recommendation for future work

The scope of this paper limits the depth of some topics and raises possible topics for future research. This section will present some of the main topics that could benefit from more research.

- The similarities between the PowerSaves for characters and MaxLander as described in Section 4.3.3 raises some question on the involvement of the commercial company Datel. How these are connected is unclear but the identical PCBs and similarities in software and firmware raises questions.
- The single largest problem in current RFID systems are 1-on-1 clones and it is clear that mechanisms using PUF could protect against this. However, it is not clear how to properly implement this cheaply and without internet access. Future research into possibilities, costs and usability is required to make it ready for implementation in toys-to-life games. Moreover, it is highly likely that proper PUF scheme could replace the signature scheme presented in Section 5.1.
- Using SPA with or without fuzzing to detect changes in hardware is as far as publicly known a novel way of automated testing of devices where there is no direct ways for another device to register feedback from the device under testing. Using the concepts shown in this thesis it could be shaped into a more generic and better system to increase the stability and security of hardware devices.

Most of the presented countermeasures are not foolproof but can indicate that something is amiss. Given enough evidence, it could point to piracy or abuse of the game. It could be beneficial to have a system in place that behaves like a security warden that could detect these things and even take penalizing actions just like a virus scanner does. The system should contains rules that should not happen in every day use like the detection of genuine readers scrip presented in Section 5.2.1.2. Moreover, if the system is connected to a master server over the internet, new rules could be added after the system gets deployed depending on new attacks and problems. Furthermore, if specific toys are cloned, rules could even be defined to block specific toys from working altogether using the unique identification on the tags. Such a system is certainly not trivial to implement, but once done well could be reused in many different games using different rules per game making it a worthwhile research project.

# References


[1] SUPERCOP. Website http://bench.cr.yp.to/supercop.html.

[2] Teensy 3.1, 2014. Website https://www.pjrc.com/teensy/teensy31.html.

[3] D. J. B. anl Peter Birkner, M. Joye, T. Lange, and C. Peters. Twisted edwards curves. In *Progress in Cryptology – AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer Berlin Heidelberg, 2008.

[4] Atmel. ATmega128 Complete, June 2011. Datasheet.

[5] S. Babbage, D. Catalano, C. Cid, B. de Weger, O. Dunkelman, C. Gehrmann, L. Granboulan, T. Güneysu, J. Hermans, T. Lange, A. Lenstra, C. Mitchell, M. Näslund, P. Nguyen, C. Paar, K. Paterson, J. Pelzl, T. Pornin, B. Preneel, C. Rechberger, V. Rijmen, M. Robshaw, A. Rupp, M. Schläffer, S. Vaudenay, F. Vercauteren, and M. Ward. ECRYPT II Yearly Report on Algorithms and Key Lengths. September 2012.

[6] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-Speed High-Security Signatures. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer Berlin Heidelberg, 2011.

[7] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers. TweetNaCl: A Crypto Library in 100 Tweets. In *Progress in Cryptology - LATINCRYPT 2014*, volume 8895 of *Lecture Notes in Computer Science*, pages 64–83. Springer International Publishing, 2015.

[8] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational), May 2006.

[9] D. Chu, J. Großschädl, Z. Liu, V. Müller, and Y. Zhang. Twisted Edwards-form Elliptic Curve Cryptography for 8-bit AVR-based Sensor Nodes. In *Proceedings of the first ACM workshop on Asia public-key cryptography*, AsiaPKC '13, pages 39–44. ACM, 2013.

[10] N. T. Courtois. The Dark Side of Security by Obscurity and Cloning MiFare Classic Rail and Building Passes Anywhere, Anytime. May 2009. http://eprint.iacr.org/2009/137/.

[11] F. D. Garcia, P. van Rossum, R. Verdult, and R. W. Schreur. Wirelessly pickpocketing a MIFARE Classic card. In *30th IEEE Symposium on Security and Privacy (S&P 2009)*, pages 3–15. IEEE Computer Society, 2009.

[12] B. Gassend, D. Clarke, M. V. Dijk, and S. Devadas. Controlled Physical Random Functions. In *Computer Security Applications Conference*, pages 149–160. IEEE, 2002.

