Methods to Protect Cryptographic Keys on Safety-Critical 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: - Safety-critical systems are embedded systems whose failure or malfunction could lead to unacceptable consequences. Despite the major worries about such systems are related to the design of its embedded software, the security is still a challenge to be faced, particularly in terms of data confidentiality, since they could store sensitive that, such as cryptographic keys, which could not be revealed by unauthorized people. Assuming that safety-critical systems are commonly arranged in unprotected areas, without being under surveillance, an attacker could easily capture the respective devices in order to disclosure its cryptographic keys. Thus, it is necessary to create solutions to keep cryptographic keys secret. In this paper is proposed methods to protect cryptographic keys based on code transformations. Since all major protections stand up to a determined attacker's strategies till a certain period of time, we propose methods taking into account what strategies the attacker can perform. We conducted a case study and a discussion to show the difficulty to disclosure cryptographic keys if were used one or more methods proposed here.

Key-Words: - Security, Safety-Critical Systems, Cryptographic Key Protection, Obfuscation

1 Introduction

The cost reduction of commercial off-the-shelf hardware components and the evolution of information systems have allowed the widespread use of embedded systems everywhere. Those systems are being used to bring safety, reliability and efficiency to distinct kind of applications, such as train control systems, flight monitoring systems and so on [1]. However, despite the several benefits provided by the emergence of embedded systems, it has caused a lot of concern, particularly when we are dealing with safety-critical systems, which are embedded systems whose malfunction or failure of could lead to unacceptable consequences, like endanger human lives, substantial economic losses and environmental damages [2]. One example of safety-critical system could be found on railways management systems. Although it could improve the performance of such system, it could lead to a catastrophic accident if any fault occurs on such system [3].

The major concern about the software of safety-critical systems is to guarantee that it behaves like expected. Therefore, they are designed in order to be protected against attacks to its core functions (memory management, process management and data communication), such as buffer overflow attacks [4]. However, besides the software design, there are other security challenges to be faced, such as data confidentiality, particularly if we are dealing with cryptographic keys.

Assuming that safety-critical systems generally is arranged in unprotected areas, i.e. without being under surveillance, their devices are susceptible to Man-Et-The-End (MATE) attacks [5]. Those attacks happen when attackers, after capturing one of those devices, pull out its embedded software to perform reverse engineering in order to extract, insert or modify code or data. For example, an attacker may perform reverse engineering of one safety-critical system in order to disclosure cryptographic keys inner its embedded software [6]. Another example of MATE attack to disclosure cryptographic keys can be found on [7], which take advantage of entropy analysis to find out where cryptographic keys could be stored.

In cryptography, there is a principle, called kerckhoffs's principle, which states that a cryptosystem should be secure even if everything about it is public, except the cryptographic key. Assuming that, if an attacker could disclosure the cryptographic key, the respective cryptosystem could not be considered reliable. For example, if the system responsible to manage railways count on information provided by devices that monitor the railways and such devices were captured, the attacker could extract its cryptographic keys in order to send messages to mislead the system and cause an accident. So, to keep safety-critical systems secure, it is necessary to guarantee that the Kerckhoffs's principle be obeyed. For such, it is

necessary to protect safety-critical systems against MATE attacks whose goal is to disclosure cryptographic keys.

One possible solution to protect cryptographic keys is using a trusted platform module (TPM) [8]. Such solution consists of a microcontroller capable to securely store a small amount of data. Although it could be considered a good solution, the cost of TPM may preclude its use, mainly when the financial resources of the project are limited. In addition, the use of TPM may require that all hardware architecture should be redesign, which also could be expensive. Thus, it may be desirable to develop cheaper solutions to protect cryptographic keys instead of use TPM.

One way to protect cryptographic keys without using a TPM is by 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 [9]. Data obfuscation could be considered a cheap pathway to enhance cryptographic key protection because it tries to hinder reverse engineering without increasing the financial costs. This is because data obfuscation consists only on simple code transformations that demand more time and effort to understand the obfuscated data.

In general, data obfuscation could be broadly classified as static and dynamic [9]. When it is static, the obfuscated data representation remains intact during runtime, independently of the user input or the environment where the software is running. On the other hand, when data obfuscation is dynamic, the software itself changes the representation of the obfuscated data during runtime. In this case, the attacker must to spend more time and effort to understand dynamic obfuscated data compared to static obfuscated data. This is 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 whose data representation will be different each time the software runs.

Assuming that while defense methods are developed, new attack strategies are created to break up them. It is necessary to consider what the attacker can do with the available code analysis tools during the development of such methods. So in this paper it is proposed methods based on software protection techniques to protect cryptographic keys taking into account the attacker's strategies to disclosure them. Such methods are described in an incremental way considering what the approach the attacker will use. The main benefits of our methods are twofold. First, the software developer could use only the methods that will counter the kind of attacker he wants to hinder and, second, the proposed methods are efficient since they do not cause a negative impact in terms of resource consumption because such methods only employ code transformations.

The remainder of this paper is organized as follows. In section 2, it is presented related works, where it is presented a comparison between the proposed work and other studies in the literature. Next, in the section 3 is described in details the proposed methods to protect cryptographic keys as countermeasures against different attacker's strategies. In the following, in the section 4, are shown a case study and a discussion to evaluate the proposed methods in order to demonstrate the difficulty to disclosure cryptographic keys protected by each proposed method in terms of the effort to perform attacks to achieve such goal. Finally, in the section 5 is presented our final remarks and future works.

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 [10] propose architectural enhancements for general-purpose processors capable to 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.* [11] 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.

Azhar et al. propose to transform cryptographic key chunks to a set of randomized functions, which are obfuscated together to the program. Thus, when such keys are required, an inverse transform is executed at the run-time to yield the original bit-strings of the key [12]. This work is similar to ours, but instead of obfuscating a set of cryptographic key chunks into randomized functions, our proposal obfuscates such chunks into instructions.

Sanjeev et al. [13] describe two methods to protect cryptographic keys. The first one isolates the memory location where cryptographic keys are stored using the Virtual Machine Monitor as a cryptographic service provider. Thus, all applications execute in the guest OS, while the cryptosystem run in a separate and secure hardware. The second method provides a way to securely retrieve and store cryptographic keys. For such, the bits of the key are scattered all over unused sector of files on the secondary storage without using the file system calls of the OS to write them.

Bansod *et al.* [14] propose the use of bit level permutation in cryptographic on automobile environment in order to accelerates cryptography, providing low cost security solution. However, such proposal despite pave a new way in securing small scale embedded system, does not cares about protecting the cryptographic key inner the embedded software, and when it comes address automobiles, the manipulation of its information could lead to disastrous results. Thus, our proposal works complementarily since our proposal is intended to protect cryptographic keys against MATE attacks.

