Automatika

Journal for Control, Measurement, Electronics, Computing and Communications

ISSN: 0005-1144 (Print) 1848-3380 (Online) Journal homepage: https://www.tandfonline.com/loi/taut20

Implementation of security module to protect programme theft in microcontroller-based applications

P. Muthu Subramanian & A. Rajeswari

To cite this article: P. Muthu Subramanian & A. Rajeswari (2019) Implementation of security module to protect programme theft in microcontroller-based applications, Automatika, 60:5, 526-534, DOI: 10.1080/00051144.2019.1578916

To link to this article: https://doi.org/10.1080/00051144.2019.1578916

AUTOMATIKA
2019, VOL. 60, NO. 5, 526–534
https://doi.org/10.1080/00051144.2019.1578916

Implementation of security module to protect programme theft in microcontroller-based applications

P. Muthu Subramanian and A. Rajeswari

Department of Electronics and Communication Engineering, Coimbatore Institute of Technology, Coimbatore, India

ABSTRACT

Source code plagiarism has become a serious threat for the development of small scale embedded industries and also the violations of intellectual property right are a threat for the development of hardware system. There are many software solutions for comparing source codes, but they are often not realistic in the present scenario. Digital watermarking scheme is one of the possible solutions for this problem. A novel watermarking technique is employed so that it can be easily and reliably detected by special techniques. In this paper, verification methods are presented to detect software plagiarism in the embedded application software without the implemented source code. All the approaches use side-channel information obtained during the execution of the suspicious code. The primary method is passive, i.e. no previous modification of the original code is required. It determines that the Hamming weights of the executed instructions of the suspicious device are used and uses string matching algorithms for comparisons with a reference implementation. The other method inserts additional code fragments as a watermark that can be identified in the power consumption of the executed source code. Proposed approaches are robust against code-transformation attacks.

ARTICLE HISTORY
Received 19 November 2018
Accepted 13 January 2019

KEYWORDS
Microcontrollers; security; embedded systems

1. Introduction

Software plagiarism and piracy is a serious problem which is estimated to cost the small scale industry billions of dollars per year. Software piracy for desktop computers is a serious problem that gained attention in the past. However, the companies working with embedded systems are also facing serious problems regarding software plagiarism and software piracy. If a designer suspects his code has been used in an embedded device, it is difficult to determine whether the code belongs to the user or the designer. The designer has to check the code in the suspected device and compare with the original code to detect the plagiarism. However, programme memory protection mechanism prevents unauthorized access to the programme memory used in today's microcontrollers. So the protection mechanism has to be defeated first to check the software plagiarism [1]. This makes testing of embedded devices from the perspective of software plagiarism very difficult, especially if it needs to be done in an automated way. The serious threat in this regard is that the IP cores with illegal copies can lead to huge monetary loss in. The negative impact of product piracy is a threat that needs to be taken seriously. It not only affects the manufacturers and consumers of the original equipment but also leads to damage in economy. Embedded Systems are the modern systems which play a vital role in the development of many consumer goods. Unless preventive technological protection is provided, sophisticated technologies permit attacks to be made on hardware and software in embedded systems. The attacks range from targeted modification to complete reverse engineering and product piracy. In the proposed system, the ways in which attacks can occur as well as the protection measures that are considered to fight product piracy through technological means are investigated. A new watermarking technique is proposed that employs side channels as the building block which can be easily and reliably detected by methods adapted from side-channel analysis [2]. The hardware measure is to embed a unique signal into a side-channel of the device that serves as a watermark. This makes the designers to check integrated circuits of their watermarked cores in the Integrated Chip. The watermark is hidden under the noise floor of the side-channel and is thus hidden from third parties. Furthermore, the proposed schemes are implemented with very few gates and are thus even harder to detect and remove. The proposed watermarks are realized in a programmable fashion to leak a digital signature. We proposed multiple levels of protection measure to this serious threat. The proposed technique requires source code rewriting and addition of hex codes in the language conversions. Real Time Software system uses a variety of methods to secure the code.

CONTACT P. Muthu Subramanian pmuthusubramanian1@gmail.com

Most of these methods require certain modifications in the source code and may not be efficient to have a comprehensive variety of attacks. Hardware detection is one of the efficient methods compared to the aforementioned software method. But still hardware methods also have disadvantages such as the designer needs to modify the hardware of the processor, which is usually not easy. Obfuscation techniques convert the programme from its original form into a new form which is not easily understood without changing the function of programme. Here, the objective is to confuse the attacker and this increase the difficulty of reverse engineering. The disadvantage of obfuscation techniques is the lack of theoretical foundation, so it cannot assess the effectiveness of measures quantitatively.

2. Related work

Programming plagrisim sign is presently related to the Hamming weight strings. The plan can be abridged in three stepping tools: First, the execution stream of the first programming is mapped to a string of Hamming weights. This is acknowledged either by measuring and assessing the power utilization of the execution as expressed above or by re-enacting the execution stream and figuring the Hamming weights from the opcodes. The branch guidelines require more than one clock cycle and thus, amid the execution more than one opcode is prefetched from the programme memory. The second step is to record a power hint of the suspicious gadget to get additionally a string of Hamming weights, meant as "t" . A coordinated duplicate of the string will be recognized in the third step by looking at the two strings s and t. A balanced duplicate is an admired supposition of programming copyright infringement and is subsequently simple to distinguish. The populace check of a bit string is frequently required in cryptography and other applications. A DPA is based on the algorithm and its power consumed by a processor. In order to detect the watermark, power traces for different values are to be verified. Combining various functions with the input values and the watermarked key, the user can verify and compute the correct assumption. Computation of the watermark and its energy trace is based on the clock cycle. Traces should be synchronized properly to avoid unwanted noise in the side-channel analysis [3].