[13] T. Goodspeed. Facedancer21. http://goodfet.sourceforge.net/hardware/facedancer21/.

[14] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Berlin Heidelberg, 2004.

[15] H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted edwards curves revisited. In *Advances in Cryptology - ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer Berlin Heidelberg, 2008.

[16] D. E. Holcomb and K. Fu. Bitline PUF: Building Native Challenge-Response PUF Capability into Any SRAM. Cryptology ePrint Archive, Report 2014/749, 2014. http://eprint.iacr.org/.

[17] M. Hutter and P. Schwabe. NaCl on 8-bit AVR microcontrollers. In A. Youssef and A. Nitaj, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer-Verlag Berlin Heidelberg, 2013. Document ID: cd4aad485407c33ece17e509622eb554, http://cryptojedi.org/papers/#avrnacl.

[18] International Organization for Standardization. ISO/IEC 14443-4:2008 - Identification cards - Contactless integrated circuit cards - Proximity cards - Part 4: Transmission protocol. *International Standard*, July 2008. Specifications.

[19] International Organization for Standardization. ISO/IEC 14443-2:2010 - Identification cards - Contactless integrated circuit cards - Proximity cards - Part 2: Radio frequency power and signal interface. *International Standard*, September 2010. Specifications.

[20] International Organization for Standardization. ISO/IEC 14443-3:2011 - Identification cards - Contactless integrated circuit cards - Proximity cards - Part 3: Initialization and anticollision. *International Standard*, April 2011. Specifications.

[21] International Organization for Standardization. ISO/IEC 14443-1:2016 - Identification cards - Contactless integrated circuit cards - Proximity cards - Part 1: Physical characteristics. *International Standard*, March 2016. Specifications.

[22] B. Kordy, P. Kordy, S. Mauw, and P. Schweitzer. ADTool: Security Analysis with Attack-Defense Trees (Extended Version). *arXiv preprint arXiv:1305.6829*, June 2013.

[23] A. Laurie. Practical attacks against RFID. *Network Security*, 2007(9):4–7, 2007.

[24] A. Lenstra, E. Tromer, A. Shamir, W. Kortsmit, B. Dodson, J. Hughes, and P. Leyland. Factoring Estimates for a 1024-Bit RSA Modulus. In C.-S. Laih, editor, *Advances in Cryptology - ASIACRYPT 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 55–74. Springer Berlin Heidelberg, 2003.

[25] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. In *Public Key Cryptography*, volume 1751 of *Lecture Notes in Computer Science*, pages 446–465. Springer Berlin Heidelberg, 2000.

[26] D. Lim, J. W. Lee, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas. Extracting Secret Keys From Integrated Circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(10):1200–1205, Oct 2005.

[27] S. Mauw and M. Oostdijk. Foundations of attack trees. In *Information Security and Cryptology - ICISC 2005*, volume 3935 of *Lecture Notes in Computer Science*, pages 186–198. Springer Berlin Heidelberg, 2006.

[28] M. Meriacs. Heart of Darkness - exploring the uncharted backwaters of HID iCLASS™ security. December 2010. http://www.openpcd.org/HID_iClass_demystified.

[29] NXP. PUF - Physical Unclonable Functions. http://www.nxp.com/documents/other/75017366.pdf.

[30] NXP Semiconductors. UM0701-02 PN532 User Manual, 2007. Datasheet.

[31] NXP Semiconductors. MF1S50yyX MIFARE Classic 1K - Mainstream contactless smart card IC for fast and easy solution development, May 2011. Datasheet.

[32] NXP Semiconductors. MF1S70yyX MIFARE Classic 4K - Mainstream contactless smart card IC for fast and easy solution development, May 2011. Datasheet.

[33] S. Peter, Bushing, Marcan, and Segher. Console Hacking 2010 - PS3 Epic Fail, 2010. https://events.ccc.de/congress/2010/Fahrplan/events/4087.en.html.

[34] P. S. Ravikanth. *Physical One-Way Functions*. PhD thesis, Massachusetts Institute of Technology, March 2001.

[35] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[36] C. M. Roberts. Radio frequency identification (RFID). *Computers & Security*, 25(1):18–26, 2006.