Elizabeth *et al.* [15] proposes two schemes to enhance security of Mobile Ad hoc Networks (Manets). The first method proposes to choose the secret key based on elliptic curve of prime field and the second proposes to use RSA algorithm to share such secret key. Before sending and receiving the keys, it is signed by the sender and after verified by the receiver thereby authenticating each other. Although this method could improve the security of Manets, it does not consider the problem cryptographic key disclosure by reverse engineering. Once the attacker dumps the embedded software on a single device of such infrastructure, he could gather the secret key and the pair of RSA keys by software analysis tools. Thus, the methods could not be sufficient to protect Manets since the attacker may send malicious messages or compromise confidential communication.

3 Protect Cryptographic Keys

In this section, we describe the proposed methods to protect cryptographic keys. These methods are described incrementally in order to make easier to the reader understand what countermeasure could be employed against certain attacker strategies.

Assuming an attacker willing to discover cryptographic keys stored inner the embedded software of safety-critical, the first step to accomplish any strategy to achieve his goal is getting the binary code from the disk or memory card where the embedded software is stored. Then, he could use at ease all available code analysis tools. For example, he could use a disassembler to translate the binary code into assembly code (notation used to represent machine code) or go further and use a decompiler to create a high-level representation of the code in a certain programming language, such as c, in order to make easier to analyze the software.

3.1 Move Cryptographic Keys

Generally, an ordinary code is composed by one code segment, usually containing program instructions, and one data segment, commonly used to store data. Since no defense mechanism is implemented to protect cryptographic keys in such code, an attacker could use a strategy called string analysis to disclosure cryptographic keys.

In the **string analysis**, an attacker simply examines the data segment, searching for variables or constant names associated to cryptography, such as the strings ‘crypt’, ‘password’ and etc. Although this strategy could be considered quite naïve, it is still useful when no defense methods are used to protect cryptographic keys.

In order to prevent string analysis, we propose to move cryptographic key from the data segment to the code segment. But, before this, it is necessary to create areas in the code segment that never gets executed, ***dead execution spots***, allowing store the cryptographic key without change the behavior of the software.

Dead execution spots could be created through code transformations capable to manipulate the software control flow. For instance, it is possible to create dead execution spots by code obfuscation techniques, such as call obfuscation, return obfuscation or false return obfuscation combined with control flow manipulation [16].

The code snippet presented in the **Table 1** shows how to create a dead execution spot using call obfuscation for ARM processors[1]. For such, the call instruction 'bl foo' at line 1 on column (a) is replaced by two instructions: 'add lr, pc, # 4' and 'ldr pc, = foo' respectively at the lines 1 and 2 on column (b). The first instruction ('add lr, pc, # 4') is responsible to save the return address, in this case, the new address of the instruction 'mul r1, r0, # 2' to the lr register, which is standard used to store the return address. Then, the second instruction ('ldr pc, = foo') is used to update the pc register with the address of the first instruction of foo, which is the function to be called. After employ such obfuscation, a dead execution is created in the line 3 of column (b) (dead exec spot).

main:	main:
1. bl foo	1. add lr,pc,#4
2. mul r1, r0,#2	2. ldr pc,=foo
foo:	**3. dead exec spot**
3. stmdb sp!,{r4-r11}	4. mul r1, r0,#2
4. add r0,r0,#1	foo:
5. ldmia sp!, {r4-r11}	5. stmdb sp!,{r4-r11}
6. ret	6. add r0,r0,#1
	7. ldmia sp!, {r4-r11}
	8. ret
(a)	(b)

Table 1. Creating dead execution spot using call obfuscation

After creating the dead execution spot capable to store the cryptographic key, such key is moved from the data segment to the code segment, i.e. to the recently created dead execution spot. Then, disassemblers will incorrectly translate the cryptographic key as program instructions.

When cryptographic keys are translated as program instructions, the assembly code, generated by disassemblers, does not represent valid code and, thus, the analysis upon that code could not be considered reliable. In addition, if the assembly code is not reliable, errors will be propagated to the next stages of code analysis, such as the decompilation stage, because it depends on the assembly code. Then the analysis upon the decompiled code could not be reliable too.

Before describing the proposed algorithm to move cryptographic key to a dead execution spot, it is necessary to describe the procedures GetCandidateInstructions (**Procedure 1**) and CreateDeadExecutionSpots (**Procedure 2**).

[1] http://www.arm.com/products/processors/index.php

Procedure 1. GetCandidateInstructions($\mathcal{P}$)

Input: Program $\mathcal{P}$
Output: Candidate Instructions $\mathcal{C}_{Instrs}$

1. $\mathcal{T}$ ← Set of obfuscation techniques
2. $\mathcal{C}_{Instrs} \leftarrow \emptyset$
3. For each instruction $i \in \mathcal{P}$
4. If i may be obfuscated by one technique $t \in \mathcal{T}$
5. $\mathcal{C}_{Instrs} \leftarrow \mathcal{C}_{Instrs} \cup i$
6. End If
7. End For

The **Procedure 1** is responsible to get all candidate instructions, *i.e.* instructions that could be obfuscated in the program $\mathcal{P}$. Such procedure depends on the data structure $\mathcal{T}$, which describes how to perform code transformations to obfuscate certain instructions. The data structure $\mathcal{T}$ could be seen as a vector composed by tuples with three columns, as seen in **Table 2**. The first column contains the label that identifies the obfuscation technique; the second column (*Opcode to Search*) contains the *opcode* that could be obfuscated and the third column (*Replace Instructions*) contains the instructions that act as the instruction that was obfuscated, i.e. the instruction whose *opcode* matches with the *opcode* in the respective second column. For example, the entry in the **Table 2** details the call obfuscation for ARM processors, which is identified by the label t_1. In this entry the *opcode* to search is 'bl' and, according to this entry, such instruction could be replaced by the instructions '*add lr, pc, #hop*' and '*ldr pc,=function_address*'. The constant *hop* represents how many addresses should be added from the address contained in the *pc* register and *function_address* represents the address of the function to be called.

Obfuscation Technique	Opcode to Search	Replace Instructions
t_1	bl *function_address*	add lr,pc,#*hop*
		ldr pc,= *function_address*

Table 2. Typical entry of the set of obfuscation techniques $\mathcal{T}$

During the execution of **Procedure 1**, each instruction of $\mathcal{P}$ is inspected to check if it could be obfuscated by at least one obfuscation technique in $\mathcal{T}$. If so, the position of the respective instruction (i) is stored in the vector of candidate instructions ($\mathcal{C}_{Instrs}$). The checking process in the step 4 of **Procedure 1** occurs as follows: the *opcode* of the instruction locate at i is compared with the *opcode*

located at the column *Opcode to Search* of all entries of $\mathcal{T}$ until one or none entry matches.

