Hiding Cryptographic Keys of Embedded Systems

RAFAEL COSTA[1, 2], DAVIDSON BOCCARDO[2], LUCI PIRMEZ[1,2], LUIZ FERNANDO RUST[2]
Federal University of Rio de Janeiro[1], National Institute of Metrology, Quality and Technology[2]
BRAZIL
rafaelcosta@ppgi.ufrj.br, {drboccardo, lfrust}@inmetro.gov.br, luci@nce.ufrj.br

Abstract: - One of the worries still present in the development of embedded systems is about the confidentiality of its sensitive data. Since those systems could be arranged in unprotected areas, an attacker, with physical access to their devices, may disclosure its sensitive data, e.g. cryptographic keys, by reverse engineering means. Although there are solutions to protect such cryptographic keys, they are usually costly. Therefore, this paper proposes different methods to protect cryptographic keys of embedded systems based on software protection techniques taking into account the attacker's reverse engineering perspective. We conducted a case study that shows the difficulty to disclosure cryptographic keys hidden by the proposed methods.

Key-Words: - Security, Embedded Systems, Data Hiding

1 Introduction

The cost reduction of hardware components and the evolution of information systems have allowed the widespread use of embedded systems. Those systems are being used to bring safety, reliability and efficiency to distinct kinds of applications, such as flight control systems, automotive monitoring systems and so on [1]. However, despite the several benefits provided by the emergence of embedded systems, there are many challenges to be faced, particularly, in terms of the security of its inner sensitive data.

Since the embedded system could be arranged in unprotected areas, i.e. without being under surveillance, those devices are susceptible to Man-Et-The-End (MATE) attacks [2]. Those attacks happen when attackers, after capturing one of those devices, try to compromise it through reverse engineering or by tampering the hardware itself or the embedded software. For instance, an attacker may perform reverse engineering on the embedded software in order to discover its sensitive data, such as its cryptographic keys [3].

One example of MATE attack to disclosure cryptographic keys can be found on [4], which finds the cryptographic key performing an entropy analysis. In general, cryptographic algorithms are used to provide confidentiality, integrity, authentication and non-repudiation. Although not all cryptographic algorithms are used to achieve all the mentioned goals, the critical element of all of them is its cryptographic keys. For example, such algorithms could be used to make data unclear for people without permissions in order to keeping its confidentiality. However, if the used cryptographic keys are disclosure, the encoded data could be revealed. So, the effectiveness of cryptographic algorithms depends on the secrecy of the used cryptographic key.

One possible solution to protect cryptographic keys is using a trusted platform module (TPM) [5]. Such solution consists of a microcontroller capable to securely store a small amount of data. Although it can be considered a good solution, the cost of TPM may preclude its use, mainly when the financial resources of projects are limited. For example, in the embedded system scenario, it may be desirable to create cheaper solutions instead of use a TPM to protect cryptographic keys since the cost to build each device must to be low. It is also important to note the addition of a TPM will lead to redesign of the hardware architecture of the embedded system.

A way to protect cryptographic keys without using a TPM is to use data obfuscation. Data obfuscation is a class of code transformations that converts an initial data representation into other representation that reveals less information about it [6]. Data obfuscation could be considered a cheap pathway to enhance cryptographic key protection because it tries to hinder reverse engineering without increasing the financial cost of embedded system projects. This is because data obfuscation consists of simple code transformations that require more time and effort to the attacker understand the obfuscated data.

In general, data obfuscation could be broadly classified as static and dynamic [6]. When it is

static, the data representation remains intact during runtime, independently of the user input or the environment where the software is running. On the other hand, when it is dynamic, the software itself changes its own data representation during runtime. In this case, i.e. when the data representation is changed dynamically, it is required more time and effort for the attacker understands such dynamic obfuscated data compared to data obfuscated statically. This happens because when an attacker looks at statically obfuscated data, he sees a difficult representation to analyze, but every time the software runs, he sees the same data representation, which is not true when it is used dynamic data obfuscation due the data representation will be different each time the software runs.

While defense methods are implemented to make difficulty to the attacker perform reverse engineering, new attack strategies to break them are created. Thus, it is necessary to consider what the attacker can do with the available code analysis tools during the development of defense methods. Assuming that, this paper proposes methods based on software protection techniques to protect cryptographic keys taking into account the attacker's perspective. Such methods are presented in an incremental way considering different attacker's strategies. We evaluate our methods through a case study, which shows the difficulty to disclosure cryptographic keys in terms of the time to do it.

The remainder of this paper is organized as follows: section 2 presents the related works; the section 3 presents in details the methods to protect cryptographic keys considering different attacker's strategies; the section 4 presents our case study and, finally, the section 5 presents our final remarks and future work.

2 Related Works

Since hardware solutions are more expensive than software solutions, we describe mainly software-based papers. Despite not all papers present proposals to protect exclusively cryptographic keys, they could be used to that purpose.

McGregor and Lee [7] propose architectural enhancements for general-purpose processors that protect cryptographic keys. They describe modest hardware modifications combined with a trusted software library that allows protected cryptographic operation, i.e. devices perform encryption and decryption using a secret cryptographic key. For such, it is proposed to store such cryptographic key in the processor. However, different from such proposal, the presented paper proposed a cryptographic key protection method that does not require a special hardware.

Hu *et. al.* [8] describe a software encryption method to protect software intellectual property. Since the security of the encrypted software relies on the secrecy of the key, this paper proposes to protect it by a key protection scheme, which hides the cryptographic key into the encrypted code. Despite this method is similar to ours, we focusing only on protect cryptographic keys, without encrypting the whole software, which could demands unnecessary computation resources.

