ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2012, Vol.41, No.4

Generation of the Secret Encryption Key Using the Signature of the Embedded System

Algimantas Venčkauskas

Computer Department, Kaunas University of Technology,
Studentų str. 50-212, LT-51368, Kaunas, Lithuania
e-mail: algimantas.venckauskas@ktu.lt

Nerijus Jusas

Computer Department, Kaunas University of Technology,
Studentų str. 50-210, LT-51368, Kaunas, Lithuania
e-mail: nerijus.jusas@ktu.lt

Irena Mikuckienė

System Analysis Department, Kaunas University of Technology,
Studentų str. 50-211, LT-51368, Kaunas, Lithuania
e-mail: irena.mikuckiene@ktu.lt

Stasys Maciulevičius

Computer Department, Kaunas University of Technology,
Studentų str. 50-209, LT-51368, Kaunas, Lithuania
e-mail: stasys.maciulevicius@ktu.lt

crossref http://dx.doi.org/10.5755/j01.itc.41.4.1162

Abstract. Program protection, programming code integrity and intellectual property protection are important problems in embedded systems. Security mechanisms for embedded systems have some specific restrictions related to limited resources, bandwidth requirements and security. In this paper we develop a secret encryption key generation algorithm by using the signature of the embedded system. We explore the qualitative characteristic of the generated keys - the entropy. Experiments showed that the generated secret keys have high entropy.

Keywords: embedded system, program protection, secret encryption key, hash function, entropy.

1. Introduction

It is hard to imagine most of today's appliances and devices, electronics, telecommunications, mechatronics, etc., without using the embedded systems. These systems face significant challenges in information security; on the one hand, they usually have very limited resources and, on the other hand, they function in a physically unsafe environment. The embedded systems usually perform critical functions: controlling important real time objects and processing important information. Therefore, their work is open to sabotage.

Security requirements for the embedded systems depend on specific areas of application [16, 26]. The following requirements are related to the general requirements for information security: integrity, availability and confidentiality. However, the specificity of the embedded systems, their mobility and operation in real time, typically have certain limitations such as processing gap, energy gap, flexibility, tamper resistance, assurance gap and cost. This is largely due to limited resources, performance and security requirements.

An important component of the embedded system that also influences its performance and vitality is software. Software security has two aspects: secure program and program protection [19]. We will explore the protection aspect of program security. The main program protection vulnerabilities are the following

[23]: violation of intellectual property (illegal copying and distribution, improper use of licenses, reverse engineering), disclosure of software code, theft of algorithms and falsification of software code.

According to the study by the Business Software Alliance (BSA) [4], the software creators lost USD 51.4 billion and the pirated software accounted for 43 % of all software with piracy growing by around 2 percent annually.

No matter what threats software is protected from, for example copying or the theft of algorithms, the attackers use a wide range of means to crack the protection: reverse engineering, including disassembly and decompilation, debuggers, disassemblers, decompilers, emulators, simulators and spoofing attacks [18].

There are many software protection methods, which are divided into software and hardware-based. Software-based protection mechanisms are integrated into the software or the algorithm, which is protected and can be added to the software code: code and data obfuscation [6], anti-debugging method [7], code encryption technology, self-modifying code and self-extracting code [13]. Hardware-based methods can significantly increase the level of protection, largely due to the fact that they are external devices in which the level of protection is controlled by the software provider and not by the end-user [12, 17, 20]. A part of the program code or data (encryption keys) required to run the program can be stored in the additional hardware (commonly Dongle or USB keys). However, this protection mechanism is relatively expensive and is generally only used for the programs which are of great commercial value.

Intermediate software/hardware methods are also used: tethering the program to a computer or device signatures (CPU, RAM, ROM, BIOS, OS, etc., serial numbers, model ID, etc.) [21, 25, 31]. Firewalls are also used for the protection of the internet programs [14]. These methods are usually used for anti-piracy in personal computers.

Gelbart et al. [8] proposed a joint compiler/hardware infrastructure for protection of the embedded system software for fully encrypted execution in which both program and data are in the memory in the encrypted form. The processor is supplemented with the Field Programmable Gate Array (FPGA)-based secure hardware component. Arora et al. [1] presented architecture for hardware-assisted run-time monitoring, wherein the embedded processor is augmented with the hardware monitor that observes the dynamic execution trace of the processor and checks whether the execution trace falls within the allowed program behaviour.

In assessing the limitations of the embedded systems [2], one of the most acceptable software protection methods is code encryption. However, key management faces additional issues: it requires an additional storage medium, the encryption keys have to be entered manually and key transfer via the network must be protected by using SSL protocol and others [27, 28].