[37] V. Saini, Q. Duan, and V. Paruchuri. Threat modeling using attack trees. *Journal of Computing Sciences in Colleges*, 23(4):124–131, April 2008.

[38] A. Salomaa. *Public-Key Cryptography*. Springer Science & Business Media, 2013.

[39] B. Schneier. Attack trees, 1999.

[40] A. Shamir and E. Tromer. On the cost of factoring RSA-1024. *RSA CryptoBytes*, 6(2):10–19, 2003.

[41] S. Sorrell. Smart toys do toys dreams of digital lives. Technical report, Juniper Research, November 2015. Whitepaper http://www.juniperresearch.com/press/press-releases/smart-toy-revenues-to-hit-$2-8bn-this-year.

[42] C. Tarnovsky. Infineon SLE 66CL PE, Feb 2010. http://www.darkreading.com/risk/researcher-cracks-security-of-widely-used-computer-chip/d/d-id/1132874.

[43] R. van Dijk and L. Sangers. Portable RFID Bumping Device. February 2016. Research paper for university course http://rp.delaat.net/2015-2016/p04/report.pdf.

[44] R. Verdult. Security analysis of RFID tags. Master's thesis, Radboud University Nijmegen, July 2008.

[45] R. Verdult. *The (in)security of proprietary cryptography*. Roel Verdult, 2015.

[46] R. Verdult, G. G. de Koning, and F. D. Garcia. A toolbox for RFID protocol analysis. In *4th International EURASIP Workshop on RFID Technology (EURASIP RFID 2012)*, pages 27–34. IEEE Computer Society, 2012.

[47] R. Verdult, G. G. de Koning, and F. D. Garcia. Tutorial: Proxmark, the Swiss Army Knife for RFID Security Research. Technical report, Radboud University Nijmegen, 2012.

[48] WCH. USB Bus Interface Chip CH375, November 2008. Datasheet version 3E.

[49] E. Wenger and P. Wolfger. Harder, Better, Faster, Stronger - Elliptic Curve Discrete Logarithm Computations on FPGAs. Cryptology ePrint Archive, Report 2015/143, 2015. http://eprint.iacr.org/.

[50] J. Westhues. Proxmark III Website, 2007. http://www.proxmark.org/.

# Glossary

Abbreviations, most of which are explained in more detail in the document.

**ACL** Access Control List. 7, 8, 27, 35
**AES** Advanced Encryption Standard. 48
**API** Application Program Interface. 42, 44
**CPRNG** Cryptographic Pseudo-Random Number Generator. 26, 27
**CRP** Challenge-Response Pair. 28
**DRM** Digital Rights Management. 50
**ECB** Electronic Codebook. 48
**ECC** Elliptic Curve Cryptography. 6, 26, 31
**ECDSA** Elliptic Curve Digital Signature Algorithm. 26
**EdDSA** Edwards-curve Digital Signature Algorithm. 26, 27
**FIB** Focused Ion Beam. 40
**GUI** Graphical User Interface. 44
**HID** Human Interface Device. 21
**HMAC** Hash-based Message Authentication Code. 40
**HSM** Hardware Security Module. 34
**I$^2$C** Inter-Integrated Circuit. 20, 28, 29
**IC** Integrated Circuit. 7, 17, 19–23, 28, 29, 31, 40, 57
**ICSP** In Circuit Serial Programmer. 18
**IT** Information Technology. 50
**LAN** Local Area Network. 34
**LED** Light-Emitting Diode. 42, 43
**MCU** Microcontroller Unit. 17–21, 28, 31, 32
**MS/s** Mega-Samples per second. 29, 43
**NFC** Near Field Communication. 5, 7, 10, 17, 20, 29, 40, 41, 56
**NUID** Non Unique Identifier. 14–16, 19
**OpenOCD** Open On-Chip Debugger. 21
**PCB** Printed Circuit Board. 17, 21, 22, 51, 57, 62
**PRNG** Pseudo-Random Number Generator. 8
**PUF** Physical Unclonable Function. 1, 28, 36, 50, 51
**PWM** Pulse Width Modulation. 43
**QA** Quality Assurance. 35, 49
**RAM** Static Random-Access Memory. 18, 43
**RF** Radio Frequency. 8
**RFID** Radio-Frequency Identification. 4, 8, 14, 25, 42, 44, 48, 50, 51
**SAM** Secure Access Module. 20
**SPA** Simple Power Analysis. 42, 46, 51
**SPI** Serial Peripheral Interface. 20, 32
**SRAM** Static Random-Access Memory. 28, 32
**UART** Universal Asynchronous Receiver/Transmitter. 32
**UI** User Interface. 48, 49
**USB** Universal Serial Bus. 10, 12, 17, 20, 21, 28, 32, 34, 37, 39, 40, 42, 44, 47, 50, 56