3 Hiding Cryptographic Keys

In this section, we present methods to protect cryptographic keys based on software protection techniques taking into account the strategies that the attacker could perform to disclosure cryptographic keys. These methods are described incrementally in order to make easier to the reader understand what the countermeasure could be applied against each attacker strategy.

Considering that an attacker is motivated to discover cryptographic keys inner the software of embedded systems, he could have different strategies to achieve his goal. However, for all strategies, we made two assumptions. The first assumption is that the attacker should be capable of getting the binary code from the disk or memory card of the captured device, which contains the software to be compromised. The second assumption, except for the entropy analysis, that the software should be at a higher-level representation rather than the binary code. For this, the attacker could use a disassembler tool to translate the binary code into the assembly code (notation to represent machine code) or go further and use a decompiler tool to create a representation on a high-level programming language, such as c, in order to make easier to analyze the software.

Despite the attacker could use at ease all available tools, including static and dynamic tools, according on the strategy that he will use, only the last method (Dynamic Camouglage Keys) tries to hinder strategies to disclosure cryptographic keys that count on dynamic code analysis tools.

3.1 Moving Keys to Code Segment

Since the attacker is dealing with an ordinary code, which generally is composed by two segments: (i) code segment, usually containing program instructions, and (ii) data segment, commonly used to store data; and no defense mechanism is used to protect its cryptographic keys, an attacker could

perform two strategies to find out cryptographic keys contained in the data segment: string analysis and pattern-matching analysis.

In the first strategy, **string analysis**, the attacker simply examines the data segment content, pursuing of variable or constant names related to cryptography. Although this strategy can be considered quite naïve, it is still useful when no method is used to protect cryptographic keys.

On the other hand, once it is used cryptographic algorithms from known cryptographic libraries, the attacker could use a **pattern-matching analysis** towards to disclosure its cryptographic keys. In such strategy, the attacker examines predetermined positions in the data segment, where cryptographic keys are possibly disposed to be, whereas the elements of known cryptographic libraries usually are stored in the same positions.

The method proposed in this work to prevent against string and pattern-matching analysis is moving the cryptographic key from the data segment to the code segment, in areas that never gets executed (***dead execution spots***) in order to maintain the software behavior. This can be done through code transformations that manipulate the software control flow. For instance, it is possible to create dead execution spots using code obfuscation techniques, such as call obfuscation, return obfuscation or false return obfuscation combined with control flow manipulation [9]. Thus, moving the cryptographic key to a dead execution spot may lead disassemblers to translate it as program instructions.

When cryptographic keys are translated as program instructions, the generated assembly code does not represent the real code and, thus, the analysis upon that code will not be reliable. In addition, if the assembly code is not reliable, errors will be propagated to the decompilation because the decompilation depends on the code generated by a disassembler. Therefore, the analysis upon the decompiled code will not be reliable too.

The algorithm 1 describes the steps of our proposal to prevent against string analysis and pattern-matching analysis. Such algorithm receives the following information as input: program $\mathcal{P}$, cryptographic key $\mathcal{K}$ and the size of the respective cryptographic key ℓ; and returns $\mathcal{P}^{II}$, the program whose cryptographic key was moved to a certain dead executions spot into the code segment. The first step of this algorithm is getting candidate instruction addresses ($\mathcal{C}_{Instrs}$) containing instructions that may be obfuscated by an obfuscation technique t from an obfuscation techniques set $\mathcal{T}$. In the following, one instruction address from $\mathcal{C}_{Instrs}$ (s) has to be randomly chosen in order to create a dead execution spot in there. For such, the *CreateDeadExecutionSpot* function applies code transformations dictated by t on s to create a dead execution spot capable to store the cryptographic key whose size is ℓ. Finally, $\mathcal{K}$ is moved to the dead execution spot in s through the function *MoveKey*.

Input: Program $\mathcal{P}$; cryptography key $\mathcal{K}$, key size ℓ
Output: Program $\mathcal{P}^{II}$

1. /* Getting candidate instructions ($\mathcal{C}_{Instrs}$) that could be obfuscated in order to create a dead execution spot */
2. $\mathcal{C}_{Instrs} \leftarrow \emptyset$
3. For each instruction $i \in \mathcal{P}$
4. If i may be obfuscated by technique $t \in \mathcal{T}$
5. $\mathcal{C}_{Instrs} \leftarrow \mathcal{C}_{Instrs} \cup i$
6. End If
7. End For
8. /* Choosing randomly the instruction address (s) where one dead execution spot will be created */
9. $s \leftarrow$ GetRandomElement($\mathcal{C}_{Instrs}$)
10. /* Creating a dead execution spot in s whose size is ℓ using a certain obfuscation technique t
11. $\mathcal{P}^{I} \leftarrow$ CreateDeadExecutionSpot ($\mathcal{P}, s, t, \ell$)
12. /* Moving $\mathcal{K}$ to the dead execution spot at s */
13. $\mathcal{P}^{II} \leftarrow$ MoveKey($\mathcal{P}^{I}, \mathcal{K}, s$)

Algorithm 1: Proposed algorithm to move cryptographic keys to dead execution spots into the code segment.

3.2 Splitting Keys

If an attacker was not able to find out a cryptographic key in the data segment, using the previous strategies (string and pattern-matching analysis), he must to concentrate his effort on searching it on the code segment. For such, he could use an entropy analysis.

In the entropy analysis, the attacker calculates the entropy of the code segment in order to obtain an indication of where the cryptographic key is located. In a simplistic way, the entropy could be understood as a measure of random or disorder [10]. Generally the entropy of an ordinary code segment is low. This is because program instructions are represented by a limited set of bytes that do not change drastically among different program instructions. However, when a cryptographic key is stored in the code segment, it disturbs the entropy of the code segment because cryptographic keys are represented by random bytes, which makes the entropy higher on the position where the cryptographic key is located [4]. Thus, the entropy could help the attacker to identify possible candidate positions in the code segment where the cryptographic keys could be.