Conversion of characters for encryption is incompatible. Encryption is measured with a block of 16 C/128 bits. Characters undergo transpositions and a substitution technique is used with a multidimensional array. One-time sub-key will be generated for the block which produces transitional result of the related length. The encryption iterations are done with the combination of a previous text block and a plain text with 8 C/64 bits which gives 192 bits and this will be the present block of text which produces a new 24-character text block. The same technique is applied till it gives a block of 256 bits and this bits are XORed with previous 256 bits other than the first block if the plain text is more than 32 bits [4]. The same 256 bits are XORed with last 256 bits other than the first block if there are more than 32 characters [5]. In both techniques if the attacker knows the internal state combinations, it is easy to reveal the watermark and the instructions used.

3. Security architecture

System architecture is proposed with two different levels of authentication. The first level specifies the changes in the assembly language code with watermarks and provides the robustness. The second level specifies the importance of the power consumption in the affected code to proof of ownership. Figure 1 shows the primary level of protection to detect software theft with minor changes to the assembly-level code. The side-channel software watermark consists of few instructions that are inserted in the assembly-level code. These instructions provide changes in machine cycle and power consumption of the processor and the pilfering can be detected by a trusted verifier with methods very similar to classic side-channel attacks.

Encrypted water-marked code in the processor will increase the usage of machine cycle when compared with the non- watermarked code. Machine cycle is based on the clock periods and frequency of the processor which will give a reasonable time analysis between the water-marked code and the normal code. Energy saving is not the goal of the proposed system, the increase of machine cycle will vary the energy consumption level and power parameters of the processor, but the authentication of the system will be maintained throughout the analysis.

Figure 2 shows the flow of security and these watermarks provide robustness at the higher level against code-transformation attacks while introducing only a

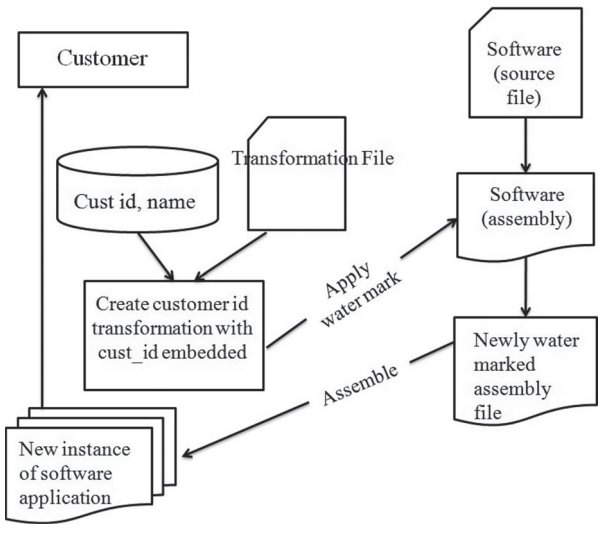

Figure 1. System architecture-watermarked code.

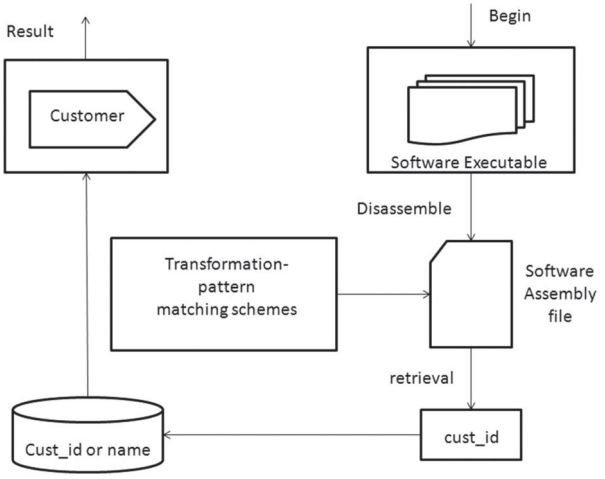

Figure 2. Flow of providing security-disassemble system.

small overhead in terms of performance and code size. The previous method can be used by a verifier to detect software plagiarism and to detect whether his code is present in an embedded system or not, but it is not possible to prove towards a third party that the code belongs to the verifier. The original software watermark only transmits bit of information-either the watermark is present or absent-but it does not provide any information about the owner of the watermark.