# Appendices

## A MaxLander USB Protocol

| API Name | Command | Response | Comment |
|---|---|---|---|
| Query | `02` | `50 6F 72 74 61 6C` | Returns static "Portal" |
| Reset | `08` | | Reset the STM32F0 |
| Field On | `10` | | |
| Field Off | `11` | | |
| Activate Tag | `12` | `00 <Sak> <Len> <?b Uid>` | |
| MFC Auth | `13 <Index>` | | Authenticates to the correct sector (derived from block index) |
| MFC Read | `14 <Index>` | `<16b Data>` | Uses the same code as CH Read after a few stubs |
| MFC Write | `15 <Index> <16b Data>` | | Uses the same code as CH Write after one stub |
| CH Unlock | `16` | | Uses Chinese backdoor to unlock Mifare Classic |
| CH Read | `17 <Index>` | `<16b Data>` | Uses the same code as MFC Read after a few stubs |
| CH Write | `18 <Index> <16b Data>` | | Uses the same code as MFC Write after one stub |
| Get Status | `19` | `<2b StatusCode>` | Zero is all good |
| Set LED | `20 <Brightness> ?? ??` | | Sets the blue LED component, the other two bytes are likely residue of an RGB LED |
| Make Key | `30 <Index>` | `<6b KeyA>` | Retrieve NFC authentication key A for a sector |
| DFU | `99 44 46 55 [20]` | | Puts the STM32F0 chip into DFU mode. After this command, send the new firmware image with size 0x8000. The last 0x20 is being send by the library, but not checked by the firmware. |

Table 15: MaxLander USB protocol. *All numbers are given in hexadecimal representation.*