ISBN: 978-1-61804-276-7

In order to prevent against entropy analysis, we propose to split the cryptographic key into a certain number of parts and store them in smaller dead execution spots instead of simply moving the entire cryptographic key to a single dead execution spot. Therefore, the entropy of the code segment could not indicate one possible candidate position in the code segment where the cryptographic key could be.

Input: Program $\mathcal{P}$; cryptography key $\mathcal{K}$, operand size n
Output: Program $\mathcal{P}^{II}$

1. $\mathcal{C}_{Instrs} \leftarrow$ GetCandidateInstructions($\mathcal{P}$)
2. /* Splitting $\mathcal{K}$ in n sized parts */
3. $\mathcal{K}_{Parts} \leftarrow$ SplitCryptographicKey($\mathcal{K}, n$)
4. /* Choosing the instruction addresses ($\mathcal{S}_{parts}$) from $\mathcal{C}_{Instrs}$ where dead execution spots should be created to store each part of $\mathcal{K}$ */
5. $\mathcal{S}_{parts} \leftarrow$ GetRandomElements($\mathcal{C}_{Instrs}, sizeof(\mathcal{K}_{Parts})$)
6. /* Creating dead execution spots capable to store cryptographic key parts of n size */
7. $\forall\, s_i \in \mathcal{S}_{Parts}$:
   $\mathcal{P}^{I} \leftarrow$ CreateDeadExecutionSpot ($\mathcal{P}, s_i, t, n$)
8. /* Moving each part of $\mathcal{K}$ to one of the created dead execution spots */
9. $\forall\, k_i \in \mathcal{K}_{Parts} \wedge \forall\, s_i \in \mathcal{S}_{Parts}$:
   $\mathcal{P}^{II} \leftarrow$ MoveKey($\mathcal{P}^{I}, k_i, s_i$)
10. /* Embed the reconstruct cryptographic key routine $\mathcal{R}$ to $\mathcal{P}^{II}$ and insert calls to $\mathcal{R}$ before program instructions that refers to $\mathcal{K}$ */
11. $\mathcal{P}^{II} \leftarrow \mathcal{P}^{I} \cup \mathcal{R}$
12. For each instruction $i \in \mathcal{P}^{II}$
13. If i refers $\mathcal{K}$
14. insert call to $\mathcal{R}$ in $i-1$
15. End If
16. End For

Algorithm 2: Proposed algorithm to split the cryptographic key in parts and store each part to a dead execution spot

Algorithm 2 describes the steps of the proposed method to prevent against the entropy analysis strategy. Such algorithm receives as input the program $\mathcal{P}$, the cryptographic key $\mathcal{K}$ and the *operand* size n; and returns $\mathcal{P}^{II}$, the program whose cryptographic key is partitioned and its parts are moved to various dead execution spots in the code segment. First, $\mathcal{C}_{Instrs}$ is got from $\mathcal{P}$ by the function *GetCandidateInstructions* as in the algorithm 1. After that, the *SplitCryptographicKey* function partitions $\mathcal{K}$ into *operand* size n parts. In the following, the function *GetRandomElements* chooses instruction addresses from $\mathcal{C}_{Instrs}$ to create the required number of dead execution spots. Next, each part of the cryptographic key (k_i) is moved to one of the created dead execution spots by the function *MoveKey*. Finally, in order to ensure that the correct cryptographic key will be used, it is necessary to insert a reconstruct cryptographic key routine ($\mathcal{R}$) to reconstruct all the parts of the cryptographic key that are distributed in the code segment before each program instruction that uses the respective cryptographic key.

3.3 Camouflaging Key as False Instructions

When the previously presented strategies, including entropy analysis, could not help the attacker to know possible candidate positions in the code segment where the cryptographic key could be, the attacker may investigate the code segment for garbage bytes (**garbage bytes investigation**).

As **garbage bytes** are bytes that do not represent any program instruction, then it is possible that such bytes could be part of a certain cryptographic key that was moved from the data segment to the code segment.

Once an attacker finds several garbage bytes in the code segment, he could combine all of them in order to construct the used cryptographic key. Besides combining all the garbage bytes demands time, an attacker with time and dedication always could reveal the desired cryptographic key and the only thing to do is slow down such attacker.

The proposed method in our work to counter garbage bytes investigation is camouflaging all cryptographic key parts into **false instructions**, which are instruction composed by a random *opcode* combined with one cryptographic key part that will look like the *operand* of such instruction.

The number of false instructions depends on the *operand* size and the cryptographic key size. After splitting the cryptographic key into *operand* size parts and combine they with random *opcodes*, it is necessary to create the required number of dead execution spots to store all generated false instructions. Notice that each false instruction may have different sizes because they have different *opcodes*. So, it is necessary to create dead execution spots having different sizes.

Algorithm 3 describes the steps of the proposed method to prevent against previous strategies and garbage byte investigation. Algorithm 3 receives as input the program $\mathcal{P}$, the cryptographic key $\mathcal{K}$ and the *operand* size n; and returns $\mathcal{P}^{IV}$, the program whose cryptographic key is camouflaged in false instructions that are distributed in the code segment, being difficulty to the attacker distinguish between them and actual instructions. The only difference between this algorithm and algorithm 2 are the steps to create false instructions and to create dead execution spots since such spots have to store false instructions instead of cryptographic key parts.

