The B-Side of Side Channel Leakage: Control Flow Security in Embedded Systems

Mehari Msgna, Konstantinos Markantonakis, and Keith Mayes

Smart Card Centre, Information Security Group,
Royal Holloway, University of London,
Egham, TW20 0EX, UK
{mehari.msgna.2011,k.markantonakis,k.mayes}@rhul.ac.uk

Abstract. The security of an embedded system is often compromised when a "trusted" program is subverted to behave differently. Such as executing maliciously crafted code and/or skipping legitimate parts of a "trusted" program. Several countermeasures have been proposed in the literature to counteract these behavioural changes of a program. A common underlying theme in most of them is to define security policies at the lower level of the system in an independent manner and then check for security violations either statically or dynamically at runtime. In this paper we propose a novel method that verifies a program's behaviour, such as the control flow, by using the device's side channel leakage.

Keywords: Side Channel Leakage, Power Consumption, Program's Control Flow, Hidden Markov Model, Principal Components Analysis, Linear Discriminant Analysis.

1 Introduction

In recent years, embedded systems have been proliferated into wide range of modern life applications. One of the main application vector of embedded systems is communication [1,2,3]. A typical embedded system application contains hardware and software components. The hardware component includes storage area, execution engine and other peripherals required to successfully execute instructions. The software component is a written procedures or rules stored in a memory pertaining to the operation of a computer system or part of the system itself.

The execution of a software program always involves incrementing the program counter (a special register which stores the address of the next instruction). Normally the program counter is incremented by "1"; however, certain instructions change its value by more than one in both directions. This kind of change is known as *Control Flow Change* and can be caused by both conditional and unconditional branching instructions. According to [4], program control flow is the most attacked target in software and such attacks are called *Control Flow Attacks*. A *Control Flow Attack* is one of the main threats for embedded systems [5,6,7]. *Control Flow Attacks* can be performed on embedded systems for

T. Zia et al. (Eds.): SecureComm 2013, LNICST 127, pp. 288–304, 2013.

two reasons. Firstly, the attacker installs his code segment on the target device. Later on when the device executes a genuine program, the attacker targets saved function return addresses to divert the control flow into his previously installed code. Secondly, the attacker does not install any code but instead when the program is executed the attacker changes the saved return addresses just in order to skip the execution of certain parts of the program.

In the literature, several countermeasures have been proposed to counteract these kinds of intrusions. To explain some of them; in [8], the authors discuss a technique that employs a dedicated hardware module to detect and prevent unintended program behaviors. In this method the program's properties are extracted through a static code analysis and the hardware module uses them to enforce a permissible program behavior at runtime. Another countermeasure, described in [9] introduces *Control-Flow Integrity (CFI)* enforcement. The CFI dictates that software execution must follow the path of a *Control-Flow Graph (CFG)* determined ahead of time. The work of Michael Frantzen and Michael Shuey [10], presents a buffer overflow prevention method. This is acheived via a kernel modification that performs transparent, automatic and atomic operations on the function return addresses before they are written into the stack and before the program transfers execution back to the saved return addresses. In [11], Aurélien et al. discussed a control flow enforcement technique based on Instruction Based Memory Access Control (IBMAC). This is done by using a simple hardware modification to divide the stack into a data and a control flow stack (or return stack). Moreover, access to the control flow stack is restricted only to return and call instructions, which prevents control flow manipulation. More countermeasures can be found in [12,13,14]. Most of the proposed countermeasures are demanding in terms of computational capability, memory usage and often rely on a hardware module that is not present on simple devices.

In this paper we present a novel approach to verify a program's control flow by using the device's side channel leakage. In our proposal we modelled the device as a *Markov Process* with hidden states, each state belonging to a part of the program. Then a verifying device extracts the control flow transition that the device had followed when executing the program from its side channel leakage (power consumption). This extracted control flow (state sequence) is then verified against a list of valid state transitions of the application which was calculated ahead of time.

