A Method for Detecting Abnormal Program Behaviour on Embedded Devices

Xiaojun Zhai, Kofi Appiah, Shoaib Ehsan, Gareth Howells, Huosheng Hu, Dongbing Gu and Klaus McDonald-Maier

***Abstract*—A potential threat to embedded systems is the execution of unknown or malicious software capable of triggering harmful system behaviour, aimed at theft of sensitive data or causing damage to the system. Commercial off-the-shelf embedded devices, such as embedded medical equipment, are more vulnerable as these type of products cannot be amended conventionally or have limited resources to implement protection mechanisms. In this paper, we present a Self-Organising Map based approach to enhance embedded system security by detecting abnormal program behaviour. The proposed method extracts features derived from processor's Program Counter and Cycles per Instruction, and then utilises the features to identify abnormal behaviour using the SOM. Results achieved in our experiment show that the proposed method can identify unknown program behaviours not included in the training set with over 98.4% accuracy.**

***Index Terms*—Embedded system security, abnormal behaviour detection, intrusion detection, Self-Organising Map.**

I. Introduction

The widespread use of embedded systems today has significantly changed the way we create, destroy, share, process and manage information. For instance, an embedded medical device often processes sensitive information or performs critical functions for multiple patients. Consequently, security of embedded systems is emerging as an important concern in embedded system design [1, 2]. Although security has been extensively explored in the context of general purpose computing and communications systems, for example via cryptographic algorithms and security protocols [3], such security solutions usually are often incompatible with particular embedded architectures. The reason for this is, that embedded architectures use custom firmware or operating systems, and are normally specific to a certain function with limited cost and resource, which makes e.g. conventional antivirus (AV) programs difficult to implement. Generally, the protection of embedded systems can be developed either at hardware or/and at software level.

From hardware perspective, Physical Unclonable Function (PUF) [4] or hardware intrinsic security [5], has been proposed to secure embedded devices physically. The manufacturing process variation is first used to identify the integrated circuits, and then the identifications are subsequently used for cryptography. There are also works focusing on detecting software failure, tampering and malicious codes in embedded architectures [1, 6]. The main disadvantage of these approaches is that they require storing sensitive data in the system as "valid" samples or templates. For example, a basic-block control-flow graph (CFG) is usually stored and used to exam the running program.

Embedded devices that are used in the medical and industrial domains usually perform a small number of repetitive functions or operate in a simplified state space. The execution space may include activities such as actuating an electrical relay, controlling a pump, or collecting sensor readings [7]. This intrinsic behaviour makes them unsuitable for conventional AV and exposes deviation in normal program execution as a means of detecting compromised activities. There are currently alternative solutions that may secure vulnerable embedded architectures [8], [9], where machine learning and pattern recognition algorithms are employed on human-machine interaction. ICMetrics (Integrated Circuit metrics) [10], is one of the on-going research areas into embedded security, which relies on the unique trace

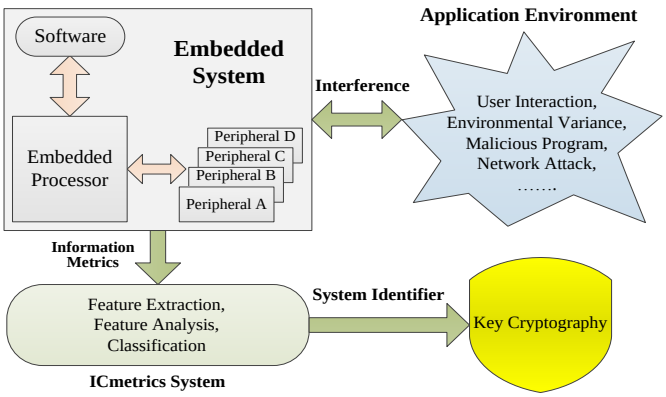

Fig. 1. A typical embedded system and ICMetrics system.

Manuscript received October 01, 2014. Manuscript revised March 27, 2015. Manuscript accepted March 29, 2015. This work was supported by the UK Engineering and Physical Sciences Research Council under grant EP/K004638/1 and the EU ERDF Interreg IVA 2 Mers Seas Zeeën Cross-border Cooperation Programme – SYSIASS project: Autonomous and Intelligent Healthcare System.

S. Ehsan, H. Hu, D. Gu, and K. McDonald-Maier are with the School of Computer Science and Electronic Engineering, University of Essex, Colchester, UK (e-mail: { sehsan, hhu, dgu, kdm}@essex.ac.uk).

X. Zhai was with University of Essex, Colchester, UK. He is now with the Department of Engineering, University of Leicester (e-mail: xz151@leicester.ac.uk)

K. Appiah was with University of Essex, Colchester, UK. He is now with the School of Science and Technology, Nottingham Trent University (e-mail: kofi.appiah@ntu.ac.uk).

G. Howells is with the School of Engineering and Digital Arts, University of Kent, Canterbury, UK (e-mail: w.g.j.howells@kent.ac.uk).

generated on the embedded architecture by its regular user or environment. The concept of ICMetrics is akin to biometrics in humans. Fig. 1 exhibits a typical embedded system and ICMetrics system.

The ICMetrics based system can offer multiple advantages over traditional static AV approach like scanning executable, instruction sequences and CFG of an application, which does not need to store user data or template and supports from operating systems. Our approach is suitable for embedded devices predominantly used in the medical and automation industry, which have limited cost and resource in the systems.

In this paper, we use Cycle per Instruction (CPI) to extract corresponding Program Counter (PC) values, and use it as ICMetrics features for correct program identification allowable to execute on the embedded architecture, and an unsupervised Self-Organising Map (SOM) is used to classify the behaviour of the embedded system. Results achieved in our experiment show that the proposed method can identify unknown program behaviours not included in the training set with great accuracy.

The remainder of the paper is structured as follows: Section II discusses the related work in this domain. The threat model utilized for this work is introduced in Section III. A SOM-based abnormal behaviour detection algorithm is presented in Section IV. To demonstrate the usefulness of the presented technique, Section V details the experimental design and results performed on an ARM Cortex-M3 embedded processor. Finally, the conclusions are presented in Section VI.

II. Related Work

This section provides a brief overview of the previous work related to embedded systems security. As mentioned in Section I, information digitization to facilitate quick access has rendered digital privacy an important issue in protecting personal data [11]. While we believe our work to be the first demonstration of how on-chip debug information [12] can be used to identify anomalies in embedded system program execution, previous research has investigated the behaviour and prevalence of code modified with the intent of harming a system or its user. 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. In their work, properties of an embedded program is extracted and used as the basis for enforcing permissible program behaviour.

Software piracy has enormous economic impact [13], making it important to protect software intellectual property rights. Software watermarks, a unique identifier embedded in a protected software to discourage intellectual property theft is presented by Collberg and Thomborson [14]. In [15], Kolbitsch et al proposed a malware detection system to complement conventional AV software by matching automatically generated behaviour models against the runtime behaviour of unknown programs. Similar to [1], Rahmatian et al [5] used a CFG to detect intrusion for secured embedded systems by detecting behavioural differences between the correct system and malware. In their system, each executing process is associated with a finite state machine (FSM) that recognizes the sequences of system calls generated by the correct program. Attacks are detected if the system call sequence deviates from the known sequence. The system promises the ability to detect attacks in most application-specific embedded processors. Wang et al [12] proposed a system call dependence graph (SCDG) birthmark software theft detection system. Software birthmarks have been defined as unique characteristics that a program possesses and can be used to identify the program. Without the need for source code, a dynamic analysis tool is used in [16] to generate system call trace and SCDGs to detect software component theft.

Yang et al [17] presented an 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. Their work proves that frame offsets detected by the identification of quantization characteristics are good indication for locating forgeries. Experiment conducted on 128 MP3 speech and music clips shows 94% rate of correctly detecting deletion and insertion using frame offset. Panagakis and Kotropoulos [18] proposed the random spectral features (RSFs) and the labelled spectral features (LSFs) as intrinsic fingerprints suitable for device identification. The unsupervised RSFs reduce the dimensionality of the mean spectrogram of recorded speech, whiles the supervised LSFs derives a mapping between the feature space where the mean spectrograms lie onto the label space. Experimental result shows that RSFs and LSFs are able to identify a telephone handset with up to 97.58% accuracy.