After splitting the cryptographic key $\mathcal{K}$ into n sized parts, for each cryptographic key part k, it is performed the following steps to create false instructions: (i) generation of a random byte that represents a valid *opcode* by the function *RandomOpcode* and (ii) attaching it with k, which will be translated as the *operand* of such instruction. Next, before moving each false instruction in $false_{instrs}$ to a dead execution spot, it is necessary to ensure that such spot is capable to store a false instruction (f_i), i.e. whose size is sizeof(f_i). Notice that all false instructions are moved randomly to the available dead execution spots. Finally it is embedded the reconstruction cryptographic key routine to the program and calls to it.

Input: Program $\mathcal{P}$; cryptography key $\mathcal{K}$, operand size n
Output: Program $\mathcal{P}^{III}$

1. $\mathcal{C}_{Instrs} \leftarrow$ GetCandidateInstructions($\mathcal{P}$)
2. $\mathcal{K}_{Parts} \leftarrow$ SplitCryptographicKey($\mathcal{K}$,n)
3. /* Creating false instructions */
4. $false_{instrs} \leftarrow \emptyset$
5. For each $k \in \mathcal{K}_{Parts}$
6. $\quad f \leftarrow$ RandomOpcode() $\circ\, k$
7. $\quad false_{instrs} \leftarrow false_{instrs} \cup f$
8. End For
9. /* Moving false instructions to dead execution spot */
10. $\mathcal{S}_{parts} \leftarrow$ GetRandomElements($\mathcal{C}_{Instrs}$, $sizeof(\mathcal{K}_{Parts})$)
11. $\forall f_i \in false_{instrs} \wedge \forall s_i \in \mathcal{S}_{Parts}$:
12. $\quad \mathcal{P}^{I} \leftarrow$ CreateDeadExecutionSpot($\mathcal{P}, s_i, t$,$sizeof(f_i)$)
$\quad \mathcal{P}^{II} \leftarrow$ MoveFalseInstruction($\mathcal{P}^{I}, f_i, s_i$)
13. $\mathcal{P}^{III} \leftarrow \mathcal{P}^{II} \cup \mathcal{R}$
14. $\mathcal{P}^{IV} \leftarrow$ InsertCallsToReconstructRoutine($\mathcal{P}^{III}, \mathcal{K}$)

Algorithm 3: Proposed algorithm to hide cryptographic key parts into false instructions

3.4 Inserting Garbage Instructions

Considering that an attacker knows that all cryptographic key parts are camouflaged in false instructions, his strategy to reveal such cryptographic key is examine the code segment in pursuit of dead execution spots containing false instructions (dead execution spots investigation). Thus, we propose to counter such strategy by increasing the time and effort to reveal the cryptographic key through adding **garbage instructions**, which are random program instructions also stored in dead execution spots. This increases the time and effort to reveal cryptographic key because the attacker must to combine more operands, i.e. *operands* of false instructions and the *operands* of garbage instructions, whereas it is difficulty to the attacker distinguish one kind of instruction from another.

Algorithm 4 describes the steps of the proposed method that camouflages cryptographic key parts into false instructions and randomly adds garbage instructions on the code segment. Such algorithm receives as input the program $\mathcal{P}$, the cryptographic key $\mathcal{K}$, the *operand* size n, number of garbage instructions m; and returns $\mathcal{P}^{VI}$, the program whose cryptographic key is hidden in false instruction and has garbage instructions distributed in the code segment, requiring to the attacker more time and effort to reveal such cryptographic key since it is difficulty to distinguishes from what instructions the operand part must to be extracted in order to reconstruct the right cryptographic key.

Input: Program $\mathcal{P}$; cryptography key $\mathcal{K}$, operand size n, number of garbage instructions m
Output: Program $\mathcal{P}^{VI}$

1. $\mathcal{C}_{Instrs} \leftarrow$ GetCandidateInstructions($\mathcal{P}$)
2. $\mathcal{K}_{Parts} \leftarrow$ SplitCryptographicKey($\mathcal{K}$,n)
3. $false_{instrs} \leftarrow$ CreateFalseInstructions($\mathcal{K}_{Parts}$)
4. /*Creating garbage instructions*/
5. $garbage_{instrs} \leftarrow \emptyset$
6. While sizeof($garbage_{instrs}$) $< m$
7. $\quad g \leftarrow$ RandomOpcode() $\circ$ RandomOperand()
8. $\quad garbage_{instrs} \leftarrow garbage_{instrs} \cup g$
9. End While
10. /*Creating dead execution spots to hold false and garbage instructions and moving such instructions to there */
11. $\mathcal{M} \leftarrow SizeOf(\mathcal{K}_{Parts}) + m$
12. $\mathcal{S}_{parts} \leftarrow$ GetRandomElements($\mathcal{C}_{Instrs}$, $\mathcal{M}$)
13. $\forall f_i \in false_{instrs} \wedge \forall s_i \in \mathcal{S}_{Parts}$:
$\quad \mathcal{P}^{I} \leftarrow$ CreateDeadExecutionSpot($\mathcal{P}, s_i, t$,sizeof(f_i))
$\quad \mathcal{P}^{II} \leftarrow$ Move($\mathcal{P}^{I}, f_i, s_i$)
14. $\forall g_j \in garbage_{instrs} \wedge \forall s_j \in \mathcal{S}_{Parts}$:
$\quad \mathcal{P}^{III} \leftarrow$ CreateDeadExecutionSpot($\mathcal{P}^{II}, s_j, t, g_j$)
$\quad \mathcal{P}^{IV} \leftarrow$ Move($\mathcal{P}^{III}, g_j, s_j$)
15. $\mathcal{P}^{V} \leftarrow \mathcal{P}^{IV} \cup \mathcal{R}$
16. $\mathcal{P}^{VI} \leftarrow$ InsertCallsToReconstructRoutine($\mathcal{P}^{V}, \mathcal{K}$)