3.1. Proposed technique

In the proposed cipher technique the character of plain text is encrypted with the secret key using a multiplication technique. The secret key should not be either 0 or 1(when the key is 0, the letter becomes 0; when the key is 1, the letter of plain text remains unchanged and becomes 1). Here the decryption is done by inversing the secret key word and multiply with chiper text character to obtain the plain text character. The choice of odd numbers and even numbers is based on the traditional encryption and decryption approach for authentication and various selection of numbers for Encryption keys such as 3,5,7,9,11 are the best authenticated keys and inverse values of the encryption keys are 9,21,15,3,19. These keys give distinctive character by the given equation.

$$((\text{Character} * \text{Secret\_Key}) \mod 26). \quad (1)$$

In the next approach, Rail fence cipher technique [6] is used in which the plain text character is written downwards diagonally as per the allotted rail (rail can be 3 or 4) as shown in Figure 3 once it is reached as per the rail number limitation again the rail starts to move upwards diagonally here the message given is TOMORROW I WILL ATTACK and after the encryption with rail = 4 the encrypted message becomes TOLKORWLACMRIITAOWT.

```
T       O       L       K
 O   R   W   L   A   C
  M R     I I     T A
   O       W       T
```

Figure 3. Rail fence encryption.

The proposed algorithm implies both encryption and decryption in the context of using substitution and transportation ciphers and the algorithm states three different steps for the authentication. The proposed algorithm uses the series of substitution and transportation ciphers for the process of encryption and decryption. Algorithm states three different authentication steps (Figure 4).

Step 1: Substitution
Step 2: Transposition
Step 3: Substitution

To have a better understanding of the algorithm, plain text "ATTACK AT TEXT" is considered eliminating the spaces of the plain text "ATTACKATTEXT". Now this text goes through the encryption process with different levels of authentication. Authentication of the processor is the primary stream concentrating on the hexadecimal changes and analysing the power. Substitution and transposition is one of the best methods for analysing the hexadecimal values and its conversion, moving to any modern algorithm will provide the same objective of producing the hexadecimal conversion out of which power is measured for the changes.

3.2. Encryption

Step I: In this step, each plain text character is multiplied by the secret keyword depending on the position of the text.

Plain Text	A	T	T	A	C	K
Position	0	1	2	3	4	5
	Even	Odd	Even	Odd	Even	Odd
Plain Text	A	T	T	E	X	T
Position	6	7	8	9	10	11
	Even	Odd	Even	Odd	Even	Odd

The plain text character is multiplied by either 5 or 11 depending on its position (even or odd) i.e. an even character is multiplied by 5 and an odd character is multiplied with 11. Integral values of all the alphabets are treated as A assigned to 0, B assigned to 1 and Z is assigned to 25. The equ.2 is applied on every character of the plain text to obtain the intermediate Cipher text (C.T.): If position of character is even. In order to identify the intermediate cipher every character of the plain text is applied with the formula given below.

Position of the character is even

$$\text{Intermediate C.T.} = [[\text{plain text character integral value} * 5] + 1] \bmod 26 \quad (2)$$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	0	2	4	6	8	10	12	14	16	18	20	22	24	0	2	4	6	8	10	12	14	16	18	20	22	24
3	**0**	**3**	**6**	**9**	**12**	**15**	**18**	**21**	**24**	**1**	**4**	**7**	**10**	**13**	**16**	**19**	**22**	**25**	**2**	**5**	**8**	**11**	**14**	**17**	**20**	**23**
4	0	4	8	12	16	20	24	2	6	10	14	18	22	0	4	8	12	16	20	24	2	6	10	14	18	22
5	**0**	**5**	**10**	**15**	**20**	**25**	**4**	**9**	**14**	**19**	**24**	**3**	**8**	**13**	**18**	**23**	**2**	**7**	**12**	**17**	**22**	**1**	**6**	**11**	**16**	**21**
6	0	6	12	18	24	4	10	16	22	2	8	14	20	0	6	12	18	24	4	10	16	22	2	8	14	20
7	**0**	**7**	**14**	**21**	**2**	**9**	**16**	**23**	**4**	**11**	**18**	**25**	**6**	**13**	**20**	**1**	**8**	**15**	**22**	**3**	**10**	**17**	**24**	**5**	**12**	**19**
8	0	8	16	24	6	14	22	4	12	20	2	10	18	0	8	16	24	6	14	22	4	12	20	2	10	18
9	**0**	**9**	**18**	**1**	**10**	**19**	**2**	**11**	**20**	**3**	**12**	**21**	**4**	**13**	**22**	**5**	**14**	**23**	**6**	**15**	**24**	**7**	**16**	**25**	**8**	**17**
10	0	10	20	4	14	24	8	18	2	12	22	6	16	0	10	20	4	14	24	8	18	2	12	22	6	16
11	**0**	**11**	**22**	**7**	**18**	**3**	**14**	**25**	**10**	**21**	**6**	**17**	**2**	**13**	**24**	**9**	**20**	**5**	**16**	**1**	**12**	**23**	**8**	**19**	**4**	**15**
12	0	12	24	10	22	8	20	6	18	4	16	2	14	0	12	24	10	22	8	20	6	18	4	16	2	14
13	0	13	0	13	0	13	0	13	0	13	0	13	0	13	0	13	0	13	0	13	0	13	0	13	0	13
14	0	14	2	16	4	18	6	20	8	22	10	24	12	0	14	2	16	4	18	6	20	8	22	10	24	12
15	**0**	**15**	**4**	**19**	**8**	**23**	**12**	**1**	**16**	**5**	**20**	**9**	**24**	**13**	**2**	**17**	**6**	**21**	**10**	**25**	**14**	**3**	**18**	**7**	**22**	**11**
16	0	16	6	22	12	2	18	8	24	14	4	20	10	0	16	6	22	12	2	18	8	24	14	4	20	10
17	**0**	**17**	**8**	**25**	**16**	**7**	**24**	**15**	**6**	**23**	**14**	**5**	**22**	**13**	**4**	**21**	**12**	**3**	**20**	**11**	**2**	**19**	**10**	**1**	**18**	**9**
18	0	18	10	2	20	12	4	22	14	6	24	16	8	0	18	10	2	20	12	4	22	14	6	24	16	8
19	**0**	**19**	**12**	**5**	**24**	**17**	**10**	**3**	**22**	**15**	**8**	**1**	**20**	**13**	**6**	**25**	**18**	**11**	**4**	**23**	**16**	**9**	**2**	**21**	**14**	**7**
20	0	20	14	8	2	22	16	10	4	24	18	12	6	0	20	14	8	2	22	16	10	4	24	18	12	6
21	**0**	**21**	**16**	**11**	**6**	**1**	**22**	**17**	**12**	**7**	**2**	**23**	**18**	**13**	**8**	**3**	**24**	**19**	**14**	**9**	**4**	**25**	**20**	**15**	**10**	**5**
22	0	22	18	14	10	6	2	24	20	16	12	8	4	0	22	18	14	10	6	2	24	20	16	12	8	4
23	**0**	**23**	**20**	**17**	**14**	**11**	**8**	**5**	**2**	**25**	**22**	**19**	**16**	**13**	**10**	**7**	**4**	**1**	**24**	**21**	**18**	**15**	**12**	**9**	**6**	**3**
24	0	24	22	20	18	16	14	12	10	8	6	4	2	0	24	22	20	18	16	14	12	10	8	6	4	2
25	**0**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**

