Detecting Compromised Programs for Embedded System Applications

Xiaojun Zhai[1], Kofi Appiah[1], Shoaib Ehsan[1], Wah M Cheung[1], Gareth Howells[2], Huosheng Hu[1], Dongbing Gu[1], and Klaus McDonald-Maier[1]

[1]School of Computer Science & Electronic Engineering
University of Essex, Colchester, UK
{xzhai, kappiah, sehsan, wmcheu, hhu, dgu, kdm}@essex.ac.uk

[2]School of Engineering and Digital Arts
University of Kent
Canterbury, UK
W.G.J.Howells@kent.ac.uk

Abstract. This paper proposes an approach for detecting compromised programs by analysing suitable features from an embedded system. Features used in this paper are the performance variance and actual program counter values of the embedded processor extracted during program execution. "Cycles per Instruction" is used as pre-processing block before the features are classified using a Self-Organizing Map. Experimental results demonstrate the validity of the proposed approach on detecting some common changes such as deletion, insertion and substitution of programs. Overall, correct detection rate for our system is above 90.9% for tested programs.

Keywords: ICmetrics, Self-Organising Map (SOM), embedded system security.

1 Introduction

As embedded systems involve various aspects of our everyday lives, they are often needed to process sensitive information or perform critical functions, which make security an important concern in embedded computer architecture design [1]. The rapid growth of embedded systems has transformed the way we create, destroy, share, process and manage information. However, this has also paved the way for unauthorised access, fraud and other related crimes [2]. Security has been extensively explored in the context of general purpose computing and communications systems, such as cryptographic algorithms and security protocols [3]. Such security measures typically provide a basis for securing embedded system rather than enabling a system's overall security. On the other hand, as embedded systems are often specific to a certain function, the resources and cost are very limited by the strict performance and power con-

straints. Consequently, it is a challenge to increase overall dependability, integrity and robust security of embedded systems [4].

Identification and security of these embedded systems are emerging as an important concern in embedded computer architecture design. Mechanisms to protect the embedded system can be either included in the hardware architecture or at software level. Physical Unclonable Function (PUF) [5] or hardware intrinsic security [6], have been proposed as physically more secure alternative to storing secrets in a digital memory [7]. The core idea behind these approaches is to use the manufacturing process variation to identify the integrated circuits, which offers a higher level of security against physical level attacks. However, they are limited by environmental variance such as changes in temperature, user interactions and software. There is much existing work focusing on detecting software failure, tampering and malicious codes in embedded systems [1, 4, 8]. These approaches require storing sensitive data in the system as "valid" samples or template. For example, a basic-block control flow graph (CFG) is usually stored and used to examine the running program.

Currently, researchers are working on alternative solutions to the above problems in the fields of digital forensics and machine learning [9]. As electronic devices and components cannot have exactly the same frequency response and latency due to tolerances in production and the different designs employed by various manufacturers, it is possible to find unique features or identifiers from the electronic devices [9]. In order to recognise the features, various machine learning algorithms can also be applied. Based on the above ideas, a new concept termed ICmetrics (Integrated Circuit metrics) was introduced [10]. Embedded systems typically consist of hardware and application specific software, and are applied in a specific environment. These could result in the embedded system performing uniquely to the others. Consequently, the structure, characteristic and behaviour of an embedded system can also be used to identify the devices. Fig.1 exhibits a typical embedded system and ICmetrics system.

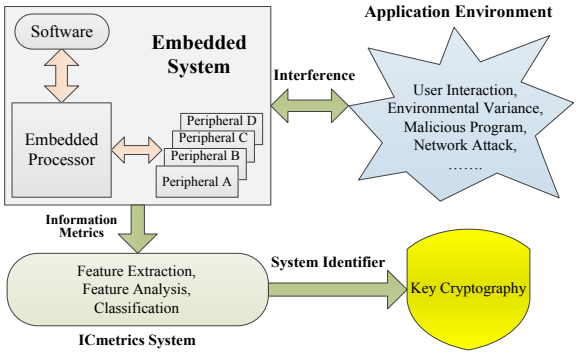

Fig. 1. A typical embedded system and ICmetrics system.