Algorithm 4: Proposed algorithm to hide cryptographic key parts into false instructions and add garbage instructions

The differences between this algorithm and algorithm 3 are: (i) the steps to create garbage instructions that do not exist in the previous algorithm and (ii) the number of dead execution spots that should be created. After getting the candidate instructions ($\mathcal{C}_{Instrs}$) and creating the false instructions by the function *SplitCryptographicKey*, this algorithm creates garbage instructions until it is created m garbage instructions. Each garbage instruction is created by combining random bytes, which represents a valid *opcode*, with random bytes that represents a valid *operand*. Next it is created the required number of

dead execution spots, which is the number of parts of the cryptographic key ($SizeOf(\mathcal{K}_{Parts})$) more the number of garbage instructions m. Then both false and garbage instructions are randomly moved to the created dead execution spots and finally, the reconstruction cryptographic key routine is attached to the program and calls to it are inserted.

3.5 Dynamically Camouflage Keys

The previously proposed methods try to hinder reverse engineering considering that the attacker only uses static analysis tools. However, such methods are not effective when he uses dynamic analysis tools, such as debuggers and emulators.

The fist strategy that takes advantage of static software is the **program diffing**. This strategy depends on the attacker has two or more copies of the same software and the cryptographic key in each copy be different. In this case, the attacker could reveal cryptographic keys by comparing the copies whereas each copy is equivalent except on the positions where the cryptographic is stored. Thus, examining the positions that are not equivalent, the attacker could easily reveal cryptographic keys.

The other strategy that takes advantage of static software is the **recurrent attacks**. The goal of this strategy is compromise the largest possible number of devices. Assuming a scenario where there are many devices containing identical software within it, once an attacker could find out the position where the cryptographic key is located, he may create a script to remotely read such position on other devices, where the cryptographic key is expected to be, in order to find out new cryptographic keys without analyze the software of all devices.

The countermeasure proposed in this work to prevent against such strategies is attaching to the embedded software an obfuscation engine that dynamically camouflages all parts of the cryptographic key in different false instructions and create random garbage instructions. Notice that the obfuscation engine could move both instructions to different program address in order to make it more difficult to identify such instructions.

The obfuscation engine could prevent against program diffing because the copy contained in each device is different since the software changes periodically differently and not only in the positions where the cryptographic key is. So, there are many positions in the code that are different, making difficult to know cryptographic key position by simply comparing two or more software. Furthermore, the obfuscation engine hinders recurrent attacks because it provides software diversity, i.e. the software within each device is different. Thus, the attacker could not take advantage of previous knowledge (cryptographic key location) to discover new cryptographic keys to propagate an attack for other devices.

The algorithm 5 describes the steps of the proposed method to counter program diffing and recurrent attacks. Such algorithm receives the following information as input: program $\mathcal{P}$, cryptographic key $\mathcal{K}$, the operand size n, the number of garbage instructions m and returns $\mathcal{P}^{VI}$ containing an obfuscation engine capable to embed an obfuscation engine that periodically hides cryptographic key in false instructions and inserts garbage instructions in different ways, by changing its shape and its location. In order to ensure that each device has distinct software, it is necessary to use schemes that gets intrinsic features of the device where the software is embedded, such as Physical unclonable Function (PUF) [11].

Input: Program $\mathcal{P}$; cryptography key $\mathcal{K}$; operand size n, number of garbage instructions m
Output: Program $\mathcal{P}^{VI}$

1. $\mathcal{C}_{Instrs} \leftarrow$ GetCandidateInstructions($\mathcal{P}$)
2. $\mathcal{K}_{Parts} \leftarrow$ SplitCryptographicKey($\mathcal{K}$,n)
3. $false_{instrs} \leftarrow$ CreateFalseInstructions($\mathcal{K}_{Parts}$)
4. $garbage_{instrs} \leftarrow$ CreateGarbageInstructions(m)
5. $\mathcal{M} \leftarrow SizeOf(\mathcal{K}_{Parts}) + m$
6. $S_{parts} \leftarrow$ GetRandomElements($\mathcal{C}_{Instrs}, \mathcal{M}$)
7. $\mathcal{P}' \leftarrow$ CreateDeadExecutionSpots($\mathcal{P}, S_{parts}$)
8. $\mathcal{P}'' \leftarrow Move(\mathcal{P}', false_{instrs} \cup garbage_{instrs}, \mathcal{S}_{Parts})$
9. $\mathcal{P}''' \leftarrow \mathcal{P}'' \cup \mathcal{R}$
10. $\mathcal{P}'^{v} \leftarrow$ InsertCallsToReconstructRoutine($\mathcal{P}'''$, $\mathcal{K}$)
11. /* Filling Obfuscation Engine $\mathcal{O}$ with information that allows it to run */
12. $\mathcal{O} \leftarrow \mathcal{O} \cup \mathcal{C}_{Instrs} \cup S_{parts} \cup false_{instrs} \cup garbage_{instrs}$
13. /* Embed $\mathcal{O}$ to $\mathcal{P}^{IV}$ and insert calls to $\mathcal{O}$ at randomly different positions in the software */
14. $\mathcal{P}^{V} \leftarrow \mathcal{P}^{IV} \cup \mathcal{O}$
15. $\mathcal{P}^{VI} \leftarrow$ insertCallsToObfuscatorEngine($\mathcal{P}^{V}$)