# B MaxLander and PowerSaves for characters PCB


Figure 31: Front of the MaxLander PCB, “SAM12” is printed on the U2 IC

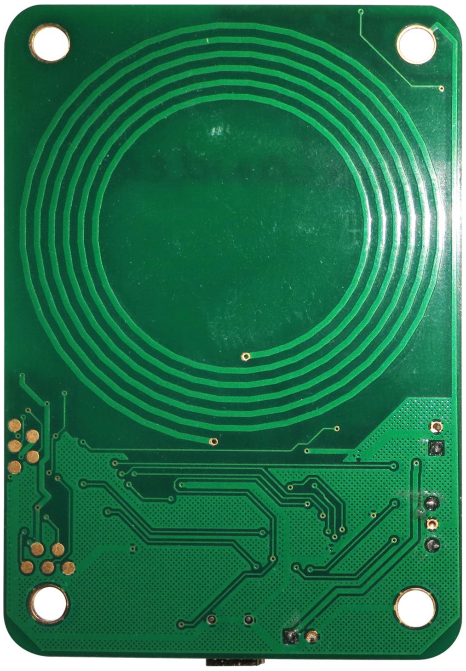

Figure 32: Back of the MaxLander PCB

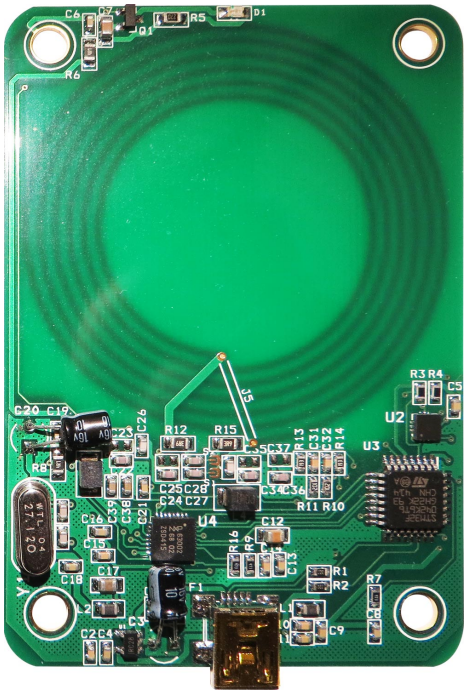

Figure 33: Front of the PowerSaves PCB, there is nothing printed on the U2 IC

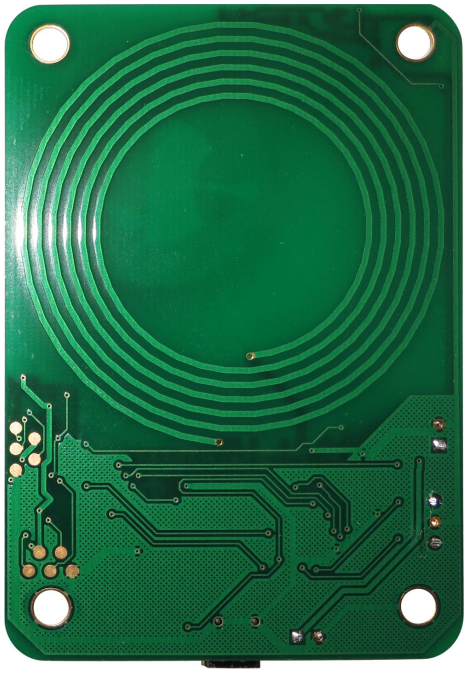

Figure 34: Back of the PowerSaves PCB

# C Detect genuine reader script

```
from lib.util import *
from lib.general import *

def detectReadTimeout(tagIndex, blockIndex):
  portal.send(packetRead(tagIndex, blockIndex))
  delay(17)
  portal.send(packet('A', [1]))

  hasQ = hasA = None
  isTimeout = False
  for i in range(0, 75):
    block = portal.recv()
    if block is not None:
      # Very small chance that this get hit, but for completeness
      if hasA == False and block[0] == ord('Q') and block[2] == blockIndex:
        isTimeout = False
        break
      elif hasA == True and block[0] == ord('Q') and block[2] == blockIndex:
        isTimeout = True
        break
      elif block[0] == ord('Q') and block[2] == blockIndex:
        hasQ = True
      elif block[0] == ord('A'):
        hasA = True
    delay(5)
  return isTimeout

tagNo = getFirstTagIndex()
if tagNo is None:
  print('No tag present to check against!')
else:
  print('Checking portal using tag {0:d}: '.format(tagNo), end='')

  # Detect flipping of the command response when the block is invalid
  # InRange      = Send valid block request (no flip): S(Q), S(A), R(Q), R(A)
  # OutsideRange = Send invalid block request (yes flip): S(Q), S(A), R(A), R(Q)
  isTimeoutInRange = detectReadTimeout(tagNo, 0)
  isTimeoutOutsideRange = detectReadTimeout(tagNo, 0x80)

  # Print result
  print('genuine' if not isTimeoutInRange and isTimeoutOutsideRange else
    'emulated')
```