In Fig. 1, the embedded system can be affected by many factors, for example, compromised software, unauthorised access, environment changing, internal structure failure and malicious code. All these effects could change the behaviour or characteristic of the embedded system significantly. Since the ICmetrics system is continually

monitoring the information metric, and behaviours derived from the embedded system change over time as well, a different system identifier could be generated. As a result of this, a different encryption key will be generated by the key cryptography mechanism [11], using a two phase approach to deal with training and recall. As the ICmetrics system only relies on the properties and features of the system, the system identifier (i.e. basic number or encryption key) can be regenerated on demand and there is no requirement to store it locally. The major advantage of the ICmetrics system is no user data or template is required to be stored, which is essential for applications that have no direct interaction with human operators. Thus, the ICmetrics can improve both security and dependability based on exploitation of the system's unique behaviour.

The information metrics used in the ICmetrics system can be collected from any aspect of the embedded system, for example, memory usage, program monitoring, processor caches, and register status checking. In this paper, we limit the focus on monitoring the system processor's status while running various programs. A method for detecting compromised programs is proposed. The method extracts suitable features from the embedded system (i.e. the performance variance and program counter (PC) register of the embedded processor), enabling it to identify the running programs using Self-Organising Map (SOM) classifier [12]. The experimental results demonstrate the effectiveness of the proposed method for identifying compromised programs. The performance variance and PC status can be one of the information metrics for the ICmetrics system.

The remainder of this paper is organised as follows. A survey of related work is presented in Section II. The proposed algorithm is introduced in Section III. The experimental setup and the implementation results are discussed in Section IV. Finally, the conclusions are presented in Section V.

2 Related Work

As most information is being digitized to facilitate quick access, digital privacy is becoming even more important in protecting personal information [13]. Arora et al [1] addressed secure program execution by focusing on the specific problem of ensuring that the program does not deviate from its intended behaviour. Similar to [1], Rahmatian et al [4] used a CFG to detect intrusion for secured embedded systems by detecting behavioural differences between the correct system and malware. An attack is detected if the system call sequence deviates from the known sequence. Yang et al [14] presents a very interesting approach for detecting digital audio forgeries mainly in MP3. Using a passive approach, they are able to detect doctored MP3 audio by checking frame offsets.

Information hiding can be used in authentication, copyright management as well as digital forensics [15]. Swaminathan et al [15] proposed an enhanced computer system performance with information hiding in the compiled program binaries. The system wide performance is improved by providing additional information to the processor without changing the instruction set architecture. In [16] Boufounos and

Rana demonstrate with the use of signal processing and machine learning techniques, to securely determine whether two signals are similar to each other. They also show how to utilize an embedding scheme for privacy-preserving nearest neighbour search by presenting protocols for clustering and authenticating applications.

ICmetrics can be defined as a unique characteristic that a program possesses when running on a particular embedded device and can be used to identify the program and hardware. In this paper we use Cycle per Instruction (CPI) to extract corresponding PC values, and use it as ICmetric for program identification. Using an unsupervised SOM to reduce the dimensionality of PC values, we introduce an offset rule similar to that presented in [14] to detect compromised programs rather than detecting digital audio forgeries. Thus using machine learning techniques [16], we are able to determine whether two PC values are similar to each other, with the use of the program binaries [15] and no prior knowledge of the source code. The following section describes our system to detect compromised programmes in details.

3 Methods for Detecting Compromised Programs

In this section, we first provide an overview of the proposed methods for detecting compromised programs, and then details of the proposed method are introduced.

3.1 Overall System Architecture

In computer systems, a program normally consists of three structure levels: (1) function call level, as represented by function call relationship; (2) internal control flow for each function, represented by a basic-block CFG; (3) instruction stream within each CFG [1]. A program is comprised of a number of micro operations, which depend on the instruction sets and the exact processor architecture that are used in the embedded system. The number of clock cycles for each instruction depends on the used hardware architecture and type of instruction, for example, most of instructions only require one clock cycle to be executed in modern pipelined processer architecture, but some instructions require multi-cycle to be executed, as they need access to memory during processing (e.g. Load, Store and Jump). In particularly, these multi-cycle instructions indicate where the functions call or the condition branch is [4]. Consequently, we can approximately detect the function call or condition branch based on the variance of the processor's performance. In addition, the value of PC register shows the instruction stream of a program, which is also a suitable source for monitoring changes at the instruction level.