The **Procedure 2** is responsible to create a dead execution spot. Such procedure requires the parameters: the program $\mathcal{P}$, the program address s and the cryptographic key size ℓ. First, this procedure should go on $\mathcal{P}$ until it comes to s. For this, it is necessary to verify for each instruction $i \in \mathcal{P}$, if i is s. When the condition of line 3 is true, an obfuscation technique capable to obfuscate the instruction at s (t) should be uniformly random chosen (line 4). Then, in the line 5, code transformations dictated by t are applied on s in order to create a dead execution spot capable to store ℓ bytes.

Procedure 2. *CreateDeadExecutionSpots*($\mathcal{P}, s, \ell$)

Input: Program $\mathcal{P}$, program address s, key size ℓ
Output: Program $\mathcal{P}^{I}$

1. $\mathcal{T}$ ← Set of obfuscation techniques
2. For each instruction $i \in \mathcal{P}$
3. If $i = s$
4. Choose uniformly random $t \in \mathcal{T}$ such that it could obfuscate the instruction at s
5. Employ code transformations dictated by t on s that manipulates the control flow to create a dead execution spot ℓ sized
6. End If
7. End For

In the following, we present the proposed algorithm to modify a program in order to protect its cryptographic key against string analysis (**Algorithm 1**). Such algorithm receives the following information as input: program $\mathcal{P}$, cryptographic key position $\mathcal{K}$ and the size of the respective cryptographic key ℓ. Then, it returns the program $\mathcal{P}^{II}$ whose cryptographic key was moved to one dead execution spot.

The first step of this algorithm is getting the candidate instruction addresses ($\mathcal{C}_{Instrs}$). For this, it is used the **Procedure 1**. Next, in the line 2, it is randomly chosen one instruction address among $\mathcal{C}_{Instrs}$, i.e. s. In the following, in the line 3, it is called the procedure *CreateDeadExecutionSpot* (**Procedure 2**) in order to create a dead execution spot capable to store the cryptographic key located in $\mathcal{K}$. Next, in the line 4, is called the function *MoveKey*, which is responsible to move all ℓ bytes of the cryptographic key from the $\mathcal{K}$ address to the dead execution spot created at s. Finally, the loop started on line 5 is used to check every instruction in the program in order to check if it uses the cryptographic key that was moved. If so, the reference to the cryptographic key in such instruction is changed, i.e. from $\mathcal{K}$ to s.

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

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

1. $\mathcal{C}_{Instrs}$ ← GetCandidateInstructions($\mathcal{P}$)
2. s ← GetRandomCandidate($\mathcal{C}_{Instrs}$)
3. $\mathcal{P}^{I}$ ← CreateDeadExecutionSpot ($\mathcal{P}, s, \ell$)
4. $\mathcal{P}^{II}$ ← MoveKey($\mathcal{P}^{I}, \mathcal{K}, s, \ell$)
5. For each instruction $i \in \mathcal{P}^{II}$
6. If i refers to $\mathcal{K}$
7. Change the reference of i from $\mathcal{K}$ to s
8. End If
9. End For

Considering that the procedure *GetRandomCandidate* is performed in constant time and the data structure $\mathcal{T}$ could be a vector whose positions could be accessed in constant time. The complexity of **Algorithm 1** could be obtained through the complexity of the procedures it calls and the loop starting at line 5.

Assuming that the number of program instructions is equal to the program size ($|\mathcal{P}|$). The complexity of *GetCandidateInstructions* is $O\ (|\mathcal{P}| * |\mathcal{T}|)$ because for each instruction, we verify in $\mathcal{T}$ what obfuscation technique can be used for obfuscate such instruction. On the other hand, the complexity of *createDeadExecutionSpots* is $O\ (|\mathcal{P}|)$ since such procedure needs to find out the program address s in $\mathcal{P}$ and the code transformations are performed in constant time. Finally, the complexity of the procedure *MoveKey* is $O(|\mathcal{P}|)$ because this procedure simply moves the entire key to one dead execution spot once, requiring to find out the program address to move the instruction. Thus, the complexity of the **Algorithm 1** is $O(|\mathcal{P}| * |\mathcal{T}|)$, *i.e* the complexity of the procedure *GetCandidateInstructions*.

3.2 Split Cryptographic Key

If the string analysis strategy could not help attackers to find out cryptographic keys, attackers must to use other strategies, such as the entropy analysis. In the **entropy analysis**, an attacker measures the entropy of the code, i.e. measure its disorder in order to have evidence where the cryptographic key could be [17].

Generally the entropy of the code segment is low because the bytes that comprise the instructions are

always the same and repeat. If certain code areas have high entropy, it could mean that such areas store cryptographic keys since they are composed by random bytes [7]. So, the entropy analysis could help attackers to identify possible candidate positions where cryptographic keys could be stored.

In order to prevent entropy analysis, we propose to split the cryptographic key and moving each part of it to several dead execution spots instead of moving the entire cryptographic key to one single dead execution spot. Therefore, the entropy analysis could not indicate candidate position where cryptographic keys could be because the random bytes are distributed among the code and do not impact on the bytes frequency of the code segment.

In the following, we present the proposed algorithm to split cryptographic key and store its parts in dead execution spots in order to prevent against entropy analysis and consequently against string analysis too (**Algorithm 2**). Such algorithm receives as input the program $\mathcal{P}$, the cryptographic key position $\mathcal{K}$ and the part size n; and returns the program $\mathcal{P}^{II}$ whose cryptographic key is splitted in parts fragment sized and moved to various dead execution spots.