Secret encryption keys are used for various purposes in the embedded systems, such as communication, data encryption, etc. Physical characteristics of the embedded system, such as physical unclonable functions (PUF) are used for the generation of keys [24, 29].

Our goal is to create a protection method of the embedded system software that does not require external hardware and infrastructure for key generation, storage and management and provides a sufficient level of security. The code of the embedded system software is stored in an encrypted form; secret encryption keys are generated in real time, on demand, before the execution of the encrypted software module.

In the following sections, we describe the proposed method of secret key generation by using the signature of the embedded system and investigate its characteristics and possibilities of its application for the protection of the embedded system software.

2. The method for generation of the secret key

The secret encryption key of the software module is generated from the headers of the program to be protected and from the signatures of the embedded system hardware and software components (CPU, RAM, ROM, BIOS, OS, etc.), using the fastest and simplest logical operations (XOR, OR, SHIFT). For convenience of description, terminology and notations used in the paper are summarized as follows:

- $K = \{k_i\}$: program encrypting key.
- $P = \{p_i\}$: header of the program to be protected. The program header is an array of structures, each describing a segment or other information the system needs to prepare the program for execution.
- $PSN = \{psn_i\}$: a serial number of the program to be protected.
- $PH = \{ph_i\}$: a hash of header of the program to be protected.
- $SS = \{ss_i\}$: signature of the embedded system.
- $ES = \{es_i\}$: signature of the embedded system components.
- $CV = \{cv_i\}$: component vendor identifiers (ID).
- $CT = \{ct_i\}$: component type ID.
- $CM = \{cm_i\}$: component model ID.
- $CSN = \{csn_i\}$: component serial number.
- $\|$: string concatenation operation.
- $\vee$: the bitwise OR operation.
- $\oplus$: the bitwise exclusive OR (XOR) operation.
- *mod n*: modulo *n* operation.
- *h(.)*: a cryptographic one-way hashing function (MD5, SHA-1…).

– *eb(s, k)*: function for the extraction of *k* bytes from the string *s*.
– *sign(.)*: function for creating the signature of the embedded system (defined below).
– *key(.)*: function for generation of the secret encryption key (defined below).

The process of generation of the secret key consists of five steps (Fig. 1).

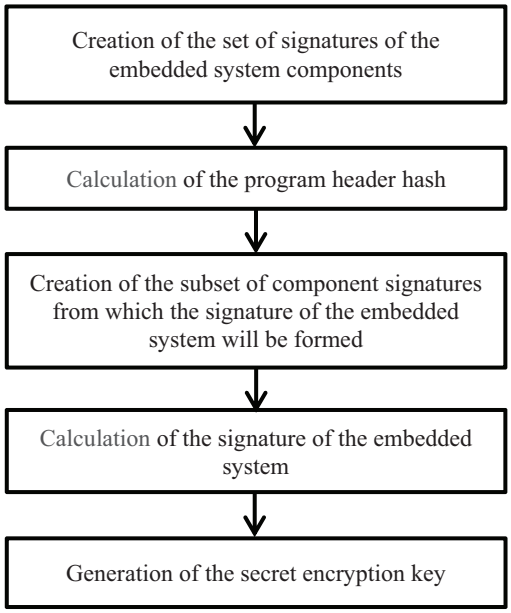

Figure 1. The process of secret key generation

Further, the method of generating a secret key by using the signature of the embedded system is described in detail.

1. Create the set of signatures of the components of the embedded system $ES = \{es_i\}, i = 1, \dots, n$. The signature is created by applying the string concatenation of *Vendor ID* (cv_i), *Type ID* (ct_i), *Model ID* (cm_i), and *Serial Number* (csn_i):

$$es_i = cv_i \,\|\, ct_i \,\|\, cm_i \,\|\, csn_i.$$

In steps 2 – 6 a subset of the component signatures is created. These signatures will be used for computing of the embedded system signature.

2. Calculate the program header hash $ph = h(p \,\|\, psn)$.
3. Create the $n \times m$ matrix $MH = \{mh_{ij}\}$ from the bytes of the program header hash $mh_{ij} = eb(ph, (i-1) \times j + i)$, where n is the number of the embedded system signatures, and $m = eb(ph, n) \bmod n$.
4. Calculate the sum s_j of the column elements in the matrix *MH*, $s_j = \sum_{i=1}^{n} mh_{ij}, j = 1, \dots, m$.
5. Create the index array of the component signatures $IND = \{ind_j\}$, where $ind_j = s_j \bmod n$ and delete repetitive indices, $ind_j \neq ind_i, \forall\, i \in \{1 \dots j-1\}$.
6. Create the subset of the component signatures $\widetilde{ES} \subseteq ES, \widetilde{es_i} = es_j$, where $j = ind_k, \forall\, ind_k \in IND,\ k = 1, \dots, m$, from which the embedded system signature will be created.
7. Create the signature of the embedded system $ss_i = sign(\widetilde{ES})$.
8. Generate the program protection key $= key(ss, salt, iteration\_count, key\_length)$, where salt, iteration_count and key_length are defined below.