Information hiding can be used in authentication, copyright management as well as digital forensics [19]. Swaminathan et al [19] 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. The proposed system employs look-up-tables for data embedding and extraction, which is subsequently stored in the program header and loaded into run-time memory at the beginning of program execution. In [20], Boufounos and Rana demonstrate with the use of signal processing and machine learning techniques, how 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.

As indicated above, software birthmarks are unique characteristic that a program possesses and can be used to identify the program [12]. Similarly, 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. Let p, q be programs. Let $f(p)$ be a set of characteristics extracted from p when running on hardware f. We say $f(p)$ is the ICMetrics of p, only if the following two conditions are satisfied:

1) $f(p)$is obtained from p running on f.

2) Program q is a copy of $p \Rightarrow f(p) = f(q)$.

The limitations with the use of system calls for program identification [1], 5] have been pointed out in [12] and are more prevalent in embedded systems settings, which typically have no operating system. The mentioned limitations are:

1) Programs with little or no system calls such as programs solely based on arithmetic operation and
2) Programs which do not have unique system call behaviours may fail to exhibit a birthmark.

Using an unsupervised SOM to reduce the dimensionality of PC values, we introduce an offset rule similar to that presented in [17] to detect compromised programs. Thus using machine learning techniques [20] we are able to determine whether two PC values are similar to each other, with the use of the program binaries [19] and no prior knowledge of the source code. Our main contributions of this paper can be summarised as follows:

1) We introduce a novel SOM based anomaly detection system, which can be used to combine with an ICMetrics system in the embedded devices predominantly adopted in the medical and automation industry.
2) Our approach introduces a way to extract and analyse the useful low level hardware information, and used them as a feature to identify an embedded system's abnormal behaviour, which allows our system to be used in a wider range of embedded systems, as it is independent to the high level software environments (e.g. Operating system, source programs).
3) In terms of performance, the results achieved in our experiment show that our approach also outperforms other existing SOM based anomaly detection systems that utilise the high level software information.

III. Threat Model

Embedded systems are used in a variety of applications in our daily life and enable sophisticated features for their users. However, these sophisticated features increase system complexity, which in turn results in a higher occurrence of bugs that require software updates to fix. Embedded systems with network access and code update support are therefore becoming increasingly mainstream. Unfortunately, this flexibility substantially increases the risk of malicious code injection in embedded systems. For example, there is a steady increase in the number and complexity of embedded processors in vehicular embedded networks (GPS, in-car entertainment, safety systems, car communication systems). This in turn has raised major software integrity issues, and it is critical to ensure that the executing instructions have not been changed by an attack.

Attacks that are harming software integrity are generally known as code injection attacks, since they inject and execute malicious code instead of correct programs. A well-known code injection attack is stack smashing. If a function does not validate whether the length of the input exceeds the buffer size, an attacker can easily overflow the buffer. By overflowing the buffer, any location on the stack in the address space after the start of the buffer can be overwritten, including the return address of the susceptible function. Using this technique, an attacker can insert malicious code sequence, and overwrite the return address to point to the malicious code. Other attacks may overflow buffers stored on the heap, or exploit integer errors, dangling pointers, or format string vulnerabilities. Most programs with these vulnerabilities are also susceptible to so-called return-into-libc attacks, where an attacker modifies a code pointer to point to the existing code, usually the library code. Return-into-libc attacks are also called arc injection, since they inject an arc in a control flow graph of a program.

The proposed system is designed to protect against the execution of malicious code that the system designer does not intend to execute. Our interest is to ensure that the software running continuously on an embedded device has essentially the same behaviour as the original program for the purposes of security and detect any possible changes on the trusted software. The basis of our proposed system of ICMetrics is akin to dynamic systems analysis, which analyse the execution of a program on an embedded architecture. Thus the system presented is mainly for flagging rather than directly stopping execution of untrusted code.

A common theme among many security attacks is hijacking the trusted code at run-time, so even if the original code is not malicious by intent, it can be manipulated by the attacker [6]. As mentioned above, a very common method is the exploitation of a buffer overflow to overwrite a return address, altering program control flow to a malicious code. We assume that the unexpected software running on the embedded device will result in a significant behavioural difference compared to the original program. The proposed system monitors the executing program continuously, while constructing its behaviour to detect any changes. It is observed that any behavioural difference in the program execution trace, for example in medical devices can be detrimental and must be flagged in real-time by monitoring the system behaviour. The proposed intrusion detection method will not prevent buffer overflow, but it could detect the abnormal behaviour caused by buffer overflow by monitoring system behaviour.

IV. Algorithm for Abnormal Program Behaviour Detection

Generally, from a software architecture point of view, there are three structural levels in a program: (*a*) function call level, as represented by function call relationship; (*b*) internal control flow for each function, represented by a basic-block CFG; and (c) instruction stream within each CFG [1]. From a hardware point of view, the processor's architecture and performance can affect the execution of instructions. For instance, multi-cycle function calls or condition branches could decrease the performance of a processor. On the other hand, as the PC register indicates where a program is in its code sequence, it can be used to represent the instruction sequence within the CFG. Consequently, we could first detect the function call and CFG based on the variance of the processor's performance, then analyse the PC values within each CFG. Finally, an overall evaluation could indicate

whether the system is compromised or not. In the proposed work, we measure the average CPI as a parameter of the processor's performance. A block diagram of the architecture of the proposed abnormal program behaviour detection system is shown in Fig. 2.

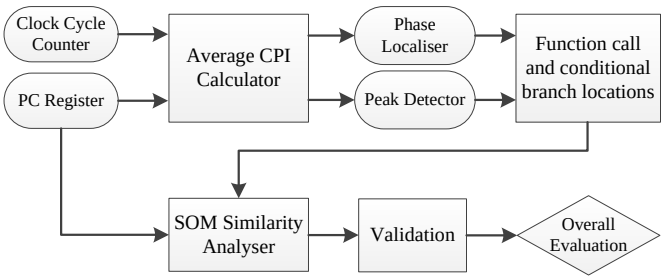

Fig. 2. Overall block diagram of the proposed abnormal program behaviour detection algorithm.

The average CPI calculator in Fig. 2 is first used to calculate average CPI value, and it continually reads clock cycle and PC data from the time counter and PC register. Sequentially, the average CPI values are used to obtain phase and peak information in the Phase and Peak Point Detector module respectively, and the information indicates where function calls or the conditional branch occur in the executing program. Afterwards, the obtained locations and their corresponding PC sequence are used in a SOM based similarity analyser for abnormal program behaviour detection. If the phase's information and the PC sequences deviate from a known program, the SOM based classifier asserts the intrusion detected output. In the last stage, the results of SOM are validated by comparing with their expected property table (i.e. number of peak within each phase and associated network node).

A. Average CPI Calculator Module

CPI is one of the most commonly used parameter for measuring processor's performance, which indicates the complexity of instructions executed within a particular period of time. Average CPI of a processor can be calculated based on (1):

$$CPI = \frac{C}{I} \tag{1}$$

where I is the total number of executed instructions, C is the number of cycles for executing I instructions. As number of cycles can be calculated by time elapsed and maximum clock frequency of a processor, the CPI can easily be accessed by modern debug facilities. In Fig. 3, an average CPI profile is generated while a program is running in an ARM cortex-M3 processor, where I and the maximum frequency are set to 2^{11} and 120 MHz respectively.