Based on the above principle, through monitoring the processor's performance, we detect changes in the function call and CFG, and then analyse the PC values within each CFG. Finally, an overall evaluation could indicate whether the program is compromised or not. In the proposed work, we measure the average CPI as the parameter of a processor's performance. Fig. 2 shows a block diagram of the proposed program monitoring system.

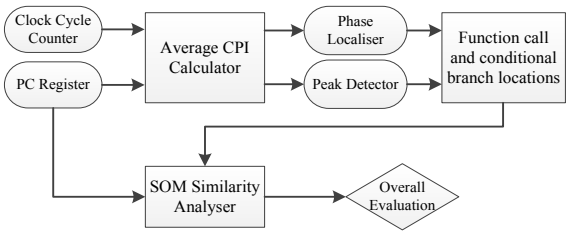

Fig. 2. Overall block diagram of the proposed monitoring system.

In Fig. 2, phase localiser and peak point detector blocks are used to obtain the function call and conditional branch location information from average CPI profile respectively, and then the obtained information will be used to extract features for the SOM classifier. The final evaluation is based on the results of the SOM classifier.

3.2 CPI Analysis

CPI indicates the complexity of instructions executed within a particular period of time. The average CPI of a processor can be calculated as described in [17]. Fig. 3 shows an average CPI profile while a program is running in an ARM cortex-M3 processor based embedded platform, where I and $f_{\max}$ are 2^{11} and 120 MHz respectively.

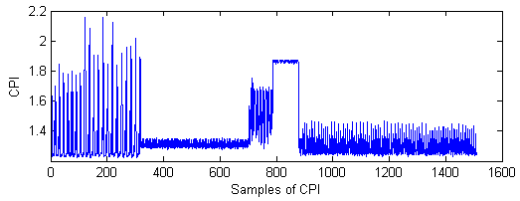

Fig. 3. Example of average CPI diagram.

As can be seen from Fig. 3, the program mainly consists of five phases, and there are also many variances (i.e. peaks) within each phase. In the following sections, we introduce a method to obtain the position information of the phases and peaks.

Phase localiser block

In the phase localiser block, there are mainly two sub-blocks: mean filter and critical point localiser. The mean filter is first used to smooth the original CPI diagram, the critical point localiser is then used to localise the positions of each phase.

Mean filter

A $1\times w$ rectangular window is used as a mask in the mean filter, the local average value within the mask is then calculated. Let $f(n)$ denote the CPI value at position n which is always the centre point of a rectangular window B with size $1\times w$. The window mean value $f_{mean}(n)$ is calculated by (1):

$$f_{mean}(n) = \sum_{n\in B} f(n)/w \tag{1}$$

Fig. 4 shows the resulting diagram after applying the mean filter on the original CPI diagram (i.e. Fig. 3), where w is set to '5'. As can be seen from Fig. 4, the variances within each phase have been significantly suppressed, and the boundaries of each phase still stay intact.

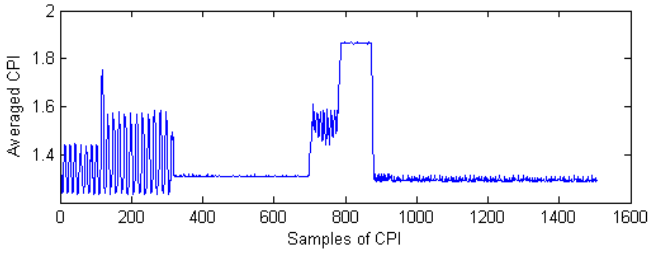

Fig. 4. Resulting CPI diagram after applying the mean filter.

Critical point localiser

As the values of two adjacent points at the boundary are normally significantly different, the proposed method is to localise the high variance points, and then select the best candidates based on pre-defined criterion.

Let f_{mean} denote averaged CPI, absolute differences between adjacent elements of f_{mean} can then be calculated by:

$$d(n) = \left| f_{mean}(n+1) - f_{mean}(n) \right| \tag{2}$$

where $1 \le n < N$, N is the total numbers of elements in array f_{mean}, $d(n)$ is n^{th} element in an array of absolute differences between adjacent elements of $f_{mean}(n)$.