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

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

1. $\mathcal{C}_{Instrs}$ ← GetCandidateInstructions($\mathcal{P}$)
2. $\mathcal{K}_{Parts}$ ← SplitCryptographicKey ($\mathcal{K}, n$)
3. For each $k_i \in \mathcal{K}_{Parts}$
4. $\quad$ $\mathcal{S}_{parts}$ ← GetRandomCandidate($\mathcal{C}_{Instrs}$)
5. End For
6. For each $s_i \in \mathcal{S}_{Parts}$
7. $\quad$ $\mathcal{P}^{I}$ ← CreateDeadExecutionSpot ($\mathcal{P}, s_i, t, n$)
8. $\quad$ $\mathcal{P}^{II}$ ← MoveKey($\mathcal{P}^{I}, k_i, s_i$)
9. End For
10. $\mathcal{P}^{II}$ ← InsertCallsToReconstructRoutine ($\mathcal{P}^{III}, \mathcal{K}$)

As in the algorithm 1, the first step of the **Algorithm 2** is to call the *GetCandidateInstructions* procedure. Then, in the line 2 is called the procedure *SplitCryptographicKey* to split the cryptographic key located at $\mathcal{K}$ in n sized parts. Notice that the size of the last part of the cryptographic key could be different to the other parts, i.e. shorter than n. The position of each part of the cryptographic key is stored in the data structure called $\mathcal{K}_{Parts}$. In the following, in the line 3 begin a loop to choose randomly addresses from $\mathcal{C}_{Instrs}$ to create dead execution spots to store each part of the cryptographic key pointed in $\mathcal{K}_{Parts}$ (k_i). Each address chosen using the *GetRandomAddress* procedure is stored at the data structure called $\mathcal{S}_{parts}$. Next, for each address s_i in $\mathcal{S}_{parts}$ it is called the procedure *CreateDeadExecutionSpot* to create one dead execution spot to store one part of the cryptographic key k_i. Then, one part of the cryptographic key is moved to the recently 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. For such, in the line 10, is called the procedure InsertCallsToReconstructRoutine (**Procedure 3**). The **Procedure 3** requires as input the program $\mathcal{P}$, the cryptographic key position $\mathcal{K}$. First, it is appended in $\mathcal{P}$ the reconstruct cryptographic key routine $\mathcal{R}$. Then, for each instruction i, is verified if such instruction refers to the cryptographic key in $\mathcal{K}$. If so, it is added before this instruction, a call instruction to execute $\mathcal{R}$.

Procedure 3. InsertCallsToReconstructRoutine($\mathcal{P}$,$\mathcal{K}$)

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

1. $\mathcal{P}^{II} \leftarrow \mathcal{P}^{I} \cup \mathcal{R}$
2. For each instruction $i \in \mathcal{P}^{II}$
3. $\quad$ If i refers $\mathcal{K}$
4. $\qquad$ insert call to $\mathcal{R}$ in $i - 1$
5. $\quad$ End If
6. End For

Assuming that the procedures *SplitCryptographicKey* and *GetRandomCandidate* are performed in constant time and the complexity of the procedures *GetCandidateInstructions, createDeadExecutionSpots* and *MoveKey* are $O\,(|\mathcal{P}| * |\mathcal{T}|)$, $O\,(|\mathcal{P}|)$ and $O(|\mathcal{P}|)$ respectively. Then, the complexity of complexity of **Algorithm 2** could be obtained through analysis of the procedure *InsertCallsToReconstructRoutine* and the loops that begin at lines 3 and 6.

Since the procedure *GetRandomCandidate* is called for each part k_i of $\mathcal{K}_{Parts}$, the complexity of the loop beginning at line 3 is $O(|\mathcal{K}_{Parts}|)$ because the number of iterations is equal to the size of $\mathcal{K}_{Parts}$, which is the size of cryptographic key located in K divided by the part size n. Similarly, the complexity of the loop beginning at line 6 depends on the size of $\mathcal{K}_{Parts}$ because $|\mathcal{S}_{parts}|$ is

equal to $|\mathcal{K}_{Parts}|$. Since the complexity of the procedures inside this loop is $O(|\mathcal{P}|)$, then the complexity of this loop is $O(|\mathcal{K}_{Parts}| * |\mathcal{P}|)$ because it is necessary to go through the program until find the program address where to create a dead execution spot and move the cryptographic key to such dead execution spot.

Finally, the complexity of the **Procedure 3** is $O\ (|\mathcal{P}|)$ because insert the call instruction takes constant time and it is necessary to verify all instructions of the program $|\mathcal{P}|$. Thus, the complexity of the **Algorithm 2** is $O(|\mathcal{K}_{Parts}| * |\mathcal{P}|)$.

3.3 Create False Instructions

When the previously strategies, i.*e.* string and entropy analysis could not help attackers to find out where the cryptographic key is, one possible strategy that attackers could use is **junky bytes investigation**. Such strategy states that the attacker must to search in the code segment for bytes that are not matched as program instructions (junky bytes) and combine those bytes in order to find out cryptographic keys.

Since junky bytes do not appear very often in the code segment, the attacker could deduce that such bytes is or belongs to a certain cryptographic key, which may have been moved from the data segment to the code segment. Thus, after the attacker finds all junky bytes, he could verify them individually or combined if it is the desired cryptographic key. Besides combining all junky bytes demands time, an attacker with time and dedication could always reveal the desired cryptographic key and the only thing to do is trying to slow down him. Although not all junky bytes belongs to cryptographic keys, this strategy is useful when the cryptographic key is splitted and moved to the code segment in several dead execution spots because the bytes of each part of the cryptographic key could not be translated as program instructions.

In order to detain junky bytes investigation strategy, we propose to camouflage all cryptographic key parts into **false instructions**, which are instruction composed by a random *opcode* attached with a single part of the cryptographic key, which will look like as 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 attach them with random *opcodes*, it is necessary to create the required number of dead execution spots to store all generated false instructions. Notice that false instructions 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 junky byte investigation. **Algorithm 3** receives as input the program $\mathcal{P}$, the cryptographic key $\mathcal{K}$ and the *operand* size n; and returns $\mathcal{P}^{III}$, 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.

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

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

1. $\mathcal{C}_{Instrs}$ ← GetCandidateInstructions($\mathcal{P}$)
2. $\mathcal{K}_{Parts}$ ← SplitCryptographicKey($\mathcal{K}$,n)
3. $false_{instrs} \leftarrow \emptyset$
4. For each $k_i \in \mathcal{K}_{Parts}$
5. $\quad f \leftarrow Get$RandomOpcode() $\circ\, k_i$
6. $\quad false_{instrs} \leftarrow false_{instrs} \cup f$
7. End For
8. $\mathcal{S}_{parts}$ ← GetRandomCandidates($\mathcal{C}_{Instrs}$, $|\mathcal{K}_{Parts}|$)
9. For each $s_i \in \mathcal{S}_{Parts}$
10. $\quad$ Choose f_i from $false_{instrs}$
11. $\quad \mathcal{P}^{I}$ ← CreateDeadExecutionSpot($\mathcal{P}, s_i, t, |f_i|$)
12. $\quad \mathcal{P}^{II}$ ← MoveFalseInstruction($\mathcal{P}^{I}, f_i, s_i$)
13. End For
14. $\mathcal{P}^{III}$ ← InsertCallsToReconstructRoutine($\mathcal{P}^{II}, \mathcal{K}$)

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 opcode by the procedure *RandomOpcode* and (ii) attaching it with k, which will be translated as the *operand* of such instruction (line 5). Each created false instruction is store in the data structure $false_{instrs}$.