In Fig. 3, the program consists of five different functions, and each function is called in a sequence. While a new function is called, the CPI value is significantly increased, which means the performance of the processor is decreased accordingly. The main reason for that is that the PC jumps to other memory location in order to execute the newly called function (as illustrated in Fig. 7 (a)), where it usually involves many multi-clock cycles instructions. As a result, the average CPI value is significantly changed. Similarly, the CPI values vary within each function, and the number of executed instructions I decides the resolution of the average CPI profile, the value of I varies from [1 n], where n is the total length of programme. The larger number of I used in the CPI profile, the less details of the CPI profile, which means some of potential abnormal behaviour of the monitored programme may not be detected. However, although with smaller number of I, we could have more sensitive of the detection mechanism, it would significantly increase the computational cost of the detection system. For instance, if I uses '1', which means that every single instruction in the programme will be examined and it does not contain any continuous pattern that can be used to identify the characteristics of the monitored programme Therefore, in this paper, the value of I is set to 2^{11}, which gives a gives the best balance of the accuracy and computational complexity of the proposed system. In the following sub-sections, we introduce a method to automatically obtain the position information of the phases (i.e. function calls) and peaks (i.e. branch conditions).

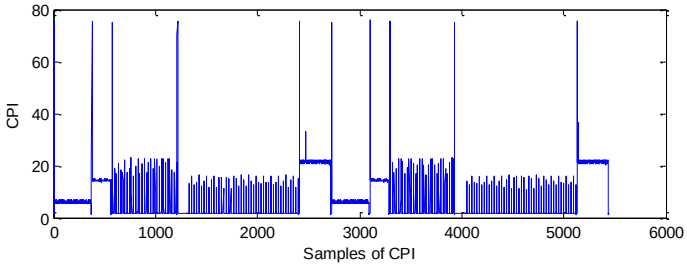

Fig. 3. Example of average CPI profile.

B. Phase and Peak Point Detector Module

The main task of this module is to obtain the locations of the phase and peak within the average CPI profile. There are two sub-blocks: local and global critical point localisers are used to localise the peak and phase positions.

1) Local Critical Point Localiser

The local critical point localiser is used to localise the local significant variance points from the average CPI profile. The proposed method first calculates absolute differences between adjacent elements in the average CPI profile, and then localises the peak value within a 1×3 rectangular range.

Let f_{mean} denotes 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 $f_{mean}(n)$.

After obtained $d(n)$, a 1×3 rectangular window is used as a mask to scan all the elements in $d(n)$. Let $d(n\text{-}1)$, $d(n)$ and $d(n+1)$ denote the three elements within the 1×3 rectangular window respectively, and the locations of the detected peaks can be calculated by:

$$p(n') = n \quad \text{for } d(n) > d(n-1) \text{ and } d(n) > d(n+1) \qquad (3)$$

where $p(n')$ is n^{th} element in an array of detected peak locations.

The main advantage of the proposed local critical point localiser is adaptively detecting the peaks without the need of setting any fixed threshold, hence the proposed local critical point localiser would not be limited on any particular scenario, and it can also detect the peaks that have minor variance. Fig. 4 shows resulting diagram after applying the local critical point localiser on the points in Fig.3.

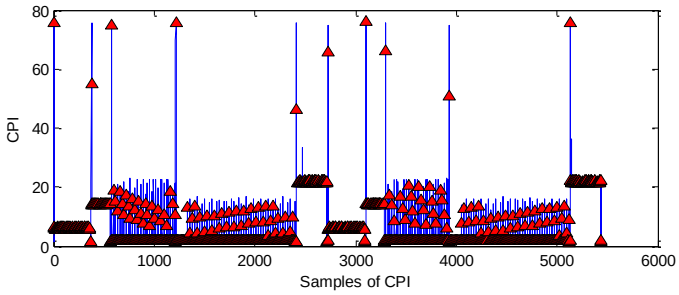

Fig. 4 Resulting diagram after applying the local critical point localiser.

2) Global Critical Point Localiser

The global critical point localiser is used to localise the global significant variance points from the average CPI profile, which indicate the locations of each phase.

Step 1: Localising the elements in $d(n)$ that are greater than $(max(d) + min(d)) / 2$. These elements represent the boundary points at each adjacent phase. The selected elements are stored in array p'.

Step 2: Calculating absolute differences between adjacent elements of array p', if the absolute differences between k^{th} and $(k+1)^{th}$ elements are greater than t, then store $p'(k)$ and $p'(k+1)$ into array p_h, where t is the number of CPI samples in a phase. The value of t depends on the minimum accepted phase length of the training programme. The smaller the value of t is, the more details of the average CPI profile can be obtained. On the other hand, in consequence the complexity of the proposed algorithm would be increased. In this paper, t is set to 50 in order to balance the complexity and performance of the proposed algorithm.

Step 3: Checking absolute difference between every adjacent phase $p_h(2k')$ and $p_h(2k'+1)$, if the difference is greater than '2' or equal to '0', then $p_h(2k'+1) = p_h(2k') + 1$. The main purpose of this step is to make sure that the adjacent phases do not include the overlapped boundaries.

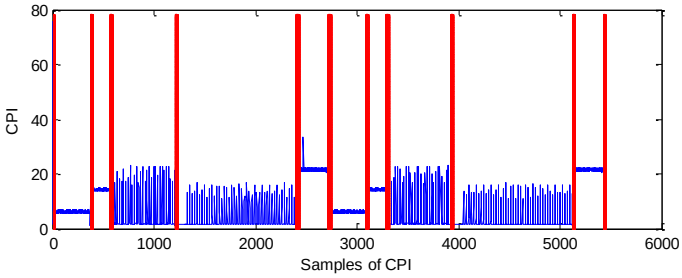

Fig. 5 Resulting diagram after applying the global critical point localiser.

Fig. 5 shows resulting diagram after applying the global critical points localiser on Fig. 3.

The obtained peak and phase locations are first converted into their corresponding locations in PC profile by (3):

$$\begin{aligned} p_s &= I \times p(n') + 1 \\ p_e &= I \times p(n') + I \\ p_{hs} &= I \times p_h(k') + 1 \\ p_{he} &= I \times p_h(k'+1) \end{aligned} \qquad (3)$$

where p_s and p_e are the start and end locations in PC profile for the n'^{th} peak respectively. p_{hs} and p_{he} are the start and end locations in PC profile for the k'^{th} phase respectively. I is the total number of executed instructions used to calculate average CPI profile.

The converted locations are used to select appropriate PC patterns for training and testing of the similarity analyser.

C. SOM based Similarity Analyser Module

The designed similarity analyser is capable of classifying and recognising between known and unknown programs while the programs are running. There are two major levels of the classification and recognition process: the function call level and the PC pattern level, where each phase and peak is measured to ascertain the originality of the program in execution. Any significant difference shows that the numbers of function calls differ, characteristic of function call and PC signature are different to the original program, and an abnormal behaviour notification could be signified. The main advantage of the proposed similarity analyser is that it governs the classification and recognition at two different levels: 1) phase and peak level, and 2) the PC pattern level. Phase and peak level are statistically analysed, and the corresponding PC patterns are classified in SOM. Consequently, even if the malicious codes have similar information of the phase and peak, it is very difficult to have the exactly same PC pattern as the original code.

Kohonen's SOM [21] is a common pattern recognition and clustering process, where intrinsic inter- and intra-pattern relationships among the stimuli and responses are learnt without the presence of a potentially biased or subjective external influence is presented, and would be adopted in this work as the basis for our classifier. We utilize the *k−means* nature of the SOM, to partition the extracted PC signatures into a user-specified number of clusters, k (number of groups). In the proposed work, the analyser uses SOM to measure similarity between known and executing programs in terms of PC pattern. The value of k should at minimum be equal to the total number of programs intended to run on the embedded hardware. The value of k used in this work is set to two times (2x) the number of known programs that can legitimately run on the embedded processor. This value of k is to handle the linear separating boundaries between known program behaviours as defined in K-means clustering; avoiding the computational overheads associated with a nonlinear kernel K-means. Specifically, we extract static properties of an embedded program to enforce permissible program behaviour at run time. The PC patterns are a set of *N*-dimensional

vectors, where the size of the vectors N is equal to number of executed instructions I. The size N of the vector, if set too large will add significant performance overhead to the application that it represents. Similarly, if it is set too small, it will not be robust enough to distinguish between applications. Thus choosing the right size of N is very important.