A threshold t_1 is first used to select the high variance elements from array d, where the indices of the elements are greater than t_1 they are stored in array d_1. After that, absolute differences between adjacent elements of d_1 are calculated to form d_2. Finally, a threshold t_2 is used to select the boundary candidates, where elements greater than t_2 are selected as the candidates. Values of t_1 and t_2 are fixed based on experimental results. In this work, t_1 and t_2 are set to 0.03 and 9 respectively. Fig. 5 shows resulting diagram after applying the critical point localiser on Fig.4.

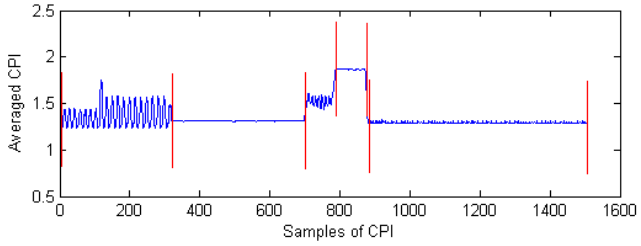

Fig. 5. Resulting diagram after applying the critical point localiser.

Peak detector block

In order to obtain positions of peaks and valleys, we apply the peak detector on array d rather than the original array f_{mean}. Pseudo-codes for detecting the peaks are summarised as follows:

```
Peak detection procedure:
Input: d_i is an array of absolute differences between ad-
jacent elements of f_mean in the i^th phase.
Output: P = {p_1, p_2, p_3, ..., p_i} where p_1 is a set of locations
for the i^th phase.
for all samples in d_i do
    if d_i(n − 1) < d_i(n) and d_i(n) > d_i(n + 1) then
      d_i'(j) = d_i(n); /* record the amplitude in array d_i'(j)*/
    end
end
t_i=mean( d_i'(j));  /*t is mean of all the elements in d_i'*/
for all samples in d_i' do
    if d_i'(j) > t then
      p_i = j;  /*mark j^th element as a peak*/
    end
end
```

Fig. 6 shows resulting diagram after applying the peak detector on array *d*.

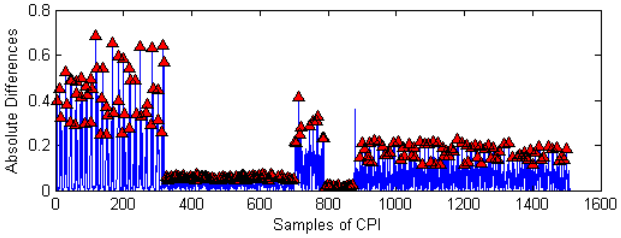

Fig. 6. Diagram following the application of the peak detector.

Similarity analyser

The similarity analyser has three different parts, each with a measure to ascertain the originality of the program in execution. The three parts are the phase, peak and SOM analysers. The first part is used to verify if the number of known phases is the same as the number of phases in the executed program. Any mismatch shows that the number of function calls differ, signifying an insertion or deletion. The second part compares the number of identified peaks within each phase. It must be noted that any difference in the number of peaks does not necessarily mean the program is compromised, but rather a variation in CPI. The first two parts of the analyser becomes useful when the system has completed a cycle. The final part of the analyser uses the SOM to measure similarity between known programs and programs currently executed.

The basic principle of the SOM is to adjust the weight vectors until the neurons represent the input data, while using a topological neighbourhood update rule to ensure that similar prototypes occupy nearby positions on the topological map. PC values extracted from the program execution trace, corresponding to the peaks in the trace are used as inputs to the SOM during training and testing. For a given network

with k neurons and N-dimensional input vector $\mathbf{K^i}$, the distance from the j^{th} neuron with weight vector $\boldsymbol{w_j}$ ($j<k$) is given by

$$D_j^2 = \sum_{l=1}^{N} \left(K_l^i - w_{jl} \right)^2 \tag{3}$$

where $\boldsymbol{w_{jl}}$ is the l^{th} component of weight vector $\boldsymbol{w_j}$. The vector components of the winning neuron $\boldsymbol{w_k}$ with minimum distance D_k are updated as follows, where $\eta \in (0,1)$ is the learning rate.

$$\Delta w_k = \eta \left(K^i - w_k \right) \tag{4}$$

Updates are only carried out during the training phase. Additionally, for every neuron in the network we maintain two extra parameters; the minimum and maximum distances of all input vectors associated with any particular neuron.
After training, the next step is to associate each of the network neurons with the corresponding program or sub-program. In this work, we use Vector Quantization (VQ) [12] to assign labels to the trained neurons in the network as follows:

- Assign labels to all the input training data. The label is an identifier for the program from which the training data has been extracted from.
- Find the neuron in the network with the minimum distance to the labelled input data.
- For each input data maintain the application label, the corresponding neuron and the distance measured. The distance is maintained as a tie breaker for applications that share similar address space.