Figure 4. Substitution cipher encoding with different keys.

Position of the character is odd

$$\text{Intermediate C.T.} = [[\text{plain text character integral value} * 11] + 1] \bmod 26 \quad (3)$$

Applying the formula in the "ATTACK AT TEXT" given below

Plain text:	A	T	T	A	C	K
	Even	Odd	Even	Odd	Even	Odd
	0	19	19	0	2	10
	A	T	T	E	X	T
	Even	Odd	Even	Odd	Even	Odd
	0	19	19	4	23	19

Table 1 shows the intermediate encrypted cipher text BCSBLHBCSTMC after the formula encoding.

STEP II: In this step, transpose of cipher text is taken in order to make the encryption more authenticated.

Here the encrypted cipher text is processed with rail = 3 such that the character is written diagonally for 3 steps (rows) i.e. Figure 5. shows the transpose of BCSBLHBCSTMC.

Figure 5. shows the encrypted message BBBTCLCMSHSC which is the transpose of BCSBLHBCSTMC

STEP III: Final stage of the algorithm; here the character is substituted with special symbols which provides more authentication hence it becomes more secured. ASCII values are assigned to each and every character. Table 2 shows ASCII values and assigned alphabets. This substitution makes the text more authenticated and unreadable.

Table 1. Formation of intermediate encrypted cipher text.

Plain text	Formula	Resulting numbers	Intermediate Cipher text
A	((0*5) + 1)mod26	1	B
T	((19*11) + 1)mod26	2	C
T	((19*5) + 1)mod26	18	S
A	((0*11) + 1)mod26	1	B
C	((2*5) + 1)mod26	11	L
K	((10*11) + 1)mod26	7	H
A	((0*5) + 1)mod26	1	B
T	((19*11) + 1)mod26	2	C
T	((19*5) + 1)mod26	18	S
E	((4*11) + 1)mod26	19	T
X	((23*5) + 1)mod26	12	M
T	((19*11) + 1)mod26	2	C

Figure 5. Transpose encryption.

Hence the proposed algorithm encrypts a message and inserts those inside the assembly language programme, since these symbols cannot be added directly to the assembly-level language, the encrypted cipher text is converted into hexadecimal values which is shown in the third row of Table 2 and this is added as a watermark inside the assembly language code instead of the normal keys using Java script and the same is tested with TI MSP 432 processor. Based on the table the text BBBTCLCMSHSC is encrypted into " " " > #, # - = (= # and the same for better encryption is now converted into 22,22,22,3E,23,2C,23,2D,3D,28,3D,23

3.3. Decryption

Decryption is the inverse processes of encryption; here the course steps are performed in reverse order.

Step I: In this step, hexadecimal value is converted into symbols and then symbols to encrypted text and its character value are shown in Table.

Decrypted text of the encrypted hexadecimal value 22,22,22,3E,23,2C,23,2D,3D,28,3D,23 followed by the symbols " " " > #, # - = (= # is BBBTCLCMSHSC

Step II: In this step, the intermediate decrypted text obtained from Step I is rearranged into its original position of the characters i.e. intermediate decrypted message BBBTCLCMSHSC is rearranged into BCSBLHBCSTMC which is written in rows and read diagonally.

Step III: The final step of decryption here the intermediate encrypted text is converted into original plain text by applying the equation.