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 $|f_i|$. First, it is necessary to choose from the candidate instructions $(\mathcal{C}_{Instrs})$ the instructions to be obfuscated in order to create dead execution spots (line 8). In the following, for each instruction s_i in $\mathcal{S}_{Parts}$ is chosen

one false instruction (f_i). Then, it is called the procedures *CreateDeadExecutionSpot* and *MoveFalseInstruction* in order to create the dead execution spot capable to store f_i and move such instruction to this dead execution spots respectively. Finally it is embedded the reconstruction cryptographic key routine to the program and calls to it.

Assuming that the procedures *SplitCryptographicKey*, *GetRandomOpcode* and *GetRandomCandidates* are performed in constant time and the complexity of the procedures *GetCandidateInstruction*, *createDeadExecutionSpot* and *MoveFalseInstruction* are $O\ (|\mathcal{P}| * |\mathcal{T}|)$, $O\ (|\mathcal{P}|)$ and $O(|\mathcal{P}|)$ respectively. Then, the complexity of complexity of **Algorithm 3** could be obtained through analysis of the loops that begin at lines 4 and 9. Notice that the complexity of *MoveFalseInstruction* is similar to the complexity of *MoveKey*.

In the loop beginning in the line 4, for each cryptographic key part (k_i) is created one false instruction (f). Then, the complexity associated to that loop is $O(|\mathcal{K}_{Parts}|)$ since the procedure *GetRandomOpcode* executes in constant time. On the other hand, the complexity of the loop beginning in the line 9 depends on the number of false instructions ($false_{instrs}$) and the complexity of the procedures *createDeadExecutionSpot* and *MoveFalseInstruction*. Thus this complexity is $O\ (|\mathcal{P}| * |false_{instrs}|)$, which is the complexity of the **Algorithm 3**.

3.4 Insert Garbage Instructions

Assuming an attacker knowing that the cryptographic key was splitted and its parts camouflaged in false instructions, he could use the **dead execution spot investigation** to reveal such cryptographic key, which states to examine the code segment in pursuit of dead execution spots since false instructions are store in such spots. Although it is difficult to find out dead execution spots because it requires sufficient test inputs to achieve all paths that produce meaningful behavior, it is possible to attackers verify areas in the code segment that never gets executed (dead execution spots) and, thus, find out all the parts of cryptographic keys, which are the operands of the false instructions in such dead execution spots.

In order to counter dead execution spot investigation, we propose inserting **garbage instructions**, which are random program instructions in dead execution spots too. So, attackers require more time and effort to reveal the cryptographic key because he could not distinguish false instructions from garbage instructions.

Algorithm 4 describes the steps of the proposed method that camouflages cryptographic key parts into false instructions and insert garbage instructions. 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 $\mathcal{P}^{III}$, which is the program whose cryptographic key is hidden in false instruction and has garbage instructions distributed in the code segment.

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

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

Output: Program $\mathcal{P}^{III}$

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. $garbage_{instrs} \leftarrow \emptyset$
5. While $|garbage_{instrs}| < m$
6. $g \leftarrow$ RandomOpcode() $\circ$ RandomOperand()
7. $garbage_{instrs} \leftarrow garbage_{instrs} \cup g$
8. End While
9. $\mathcal{M} \leftarrow |\mathcal{K}_{Parts}| + m$
10. $\mathcal{S}_{parts} \leftarrow$ GetRandomCandidates($\mathcal{C}_{Instrs}, \mathcal{M}$)
11. For each $s_i \in \mathcal{S}_{Parts}$
12. $h \leftarrow f_i$ from $false_{instrs}$ or $g_i \in garbage_{instrs}$
13. $\mathcal{P}^{I} \leftarrow$ CreateDeadExecutionSpot($\mathcal{P}, s_i, t, h$)
14. $\mathcal{P}^{II} \leftarrow$ MoveInstruction($\mathcal{P}^{I}, h, s_j$)
15. End For
16. $\mathcal{P}^{III} \leftarrow$ InsertCallsToReconstructRoutine($\mathcal{P}^{II}, \mathcal{K}$)

The differences between this algorithm and algorithm 3 are: (i) the steps to create garbage instructions that do not exist in the previous algorithm (lines 4 to 8) and (ii) the number of dead execution spots that should be created (line 9). After getting the candidate instructions ($\mathcal{C}_{Instrs}$) and creating the false instructions by the procedures *GetCandidateInstructions* and *SplitCryptographicKey* respectively, this algorithm creates m garbage instructions. Each garbage instruction is created by a random *opcode* with random *operands (*line 6*)*. Next, it is chosen $\mathcal{M}$ addresses from $\mathcal{C}_{Instrs}$ and stored at $\mathcal{S}_{parts}$. In the following, for each $s_i \in \mathcal{S}_{Parts}$ is chosen one instruction from $false_{instrs}$ or from $garbage_{instrs}$ (h) and for each instruction h is created one dead execution spot to store it and then h is moved to the

created dead execution spot (lines 13 and 14 respectively). Finally, the reconstruction cryptographic key routine is attached to the program and calls to it are inserted before each instruction that refers to $\mathcal{K}$.

Assuming that the procedures *SplitCryptographicKey*, *GetRandomOpcode, GetRandomOperand* and *GetRandomCandidates* are performed in constant time and the complexity of the procedures *GetCandidateInstruction*, *createDeadExecutionSpot* and *MoveInstruction* are $O\ (|\mathcal{P}| * |\mathcal{T}|)$, $O\ (|\mathcal{P}|)$ and $O(|\mathcal{P}|)$ respectively. Then, the complexity of complexity of **Algorithm 4** could be obtained through analysis of the loop that begin at lines 4 to create garbage instructions (lines 4 to 8), which is $O\ (|garbage_{instrs}|)$ and the loop starting at line 10, which is $O\ (|\mathcal{P}| * |\mathcal{M}\ |)$ since it is necessary for each garbage and false instruction (total of instructions is $\mathcal{M}$) to go create dead execution spots and move such instruction to that spot, which requires $O\ (|\mathcal{P}|)$. So, the complexity of the **Algorithm 4** is $O\ (|\mathcal{P}| * |\mathcal{M}\ |)$.

3.5 Insert Obfuscation Engine

The previously methods try to hinder reverse engineering assuming attackers can only use static code analysis tools. However, such methods are not effective when he can use dynamic code analysis tools, such as debuggers and emulators because the software is static. Beyond that, attackers could have different strategies taking advantage that the software is static.