For each network neuron, we estimate the number of programs that are associated with that neuron. If only one program is associated with a neuron and the number of data points exceeds 5% of the total number of program data points, the neuron is exclusively assigned to that very program. For all programs with more than 5% of data points associated with a neuron, we create a codebook with an entry for the neuron, and the corresponding programs, each with its distance range (i.e. minimum distance and maximum distance).

4 Experimental Results

An embedded system based on a STMicroelectronics STM32F207IG microcontroller equipped with an ARM 32-bit Cortex-M3 processor is used in the proposed work [18]. A combination of KEIL µVision IDE, and ULINKpro Debug and Trace Unit [19] is used to download the program and trace the instructions executed in the microcontroller. High-speed data and instruction trace are streamed directly to the host computer allowing off-line analysis of the program behaviour [19]. MATLAB is used to implement the proposed method prior to hardware implementation. It should be noted that our experimental platform limits the complexity of test programs, as it

comes with only 128KB of on-chip RAM and 2MB of external SRAM, for which only 1MB is usable when the tracing port is enabled. This limitation falls within the scope of our initial embedded architecture, expected to have minimal memory, power and computational resources. The concept presented here is very scalable; as the available resources increase the complexity of applications can also be increased.

As our initial focus is dedicated and constrained embedded systems, five algorithms from the automotive package of the MiBench benchmark suite [20] are selected: angle conversion (AC); bit count (BC); cubic function (CF); random numbers (RN); and square roots (SR). These five algorithms are mixed together as a single program, and this program is treated as original. We also propose five further compromised programs formed by various combinations of the five algorithms. In each combination AC, BC, CF, RN, and SR are executed twice. In addition to the above, we also use an "unknown" algorithm "Fibonacci Series (FS)" to replace AC, BC, CF, RN, and SR to represent another five compromised programs for testing. Since the FS algorithm consists of some similar sub-functions to the known algorithms, this experimental setup is more suitable for evaluating the proposed system. At the beginning of the test, we run the original program five times separately in the embedded platform, and all the program execution trace profiles are stored into five different files respectively. One of the files (i.e. the training file) is used for training the SOM classifier and the remainder are used for testing.

During training, PC values from the "training file" are used as input to the SOM. The size of the training vectors is 2048, taken from 2048 PC values for each peak in the training file. The vector values are then normalised before feeding them into the SOM. The epoch use for training is set to 1000, after which VQ is used to assign labels to the neurons. The outputs of the training are network weights, a record of each phase, the corresponding neuron(s), and associated minimum and maximum distance for the phase. In these experiments the network size has been fixed to 20 each of length 2048. For testing, each of the test files (27 files in total) is fed into the trained network to generate individual output files. The output after testing is the peak similarity (P_s), the correct detection rate (true positive (T_p) and true negative (T_n)) representing the correct detection rate of the SOM for known testing programs and unknown testing programs respectively, rate of misclassified unknown testing programs (false positive (F_p)) and rate of programs misclassified as compromised (false negative (F_n)). Fig. 7 shows the training and testing results for the original program.

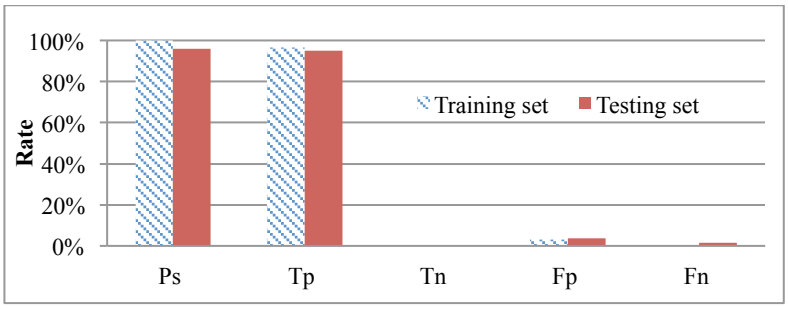

Fig. 7. Training and testing results for the original program.