Now, we describe the set of *sign* functions. The signature of the embedded system is created by processing byte strings of the component signatures. All *sign* functions can be performed using bit operations (bitwise OR, bitwise XOR, bitwise AND, SHIFT).

Function *sign1*. Bitwise XOR is used to create a signature of the embedded system (Fig. 2).

```
input: ess, m // subset of m component
                  // signatures
output: ss // signature of the embedded system
l := maxlength ess // max of the length of the
                  // component signatures
for j = 1 to l do
  ss (j) := ess (1, j)
end for
for i = 2 to m do
  for j = 1 to l do
    ss (j) := ss (j) XOR ess (i, j)
  end for
end for
```

Figure 2. Function ***sign1***

Function *sign2*. Bitwise OR is used to create a signature of the embedded system (Fig. 3).

```
input: ess, m // subset of m component
                  // signatures
output: ss // signature of the embedded system
l := maxlength ess // max of the length of the
                  // component signatures
for j = 1 to l do
  ss (j) := ess (1, j)
end for
for i = 2 to m do
  for j = 1 to l do
    ss (j) := ss (j) OR ess (i, j)
  end for
end for
```

Figure 3. Function ***sign2***

Function *sign3*. Bitwise OR and XOR are used to create a signature of the embedded system (every other component signature are processed by XOR and OR operation, starting with XOR, see Fig. 4).

```
input: ess, m // subset of m component
                // signatures
output: ss // signature of the embedded system
l := maxlength ess // max of the length of the
                // component signatures
for j = 1 to l do
  ss (j) := ess (1, j)
end for
for i = 2 to m-1 step 2 do
  for j = 1 to l do
    ss (j) := ss (j) XOR ess (i, j)
  end for
  for j = 1 to l do
    ss (j) := ss (j) OR ess (i+1, j)
  end for
end for
```

Figure 4. Function ***sign3***

Function *sign4*. Bitwise OR and XOR are used to create a signature of the embedded system (every other byte of the component signature are processed by XOR or OR operation, starting with XOR, see Fig. 5).

```
input: ess, m // subset of m component
                // signatures
output: ss // signature of the embedded system
l := maxlength ess // max of the length of the
                // component signatures
for j = 1 to l do
  ss (j) := ess (1, j)
end for
for i = 2 to m do
  for j = 1 to l-1 step 2 do
    ss (j) := ss (j) XOR ess (i, j)
    ss (j+1) := ss (j+1) OR ess (i, j+1)
  end for
end for
```

Figure 5. Function ***sign4***

The length of the signatures created using *sign* functions is equal to the maximum length of the component signature, $l = maxlength(es_i), es_i \in \widetilde{ES}$, $i = 1, \dots, m$.

The strings of signatures of the embedded system are of a variable length. The encryption key should contain strings of a fixed length. The length of these strings depends on the requirements of the encryption key length. The encryption key must have a high value of entropy. Hash functions such as MD5 and SHA are used to create secret keys of a fixed length from the strings of variable length [9]. However, the keys generated only with a hash function are not strong enough against brute force attacks [11, 32]. *Key Derivation Functions* [10] generate strong keys, where extra data (*random salt* and *an iteration count)* are also used alongside with byte strings. The main problem of these functions is how to define these additional data: *salt* and *iteration count* [15]. The proposed method uses the number of columns of the matrix *MH* as a *salt*, and the number of component signatures in the subset $(\widetilde{ES})$ as *iteration count*:

$$salt = eb(ph, n) \bmod n,$$

$$iteration\_count = count(\widetilde{ES}).$$

So, the function of generation of the program protection key is defined as follows:

$$k = \text{key}\ (ss, salt, iteration\_count, key\_length).$$

In the following section, we investigate the characteristics of the method for generation of the secret key by using the signature of the embedded system.

3. Evaluation of the proposed method

The encryption keys have to be generated truly *randomly,* to contain sufficient *entropy* and be of sufficient *length* [30]. Since our problem is the protection of the embedded system software, the key generation process must be carried out without any additional hardware and infrastructure cost [2, 26]. The proposed method generates the secret keys only from the signature of the embedded system. Further, we investigate whether the entropy of the key is high enough (close to 1). The maximum value of entropy of a bit string is 1 [30, 5, 3].