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

The other strategy that takes advantage of static software is the **recurrent attack**. The goal of this strategy is compromise the largest possible number of devices. Assuming a scenario where there are many devices containing identical software embedded in it, once an attacker could find out the program address where the cryptographic key is located in one of such copies, he may create a script to remotely read such address 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 inserting to the embedded software an obfuscation engine that dynamically modify both false and 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 will be different since this software periodically changes its code differently and not only in the program addresses where the cryptographic key is. So, there are many program addresses in the code that are different, making difficult to know cryptographic key position by simply comparing two or more copies. 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 dynamic analysis and the strategies 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 $\mathcal{P}^{III}$ containing an obfuscation engine capable to periodically hides cryptographic key in false instructions and inserts garbage instructions in different ways, by changing its shape and its position inside the code. 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) [18], in order to be the seed to the random functions that creates new false instructions and/or garbage instructions.

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 13 is reached, the algorithm 5 behaves as the algorithm 4. After this line, the algorithm 5 fills $\mathcal{O}$ with the following information in order to ensure that it has the required information to create new dead execution spots and move the existing false and garbage instructions: $\mathcal{C}_{Instrs}$, $false_{instrs}$, $garbage_{instrs}$ and S_{parts}. $\mathcal{C}_{Instrs}$ is required to the $\mathcal{O}$ knows where new dead execution

spots could be created; S_{parts} is provided to O in order that it knows the dead execution spots where is stored false and garbage instructions. Finally, it 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 O and inserts instructions that calls it randomly in different program addresses in the software.

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

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

Output: Program $\mathcal{P}^{III}$

1. $\mathcal{C}_{Instrs}$ ← GetCandidateInstructions($\mathcal{P}$)
2. $\mathcal{K}_{Parts}$ ← SplitCryptographicKey($\mathcal{K}$,n)
3. $false_{instrs}$ ← CreateFalseInstructions($\mathcal{K}_{Parts}$)
4. $garbage_{instrs}$ ← CreateGarbageInstructions(m)
5. $\mathcal{M} \leftarrow |\mathcal{K}_{Parts}| + m$
6. $\mathcal{S}_{parts}$ ← GetRandomCandidates($\mathcal{C}_{Instrs}$, $\mathcal{M}$)
7. For each $s_i \in \mathcal{S}_{Parts}$
8. $\quad h \leftarrow f_i$ from $false_{instrs}$ or $g_i \in garbage_{instrs}$
9. $\quad \mathcal{P}^{I}$ ← CreateDeadExecutionSpot($\mathcal{P}, s_i, t, h$)
10. $\quad \mathcal{P}^{II}$ ← MoveInstruction($\mathcal{P}^{I}, h , s_j$)
11. End For
12. $\mathcal{P}^{III}$ ← InsertCallsToReconstructRoutine($\mathcal{P}^{II}, \mathcal{K}$)
13. $O \leftarrow O \cup \mathcal{C}_{Instrs} \cup false_{instrs} \cup garbage_{instrs}$
14. $\mathcal{P}^{IV} \leftarrow \mathcal{P}^{III} \cup O$
15. $\mathcal{P}^{V}$ ← insertCallsToObfuscatorEngine($\mathcal{P}^{IV}$)

During the execution of 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, 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 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.

The complexity of **Algorithm 5** is identical to the complexity of the **Algorithm 4**, i.e. $O\ (|\mathcal{P}| * |\mathcal{M}\ |)$ because the instructions at lines 13 and 14 operates at constant time and the complexity of the procedure *insertCallsToObfuscatorEngine* is identical of the complexity of *InsertCallsToReconstructRoutine*, i.e. $O\ (|\mathcal{P}|)$.

3.6 Insert Anti-Debugging Techniques

When attackers could use dynamic analysis tools to seek cryptographic keys, even if the software is dynamic, i.e. changes its own code during runtime, they could disclosure cryptographic keys since it is not an absolute protection. Thus, to improve cryptographic key protection, we could insert anti-debugging techniques in order to prevent against dynamic analysis tools [19].

Debuggers such as GDB[2] (The GNU Project Debugger) and OllyDbg[3] provide an interface that links the hardware subsystem and the human analyst. Anti-debugging tricks work by detecting or exploiting specific debugging subsystems. Software that employs anti-debugging techniques can determine if it's being debugged by identifying the debugging process whether from the software or hardware. For example, when it is detected an unexpected pause in execution, it could give evidence that an attacker has paused the software to analyze it. The evidence of the debugging process could be simply detected by take the time in two periods and compare them. If the time difference is higher than usual time, it implies that a debugger is used to analyze the software. On the other hand, hardware debugging could be detected by checking debug registers for specific values since hardware debuggers use such registers to place breakpoints on processes.

4 Case Study and Discussion

In this section, we present a case study showing the difficulty to disclosure cryptographic keys by the proposed protection methods. In the following, we show after employ each of the proposed methods, how the effort to disclosure cryptographic key 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, representing the time required to the attacker to perform a certain task; and α_i is a constant factor that weights the respective difficulty to perform the related task.

[2] https://www.gnu.org/software/gdb/

[3] http://www.ollydbg.de/

For our experiments, we used the rijndael application of an embedded benchmark suite, called MiBench [20]. This application is an implementation of the AES symmetric cipher to ARM processors. In its original form, such application does not have any method to prevent MATE attacks.

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[4], a commercial multi-processor disassembler and debugger, in order to find out its cryptographic key in the data segment (.rodata). Figure 1 shows that it is possible to find out the cryptographic key in its application since it could be found after identifying the string CRYPTO_KEY stored at the program address 0x020311FC. For such, attackers must to spend the effort $E = T_1 \times \alpha_1$, where T_1 is the time to examine one string in the data segment and α_1 is related to number of strings in the data segment.

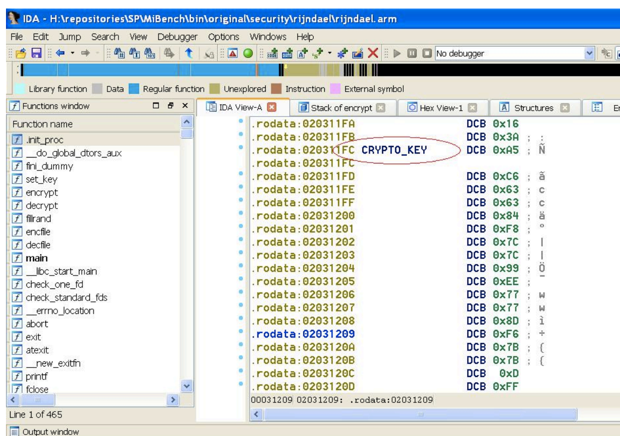

Figure 1. Cryptographic key revealed by string analysis

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 [16].