The best value of N is the minimum number of PC values that offers the best analyser performance. The value of N should define the permissible behaviour of a program by identifying suitable properties or invariants that are indicators of untampered execution, thus very unlikely to be violated when program is compromised. After a number of empirical experiments, the value of N in this work has been set to 2^{11} following an examination of the test data.

To enable continuous analysis, the system presented here requires just 2^{11} PC values at a time to infer its corresponding application. Because the system is based on a SOM, a variant of the *k–means* algorithm, the value of k should also be set. The value of k is set depending on the total number of algorithms or programs under investigation, and the number of distinct phases in any particular application. At minimum, the value of k should be equal or greater to the number of algorithms under investigation. The value of k in this work has been set to 20 as the testing database has 10 different programs. However, this can naturally be adapted to requirement of different usage scenarios according to the above given guidelines.

PC values extracted from the PC profile, corresponding to the peaks in the CPI profile 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

$$D_j^2 = \sum_{l=1}^{N} \left(K_l^i - w_{jl} \right)^2 \tag{4}$$

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.

$$\Delta w_k = \eta \left(K^i - w_k \right) \tag{5}$$

The update is done only at the training phase. Additionally, for every neuron in the network we maintain four extra parameters: the minimum, maximum, mean and standard deviation of 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) [21] to assign labels to neurons in the network as follows:

1) Assign labels to all training data. The label is an identifier for the program from which the training data has been extracted.
2) Find the neuron in the network with the minimum distance to the labelled input data.
3) 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.

In each phase of the original training program, we first count a group of input vectors that are associated with each neuron, and then calculate mean value and standard deviation of the group of distances, alongside the minimum and maximum distances (D_{min} and D_{max}) by:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} D_i \tag{6}$$

$$D_{\min} = \mu - (1+\alpha) \times \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (\mathrm{D}_i - \mu)^2} \tag{7}$$

$$D_{\max} = \mu + (1+\alpha) \times \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (\mathrm{D}_i - \mu)^2} \tag{8}$$

where D denotes the group of distances, α denotes errors of the standard deviation to accommodate any quantization errors in the calculation process. The value of α in this work has been set to 2.5%.

Sequentially, a statistical table T_k is generated for the k^{th} phase, where detailed attribute information (e.g. minimum and maximum distances, number of input vectors that are associated with each neuron and their standard deviation) are recorded for the phase. On the same principle, each phase is associated with its corresponding statistical table.

In the testing stage, each input vector is assigned to a neuron that has the shortest distance. Let $\mathbf{K^i}$ denotes the input vector and it is assigned to the j^{th} neuron with distance D_i, the proposed algorithm first compares the distance D_i with the minimum and maximum distances of the j^{th} neuron from all the statistical tables, and then decides whether the input vector belongs to the phase. Generally, the successful input vector should meet the following two conditions:

1) The distance D_i should meet the condition $D_{min} < D_i < D_{max}$, where D_{min} and D_{max} are minimum and maximum distance of the j^{th} neuron at the k^{th} phase.
2) The j^{th} neuron is a dominant neuron in the k^{th} phase, which means the occupancy of the neuron in the original statistical table is greater than 3% of total numbers of input vectors.

The successful candidate neurons are labelled to reflect their corresponding phase numbers. Otherwise, the candidate is marked as '-1', which indicates the input vector is unknown. Consequently, the known program's phase should consist of a set of known phase number; the dominant phase number to indicate the result of the phase. After obtaining the results of each phase, another statistical table T_k' is generated, which contains the same type of information as T_k. A validation process is performed in the next stage to examine the similarity of these tables.

D. Validation Module

The validation module is designed to validate the results from the SOM analyser. Usually, most of the input vectors can be classified using the SOM analyser. However, due to the variance of circumstances, the trace of program cannot always be exactly the same as the original training program, thus a global validation stage becomes necessary to improve the overall classification results.

In general, the results from the SOM analyser could consist

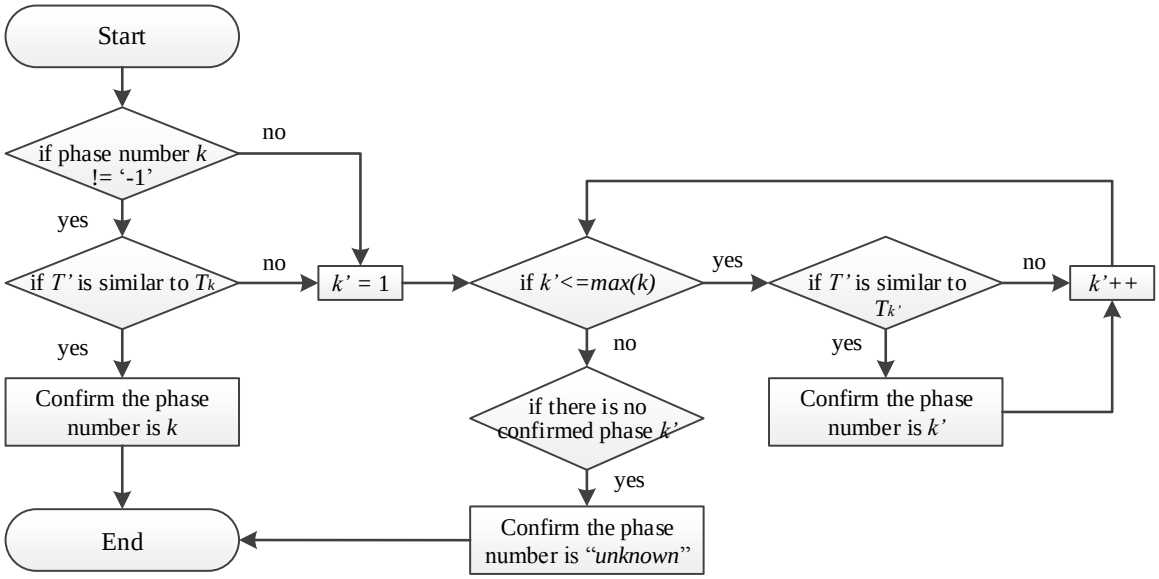

Fig. 6 Overall flow chart of the validation module.

of two categories: known and unknown samples. For the known samples, the SOM reports their potential phase number. For the unknown samples, the SOM analyser marks the phase number with '-1'. Thus, the validation module processes the two cases separately. Fig. 6 shows flow chart of the validation module.

As shown in Fig. 6, the main task is to validate the similarity between the testing statistical table *T'* and the original statistical table *T*. In order to examine the similarity of two tables, histograms of the associated neurons from the two tables are used. Pseudo-codes for calculating the similarity of the statistical tables are summarised as follows:

Calculating the similarity of the statistical tables:

1. Input: T_k and $T_{k'}$ are statistical tables for the original phase *k* and the testing phase *k'* respectively.
2. Output: Similarity between phase *k* and *k'*.
3. sort the T_k by descending order of neuron's occupancy;
4. /* *look-up the statistical table* T_k */
5. **for all** *neuron nodes* in T_k **do**
6. **if** occupancy of the j^{th} neuron > 3% **then**
7. *d*(*i*) = *j*; /* record the number of the neuron in array *d* */
8. *i* ++;
9. **end**
10. **end**
11. /* *look-up the statistical table* $T_{k'}$ */
12. **for all** *neuron nodes* in $T_{k'}$ **do**
13. **if** occupancy of the j^{th} neuron > 3% **then**
14. *d'*(*i*)=*j*; /*record the number of the neuron in array *d'* */
15. *i* ++;
16. **end**
17. **end**
18. $x \in d \bigcap d'$; /* ***x*** is the intersection of *d* and *d'* */
19. /**generate output*/
20. **if** length(*x*)/length(*d*) > 80% **then**
21. the phase *k'* is similar to the phase *k*;
22. **else**
23. the phase *k'* is not similar to the phase *k*;
24. **end**

After comparing the statistical tables, the difference of the number of peaks in the original phase *k* and testing phase *k'* is then calculated. If the difference is less than 10% of total number of peaks in the original phase, it confirms the phase number is *k*.