The concept of entropy is defined by Shannon. Let us consider an information source described as a sequence of instances of a random (discrete) variable X, which can take a finite number of possible values $x_1, x_2, \dots, x_n$ with a probability respectively equal to $p_1, p_2, \dots, p_n$ (in other words $p(x_i) = p_i$). Then the source entropy is defined by:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \cdot log\big(p(x_i)\big).$$

In a practical way, source entropy is computed from the observed frequency for every character by means of the previous formula.

So, the entropy S of the string is defined as follows:

$$S = -\sum_{i=1}^{n} p(x_i) \cdot log_2\big(p(x_i)\big),$$

$$p(x_i) = num(x_i) \,/\, m,$$

where n is the number of possible values of character, m is the length of the string, and $num(x_i)$ is the number of appearances of character x_i in the string.

The initial data of the experiment (the header of the program to be protected, the signatures of the embedded system hardware and software components (*Vendor ID, Type ID, Model ID* and *Serial Number*), their lengths and number) are generated by using the random string and number generators. 20 sets of signatures (from 2 to 7 elements) are generated. While the entropy changes marginally when the number of elements exceeds 7, we evaluated the signatures of the system consisting of 2 to 7 elements.

The entropy of these sets of signatures is presented in Tables 1-4 (the number of component signatures, the length of the system signature and the entropy).

Table 1. The entropy of signatures, *sign1* function

	n=2		*n=3*		*n=4*		*n=5*		*n=6*		*n=7*	
Test N.	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy
1	120	1.000	120	0.954	120	0.954	120	0.954	256	1.000	256	0.544
2	120	0.811	120	0.954	120	0.544	120	0.954	120	1.000	256	0.954
3	256	0.954	256	0.811	256	0.954	256	0.954	256	0.811	256	1.000
4	64	0.811	64	0.954	64	0.954	64	0.954	64	0.544	120	0.811
5	256	0.954	256	0.954	256	0.954	256	0.954	256	0.954	256	1.000
6	64	0.954	64	0.954	64	0.544	64	0.544	64	0.544	64	1.000
7	64	0.811	64	0.954	64	0.544	64	0.954	64	0.811	64	0.954
8	120	0.811	120	0.954	256	0.954	256	0.954	256	1.000	256	0.811
9	64	0.954	64	1.000	64	1.000	64	1.000	64	0.811	64	1.000
10	120	0.954	120	1.000	120	0.811	120	1.000	120	1.000	256	0.954
11	64	0.954	256	0.954	256	0.954	256	0.954	256	0.954	256	0.954
12	256	0.954	256	0.954	256	0.954	256	0.954	256	0.811	256	0.954
13	256	1.000	256	0.954	256	0.544	256	0.954	256	0.811	256	0.954
14	256	1.000	256	0.954	256	0.954	256	0.954	256	0.811	256	0.811
15	256	1.000	256	1.000	256	1.000	256	1.000	256	0.811	256	0.954
16	256	0.954	256	0.811	256	0.811	256	1.000	256	0.811	256	0.954
17	64	0.811	120	0.954	120	0.954	120	0.954	120	0.000	120	0.954
18	56	0.544	256	0.954	256	0.954	256	0.954	256	0.954	256	0.954
19	256	1.000	256	0.954	256	0.954	256	0.954	256	0.811	256	0.954
20	256	0.954	256	1.000	256	0.811	256	0.811	256	1.000	256	0.544

Table 2. The entropy of signatures, *sign2* function

	n=2		*n=3*		*n=4*		*n=5*		*n=6*		*n=7*	
Test N.	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy
1	120	0.811	120	0.811	120	0.811	120	0.811	256	0.811	256	0.811
2	120	0.811	120	0.811	120	0.811	120	0.811	120	0.811	256	0.811
3	256	0.954	256	1.000	256	0.954	256	0.954	256	0.954	256	0.954
4	64	1.000	64	1.000	64	1.000	64	1.000	64	0.954	120	1.000
5	256	0.954	256	0.811	256	0.811	256	0.811	256	0.544	256	0.544
6	64	0.954	64	0.954	64	0.954	64	0.954	64	0.811	64	0.954
7	64	0.954	64	0.954	64	0.954	64	0.954	64	0.811	64	0.811
8	120	0.811	120	0.811	256	0.811	256	0.811	256	0.811	256	0.811
9	64	0.811	64	0.811	64	0.811	64	0.811	64	0.811	64	0.811
10	120	0.811	120	0.954	120	0.954	120	0.954	120	0.954	256	0.954
11	64	0.954	256	0.811	256	0.811	256	0.811	256	0.811	256	0.811
12	256	0.954	256	0.954	256	0.811	256	0.811	256	0.811	256	0.811
13	256	0.954	256	0.954	256	0.954	256	0.811	256	0.544	256	0.544
14	256	0.811	256	0.954	256	0.954	256	0.954	256	0.954	256	0.954
15	256	0.811	256	0.811	256	0.811	256	0.811	256	0.811	256	0.544
16	256	0.954	256	0.811	256	0.811	256	0.811	256	0.811	256	0.811
17	64	0.954	120	0.811	120	0.811	120	0.811	120	0.544	120	0.811
18	56	0.954	256	0.954	256	0.811	256	0.811	256	0.811	256	0.811
19	256	0.811	256	0.811	256	0.811	256	0.544	256	0.544	256	0.544
20	256	0.954	256	1.000	256	1.000	256	1.000	256	1.000	256	0.954