Position of the character is even

$$\text{Plain Text} = [[\text{encrypted character integral value} - 1] * 21] \bmod 26 \quad (4)$$

Here value 21 is the inverse of value 5

Position of the character is odd

$$\text{Plain Text} = [[\text{encrypted character integral value} - 1] * 19] \bmod 26 \quad (5)$$

Here value 19 is the inverse of value 11

Table 3. The original plain text obtained out of an encrypted message.

Intermediate Cipher text	Formula	Resulting numbers	Plain text
B	((1 − 1) * 21)mod26	0	A
C	((2 − 1) * 19)mod26	19	T
S	((18 − 1) * 21)mod26	19	T
B	((1 − 1) * 19)mod26	0	A
L	((11 − 1) * 21)mod26	11	C
H	((7 − 1) * 19)mod26	10	K
B	((1 − 1) * 21)mod26	0	A
C	((2 − 1) * 19)mod26	19	T
S	((18 − 1) * 21)mod26	19	T
T	((19 − 1) * 19)mod26	4	E
M	((12 − 1) * 21)mod26	23	X
C	((2 − 1) * 19)mod26	19	T

Table 3 shows the original plain text is obtained from the above cipher text BCSBLHBCSTMC.

3.4. Power analysis

The second layer of security is provided by analysing the power consumed by overall process and also by the each instruction level. Nowadays, software constitutes a major part of systems like embedded computing applications drives where power is a constraint, and also has a significant contribution to the overall power consumption. In order to analyse systematically and assess this impact, it is important to start at the most practical and fundamental level –the instruction level. The instruction-level power models are derived based on the power supply current measurement technique. Each instruction takes a specific number of machine cycles to complete its operation. The number of machine cycles varies from one instruction to another and the time taken for each instruction set for execution is calculated and by using that overall time elapsed is found. With the help of time elapsed and operating frequency, power consumption of the processor with encryption is measured.

4. Experimental results

4.1. Plagiarism detection

Figure 6 shows the first level of verification of software plagiarism using the watermark added to the assembly-level code. The initial step of this verification is that the normal C code has been written in Turbo C++ and the C code has been converted into assembly-level code using GCC compiler and the steps to add the watermark

Table 2. Conversion of intermediate cipher text to symbols and hexadecimal values.

Alphabet	A	B	C	D	E	F	G	H	I	J	K	L	M
Symbol	!	"	#	$	%	&	'	(	)	*	+	,	−
HD	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
Alphabet	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Symbol	.	/	:	;	<	=	>	?	@	[	\	]	^
HD	2E	2F	3A	3B	3C	3D	3E	3F	40	5B	5C	5D	5E

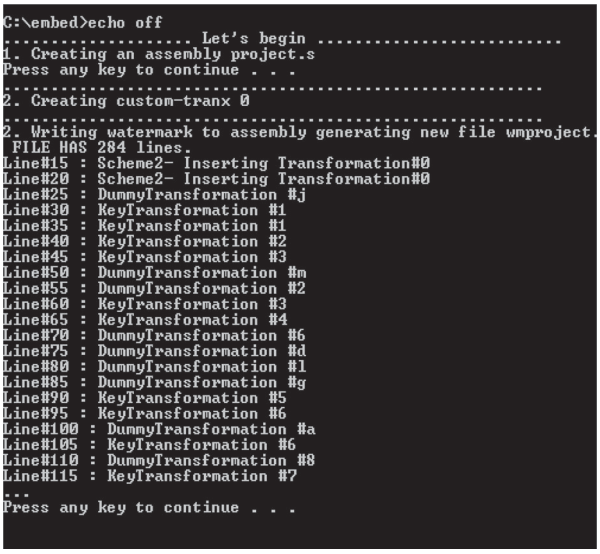

Figure 6. Key information.

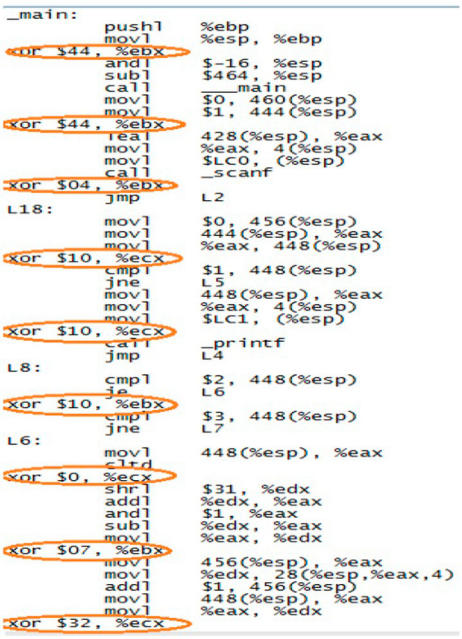

Figure 7. Keys in the assembly-level code.