In general, the SOM analyser could locally calculate the similarity for a pair of input vectors (i.e. peaks). However, it has limitation on globally indicating a group of peaks (i.e. phases). The validation stage can be used to remedy this problem. In the experimental result section, we show the improvement of the SOM results when the validation stage is applied subsequently.

V. Experimental Setup and Results

An embedded system based on a Keil MCBSTM32F200 evaluation board equipped with an ARM 32-bit Cortex-M3 processor-based microcontroller is used in the proposed work [22], which consists of various peripheral interfaces (e.g. touchscreen, Ethernet port, serial port, analogue voltage control for Analogue-to-digital converter (ADC) input and debug interface). A combination of KEIL μVision IDE, and ULINKpro Debug and Trace Unit [23] 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 [23]. MATLAB is used to implement the proposed method prior to hardware implementation. It is worth noting that the experimental platform is a typical low cost ARM-based embedded development board, and 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. Thus we can only analyse a limited number of programs at a time, and the complexity of the tested programs are also limited. These limitations fall within the scope of our initial embedded architecture, expected to have minimal memory, power and

computational resources. The concept presented here is naturally scalable; as the available resources increase, the complexity of applications can also be increased.

A. Benchmark Test Suite

In the proposed work, seven algorithms from the automotive package of the widely recognised EEMBC benchmark suite [24] are selected, in which five algorithms (i.e. the first five benchmarks in Table I) are used to train and test the SOM analyser and the other benchmarks are only used in the testing. Details of the used benchmarks are presented in Table I.

As can be seen from Table I, the seven benchmarks are set with different parameters and performing various functions. For instance, the benchmark "*a2time*" is used to perform angle to time calculation, where "NUM_TEST" indicates the number of sets of input test data stimuli, and "TENTH_DEGREES" indicates the number of 1/10 degrees in a circle. Overall, they do not only have different complexities and characteristics, but also contain similar sub-functions, which make them suitable test candidates for the proposed experiments.

In order to train with all five benchmarks, they are mixed together to form a *new program*, where each benchmark is treated as a separate function call. The new program is executed twice in order to generate enough training samples. For testing, a random function call generator is used to switch between benchmarks form the test samples. The next section explains how the random function call switching works.

TABLE I
DETAILS OF THE USED BENCHMARKS

Benchmarks	Description	Parameters
a2time	Angle to Time Conversion	NUM_TESTS: 500 TENTH_DEGREES: 3600
rspeed	Road Speed Calculation	NUM_TESTS: 500 SPEED_SCALE_FAC: 36000
bitmnp	Bit Manipulation	NUM_TESTS: 128 INPUT_CHARS: 20 CHAR_COLUMNS: 5
idctrn	Inverse Discrete Cosine Transform	NUM_TESTS: 8192 COS_SCALE_FAC: 4096 COS_SCALE_EXP: 12
puwmod	Pulse Width Modulation	NUM_TESTS: 2420 MAX_PHASE: 20
tblook	Table lookup and interpolation	NUM_TESTS: 232 NUM_X_ENTRIES: 50 NUM_Y_ENTRIES: 50
ttsprk	Tooth to Spark	NUM_TESTS: 200 CYLINDERS: 4

B. Random Function Call Generator

In order to check the performance of the proposed system for complex test samples in a variety of scenarios, a random function call generator is used to randomly select the benchmarks and form a new program. Thus, the function call sequence of the new program is varied at every run. Consequently, a set of unique test programs can be generated. In addition, since the testing program is combined with different function calls and randomly mixed during the run-time of the embedded system, the testing methodology could be used to verify the performance of the proposed system in the scenarios that have dynamic variance (e.g. different program flow, interrupt, inputs, etc.).

The random function call generator mainly consists of two components: a true random number generator and a switch statement. In order to generate true random numbers, an ADC and a potentiometer are used to generate a random seed, which is subsequently used as an input seed for a pseudo-random number generator. In general, the ADC reads the voltage from the potentiometer and converts it into a 12-bit digital number. As the voltage of the potentiometer is adjustable and sensitive, the voltage value is not constant, even without turning the potentiometer. Thus, after the conversion, the digital number is always different, which allows the pseudorandom number generator to create a true random number. For instance, if a program consists of n different function calls, a random number x is first generated, where $1 < x < n$. Subsequently, the random number x is used to select which benchmark will be called. The generated random number is used in a switch statement, which actives the corresponding function call (e.g. if $x = 1$, "*a2time*" will be called). In this experiment, the potentiometer is manually adjusted for every run, which further ensures the voltage is completely different from the previous one.

In addition, the random function call generator can also record the function call sequence for every execution, which means a complete reference table can be generated at the end of testing. With comparing the test output of the SOM with the expected output from reference table, an accurate and complete evaluation result can be generated.

C. Program Database

A total of 104 programs are generated using the random function call generator presented in the previous section. The 104 programs used for testing can further be divided into the following three subcategories:

1) *Programs with original function call sequence*: Programs in this category consist of fixed function call sequence, which are the same as the one used in training. There are 21 programs, out of the 104, which are taken from this category.
2) *Programs with random generated function call sequence (known)*: Programs in this category consist of randomly generated function calls in the sequence, with all the functions drawn from the training samples. The number of samples in this category is 42.
3) *Programs with randomly generated function call sequence (unknown)*: Programs in this category consist of randomly generated function calls in the sequence with two unknown functions included. The number of samples in this category is 41.