Table 3. The entropy of signatures, *sign3* function

	n=2		*n=3*		*n=3*		*n=5*		*n=6*		*n=7*	
Test N.	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy
1	120	1.000	120	0.811	120	1.000	120	0.811	256	0.954	256	1.000
2	120	0.811	120	1.000	120	0.811	120	1.000	120	0.954	256	0.811
3	256	0.954	256	0.954	256	0.811	256	1.000	256	0.954	256	0.954
4	64	0.811	64	1.000	64	0.811	64	1.000	64	0.811	120	0.954
5	256	0.954	256	0.811	256	1.000	256	0.811	256	0.811	256	0.544
6	64	0.954	64	1.000	64	0.811	64	0.954	64	0.954	64	0.954
7	64	0.811	64	1.000	64	0.811	64	1.000	64	0.954	64	0.954
8	120	0.811	120	0.811	256	0.811	256	0.954	256	0.811	256	1.000
9	64	0.954	64	0.954	64	0.954	64	0.954	64	0.954	64	0.954
10	120	0.954	120	0.954	120	0.954	120	1.000	120	1.000	256	0.954
11	64	0.954	256	0.811	256	1.000	256	0.811	256	1.000	256	0.954
12	256	0.954	256	0.954	256	0.954	256	0.954	256	0.811	256	1.000
13	256	1.000	256	0.954	256	0.544	256	0.954	256	0.811	256	0.954
14	256	1.000	256	1.000	256	0.811	256	1.000	256	0.544	256	1.000
15	256	1.000	256	0.811	256	1.000	256	0.811	256	0.811	256	0.544
16	256	0.954	256	0.811	256	0.811	256	1.000	256	0.811	256	0.811
17	64	0.811	120	0.811	120	0.811	120	1.000	120	0.544	120	0.954
18	56	0.544	256	0.954	256	0.954	256	0.954	256	0.954	256	0.811
19	256	1.000	256	0.811	256	1.000	256	0.544	256	1.000	256	0.954
20	256	0.954	256	1.000	256	0.811	256	0.954	256	0.954	256	0.954

Table 4. The entropy of signatures, *sign4* function

	n=2		*n=3*		*n=4*		*n=5*		*n=6*		*n=7*	
Test N.	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy	Bits N.	Entropy
1	120	1.000	120	0.954	120	0.811	120	0.954	256	0.954	256	0.954
2	120	1.000	120	1.000	120	0.954	120	1.000	120	1.000	256	0.811
3	256	1.000	256	0.954	256	0.954	256	1.000	256	1.000	256	1.000
4	64	0.954	64	1.000	64	0.954	64	1.000	64	0.811	120	0.811
5	256	1.000	256	0.811	256	0.954	256	0.811	256	0.811	256	0.811
6	64	0.954	64	0.954	64	0.811	64	0.954	64	0.954	64	0.954
7	64	1.000	64	0.954	64	0.954	64	1.000	64	1.000	64	0.954
8	120	0.954	120	0.954	256	1.000	256	0.954	256	0.954	256	0.954
9	64	1.000	64	0.954	64	0.811	64	0.954	64	0.954	64	0.954
10	120	0.811	120	0.954	120	0.954	120	1.000	120	1.000	256	1.000
11	64	1.000	256	0.811	256	0.954	256	0.811	256	1.000	256	1.000
12	256	1.000	256	1.000	256	0.954	256	0.811	256	1.000	256	1.000
13	256	1.000	256	0.954	256	0.811	256	1.000	256	1.000	256	0.811
14	256	1.000	256	0.954	256	1.000	256	0.954	256	0.954	256	0.954
15	256	1.000	256	1.000	256	0.954	256	1.000	256	0.811	256	0.811
16	256	1.000	256	0.811	256	0.954	256	1.000	256	0.811	256	1.000
17	64	1.000	120	0.811	120	0.954	120	1.000	120	0.811	120	1.000
18	56	0.954	256	0.954	256	1.000	256	0.954	256	0.954	256	0.954
19	256	1.000	256	0.954	256	0.811	256	0.954	256	0.544	256	0.954
20	256	0.954	256	1.000	256	0.811	256	0.954	256	0.954	256	0.954