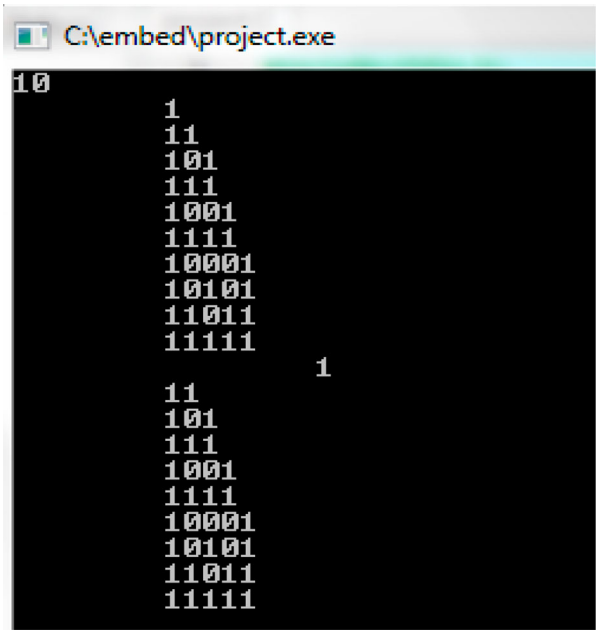

Figure 8. Water mark-added code.

code into the assembly-level code have been implemented using JavaScript. The watermark is inserted inside the code at any random place by the compiler and the watermark keyword can be chosen by the user. Same configurations are compiled and tested with TI MSP boards. MSP432 is the processor with code composer studio and inbuilt energy trace analysis, power consumed by the code is analysed for further extraction using the software tool. Power analysis is not restricted to a TI-based processor. Power can be measured in all the processors using hardware techniques.

Dummy key information can also be added into the assembly-level code in order to confuse the code hacker and the exact key information is known by the owner. The dummy keys are inserted in the assembly-level code at random places in between the exact key words Figure 7. The watermark code does not affect the output or function of the embedded system. The key functions are included in the separate registers which will not be included in the main programme. The output of the suspicious code can be compared with the original code using the string matching algorithms and the plagiarism can be detected.

Figure 8 shows the substitution keys which are added inside the code. The code is written in JavaScript language for the substitution algorithm. In the proposed system, plaintext "ENCRYPTION" is converted into cipher text "% - # } | ‘] (∼ -" using Substitution Cipher Algorithm. Since these symbols cannot be added directly to the assembly-level language, the cipher text is then converted into hexadecimal values using JavaScript language which can be added into the assembly-level language as the watermark instead of the normal keys which were implemented in the first stage as shown in Figure 9.

The Hexadecimal values for the word ENCRYPTION are 25 2D 23 7D 60 5D 28 2E 2D. The addition of encrypted hexadecimal values as watermark inside the assembly-level code does not affect the output of the programme as shown in Figure 10.

Figure 11 shows the encrypted hex value which is obtained from the watermarked executable file using the string matching algorithm which is designed in Java.

4.2. Decryption

The next step is the decryption of the hexadecimal values which are inserted inside the assembly-level code. Decryption is the reverse process of the encryption.

```
xor $25, %ebx
    call   ___main
    movl   $0, 460(%esp)
    movl   $1, 444(%esp)
    leal   428(%esp), %eax
    movl   %eax, 4(%esp)
    movl   $LC0, (%esp)
    call   _scanf
xor $25, %ebx
    jmp    L2
L18:
    movl   $0, 456(%esp)
    movl   444(%esp), %eax
    movl   %eax, 448(%esp)
    cmpl   $1, 448(%esp)
    jne    L5
xor $2D, %ecx
    movl   448(%esp), %eax
    movl   %eax, 4(%esp)
    movl   $LC1, (%esp)
    call   _printf
    jmp    L4
L8:
xor $2D, %ecx
    cmpl   $2, 448(%esp)
    je     L6
    cmpl   $3, 448(%esp)
    jne    L7
```

Figure 9. Hexadecimal values added as watermark.

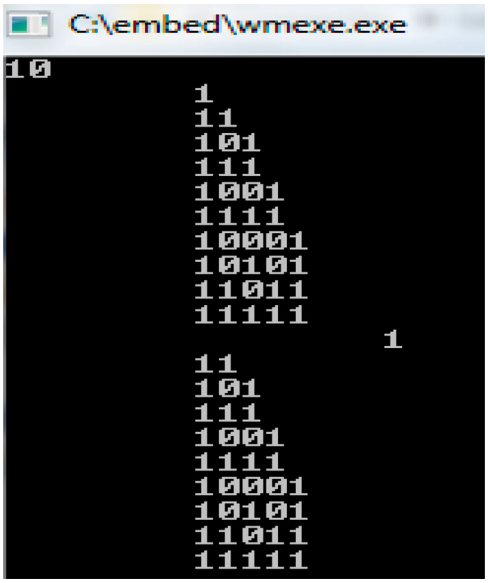

Figure 10. Encrypted watermark-added code.

First, the obtained output hexadecimal values are converted into the cipher text format as defined by the user. Then the cipher text is converted into the normal plain text using the script code written separately for the decryption process as shown in Figure 12. This is more robust for the code hacker to detect the watermark.

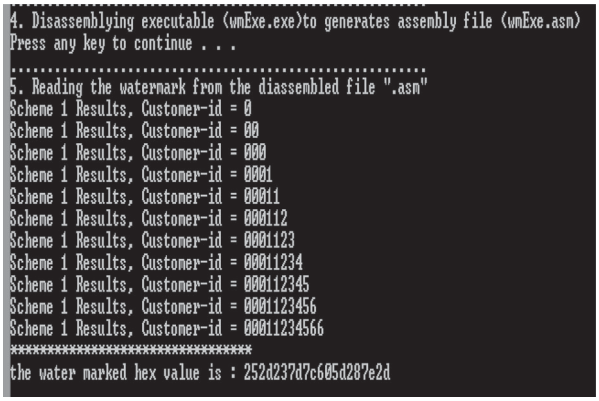

Figure 11. Output of the hex obtained from executable file.