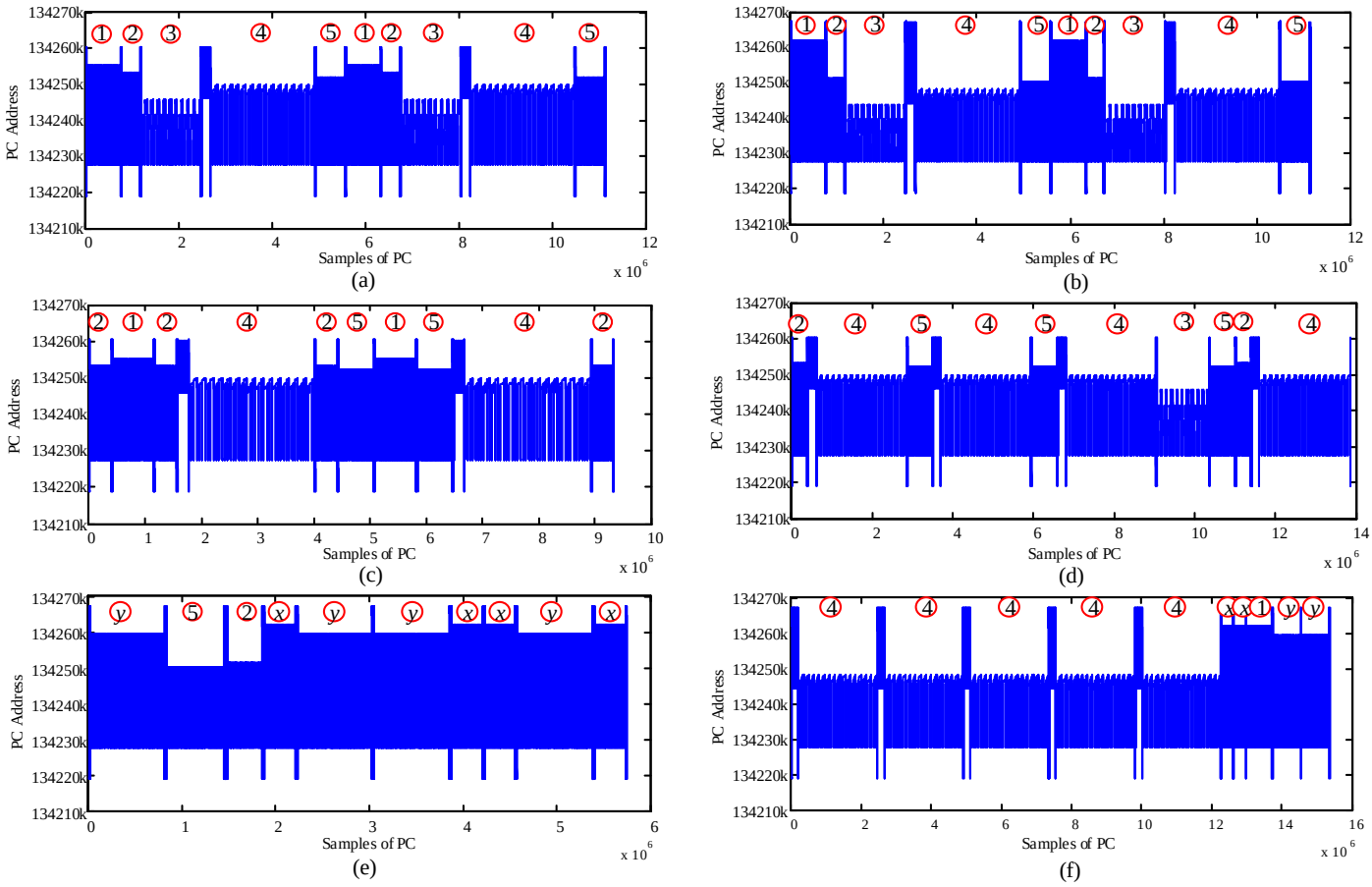

Fig. 7. Examples of PC profiles from the used program database. (a) Program used for training; (b) Program from category 1; (c) and (d) Programs from category 2; (e) and (f) Programs from category 3.

In the experiment, the first category is used to simulate instances where the embedded system is not modified, such as programs running with factory setting. The second category is used to simulate the circumstances of an embedded system with normal behaviour; for instance, the programs with legitimate credentials to run on the embedded system. Finally, the last category is used to simulate tampered systems with unknown programs; for example, the system may launch some unknown programs, triggered by buffer overflow attack. Thus, our threat model is well covered by the three set of categories. Fig. 7 shows some examples of PC profiles extracted from test programs.

In Fig. 7, numbers inside the red cycles are labels for the different benchmarks, where '1', '2', '3', '4', and '5' represent the five known benchmarks respectively with '*x*' and '*y*' representing the two unknown benchmarks. As can be seen from Fig. 7 (a) and (b), although they contain exactly the same benchmark codes and sequences, the PC addresses and outlines of each benchmark are slightly different. Especially, when the sequence of the benchmark is randomly mixed (for example, Fig. 7 (c) and (d)), the resulting PC profiles are completely different. This could help with examination of the trained SOM analyser on false negative rate. In Fig. 7 (e) and (f), the profile of the unknown programs '*x*' and '*y*' are quite similar to the known programs '1' and '2' respectively, which is used to simulate the possible attacks that try to model their peaks and phase information like the genuine programme Using the true/false positive and negative rates from the trained SOM analyser, different programs with similar profiles can further be examined.

D. System Implementation

The abnormal program behaviour detection system has been successfully implemented in MATLAB for off-line data analysis. The system implementation is divided into three parts:

1) CPI-related module

This module is first used to extract useful information from the program's tracing file, and then it calculates the average CPI for every run. The program's tracing file contains two types of information: PC address and time tag for every executed instruction. The PC addresses are only recorded in a file that will be used in the SOM-based similarity analyser module. However, the corresponding time tags are used to calculate CPI profile for the executed programs. In this work, the number of instructions is set to 2048. The frequency of the ARM cortex-M3 microcontroller used runs at 120 MHz, thus, the average CPI for every 2048 instructions can then be

calculated by (1). Subsequently, the phase and peak point detector localises the peaks and phases from the average CPI profile. The obtained peak and phase locations are finally converted into their corresponding locations in PC profile by (3).

2) SOM-based similarity analyser module

The start and end locations of each peak can be used to select a serial of PC addresses, and this forms an input vector with 1×2048 elements which is subsequently fed into the SOM-based similarity analyser. The maximum number of nodes and iterations for the SOM are set to 20 and 1000 respectively. A statistical table for each phase and estimated outputs for each peak are generated after the training process. The same process is repeated during the testing. The generated results are then used in the validation module.

3) Validation and evaluation module

The algorithm stated in Section IV-D is implemented in this module. Based on the validation results, the peaks and phases of each input program are finally classified. The final evaluation result consists of two levels: peak and phase levels. At the peak level, the final report does not only include results for every single program, but also the entire database. The measurements of the evaluation mainly includes correct recognition rate (true positive (T_p) and true negative (T_n)), rate of misclassified samples (false positive (F_p)), and rate of samples incorrectly classified as unknown (false negative (F_n)). Based on the measurements, accuracy, precision and recall rates for the proposed system can be calculated.

Accuracy

It is the rate of correctly labelled samples, which can be calculated by (T_p+ T_n) / total number of samples.

Precision

It is the rate of positively labelled samples whose labels are correct, which measures the classifier's resistance to false positives and can be calculated by T_p / (T_p +F_p).

Recall

It is the rate of samples that should have been positively labelled that are correctly positively labelled, which measures the classifier's resistance to false negatives and can be calculated by T_p / (T_p +F_n).

A classifier's precision and recall results provide insight into what types of errors the classifier tends to make, rather than only reporting the number of misclassified samples.

E. Experimental Results

In this experiment, the proposed system classifies the programs' peaks and phases into different categories, where the known peaks and phases will be assigned their corresponding names and unknown ones will be labelled as '-1'. Overall, the proposed system has 99.9% and 97.7% successful identification rates for 1040 program phases and 145763 peaks respectively. Additionally, the proposed system identifies unknown programs' peaks that were not in the training set with over 98.4% accuracy. In the following sub-sections, the analyses of the experimental results are categorised by program type.

1) Programs with original function call sequence

In this category, there are total 21 programs, which include 31884 peak samples. Overall, the proposed system has 97.9% successful identification rates for the peaks. Results of accuracy, precision and recall rates for each program are illustrated in Fig. 8.

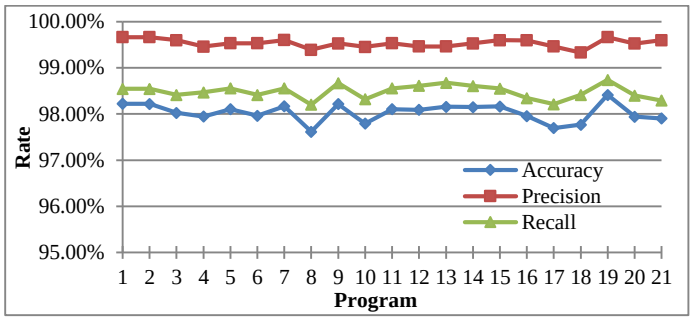

Fig. 8. Results of accuracy, precision and recall rates for category 1.

2) Programs with random generated function call sequence (include only known benchmarks):

In this category, there are 42 programs, which include 57242 peak samples. Overall, the proposed system has 97.1% successful identification rates for the peaks. Results of accuracy, precision and recall rates for each program are illustrated in Fig. 9.

3) Programs with random generated function call sequence (include known and unknown benchmarks):

In this category, there are 41 programs, which include 56637 peak samples. Overall, the proposed system has 97.5% successful identification rates for the peaks. Results of accuracy, precision and recall rates for each program are illustrated in Fig. 10.

In general, as the complexity of the test categories are varied, the first category has the smoothest and best accuracy, precision and recall rates. In contrast, the accuracy, precision and recall rates of the second and third categories are relatively lower, than the first one. Also, the types and the lengths of each tested program in the last two categories are different, which causes the resulting rates of each program have relatively higher variance than the first one.

As indicated in Table I, the database employed mainly consists of seven different benchmarks, where five of them are in the training set and the remainder are not in the training set. Table II summarises the results of each benchmark in terms of accuracy, precision and recall rates.