For evaluation of the entropy dependence on the used function and on the number of component signatures, we calculate the estimates of entropy: average, standard deviation and prediction interval [22]. The summarized results of the entropy of the signature dependence on the used function are presented in Table 5. As one can see in Table 5, the best quality of signature of the embedded system is obtained by using the function *sign4*: the maximal value of entropy is 1.000, the least standard deviation is 0.078 and the lowest limit of prediction interval is 0.861. The quality of signatures generated by function *sign3* is also relatively high: standard deviation is 0,117 and the lowest limit of the prediction interval is

0.781. Thus, the function of signature generation has to be constructed by using operations XOR and OR.

Table 5. Dependence of the signature entropy on the function

Function	Average	Standard deviation	Prediction interval	
			min	Max
sign1	0.894	0.150	0.744	1.000
sign2	0.852	0.115	0.737	0.967
sign3	0.898	0.117	0.781	1.000
sign4	0.939	0.078	0.861	1.000

The secret encryption keys are generated from the embedded system signature by using *Key Derivation Functions*. These functions use hash functions, such as MD5, SHA and SHA-2 etc. Further, we investigate the influence of the hash function algorithm on the value of entropy. Since the signature of the embedded system generated by using *sign4* function has the best entropy, we investigate the key generated by this function. The entropy of keys formatted from 2 to 7 component signatures by using *sign4* function and MD5, SHA and SHA-2 hash functions is presented in Table 6.

Table 6. The entropy of keys generated by using *sign4* function and MD5, SHA and SHA-2 hash functions

	n=2			*n=3*			*n=4*			*n=5*			*n=6*			*n=7*		
Test N.	MD5	SHA	SHA2	MD5	SHA	SHA2	MD5	SHA	SHA2	MD5	SHA	SHA2	MD5	SHA	SHA2	MD5	SHA	SHA2
1	1.000	1.000	1.000	1.000	1.000	1.000	0.978	0.997	0.997	0.999	0.991	0.998	0.989	0.998	0.999	1.000	0.991	1.000
2	0.991	0.999	0.995	0.961	0.995	0.996	0.993	0.996	0.999	0.997	0.989	0.999	0.989	0.993	0.998	0.997	0.991	0.996
3	0.993	1.000	0.975	0.990	0.997	0.998	0.999	1.000	1.000	0.993	0.996	0.999	0.999	1.000	1.000	1.000	0.997	1.000
4	0.989	0.993	0.998	0.987	0.996	0.993	0.987	0.990	1.000	0.990	1.000	0.996	1.000	1.000	0.999	0.996	0.998	0.999
5	0.999	0.999	0.997	1.000	1.000	0.997	0.998	0.996	0.990	0.985	1.000	0.999	0.989	0.997	0.997	0.982	1.000	1.000
6	1.000	0.996	1.000	0.965	0.998	1.000	0.998	0.989	1.000	0.996	0.974	0.999	1.000	0.994	0.998	1.000	1.000	0.999
7	0.998	0.998	0.999	0.998	0.996	1.000	0.986	1.000	0.996	0.972	0.999	0.994	0.993	0.990	0.999	1.000	0.971	0.998
8	0.998	0.996	1.000	0.999	0.997	0.999	1.000	0.999	0.999	0.992	0.991	0.997	0.999	0.999	1.000	1.000	0.999	1.000
9	0.989	0.991	0.998	0.998	0.993	0.997	0.998	1.000	0.999	0.992	1.000	0.998	0.989	0.996	1.000	1.000	0.999	0.999
10	0.999	0.999	0.998	0.986	0.991	0.999	1.000	1.000	0.997	0.994	1.000	1.000	0.986	0.998	0.999	0.986	0.998	0.999
11	1.000	0.994	0.999	0.996	0.998	1.000	0.974	0.996	0.999	1.000	0.999	1.000	0.996	0.997	1.000	0.998	0.996	0.997
12	0.999	0.999	0.997	0.986	0.989	0.999	0.992	1.000	0.990	1.000	0.995	1.000	1.000	1.000	0.993	0.996	1.000	0.999
13	1.000	1.000	1.000	0.997	0.987	1.000	0.999	0.995	0.998	0.986	0.994	0.999	1.000	0.998	0.998	0.999	0.991	0.999
14	1.000	1.000	1.000	0.998	1.000	0.994	0.994	0.993	0.996	1.000	1.000	0.999	0.987	1.000	1.000	1.000	1.000	0.998
15	0.978	1.000	1.000	1.000	1.000	0.999	1.000	0.998	1.000	0.996	0.999	0.995	0.999	1.000	1.000	0.998	0.995	1.000
16	0.999	0.999	0.997	0.994	1.000	0.995	0.998	0.996	0.999	0.989	0.991	0.996	1.000	0.999	0.999	1.000	0.999	0.999
17	0.998	0.998	0.999	0.954	0.994	0.993	0.997	0.976	1.000	0.999	0.999	1.000	1.000	0.999	0.994	0.998	0.997	0.978
18	0.997	0.991	1.000	0.999	1.000	0.998	0.967	1.000	1.000	0.976	0.999	0.999	0.995	1.000	0.999	0.997	0.999	0.999
19	0.978	1.000	1.000	0.982	0.993	1.000	0.999	0.999	1.000	0.997	0.986	0.997	1.000	0.997	0.997	0.986	1.000	0.993
20	1.000	0.998	0.998	1.000	0.996	0.999	1.000	0.997	0.999	0.996	0.999	1.000	1.000	1.000	0.997	0.985	0.991	0.996