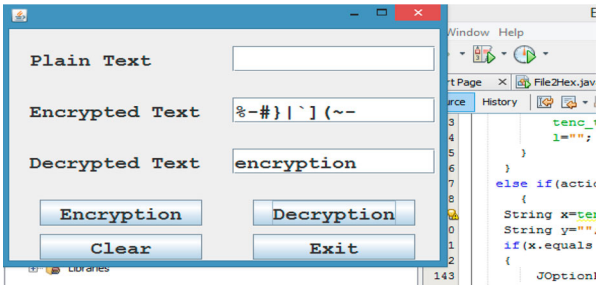

Figure 12. Decryption of cipher text.

4.3. Energy comparison

The difference between the original code and the pirated code is found by comparing the energy trace chart for the encrypted and unencrypted codes. Figure 13 shows the energy consumption and power consumption of the normal code and the pirated code. The addition of watermark inside the code will lead only to a small change in the power consumption. So, the power varies in the range of mV based on the defined clock frequency of the processor and its time

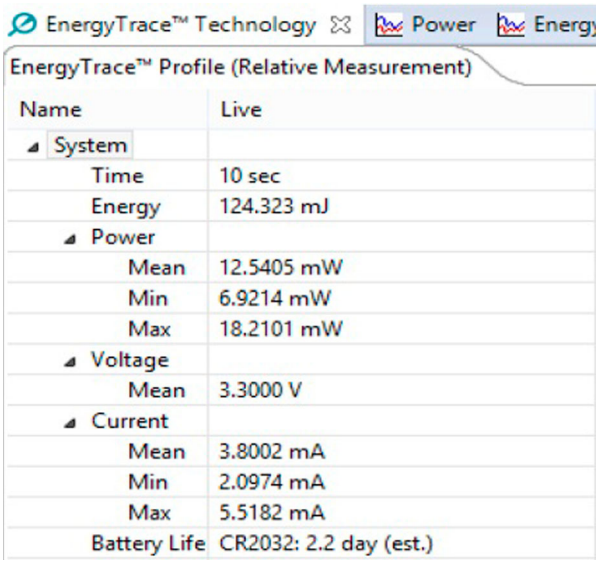

Figure 13. Energy trace chart of the normal code.

EnergyTrace™ Profile (Relative Measurement)

Name	Live
System	
Time	10 sec
Energy	127.334 mJ
Power	
Mean	12.8973 mW
Min	6.7634 mW
Max	20.7062 mW
Voltage	
Mean	3.3000 V
Current	
Mean	3.9083 mA
Min	2.0495 mA
Max	6.2746 mA
Battery Life	CR2032: 2.2 day (est.)

Figure 14. Energy trace for the watermark-added code.

in nano seconds with instruction machine cycles. The energy trace is done for the code in which the watermark is not added. The obtained power for the code is 12.5405 mW and the energy consumed is 124.323 mJ.

Figure 14 shows that for the watermark-added code, the power consumed is 12.8973 mW and the energy consumption is 127.334 mJ. From the comparison of the above energy trace charts, it is possible to find the difference between the pirated code and the watermark-added code. Here the energy difference and the power difference is the key factor for proof.

Figure 15 and Figure 16 show the power comparison between the water mark-added code and the unencrypted code using power vs time analysis graph in Code Composer Studio energy trace analysis.

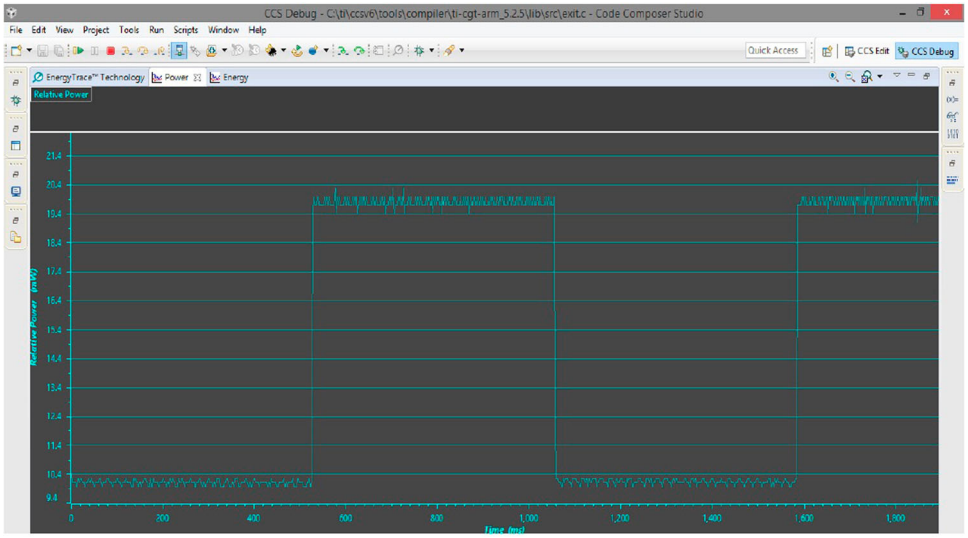

Figure 15. Energy trace for original code-expanded.