# D Key derivation function

## D.1 Get Key

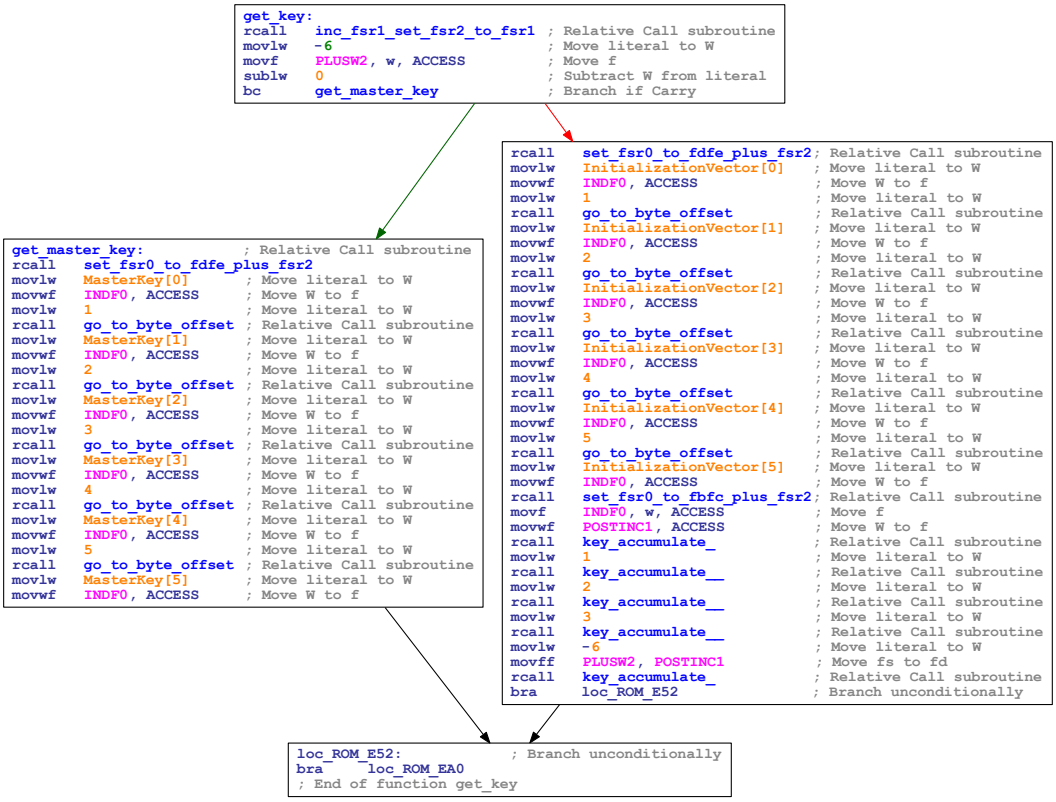


Figure 35: Get key function after naming and commenting (some simplification)