Table 3 shows how to create a dead execution spot in ARM using call obfuscation. In this example, the bytes of the crypto_key (0xFFFF) are moved from the data segment (.data) of the code snippet in the column (a) to the created dead execution spot at line 3 of the code snippet in the column (b). We follow the same knowledge to create in the rijndael application one dead execution spot. In the Figure 2 it is possible to see the cryptographic key CRYPT_KEY, first shown in the Figure 1, translated as program instruction from the program address 0x020234C8. Notice that not all bytes of CRYPT_KEY were translated as program instructions since some of them were translated as junky bytes, such as the bytes at 0x020234D0.

.data	.text
crypto_key: .word 0xFFFF	main:
.text	1. **add lr,pc,#4**
main:	2. **ldr pc,=foo**
7. bl foo	3. **0xFFFF**
8. mul r1, r0,#2	4. mul r1, r0,#2
9. bl bar	5. bl bar
10. div r1,0,#3	6. div r1,0,#3
foo:	foo:
11. stmdb sp!,{r4-r11}	1. stmdb sp!,{r4-r11}
12. add r0,r0,#1	2. add r0,r0,#1
13. ldmia sp!, {r4-r11}	3. ldmia sp!, {r4-r11}
14. ret	4. ret
bar:	bar:
1. stmdb sp!,{r4-r11}	9. stmdb sp!,{r4-r11}
2. sub r0,r0,#1	10. sub r0,r0,#1
3. ldmia sp!, {r4-r11}	11. ldmia sp!, {r4-r11}
4. ret	12. ret
(a)	(b)

Table 3. Original code sample (a) and obfuscated code (b)

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, we created a script in the IDA PRO to calculate for each code block of 1024 bytes the shannon entropy [17]. After calculate the entropy for all code blocks, we calculate 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.

The Figure 2 shows the program addresses 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 the results of the script, we are able to find out the cryptographic key since one of program addresses returned by the script is the program address 0x020234C8, *i.e.* the program address where the dead execution spot were created to store the cryptographic key.

[4] http://www.hex-rays.com/products/ida/

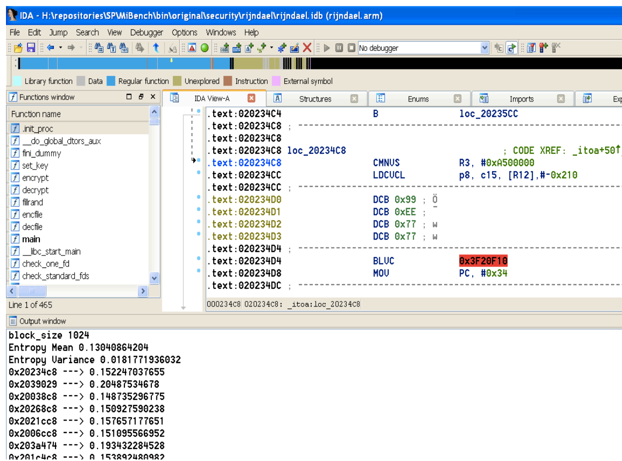
Figure 2. Entropy Calculation

The effort to disclosure the cryptographic key using the entropy analysis strategy is $E = T_2 \times \alpha_2$, where T_2 is the time to examine each program address returned by the script and α_2 is a constant factor related to the number of program addresses returned by the script to calculate the entropy regardless the effort to create such script. Notice that if the attacker has performed the string analysis before the entropy analysis, then the total effort to disclosure the desired cryptographic key is $E = T_1 \times \alpha_1 + T_2 \times \alpha_2$ and so on.

The second countermeasure proposed in this paper, described in the subsection 3.2, is split the cryptographic key and store its parts in a distinct dead execution spots, 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.

If the entropy analysis could not help attackers, they could use other strategy, such as the junky bytes examination, i.e. examine the code segment for junky bytes. Such bytes appear in the code segment because disassemblers could not make the correspondence between these bytes with a certain program instruction. However, 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 junky bytes examination is $E = T_3 \times \alpha_3 + T_4 \times \alpha_4$, where T_3 is the time to find out one junky byte, α_3 the total number of junky bytes, T_4 the time to combine the junky bytes in a certain order and α_4 dictate the number of combinations, which depends on the number of junky bytes found in the code segment. Notice that some parts of the cryptographic key will not be translated as garbage instructions. Thus such strategy could not be absolutely effective.

Next, to counter the attacker to find out junky bytes, it is applied the method described in the subsection 3.3. Then, the cryptographic key parts, which before were translated as junky bytes, now are operand of false instructions, making difficult to attackers to use junky bytes investigation to find out cryptographic keys.

Assuming that attackers know that the cryptographic key parts are camouflaged as false instructions, they could try to identify such instructions in the code segment. However, distinguish false instructions from actual instructions is difficult because false instructions do not have a standard format that differentiate it from actual instructions. Thus, attackers should try to identify all the dead execution spots in the code segment (dead execution investigation).

One way to identify dead execution spots is by analyzing the software 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 reliable, making difficulty to identify such dead execution spots.

The effort to find out cryptographic keys by dead execution investigation is $E = T_5 \times \alpha_5 + T_6 \times \alpha_6$. Such effort depends on the identification of dead execution spots among all program instructions 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. For so, T_5 is the time to find out one dead execution spot, α_5 depends on the number of dead execution spots regardless of the attacker's capacity to identify dead execution spots. 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 to counter such attacker was inserting garbage instructions in dead execution spots too in order to increase the number of combinations to disclosure the cryptographic key.

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 from actual instructions is difficulty, the distinction between false and garbage instructions is even more difficult. 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 changes randomly the software running on each device, thus making program diffing could be considered impossible. This is because each copy of software has different code, since the n parts of the cryptographic key are arranged in different places and in different false instructions. Notice that if only the false instructions are changed, the obfuscation engine could give indications of the location of these keys. However, the obfuscation engine also generates garbage instructions at runtime. Thus when comparing copies of two different devices, it does not help the attacker to locate a cryptographic key as the copies are very different and the effort to understand the differences between them makes it almost impossible.