Algorithm 5: Proposed algorithm to embed the obfuscation engine into a program

The steps of algorithm 5 are similar to the algorithm 4 except for the steps to embed the obfuscation engine $\mathcal{O}$. This is done to guarantee that before $\mathcal{O}$ runs for the first time, the software will be different for each device. Thus, until the line 10 is reached, the algorithm 5 behaves as algorithm 4. After this line, the algorithm 5 fills $\mathcal{O}$ with the following information in order to ensure that it behaves as expected: $\mathcal{C}_{Instrs}$, $false_{instrs}$, $garbage_{instrs}$ and S_{parts}. $\mathcal{C}_{Instrs}$ is required to the $\mathcal{O}$ knows where new dead execution spots could be created; S_{parts} is proved to $\mathcal{O}$ in

order that it knows the dead execution spots where is located false and garbage instructions. Finally, must to know $false_{instrs}$ and $garbage_{instrs}$ in order to know what kind of instruction is in each dead execution spots informed by S_{parts}. The last steps of this algorithm are responsible to embed the $\mathcal{O}$ and inserts instructions that calls it randomly in different positions in the software.

During the execution of $\mathcal{O}$, it creates new false and garbage instructions, which could be stored in new dead execution spots or simply moved among the existing ones. For this, $\mathcal{O}$ randomly chooses n false instructions and m garbage instructions to be modified in this moment. In the following, it is chosen how many false and garbage instructions should be modified in its shape, in its locations or both. Then it is performed the respective steps to do such actions, i.e. the steps to create new false and garbage instructions, the steps to create new dead execution spots and the steps to move false and garbage instructions to new locations. Finally, before $\mathcal{O}$ returns the control flow to the software, it ensures that the addresses of dead execution spots are known and what kind of instruction is stored in each of them.

4 Case Study

In this section, we present a case study showing the difficulty to disclosure a cryptographic key. After applying each of the proposed methods, it is presented how the difficulty increases.

Since there is not an absolute metric to evaluate software protection methods in the literature, we propose an effort metric (E), which measures the difficulty to achieve a goal, such as disclosure cryptographic keys. This metric is expressed as $E = \sum_{i=1}^{n} T_i \times \alpha_i$, where T_i is a time-based factor, which represents the time required to the attacker to perform a certain task; and α_i is a constant factor that weights the difficulty to the attacker to perform the task related to the time-based factor.

For our experiments, we used the rijndael application of an embedded benchmark suite, called MiBench [12]. This application is an implementation of the AES symmetric cipher to ARM vendor devices [13]. In its original form, such application does not have any method to prevent it against reverse engineering.

The first strategy to disclosure its cryptographic keys is the string analysis. Considering that the attacker knew nothing about the rijndael application a priori, he could examining its code with IDA PRO, a commercial multi-processor disassembler and debugger [14] in order to find out its cryptographic key in the data segment (.rodata). Figure 1 shows that it is possible since the cryptographic key used in this application could be found after identify the string CRYPTO_KEY stored at the program address 0x020311FC. For such, the attacker must to spend the effort $E = T_1 \times \alpha_1$, where T_1 is the time to examine an element in the data segment and α_1 is related to the data segment size and the number of strings in it.

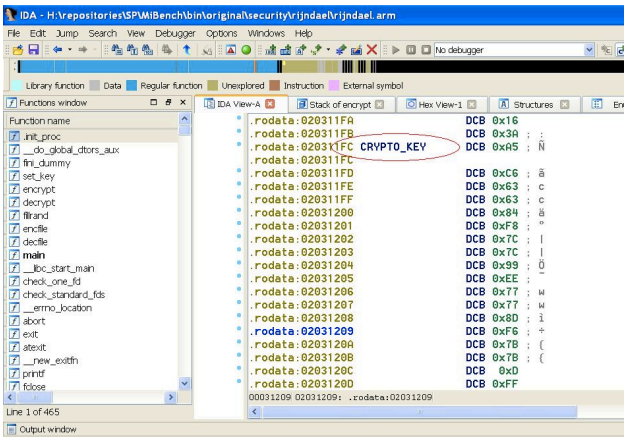

Figure 1. Cryptographic key revealed by string analysis

The first countermeasure, as described in subsection 3.1, to protect the rijndael application consists in moving the cryptographic key to the code segment, more specifically in a dead execution spot. In our experiments, we create a dead execution spot in the code segment of the rijndael application through call obfuscation. A way to perform call obfuscation in ARM is described on [9].

(a)	(b)
.data crypto_key: .word 0xFFFF .text main: 1. bl foo 2. mul r1, r0,#2 3. bl bar 4. div r1,0,#3 foo: 1. stmdb sp!,{r4-r11} 2. add r0,r0,#1 3. ldmia sp!, {r4-r11} 4. ret	.text main: 1. add lr,pc,#4 2. ldr pc,=foo **3. 0xFFFF** 4. mul r1, r0,#2 5. bl bar 6. div r1,0,#3 foo: 1. stmdb sp!,{r4-r11} 2. add r0,r0,#1 3. ldmia sp!, {r4-r11} 4. ret

Figure 2. Original code sample (a) and obfuscated code (b)

Figure 2 shows an example that shows how to create a dead execution spot in ARM architecture using call obfuscation. For such, the call instruction 'bl foo' at line 1 on column (a) is replaced by the following two instructions at the lines 1 and 2 on column (b): 'add lr, pc, # 4' and 'ldr pc, = foo'. The first instruction ('add lr, pc, # 4') is responsible to

ISBN: 978-1-61804-276-7

save the return address, in this case, the new address of the instruction 'mul r1, r0, # 2' to the lr register, which is used, as standard, to store the return address. Then, the second instruction ('ldr pc, = foo') updates the pc register with the address of the first instruction of foo. Thus, it is created a dead execution in the line 3 of column (b) capable to store 4 bytes. In this example, the bytes of (0xFFFF) of the crypto_key is moved from the data segment (.data) to the created dead execution spot.