The rest of the paper is structured as follows. Section 2 briefly provides background information on side channel leakage. Section 3 discusses the proposed control flow verification methodology. Section 4 discusses our experimental results. Finally, section 5 concludes the paper.

2 Side Channel Leakage

Side channel leakage is information revealed by a device about its internal state while processing a certain procedure. Smart cards and other embedded devices use electric current to turn transistors on and off. The instantaneous electric

current that the device consumes depends on how many transistors that the executed instructions and data turn on and off. This difference in the electric current is then reflected in the power consumption and electromagnetic emission of the device. The power consumption and/or electromagnetic emission can then be recorded and analysed to extract secret information from the target device.

In the context of cryptology, side channel leakage can be employed in retrieving cryptographic secret keys from target devices, such as smart cards. Side channel information such as timing [15,16,17], power consumption [18,19,20] and electromagnetic emission [21,22,23] have been used in attacking implementations of cryptographic algorithms including AES [24], DES [25] and RSA [26].

Besides extracting cryptographic keys, side channel information has also been used to reverse engineer embedded device applications [27,28,29]. This is done by constructing a power consumption template of the target device using an identical reference device. Then use the templates to recognise executed instructions from the target device's power consumption waveform. In addition, side channel information can also be used by device manufacturers and application developers to detect cloned devices and design advanced applications. Instruction-level power consumption model of an embedded device has been used to design a low-power consuming applications for mobile embedded devices where batteries are the main power source [30,31]. In [32], the authors discuss, theoretically, how side channel leakage can be used to fingerprint a smart card platform and then use it later to detect cloned cards.

3 Control Flow Verification

An application is a combination of basic blocks. A basic block is a linear sequence of executable instructions with only one entry point (the first instruction executed) and one exit point (the last instruction executed) [33]. After executing one basic block the processor jumps into another basic block based on the branching instruction executed at the end of the current basic block. This branching instruction can be conditional or unconditional. A basic block may have many predecessors and many successors. It might also be its own successor. Program entry basic blocks might not have predecessors that are within the program and program ending basic blocks never have successors within the program itself.

An embedded device, with one or two programs installed in its non-volatile memory, can be modelled as a state machine with each state corresponding to a basic block of the program(s). When the program is being executed we can not directly observe the states that the processor is going through but we can observe the side channel information emitted by the device. Such information can be the power consumption [18,34] or the electro-magnetic emission [22,21,23] of the device. The side channel information emitted by the device is directly dependent on the states executed by the processor.

The questions here are, by only using this observable physical emission can we reconstruct the state sequence that the processor went through when executing the program? Furthermore, once the sequence is reconstructed can we verify it?

3.1 Control Flow Reconstruction

To reconstruct the state sequence that a device followed during the execution of a program from its side channel leakage we modelled the device as a *Hidden Markov Model (HMM)* [35,36]. A *Markov Model* is a memoryless system with a finite number of hidden states. It is called memoryless because the next state depends only on the current state.

In such a model the states are not directly observable. However, there has to be (at least) one observable output of the process that reveals partial information about the state sequence that the device has followed. Fig. 1, illustrates a *Markov Process* with five hidden states (i.e A to E).