Entropy estimates – average, standard deviation and prediction interval, dependence on the used hash function are presented in Table 7.

Table 7. Dependence of the entropy of keys on the used hash function

Function	Average	Standard deviation	Prediction interval	
MD5	0.994	0.008	0.985	1.000
SHA	0.995	0.007	0.988	1.000
SHA-2	0.998	0.003	0.994	1.000

All hash functions generate high-entropy cryptographic keys. However, the keys, generated by using function SHA-2, have the least standard deviation (0.003) and the highest of the lowest limit of the prediction interval (0.994).

To summarize, it can be stated that the entropy of the secret encryption key generated by using the signature creation functions based on OR and XOR operations and SHA-2 hash function is highest.

4. Conclusions

In this paper, we present the method for generation of the secret encryption key by using signature of the embedded system and evaluate the entropy of keys and the efficiency of hash functions.

The proposed method effectively generates high-entropy keys (entropy value close to 1) without any additional hardware and infrastructure cost, which is vital for the embedded systems with limited resources.

The entropy of the secret encryption key generated by using the signature creation functions based on OR and XOR operations and SHA-2 hash function is highest.

In future, we are going to develop a prototype tool of the software protection of the embedded system using the proposed method for generation of the secret

encryption key and investigate the possibilities of its use.

References

[1] **D. Arora, S. Ravi, A. Raghunathan, N. J. Jha.** Secure Embedded Processing through Hardware-assisted Run-time Monitoring. In: *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'05), IEEE Computer Society*, 2005, 178–183.

[2] **S. Babar, A. Stango, N. Prasad, J. Sen, R. Prasad.** Proposed embedded security framework for Internet of Things (IoT). Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronics Systems Technology (Wireless VITAE), 2nd International Conference, 2011, 1–5.

[3] **J. B. Bedrune, F. Eric, R. Frederic.** Cryptography: all-out attacks or how to attack cryptography without intensive cryptanalysis. *Journal in Computer Virology*, 2010, *Vol.*6, *Issue* 3, 207–237. Available at: http://dx.doi.org/10.1007/s11416-008-0117-x.

[4] BSA. Seventh Annual BSA and IDC Global Software Piracy Study. May 2010.

[5] **E. Carrera.** Scanning data for entropy anomalies. 2007. http://blog.dkbza.org/2007/05/scanning-data-for-entropy-anomalies.html.

[6] **C. Collberg, C. Thomborson, D. Low.** A taxonomy of obfuscating transformations. *Technical Report 148, Department of Computer Sciences, the University of Auckland*, July 1997.

[7] **M. N. Gagnon, S. Taylor, A. K. Ghosh.** Software Protection through Anti-Debugging. *IEEE Security and Privacy,* May 2007, Vol.5(3), 82–84. Available at: http://dx.doi.org/10.1109/MSP.2007.71.

[8] **O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, J. Zambreno.** CODESSEAL: Compiler/FPGA Approach to Secure Applications. In: *Proceedings of IEEE International Conference on Intelligence and Security Informatics*, 2005, 530–536.

[9] **Ch. Henke, C. Schmoll, T. Zseby.** Empirical evaluation of hash functions for multipoint measurements. *SIGCOMM Comput. Commun.*, July 2008, Rev.38(3), 39–50.

[10] International Organization for Standardization. ISO/IEC FCD 18033–2, IT Security techniques—Encryption Algorithms—Part 2: Asymmetric Ciphers, 2004.

[11] **A. Joux, T. Peyrin.** Hash functions and the (amplified) boomerang attack. Proceedings of the 27th annual international cryptology conference on Advances in cryptology (CRYPTO'07), 2007, 244–263.