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

The Table 4 presents an example showing how an application could changes due the obfuscation engine operation at two different times T1 and T2 respectively shown in column (a) and (b). At T1, the cryptographic key part (0xE0C7) is camouflaged in the false instruction 'addeq r8, r4, # 199' on dead execution spot created by call obfuscation. In T2, 0xE0C7 is camouflaged within the false instruction 'subne r8, r4, # 199'. Such instruction is stored in a dead exexcution spots created with return obfuscation. For this, the ret instruction of the function bar is replaced by the instructions 'add r3, lr, # 4' and 'b r3', which manipulate the control flow in order to create a dead execution spot between the call instruction 'bl foo' and the instruction 'mul r1, r2, #2'.

.text	.text
main:	main:
1. add lr,pc,#4	1. add lr,pc,#4
2. ldr pc,=foo	2. bl foo
3. addeq r8,r4,#199	**3. subne r8,r4,#199**
4. mul r1, r0,#2	4. mul r1, r0,#2
5. bl bar	5. bl bar
6. div r1,0,#3	6. div r1,0,#3
foo:	foo:
7. stmdb sp!,{r4-r11}	7. stmdb sp!,{r4-r11}
8. add r0,r0,#1	8. add r0,r0,#1
9. ldmia sp!, {r4-r11}	9. ldmia sp!, {r4-r11}
10. ret	10. add r3,lr,#4
	11. b r3
(a)	(b)

Table 4 code examples that show two samples created by obfuscation engine operation at two different times T1 (a) and T2 (b)

5 Conclusion

Security of cryptographic mechanisms is ultimately based on the assumption that cryptographic keys are kept secret. This assumption is very difficult to accommodate because with time and creativity, and attacker always could achieve his goal. However, in this work, we presented methods to hinder script kiddies and slow down skilled attackers. Such methods could be considered appropriate for safety-critical systems because it decreases the risk of disclosure cryptographic keys by reverse engineering, without financial costs and with little impact against resource consumptions, such as memory and processing.

5.1 Future Works

For future works, we intend to improve the evaluation of proposed methods by analyzing the effectiveness and efficiency of human analysis through forms that subjectively measure the weights of alphas used to measure the effort to perform the attacks to disclosure cryptographic keys [21] [22].

Acknowledgment:
This work is partly supported by P&D Eletrobrás through the process DR/069/2012 for Luiz Fernando Rust; by the National Council for Scientific and Technological Development (CNPq) through processes 477223/2012-5, 473851/2012-1, 4781174/2010-1, 309270/2009-0 for Luci Pirmez; 563096/2010-1 for Davidson Boccardo; 550125/2012-4 for Cleber Gomes; by the Financier of Studies and Projects (FINEP) through processes 01.10.0549.00 and 01.10.0064.00 for Luci Pirmez;

References:

[1] P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*, 2nd ed. Springer Netherlands, 2011.

[2] J. Knight, "Safety Critical Systems: Challenges and Directions," in *International Conference on Software Engineering*, 2002, pp. 547–550.

[3] H. Dong, B. Ning, B. Cai, and Z. Hou, "Automatic train control system development and simulation for high-speed railways," *IEEE Circuits and Systems Magazine*, vol. 10, no. 2, pp. 6–18, 2010.

[4] A.-L. Carter, "Safety-critical versus security-critical software," 2010.

[5] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat, and M. Khurram Khan, "Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions," *J. Netw. Comput. Appl.*, vol. 48, pp. 44–57, 2015.

[6] K. Fysarakis, G. Hatzivasilis, K. Rantos, A. Papanikolaou, and C. Manifavas, "Embedded Systems Security Challenges," in *Measurable security for Embedded Computing and Communication Systems (MeSeCCS 2014), within the International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2014)*, 2014, pp. 1–12.

[7] A. Shamir and N. Van Someren, "Playing 'hide and seek' with Stored Keys," in *Proceedings of the Third International Conference on Financial Cryptography*, 1999, pp. 118–124.

[8] S. L. Kinney, *Trusted Platform Module Basics: Using TPM in Embedded Systems (Embedded Technology)*. Newnes, 2006.

[9] J. Nagra and C. Collberg, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Pearson Education, 2009.

[10] J. P. McGregor and R. B. Lee, "Protecting cryptographic keys and computations via virtual secure coprocessing," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 1, pp. 16–26, 2005.

[11] J. Hu, Q. Wen, W. Tang, and A.-F. Sui, "A Key Hiding Based Software Encryption Protection Scheme," in *International Conference on Communication Technology (ICCT)*, 2011, pp. 719–722.

[12] I. Azhar, N. Ahmed, A. G. Abbasi, A. Kiani, and A. Shibli, "Keeping Secret Keys Secret in Open Systems," in *International Conference on Open Source Systems and Technologies (ICOSST)*, 2014, pp. 100–104.

[13] S. Sanjeev, J. Lodhia, R. Srinivasan, and P. Dasgupta, "Protecting cryptographic keys on client platforms using virtualization and raw disk image access," in *Proceedings - 2011 IEEE International Conference on Privacy, Security, Risk and Trust and IEEE International Conference on Social Computing, PASSAT/SocialCom 2011*, 2011, pp. 1026–1032.

[14] G. Bansod, A. Gupta, A. Ghosh, G. Bishnoi, C. Sawhney, and H. Ankit, "Experimental analysis and implementation of bit level permutation instructions for embedded security," *WSEAS Trans. Inf. Sci. Appl.*, vol. 10, no. 9, pp. 303–312, 2013.

[15] E. Elizabeth, S. Subasree, and S. Radha, "Enhanced Security Key Management Scheme for MANETS," *WSEAS Trans. Commun.*, vol. 13, pp. 15–25, 2014.

[16] R. Costa, D. Boccardo, C. Gomes, L. F. R. C. Carmo, and L. Pirmez, "Sensitive Information Protection for Advanced Metering Infrastructure," in *10th International Congress on Electrical Metrology (SEMETRO'13)*, 2013, pp. 6–9.

[17] G. J. Croll, "BiEntropy-The Approximate Entropy of a Finite Binary String," *eprint arXiv:1305.0954, Presented at ANPA 34, Rowland's*. pp. 1–16, 2013.

[18] G. E. Suh and S. Devadas, "Physical Unclonable Functions for Device Authentication and Secret Key Generation," in *Proceedings of the 44th Annual Design Automation Conference*, 2007, pp. 9–14.

[19] M. N. Gagnon, S. Taylor, and A. K. Ghosh, "Software Protection through Anti-Debugging," *Secur. Privacy, IEEE*, vol. 5, no. 3, pp. 82–84, 2007.

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, 2001, pp. 3–14.

[21] M. Ceccato, "On the Need for More Human Studies to Assess Software Protection," in *Workshop on Continuously Upgradeable Software Security and Protection*, 2014, pp. 55–56.

[22] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques," *Empir. Softw. Eng.*, vol. 19, pp. 1040–1074, 2014.