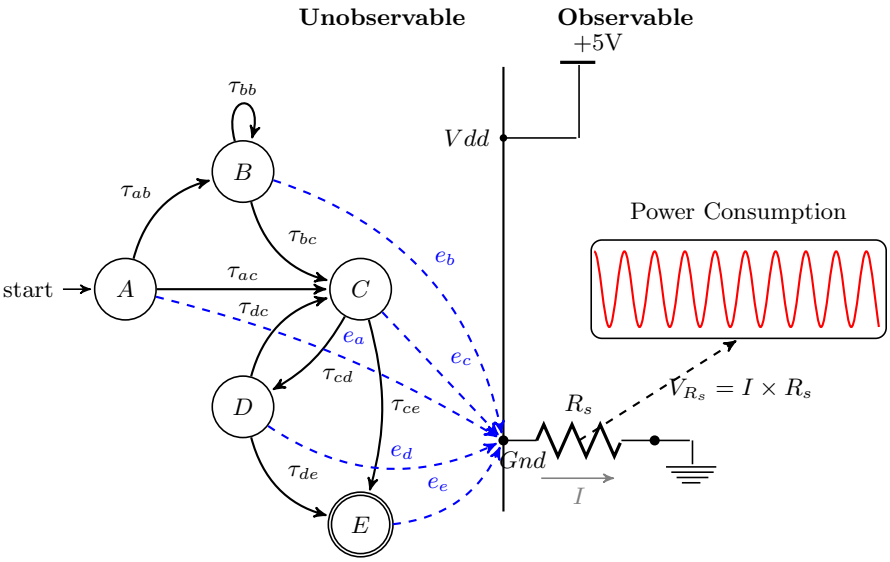

Fig. 1. A Markov model representing a device executing a program with five states (A, B, C, D and E). The power consumption is the observable output that reveals partial information about the state sequence of the device.

In case of the *Markov Process* illustrated in Fig. 1, the hidden states are the program's basic blocks and the observable output is the power consumption of the device. This observable output is measured via a resistor (R_s) connecting the ground pin of the device and ground pin of the voltage source.

Building the Hidden Markov Model. Building a *Hidden Markov Model (HMM)* requires a set of finite states q_i's, a transition probability distribution matrix $\mathbf{T} = \{\tau_{ij}\}$, emission probability distribution matrix $\mathbf{E} = \{e_i\}$ and initial state distribution $\boldsymbol{\pi}$. Given these probability distribution matrices, the HMM is defined as $\lambda = (\mathbf{T}, \mathbf{E}, \boldsymbol{\pi})$.

The transition probability distribution τ_{ij}, is the probability that the next state is q_j if the current state is q_i, for $1 \leq i, j \leq S$ where S is the number of states. If we denote s_t the current state of the system at a time t, $\tau_{ij} = \mathcal{P}(s_{t+1} = q_j \mid s_t = q_i)$ is the probability of transitioning from state q_i to state q_j. Given an observation (power consumption) $\mathcal{O}_t$ at a time t, the emission probability distribution $e_i(\mathcal{O}_t) = \mathcal{P}(\mathcal{O}_t \mid s_t = q_i)$ is the probability that $\mathcal{O}_t$ was emitted by the state q_i. To compute $e_i(\mathcal{O}_t)$ first we need to build a power consumption template for each state. The template of each state is generated by computing the mean, μ_{q_i}, and the covariance, σ_{q_i} of the state's power consumption traces.

Let us consider N L-dimensional power consumption traces $\{x_n\}$ generated by the device while executing the state q_i were recorded. The mean, μ_{q_i}, and covariance, σ_{q_i}, are calculated using the computation in equations (1) and (2) respectively.

$$\mu_{q_i} = \frac{1}{N} \sum_{n=1}^{N} x_n \tag{1}$$

$$\sigma_{q_i} = \frac{1}{N} \sum_{n=1}^{N} (x_n - \mu_{q_i})(x_n - \mu_{q_i})^T \tag{2}$$

where N is the number of power traces recorded for state q_i and $(x_n - \mu_{q_i})^T$ is the transpose of $(x_n - \mu_{q_i})$. These templates can be built beforehand using an identical reference device and a target program. Assuming the power traces are derived from a *Multivariate Gaussian Normal Distribution Model* [37], the emission probability distribution $e_i(\mathcal{O}_t)$ is computed using the equation in (3).

$$e_i(\mathcal{O}_t) = \frac{1}{(2\pi)^{L/2}\sqrt{\sigma_{q_i}}} \exp(-\frac{1}{2}(\mathcal{O}_t - \mu_{q_i})\sigma_{q_i}^{-1}(\mathcal{O}_t - \mu_{q_i})^T) \tag{3}$$