Table 1. Outputs of SOM using compromised testing programs

	P_s (%)		T_p (%)		T_n (%)		F_p (%)		F_n (%)	
	T_1[a]	T_2[b]	T_1	T_2	T_1	T_2	T_1	T_2	T_1	T_2
AC	98.0	98.0	86.5	82.7	0	0	3.9	5.8	9.6	11.5
BC	95.8	94.4	86.7	86.8	0	0	6.7	2.9	6.7	10.3
CF/**FS**	90.9	**0**	40.0	**0**	0	**50.0**	0	**50.0**	60.0	**0**
RN	40.0	80.0	87.5	75.0	0	0	0	0	12.5	25.0
SR	98.7	94.7	68.8	73.8	0	0	1.3	1.3	29.9	25.0

a. Without unknown algorithm; b. With unknown algorithm

As shown in Fig. 7, the training and testing results have very similar performance, but P_s for a particular algorithm may vary when it is executed at different times. This is because CPI does not remain exactly the same as the original value in the training file. In Table I, RN has been repeated twice in T_1 and we replaced CF with FS representing the unknown algorithm in T_2. P_s of 40% in Table I shows RN has the least peak similarity compare to the other algorithms, suggesting RN has been compromised. However, the result from the T_p (87.5%) shows that the algorithm is known to the SOM. Overall, the correct detection rate for our system is above 90.9% for uncompromised programs. P_s of 0% in Table I shows that FS is completely different from CF in the original training file. The result for T_p (0%) in Table I shows that FS is unknown to the SOM. T_n (50%) means that 50% of codes are unknown to the SOM and F_p (50%) means that 50% of codes are known to the SOM but they appeared in the wrong section. As an unknown program is introduced, the overall SOM recognition rate for each algorithm is reduced, which indicates the original program has been compromised. In our experiments, the threshold for detecting a compromised program is set to 50%. Hence a program is treated as compromised if T_n is greater than 50%. Fig. 8 shows the PC profile when the CF is replaced by the FS.

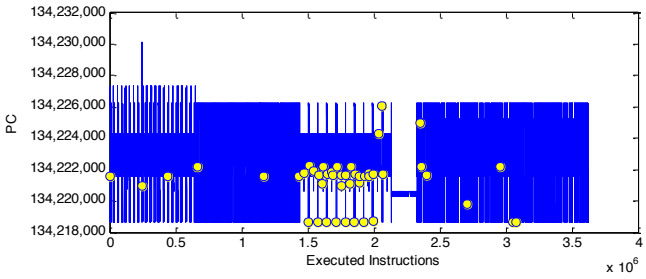

Fig. 8. PC profile when the CF is replaced by the FS.

In Fig. 8, the yellow circles indicate selected peaks from compromised part of the executed intrusions. As can be seen from the figure, most of the circles are concentrated at FS section, with the very few sparsely distributed over the remaining programs, contributing to the marginal error.

5 Conclusion

In this paper, we have presented an approach for detecting compromised programs by analysing CPI and PC from an embedded system. Through monitoring the processor's CPI, we detect changes in the function call and CFG, and then analyse the PC values within each CFG using SOM. The results achieved show that the proposed algorithm can be used to detect the changes in a program, and the information metrics can further be generated based on the outputs from the SOM. For example, different basic numbers could be generated based on the results of SOM, as a result of this, different encryption keys can be generated by the key cryptography mechanism, using the recall phase. Since the main aim of this research work is to implement a real-time security solution for complex embedded computer architectures, more evaluation on realistic attacks for the proposed algorithms will further be investigated. Moreover, the proposed algorithm can be used in combination with other ICmetric approaches to evaluate commercial embedded system benchmarks. For evaluation parameters of real-time detection system, the proposed algorithm can also be implemented with a soft-core processor on FPGA as part of an on-line protection system. The online implementation will have the capability of extracting execution trace from customised tracing interfaces directly located on the processor, determine the behaviour in real-time, and subsequently halting the program to prevent any harmful effect on the embedded system architecture.

Acknowledgment

The authors gratefully acknowledge the support of the EU ERDF Interreg IVa 2 Mers Seas Zeeën Cross-border Cooperation Programme – SYSIASS project: Autonomous and Intelligent Healthcare System (project's website http://www.sysiass.eu/).