The dead execution spot of the rijndael application was created in the program address 0x020234C8. In the Figure 3 could be seen the bytes of the cryptographic key in such dead execution spot. Notice that not all bytes were translated as program instructions, some of them are showed as garbage bytes.

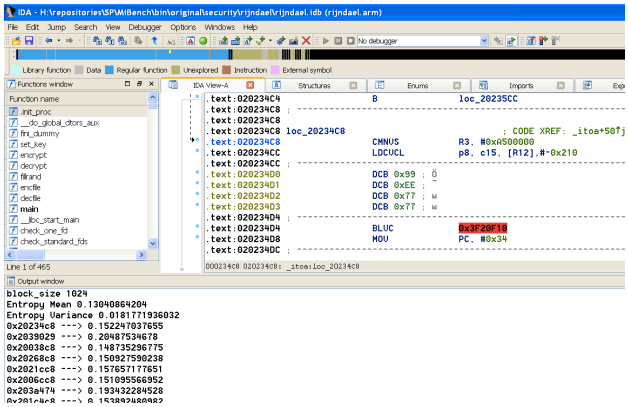

Figure 3. Entropy calculation

Despite the movement of the cryptographic key to the code segment is effective against string analysis, such method is not effective against entropy analysis. To perform the entropy analysis, he created a script in the IDA PRO to calculate for each code block of 1024 bytes the Shannon entropy [10]. After calculate the entropy for all code blocks, it is calculated the entropy mean and the entropy variance. Finally, if the entropy of a code block is greater than the entropy mean more the entropy variance. Then the first program address of such code block is returned. Also, in the Figure 3, is shown the possible candidate program address where the cryptographic key could be located, which are the program addresses of the code block, whose entropy is greater than the entropy mean (0,13040864204) more the entropy variance (0,0181771936).

After examining each candidate program address, the attacker is able to find out the cryptographic key stored in the code segment since one of such candidates is the program address 0x020234C8, i.e. the program address where the cryptographic key is located. The effort to disclosure such cryptographic key using the entropy analysis strategy is $E = T_2 \times \alpha_2$, where T_2 is the time to examine each candidate program address and α_2 is a constant factor that depends on the attacker's capacity to create the script to calculate the entropy and the number of possible candidate program address that were found by such script. Notice that the attacker has performed the string analysis before the entropy analysis, then the total effort to disclosure the desired cryptographic key is $E = T_1 \times \alpha_1 + T_2 \times \alpha_2$

The second countermeasure to protect the cryptographic key, described in the subsection 3.2, is split the cryptographic key and store each part in a distinct dead execution spot, which is randomly disposed in the code segment. In this case, the script that calculates the entropy does not return any possible candidate program address. Thus, the cryptographic key could not been disclosure by entropy analysis.

When the entropy analysis could not help the attacker, he could use the garbage bytes examination, i.e. examine the code segment for garbage bytes. Such bytes appear in the code segment because disassemblers cannot make the correspondence between these bytes with a certain program instruction. On the other hand, such bytes could also be incorrectly translated, as program instructions. It happens when an assumption used by disassemblers are not followed. For example, when disassemblers detect a call instruction, they assume that the return address is the subsequent address after the call instruction. However, since the return address is manipulated to redirect the control flow to other program address, disassemblers still translates the bytes located at return address that they consider as real program instructions.

The effort to disclosure cryptographic key using garbage examination is $E = T_3 \times \alpha_3 + T_4 \times \alpha_4$, where T_3 is the time to find out one garbage byte, α_3 the constant factor related to the number of parts of the cryptographic key, T_4 is the time to combine the collected garbage bytes in a certain order and α_4 dictate the number of combinations, which depends on the number of garbage bytes found in the code segment. Notice that some parts of the cryptographic key will not be translated as garbage instructions, such strategy could not be absolutely effective.

Next, to counter the attacker to find out garbage bytes, it is applied the method described in the subsection 3.3. Assuming that the attacker knows that the cryptographic key parts are camouflaged as false instructions, he could try to identify them in the code segment. However, distinguish the false

ISBN: 978-1-61804-276-7

instructions from actual instructions is difficult because false instructions do not have a format that differentiate the actual instructions. Thus, the attacker should try to identify all the dead execution spots in the code segment (dead execution investigation) before combine the operand part of the false instructions.

One way to identify dead execution spots is by analyzing the control flow. However, to perform such analysis, it is necessary to generate the Control Flow Graph (CFG) of the program and since the CFG depends on the assembly code created by disassemblers and the method to create dead execution spots could infringe certain disassembler assumptions, the CFG created could not be trusted, making difficulty to identify such dead execution spots.

The effort to find out cryptographic key by dead execution investigation is $E = T_5 \times \alpha_5 + T_6 \times \alpha_6$. Such effort depends on to identify the dead execution spots among all program instructions in the code segment and the time to combine the cryptographic key parts ($k_1, k_2, k_3, \ldots, k_n$) since the false instructions could be disorderly arranged in the code segment. In this effort, T_5 is the time to find out one dead execution spot, α_5 depends on the number of dead execution spots and the attacker's capacity to identify dead execution spots, which is difficult because it could be created using several code manipulations. In addition, T_6 is the time to combine the operand bytes of each false instruction in a certain order and α_6 dictate the number of combinations, which depends on the number of dead execution spots found in the code segment.