As can be seen from Table II, the overall performance of the proposed system with validation process is much higher than without the validation process. The reason is that the extra similarity comparisons between original and test statistical tables help the proposed system to re-estimate the results of the SOM analyser. Especially, when there is a unknown benchmark with similar sample peaks to the known benchmark that appears in the test program. The result of

TABLE II
PERFORMANCE RESULTS FOR THE USED BENCHMARKS

Benchmarks	Without Validation			With Validation		
	Accuracy (%)	*Precision* (%)	*Recall* (%)	*Accuracy* (%)	*Precision* (%)	*Recall* (%)
a2time	8.1	94.2	8.1	98.0	99.5	98.5
rspeed	37.7	95.3	38.5	94.9	98.1	96.7
bitmnp	76.2	99.7	76.5	97.6	99.7	97.9
idctrn	87.7	99.8	87.8	98.2	99.8	98.4
puwmod	47.6	98.2	48.1	97.0	99.1	97.8
tblook	98.4	0	0	98.4	0	0
ttsprk	99.1	0	0	99.1	0	0
Overall Performance	68.0	99.3	66.2	97.7	99.5	98.0

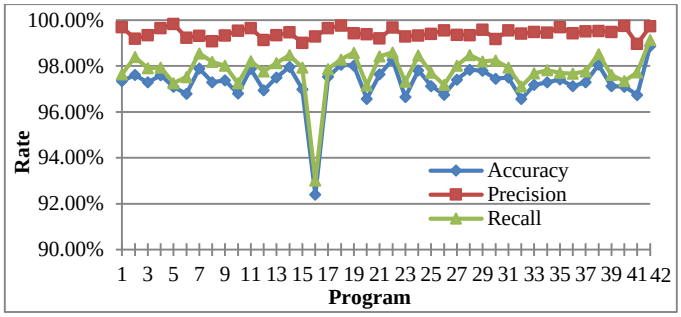

Fig. 9. Results of accuracy, precision and recall rates for category 2.

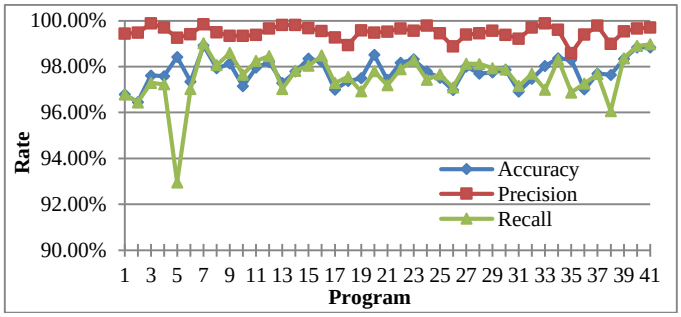

Fig. 10. Results of accuracy, precision and recall rates for category 3.

using validation process is significant higher than without validation process in the system, for example, the accuracy and recall rates of the first benchmark '*a2time*' is significantly lower than by using validation process. The reason is that '*a2time*' and '*tblook*' have very similar distances to the sample SOM node, which cause them to be classified into same cluster. For the known benchmarks, as the test samples are not exactly the same as the samples in the training set, the accuracy and recall rates are also lower, than the result using the validation process. For the unknown benchmarks, the results with and without validation are constant, as there are no positive samples in the sets, the precision and recall rates are zero.

It is worth noting that our work is independent of the processor's architecture or operating system's kernel, thus making it compatible with most modern embedded systems. Hence, the proposed work is particularly suitable for providing possible security solutions to commercial off-the-shelf (COTS) products, where the products have many restrictions on modifying their internal programs or hardware architectures. The proposed system can be run on a non-intrusive debug facility, a non-intrusive infrastructure that is generally used during device software development at present in all production devices, that connects to the targeted embedded device through a debug interface [25], [26], which means that the proposed system would not affect the performance of the monitored embedded system in terms of additional memory and processor usage. When an end user downloads a new program in the embedded device, a training process will start; the new trained parameters of the SOM and the statistic information of monitored program can then be generated and stored in the debug facility, which can only be accessed by the debug facility. The proposed system naturally combines the embedded system's hardware and software together, introducing a new potential direction to secure an embedded device. In one of the authors' previous works [27], an implementation of the conventional SOM on a Xilinx Virtex-4 with 40 neurons required only 22.1% of the available 5,184 Kb Block RAM. The debug facility targeted for our initial on-chip prototyping is utilising a mid-range Xilinx Virtex-6 FPGA having 25,344 Kb (max.) Block RAM; thus a similar implementation should utilise approximately 5% of the available Block RAM. Again, the Virtex-4 design implementation clocked at 25MHz could train with approximately 10,000 patterns per second. As a result of this, the hardware implementation of the SOM produces a significant speed improvement, which is 30 times faster than the original SOM implemented on a state-of-art PC [27]. Hence, the preferred implementation is to follow a hardware acceleration approach that facilitated rapid SOM processing suitable for real-time execution.

VI. Conclusions

In this paper, a Self-Organising Map based approach is proposed to enhance embedded system security by detecting abnormal behaviour, in which features derived from internal embedded processor are extracted and used in the SOM to identify abnormal behaviour in embedded devices. The proposed method can also be combined with ICMetrics system, as different behaviours can be represented with different basic numbers, hence, different encryption keys can be generated by the key cryptography mechanism, using the recall phase. Results achieved in our experiment show that the proposed method can identify unknown behaviours not in the training set with over 98.4% accuracy. The proposed work provides protection at different levels for embedded architectures such as function call sequence, internal control flow and instruction stream within each function. 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. For evaluation of real-time detection system, the proposed method can also be implemented with a soft-core processor on FPGA as part of an on-line protection system, and subsequently halting the program to prevent abnormal behaviours in the system, or even alongside existing debug IP in a direct Systems-on-Chip implementation.

References

[1] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," in *Proceedings Design, Automation and Test in Europe*, 2005, pp. 178-183

[2] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 1295-1308, 2006.

[3] P. Dongara and T. N. Vijaykumar, "Accelerating private-key cryptography via multithreading on symmetric multiprocessors," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2003, pp. 58-69.

[4] G. E. Suh and S. Devadas, "Physical Unclonable Functions for Device Authentication and Secret Key Generation," in *44th ACM/IEEE Design Automation Conference*, 2007, pp. 9-14.

[5] H. Handschuh, G.-J. Schrijen, and P. Tuyls, "Hardware Intrinsic Security from Physically Unclonable Functions," in *Towards Hardware-Intrinsic Security*, A.-R. Sadeghi and D. Naccache, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 39-53.

[6] M.Rahmatian, H. Kooti, I. G. Harris, and E. Bozorgzadeh, "Hardware-Assisted Detection of Malicious Software in Embedded Systems," *IEEE Embedded Systems Letters*, vol. 4, pp. 94-97, 2012.

[7] Shane S. Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Kevin Fu, *et al.*, "WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices," in *Proceedings of USENIX Workshop on Health Information Technologies*, 2013.

[8] D. Garcia-Romero and C. Y. Espy-Wilson, "Automatic acquisition device identification from speech recordings," in *IEEE International Conference onAcoustics Speech and Signal Processing* 2010, pp. 1806-1809.

[9] C. Hanilci, F. Ertas, T. Ertas, and O. Eskidere, "Recognition of Brand and Models of Cell-Phones From Recorded Speech Signals," *IEEE Transactions on Information Forensics and Security* vol. 7, pp. 625-634, 2012.

[10] Y. Kovalchuk, K. D. McDonald-Maier, and G. Howells, "Overview of ICmetrics technology-security infrastructure for autonomous and intelligent healthcare system," *International Journal of u- and e-Sevice, Science and Technology*, vol. 4, pp. 49-60, 2011.

[11] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, and W. Joosen, "A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements," *Requirements Engineering*, vol. 16, pp. 3-32, 2011/03/01 2011.

[12] K. D. Maier, "On-chip debug support for embedded Systems-on-Chip," in *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, 2003, pp. V-565-V-568 vol.5.

[13] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, *et al.*, "Dynamic path-based software watermarking," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004, pp. 107-118.