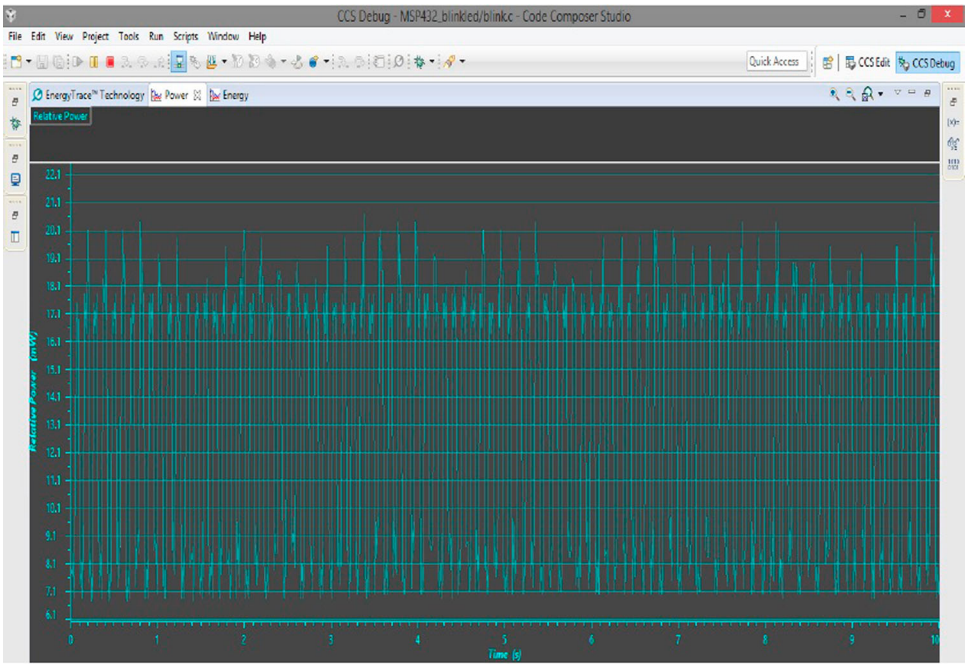

Figure 16. Energy trace for altered code-compressed.

Table 4. Power analysis comparison of the watermarked code and the original code.

Code level	Original code 1	Water mark code 1	Water mark code 2
		SYSTEM	
Time	10 sec	10 sec	10 sec
Energy	61.871 mJ	62.117 mJ	66.415 mJ
		POWER	
Mean	6.1864 mW	6.2113 mW	6.6827 mW
Min	6.0888 mW	6.0872 mW	5.9057 mW
Max	6.3687 mW	6.9478 mW	7.7105 mW
		VOLTAGE	
Mean	3.300 V	3.300 V	3.300 V
		CURRENT	
Mean	1.8747 mA	1.8822 mA	2.0250 mA
Min	1.8451 mA	1.8446 mA	1.7896 mA
Max	1.9299 mA	2.1054 mA	2.3365 mA

Power analysis comparison between the original code and the water-marked code is tested with Code Composer Studio in TI MSP boards and the result concludes that when there is a change in the code with encryption standard, the original code is not affected and there is a power variation. Table 4 shows the detailed analysis of the code which is inside the processor and this gives an authentication proof to the developer.

5. Conclusion

Software theft is reduced using the proposed watermark technique. An approach has been proposed in which substitution and transposition cipher techniques are determined. Conversion of special symbols makes the analysis challenging. Substitution of cipher encoding with different keys is done with upper case characters. There is a possibility that the compiler can filter out the watermark on compilation. Thus watermark survives this optimization using substitution cipher algorithm with hex file conversions and power analysis in our technique. The obscurity in the resulting files made the job of reverse engineering more challenging. In future, substitution cipher encoding for different keys can be extended for lower case alphabets and numeric values. The usage of digital signature proves the legitimate ownership of the watermark.

Disclosure statement

No potential conflict of interest was reported by the authors.

ORCID

P. Muthu Subramanian http://orcid.org/0000-0003-0885-6623

References

[1] Al-Wosabi AAA, Shukur Z. Software tampering detection in embedded systems. J Theor Appl Inf Technol. June 2015;76(2):211–221.
[2] Agrawal D, Archambeault B, Rao J, et al. The EM side-channel(s): attacks and methodologies. In Proceedings Workshop on Cryptographic Hardware and Embedded Systems 2002, Aug. 2002.
[3] Becker GT, Burleson W, Paar C. Side-channel watermarks for embedded software. Proceedings IEEE 9th Int. New Circuits and Systems Conf. (NEWCAS), pp. 478–481 Jun. 2011.
[4] Pal JK, Mandal JK, Gupta S. Composite transposition substitution chaining based cipher technique. Proceedings of 16th International Conference on Advanced Computing and Communications, December 2008.
[5] Pal JK, Mandal JK. A novel block cipher technique using binary field arithmetic based substitution (BCTB-FABS). Second International conference on Computing, Communication and Networking Technologies, 2010.
[6] https://www.geeksforgeeks.org/rail-fence-cipher-encryption-decryption/