If an attacker discovers all the dead execution spots, and consequently, the false instructions, the proposed solution was adding garbage instructions also in dead execution spots in order to increase the time to disclosure the cryptographic key. Thus the effort to do this is $E = T_5 \times \alpha_5 + T_6 \times \alpha_6 + T_7 \times \alpha_7$. Beyond such effort must to consider the time to find out false instructions, it has to consider the time to find out garbage instructions (T_7) and its respective constant factor (α_7). Similarly as the distinction between false instructions is difficulty, the distinction between false and garbage instructions also is difficulty. For instance, to reconstruct the cryptographic key $\mathcal{K}$, the attacker must to discover the false instructions among n + m dead execution spots whereas m is the number of garbage instructions and n the number of false instructions.

In order to prevent against program diffing and recurrent attacks, it is applied the last method, described in the subsection 3.5. Since the obfuscation engine change randomly the software running on each device, a comparison of attack can be made impossible. This is because each copy of software is different, since the n key parts are arranged in different places and in different directions false. However, if only the false statements changed, the obfuscation engine could give indications of the location of these keys. However, the obfuscation engine also generates garbage statements at run time. Thus when comparing copies of two different devices, it does not help the attacker to locate a cryptographic key as the software are very different and the effort to understand the differences compares the effort to analyze the two copies altogether.

Software obfuscation diversity provided by the engine is also useful to contain recurrent attacks. This is because the software is constantly changing and therefore cannot take advantage an earlier attack to compromise the same device in the future or commit other devices that have the same software.

(a)	(b)
.data crypto_key: .word 0xFFFF .text main: 1. bl foo 2. mul r1, r0,#2 3. bl bar 4. div r1,0,#3 foo: 1. stmdb sp!,{r4-r11} 2. add r0,r0,#1 3. ldmia sp!, {r4-r11} 4. ret	.text main: 1. add lr,pc,#4 2. ldr pc,=foo **3. 0xFFFF** 4. mul r1, r0,#2 5. bl bar 6. div r1,0,#3 foo: 1. stmdb sp!,{r4-r11} 2. add r0,r0,#1 3. ldmia sp!, {r4-r11} 4. ret

Figure 2. code examples that show two samples created by obfuscation engine operation

The Fig. 4 presents an example that shows how the code of an application changes due to operation of the obfuscation engine at two different times T1 and T2 respectively shown in Fig 2 (a) and Figure 2 (b). In T1, a part of a cryptographic key (0xE0C7) is camouflaged in the false statement 'addeq r8, r4, # 199' on a snippet of non-executable code created through a obfuscation call. In T2, 0xE0C7 is camouflaged within the false instruction 'subne r8, r4, # 199'. Such instruction is stored in a snippet of non-executable code created using an obfuscation return [3,13]. For this, the ret instruction of the function bar is replaced by the instructions 'add r3, lr, # 4' and 'r3 b', able to manipulate the flow of control in order to create a dead execution spot between the function call 'bl bar' and 'div r8, r4, # 199' instruction.

ISBN: 978-1-61804-276-7

5 Conclusion

In this work, we present methods to protect cryptographic keys by hiding them into the code segment. Such methods could be considered appropriate for embedded systems because it decreases the risk disclosure by reverse engineering, without financial costs. For future works, we would increment the cryptographic key protection with anti-debugging and anti-emulation techniques in order to prevent against dynamic analysis tools.

References:

[1] P. Marwedel, “Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems,” TU Dortmund, Informatik, 2011

[2] A. Akhunzada, M. Sookhak, N. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat and M. Khan, “Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions,” Journal of Network and Computer Applications, vol. 48, pp. 44-57, Feb. 2015.

[3] K. Fysarakis, G. Hatzivasilis, K. Rantos, A. Papanikolaou and C. Manifava, “Embedded Systems Security Challenges,” Measurable security for Embedded Computing and Communication Systems (MeSeCCS 2014), within the International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2014), At Lisbon, Portugal, 2014

[4] A. Shamir and N. Someren. “Playing "Hide and Seek" with Stored Keys,” in Proc. 3rd International Conference on Financial Cryptography (FC '99), Matthew K. Franklin (Ed.). Springer-Verlag, London 1999, pp. 118-124.

[5] S. Kinney, “Trusted Platform Module Basics: Using TPM in Embedded Systems (Embedded Technology),” Newnes, 2006

[6] C. Collberg and J. Nagra. “Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection,” Addison-Wesley Professional, Ed. 1, 2009.

[7] John P. McGregor and Ruby B. Lee. 2005. “Protecting cryptographic keys and computations via virtual secure coprocessing”, SIGARCH Comput. Archit. News 33, 1, 16-26, (2005)

[8] Jian Jun Hu; Qiaoyan Wen; Wen Tang; Ai-Fen Sui, “A key hiding based software encryption protection scheme”, Communication Technology (ICCT), IEEE 13th International Conference on , pp.719,722, 25-28. (2011)

[9] Costa, Rafael, Boccardo, Davidson, Gomes, Cleber, Carmo, Luiz ; Pirmez, Luci “Sensitive Information Protection for Advanced Metering Infrastructure”, 10th International Congress on Electrical Metrology (SEMETRO'13), (2013)

[10] G. Croll, “BiEntropy - The Approximate Entropy of a Finite Binary String”, eprint arXiv:1305.0954, Presented at ANPA 34, Rowland's

[11] Suh, G. Edward, and Srinivas Devadas. “Physical unclonable functions for device authentication and secret key generation” Proceedings of the 44th annual Design Automation Conference. ACM, 2007.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC '01). IEEE Computer Society, Washington, DC, USA, 3-14.

[13] HEX-RAYS (2012) "Executive Summary: IDA PRO - at the cornerstone of IT security", http://www.hex-rays.com/products/ida/ida-executive.pdf, June.

[14] ARM7DM Data Sheet, 1994 :Advanced RISC Machines Ltd.