## D.2 Accumulate

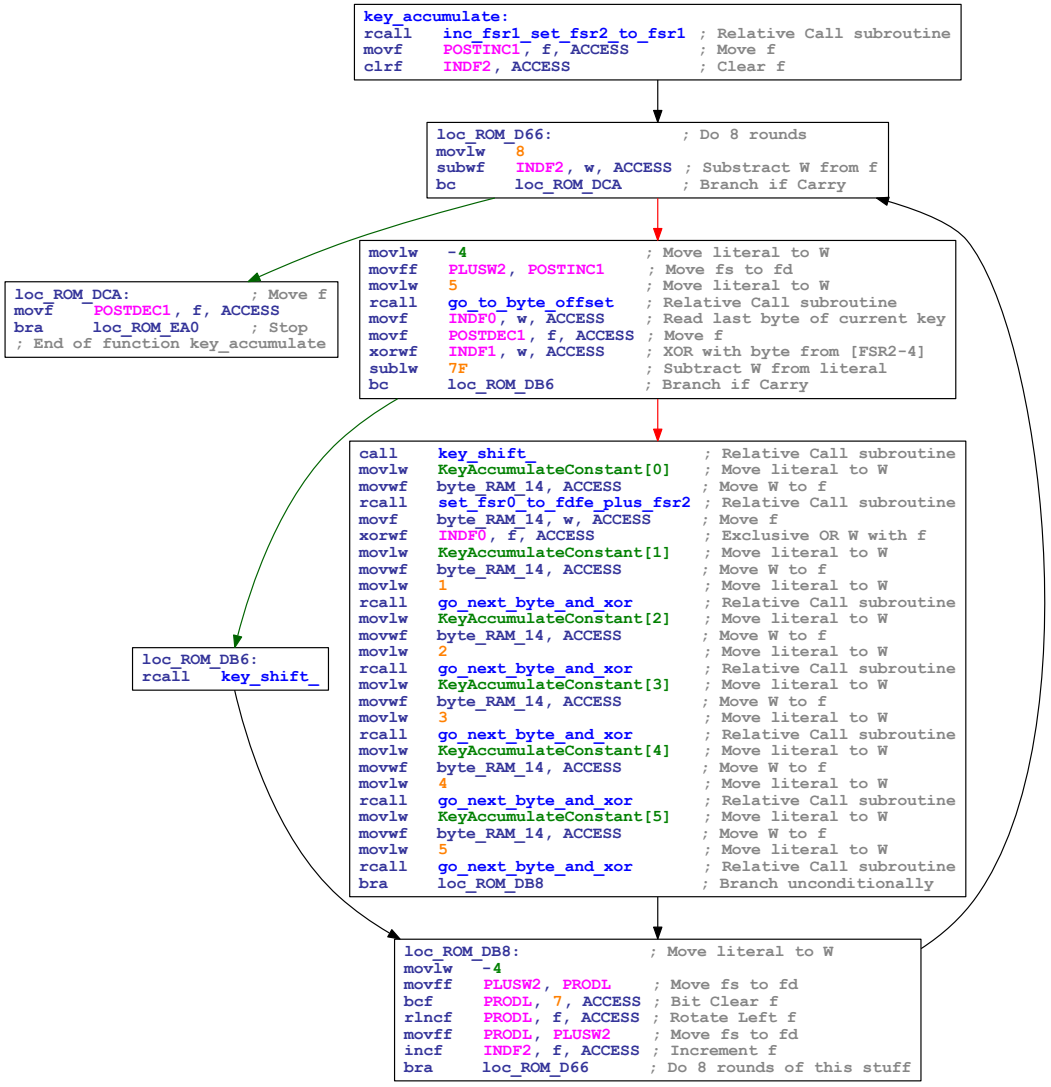


Figure 36: The key accumulate function after naming and commenting (some simplification)

## D.3 Shift

```
key_shift:
; FUNCTION CHUNK AT ROM:0EA0 SIZE 00000008 BYTES
rcall    inc_fsr1_set_fsr2_to_fsr1              ; Relative Call subroutine
movlw    4                                      ; Move literal to W
rcall    next_and_rol_and_and1                  ; Relative Call subroutine
movlw    5                                      ; Move literal to W
rcall    next_and_addw_and_or_ram14             ; Relative Call subroutine
movlw    5                                      ; Move literal to W
rcall    write_FSR1_byte_to_FSR0                ; Relative Call subroutine
movlw    3                                      ; Move literal to W
rcall    next_and_rol_and_and1                  ; Relative Call subroutine
movlw    4                                      ; Move literal to W
rcall    next_and_addw_and_or_ram14             ; Relative Call subroutine
movlw    4                                      ; Move literal to W
rcall    write_FSR1_byte_to_FSR0                ; Relative Call subroutine
movlw    2                                      ; Move literal to W
rcall    next_and_rol_and_and1                  ; Relative Call subroutine
movlw    3                                      ; Move literal to W
rcall    next_and_addw_and_or_ram14             ; Relative Call subroutine
movlw    3                                      ; Move literal to W
rcall    write_FSR1_byte_to_FSR0                ; Relative Call subroutine
movlw    1                                      ; Move literal to W
rcall    next_and_rol_and_and1                  ; Relative Call subroutine
movlw    2                                      ; Move literal to W
rcall    next_and_addw_and_or_ram14             ; Relative Call subroutine
movlw    2                                      ; Move literal to W
rcall    go_to_byte_offset                      ; Relative Call subroutine
rcall    write_FSR1_byte_to_FSR0_read_also_in_w ; Relative Call subroutine
rcall    rol_and_and1                           ; Relative Call subroutine
movlw    1                                      ; Move literal to W
rcall    next_and_addw_and_or_ram14             ; Relative Call subroutine
movlw    1                                      ; Move literal to W
rcall    go_to_byte_offset                      ; Relative Call subroutine
rcall    write_FSR1_byte_to_FSR0_read_also_in_w ; Relative Call subroutine
addwf    WREG, w, ACCESS                        ; Add W and f
movwf    POSTINC1, ACCESS                       ; Move W to f
rcall    set_fsr0_to_fdfe_plus_fsr2             ; Relative Call subroutine
movf     POSTDEC1, f, ACCESS                    ; Move f
movf     INDF1, w, ACCESS                       ; Move f
movwf    INDF0, ACCESS                          ; Move W to f
bra      loc_ROM_EA0                            ; Branch unconditionally
; End of function key_shift
```

```
; START OF FUNCTION CHUNK FOR key_shift
loc_ROM_EA0:                ; Move f
movf     POSTDEC1, f, ACCESS
movff    INDF1, FSR2L       ; Move fs to fd
return   0                  ; Return from Subroutine
; END OF FUNCTION CHUNK FOR key_shift
```