References

1.Arora, D., Ravi, S., Raghunathan, A., Jha, N.K.: Secure embedded processing through hardware-assisted run-time monitoring. In: Proceedings Design, Automation and Test in Europe, pp. 178-183 (2005)
2.Corporation, F.-S.: F-Secure reports amount of malware grew by 100% during 2007, Helsinki, Finland (2007)
3.Dongara, P., Vijaykumar, T.N.: Accelerating private-key cryptography via multithreading on symmetric multiprocessors. In: IEEE International Symposium on Performance Analysis of Systems and Software, pp. 58-69. (2003)
4.M.Rahmatian, Kooti, H., Harris, I.G., Bozorgzadeh, E.: Hardware-Assisted Detection of Malicious Software in Embedded Systems. IEEE Embedded Systems Letters 4, 94-97 (2012)
5.Suh, G.E., Devadas, S.: Physical Unclonable Functions for Device Authentication and Secret Key Generation. In: 44th ACM/IEEE Design Automation Conference, pp. 9-14. (2007)

6.Handschuh, H., Schrijen, G.-J., Tuyls, P.: Hardware Intrinsic Security from Physically Unclonable Functions. In: Sadeghi, A.-R., Naccache, D. (eds.) Towards Hardware-Intrinsic Security, pp. 39-53. Springer Berlin Heidelberg (2010)
7.Hospodar, G., Maes, R., Verbauwhede, I.: Machine learning attacks on 65nm Arbiter PUFs: Accurate modeling poses strict bounds on usability. In: IEEE International Workshop on Information Forensics and Security (WIFS), pp. 37-42. (2012)
8.Arora, D., Ravi, S., Raghunathan, A., Jha, N.K.: Secure embedded processing through hardware-assisted run-time monitoring. In: Proceedings of Design, Automation and Test in Europe, pp. 178-183 Vol. 171. (2005)
9.Hanilci, C., Ertas, F., Ertas, T., Eskidere, O.: Recognition of Brand and Models of Cell-Phones From Recorded Speech Signals. IEEE Transactions on Information Forensics and Security 7, 625-634 (2012)
10.Kovalchuk, Y., McDonald-Maier, K.D., Howells, G.: Overview of ICmetrics technology-security infrastructure for autonomous and intelligent healthcare system. International Journal of u- and e- Sevice, Science and Technology 4, 49-60 (2011)
11.Howells, G., Papoutsis, E., Hopkins, A., McDonald-Maier, K.: Normalizing Discrete Circuit Features with Statistically Independent values for incorporation within a highly Secure Encryption System. In: Second NASA/ESA Conference on Adaptive Hardware and Systems, pp. 97-102. (2007)
12.Kohonen, T.: Learning vector quantization. In: Michael, A.A. (ed.) The handbook of brain theory and neural networks, pp. 537-540. MIT Press (1998)
13.Deng, M., Wuyts, K., Scandariato, R., Preneel, B., Joosen, W.: A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. Requirements Eng 16, 3-32 (2011)
14.Yang, R., Qu, Z., Huang, J.: Detecting digital audio forgeries by checking frame offsets. Proceedings of the 10th ACM workshop on Multimedia and security, pp. 21-26. ACM, Oxford, United Kingdom (2008)
15.Swaminathan, A., Mao, Y., Wu, M., Kailas, K.: Data Hiding in Compiled Program Binaries for Enhancing Computer System Performance. In: Barni, M., Herrera-Joancomartí, J., Katzenbeisser, S., Pérez-González, F. (eds.) Information Hiding, vol. 3727, pp. 357-371. Springer Berlin Heidelberg (2005)
16.Boufounos, P., Rane, S.: Secure binary embeddings for privacy preserving nearest neighbors. In: IEEE International Workshop on Information Forensics and Security (WIFS), pp. 1-6. (2011)
17.Annavaram, M., Rakvic, R., Polito, M., Bouguet, J., Hankins, R., Davies, B.: The fuzzy correlation between code and performance predictability. In: the 37th International Symposium on Microarchitecture (MICRO), pp. 93-104. (2004)
18.STMicroelectronics. STM32F207G DATA Sheet http://www.st.com/ (Accessed on Jan 2013)
19.KEIL. Keil uVision IDE Data Sheet. http://www.keil.com/uvision/ (Accessed on Jan 2013)
20.Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: IEEE International Workshop onWorkload Characterization, pp. 3-14. (2001)