[14] C. Collberg and C. Thomborson, "Software watermarking: models and dynamic embeddings," presented at the Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Antonio, Texas, USA, 1999.

[15] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," presented at the Proceedings of the 18th conference on USENIX security symposium, Montreal, Canada, 2009.

[16] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," presented at the Proceedings of the 16th ACM conference on Computer and communications security, Chicago, Illinois, USA, 2009.

[17] R. Yang, Z. Qu, and J. Huang, "Detecting digital audio forgeries by checking frame offsets," presented at the Proceedings of the 10th ACM workshop on Multimedia and security, Oxford, United Kingdom, 2008.

[18] Y. Panagakis and C. Kotropoulos, "Telephone handset identification by feature selection and sparse representations," in *IEEE International Workshop on Information Forensics and Security (WIFS)*, 2012, pp. 73-78.

[19] A. Swaminathan, Y. Mao, M. Wu, and K. Kailas, "Data Hiding in Compiled Program Binaries for Enhancing Computer System Performance," in *Information Hiding*. vol. 3727, M. Barni, J. Herrera-Joancomartí, S. Katzenbeisser, and F. Pérez-González, Eds., ed: Springer Berlin Heidelberg, 2005, pp. 357-371.

[20] P. Boufounos and S. Rane, "Secure binary embeddings for privacy preserving nearest neighbors," in *IEEE International Workshop on Information Forensics and Security (WIFS)*, 2011, pp. 1-6.

[21] T. Kohonen, "Learning vector quantization," in *The handbook of brain theory and neural networks*, A. A. Michael, Ed., ed: MIT Press, 1998, pp. 537-540.

[22] STMicroelectronics. *STM32F207IG Data Sheet*. Available: http://www.st.com/internet/mcu/product/245085.jsp

[23] KEIL. *Keil µVision IDE Data Sheet*. Available: http://www.keil.com/uvision/

[24] The Embedded Microprocessor Benchmark Consortium (EEMBC). (2013). *AutoBench™ 1.1 Benchmark Software*. Available: http://www.eembc.org/benchmark/automotive_sl.php

[25] A. Hopkins and K. D. McDonald-Maier, "Debug Support Strategy for Systems-on-Chips with Multiple Processor Cores," IEEE Transactions on Computers, vol. 55, issue 2, pp. 174-184, 2006.

[26] A. Hopkins and K. D. McDonald-Maier, "Debug Support for Complex Systems-on-Chip: A Review," IEE Proceedings-Computers and Digital Techniques, vol. 153, no. 4, pp. 197-207, 2006.

[27] K. Appiah, A. Hunter, P. Dickinson and H. Meng, "Implementation and Applications of Tri-State Self-Organizing Maps on FPGA," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 8, pp. 1150-1160, 2012.

Xiaojun Zhai received a BSc degree in Communication Engineering from North China University of Technology, Beijing, China, in 2006 and an MSc degree in Embedded Intelligent Systems from the University of Hertfordshire, U.K. in 2009. After that, he completed his PhD degree in Electronic and Electrical Engineering at the University of Hertfordshire, U.K., in 2013, where he also worked on part-time basis as a visiting lecturer until February 2013. After completing the PhD, he joined the Embedded and Intelligent Systems (EIS) Research Group at the University of Essex in Colchester as a Senior Research Officer in January 2013. He is currently appointed as a research associate in the Embedded Systems and Communications Research Group at the University of Leicester,

U.K. His research interests mainly include Custom Computing using FPGAs, Embedded Systems, System-on-Chip (SoC) design, Real-time Image Processing, Intelligent Systems, and Neural Networks.

Kofi Appiah (M'04) received the MSc degree in Computer Science from University of Oxford, UK and an MSc in Electronic Engineering degree from Royal Institute of Technology, Sweden. He completed his PhD at University of Lincoln, UK in 2010, where he also worked on three research projects as a Research Assistant. He joined the Embedded and Intelligent Systems (EIS) Research Group at University of Essex in Colchester as Senior Research Officer in December 2012. Kofi also worked on part-time basis as a Development Engineer with Metrarc Ltd, Cambridge, before joining NTU in November 2013 as a lecturer. His current research interests include embedded computer vision and security, and highly parallel software/hardware architectures.

Shoaib Ehsan received his BSc Electrical Engineering degree from the University of Engineering and Technology, Taxila, Pakistan in 2003. He completed his PhD in Computing and Electronic Systems (with specialization in computer vision) at the University of Essex, Colchester, UK in 2012. He has extensive industrial and academic experience in the areas of embedded systems, embedded software design, computer vision and image processing. His current research interests are in intrusion detection for embedded systems, local feature detection and description techniques, image feature matching and performance analysis of vision systems. He is a recipient of the University of Essex Post Graduate Research Scholarship and the Overseas Research Student Scholarship. He is a winner of the prestigious Sullivan Doctoral Thesis Prize awarded annually by the British Machine Vision Association.

Dr. Gareth Howells is a Reader in Secure Electronic Systems at the University of Kent,UK. He has been involved in research relating to security, biometrics and pattern classification techniques for over twenty five years and has been instrumental in the development of novel ICMetric based security technology deriving secure encryption keys from the operating characteristics of digital systems. He has been awarded, either individually or jointly, several major research grants relating to the pattern classification and security fields, publishing over 160 papers in the technical literature. Recent work has been directed towards the development of secure device authentication systems which has received significant funding from several funding bodies and is currently in the process of being commercially exploited.

Huosheng Hu is Professor in the School of Computer Science and Electronic Engineering at the University of Essex, leading the robotics research. He received the MSc degree in industrial automation from the Central South University in China and the PhD degree in robotics from the University of Oxford in the U.K. His research interests include behaviour-based robotics, human-robot interaction, service robots, embedded systems, data fusion, learning algorithms, mechatronics, and pervasive computing. He has published over 450 papers in journals, books and conferences in these areas, and received a number of best paper awards. Prof. Hu is a Fellow of Institute of Engineering & Technology, a Fellow of Institute of Measurement & Control, a senior member of IEEE, a founding member of IEEE Robotics & Automation Society Technical committee on Networked Robots. He has been a Program Chair or a member of Advisory/Organising Committee for many international conferences such as IEEE IROS, ICRA, ICMA, ROBIO, and IASTED RA and CA conferences. He is currently Editor-in-Chief for International Journal of Automation and Computing, Editor-in-Chief for Robotics Journal, and Executive Editor for International Journal of Mechatronics & Automation.

Dongbing Gu received the B.Sc. and M.Sc. degrees in control engineering from the Beijing Institute of Technology, Beijing, China, and the Ph.D. degree in robotics from the University of Essex, Essex, UK. He was an Academic Visiting Scholar with the Department of Engineering Science, University of Oxford, Oxford, UK, from October 1996 to October 1997. In 2000, he joined the University of Essex as a Lecturer. Currently, he is a Professor with the School of Computer Science and Electronic Engineering, University of Essex. His current research interests include multiagent systems, wireless sensor networks, distributed control algorithms, distributed information fusion, cooperative control, reinforcement learning, fuzzy logic and neural network-based motion control, and model predictive control.

Klaus D. McDonald-Maier received Dipl.-Ing. and MS degrees in electrical engineering from the University of Ulm, Germany, and the École Supérieur de Chimie Physique Électronique de Lyon, France, in 1995, respectively. In 1999, he received a doctorate in computer science from the Friedrich Schiller University, Jena, Germany. Klaus McDonald-Maier worked as a systems architect on reusable microcontroller cores and modules at Infineon Technologies AG's Cores and Modules Division in Munich, Germany and as a lecturer in electronic engineering at the University of Kent, Canterbury, United Kingdom. In 2005, he joined the University of Essex, Colchester, United Kingdom, where he is a Professor in the School of Computer Science and Electronic Engineering. His current research interests include embedded systems and system-on-chip (SoC) design, security, development support and technology, parallel and energy efficient architectures and the application of soft computing and image processing techniques for real world problems. He is a member of the VDE and the BCS, a senior member of the IEEE and a Fellow of the IET.