Figure 37: The key shift function after naming and commenting (some simplification)

# E The reader printed circuit board

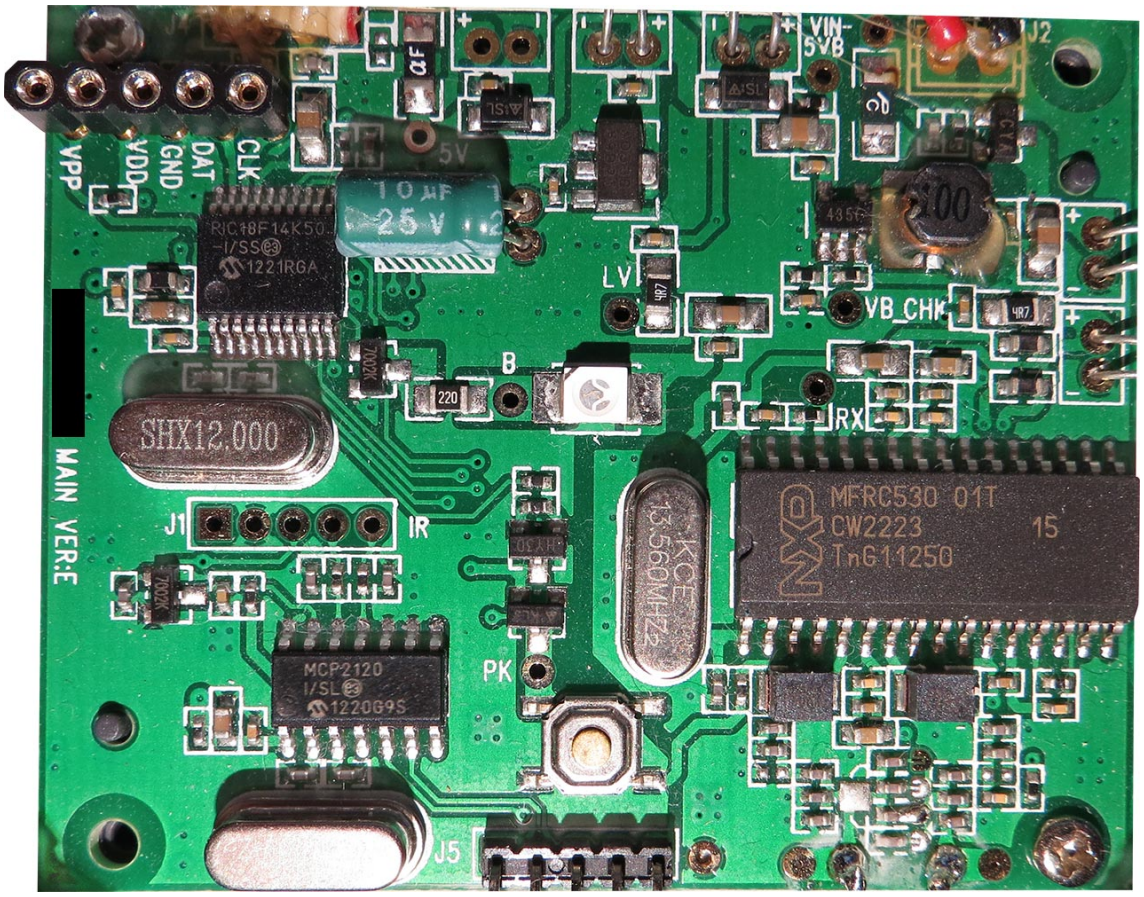


Figure 38: reader PCB, the top left header for CLK, DAT, GND, VDD and VPP was added later