[12] **I. J. Jozwiak, A. Liber, K. Marczak.** A Hardware-Based Software Protection Systems-Analysis of Security Dongles With Memory. Proceedings of the International Multi-Conference on Computing in the Global Information Technology (ICCGI'07), IEEE Computer Society, 2007, 28–38.

[13] **Y. Kanzaki, A. Monden, M. Nakamura, K. Matsumoto.** Exploiting self-modification mechanism for program protection. *Proceedings of the 27th Annual International Computer Software and Applications Conference, IEEE Computer Society,* 2003, 170-179.

[14] **E. Kazanavicius, R. Paskevicius, A. Venckauskas, V. Kazanavicius.** Securing Web Application by Embedded Firewall. *Electronics and Electrical Engineering*, 2012, No.3(119), 65–68.

[15] **A. D. Kent, L. M. Liebrock.** Secure Communication via Shared Knowledge and a Salted Hash in Ad-Hoc Environments. *Computer Software and Applications Conference Workshops (COMPSACW), IEEE 35th Annual*, July 2011, 122–127.

[16] **P. Kocher, R. Lee, G. McGraw, A. Raghunathan.** Security as a new dimension in embedded system design. *Proceedings of the 41st annual Design Automation Conference (DAC '04), ACM,* 2004, 753–760.

[17] **A. Liutkevicius, A. Vrubliauskas, E. Kazanavicius.** Assessment of Dongle-based Software Copy Protection Combined with Additional Protection Methods. *Electronics and Electrical Engineering,* 2011, No.6 (112), 111–116.

[18] **A. Main, P. C . Oorschot.** Software protection and application security: Understanding the battleground. *International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography*, June 2003.

[19] **G. McGraw.** Software security. *IEEE Security & Privacy, Vol.* 2 (2), 2004, 80–83. Available at: http://dx.doi.org/10.1109/MSECP.2004.1281254.

[20] **L. MeiHong, L. JiQiang.** USB Key-Based Approach for Software Protection. *International Conference on Industrial Mechatronics and Automation,* 2010, 151-153.

[21] **S. Mumtaz, S. Iqbal, I. Hameed.** Development of a Methodology for Piracy Protection of Software Installations. *9th International Multitopic Conference, IEEE INMIC 2005, 24-25 Dec.*. 2005, 1–7.

[22] **W. Navidi.** Statistics for engineers and scientists. *McGraw-Hill, New York*, 2011.

[23] NIST. National Vulnerability Database Version 2.2. http://nvd.nist.gov/home.cfm.

[24] **E. Papoutsis, G. Howells, K. McDonald-Maier, A. Hopkins.** Key Generation for Secure Inter-satellite Communication. *Adaptive Hardware and Systems, AHS 2007, Second NASA/ESA Conference*, 2007, 671–681.

[25] PC GUARD. Professional software protection and licensing system. http://www.sofpro.com.

[26] **S. Ravi, A. Raghunathan, S. Hattangady, P. Kocher.** Security in embedded systems: Design challenges. *ACM Trans. Embed. Comput. Syst. 3, 3,* August 2004, 461–491.

[27] **E. Sakalauskas, A Katvickis, G Dosinas.** Key Agreement Protocol over the Ring of Multivariate Polynomials. *Information Technology and Control*, 2010, *Vol.* 39(1), 51–54.

[28] **Ting-Yi Chang, Min-Shiang Hwang, Wei-Pang Yang**. An Improved Multi-stage Secret Sharing Scheme Based on the Factorization Problem. *Information Technology and Control*, 2011, *Vol.*40(3), 246–251.

[29] **G. E. Suh, S. Devadas.** Physical unclonable functions for device authentication and secret key generation. *Proceedings of the 44th annual Design Automation Conference*, June 2007, 9–14. Available at: http://dx.doi.org/10.1145/1278480.1278484.

[30] **H. C. Tilborg** (Ed). Encyclopedia of Cryptography and Security. *Springer*, 2005. Available at: http://dx.doi.org/10.1007/0-387-23483-7.

[31] **A. Venckauskas, N. Jusas, L. Kizauskienė, E. Kazanavicius, V. Kazanavicius.** Security method of embedded software for mechatronic systems. *Mechanika,* 2012, *Vol*.18(2), 196–202. Available at: http://dx.doi.org/10.5755/j01.mech.18.2.1572.

[32] **X. Y. Wang, H. B Yu.** How to break MD5 and other hash functions. *Advances in Cryptology, EUROCRYPT 2005, Lecture Notes in Computer Scince*, 2005, Vol. 3494, pp. 19–35. Available at: http://dx.doi.org/10.1007/11426639_2.

Received February 2012.