Now, if we take a number of observations $\mathcal{O} = \{\mathcal{O}_t, \mathcal{O}_{t+1}, \mathcal{O}_{t+2}, \cdots, \mathcal{O}_{t+n}\}$, the emission probability distribution matrix $\mathbf{E}$ becomes:

$$\mathbf{E} = \begin{bmatrix} e_1(\mathcal{O}_t) & e_1(\mathcal{O}_{t+1}) & e_1(\mathcal{O}_{t+2}) & \cdots & e_1(\mathcal{O}_{t+n}) \\ e_2(\mathcal{O}_t) & e_2(\mathcal{O}_{t+1}) & e_2(\mathcal{O}_{t+2}) & \cdots & e_2(\mathcal{O}_{t+n}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ e_S(\mathcal{O}_t) & e_S(\mathcal{O}_{t+1}) & e_S(\mathcal{O}_{t+2}) & \cdots & e_S(\mathcal{O}_{t+n}) \end{bmatrix} \tag{4}$$

Normally when an application is invoked, the execution always starts at the program entry point (*main()*). Therefore, the initial state distribution for the first basic block is always 1 and 0 for the other basic blocks. For example, for the system depicted in Fig. 1 the execution of the application always starts at A. So, the initial state distribution becomes $\pi_A = 1$ and $\{\pi_B, \pi_C, \pi_D, \pi_E\} = 0$.

To successfully compute $\mathbf{E}$ using equation (3), all observations $\{\mathcal{O}_t, \cdots, \mathcal{O}_{t+n}\}$ must have equal dimensionality. In other words, the power consumption traces generated by all states must have the same number of sample points. However, in reality this may not always be true. In addition, the dimension of the emissions

(power traces) may be too large for a robust and fast classification. A common way to attempt to resolve this problem is to use a dimensionality reduction technique. In doing this we have to maintain as much information about the original emission (power consumption) as possible. Two of the most popular techniques for this purpose are: *Principal Components Analysis (PCA)* and *Fisher's Linear Discriminant Analysis (F-LDA)*.

Principal Components Analysis (PCA) is a technique used to reduce the dimension of an observation while keeping as much of its variance as possible [38]. This is achieved by orthogonally projecting the observation onto a lower dimensional subspace vector.

Let us consider an N L-dimensional observations of emissions $\{x_n\}$, where $n = 1, ..., N$ and their covariance matrix σ. A lower dimensional subspace in this Euclidean space can be defined by a D-dimensional unit vector $\overrightarrow{u_1}$, where $D < L$. The projection of each observation, x_n, onto that subspace is given by $\overrightarrow{u_1}^T x_n$. Now if we stack up all the emissions into a matrix of $N \times L$ matrix, where L is the number of samples of each observation, the projection of each row of the matrix is represented as $U^T X$, where U is a matrix of *eigenvectors* of the covariance matrix σ. The projection of the observations onto a D-dimensional subspace that maximizes the projected variance is given by D *eigenvectors* [39] $\overrightarrow{u_1}, \ldots, \overrightarrow{u_d}$ with the D largest *eigenvalues* $\lambda_1, \ldots, \lambda_d$.

Fisher's Linear Discriminant Analysis (F-LDA) is a method used in statistics, pattern recognition and machine learning to find a linear combination of features which characterises two or more class observations [40,41,42]. The resulting combination may be used as a linear classifier for dimensionality reduction before classification. However, instead of maximising the variance of the original data like PCA, information regarding the covariance of different classes is taken into consideration. These are the "between-class" and "within-class" covariance matrices.

Now, let us consider again the N L-dimensional observations for each class. Then the "within-class" covariance σ_W is computed as,

$$\sigma_W = \sum_{i=1}^{S} \sum_{w \in x_i} (w - \mu_{q_i})(w - \mu_{q_i})^T = \sum_{i=1}^{S} N_{q_i} \sigma_{q_i} \tag{5}$$

In the above equation, N_{q_i}, σ_{q_i} and w are the number of observations, the covariance and the power traces of class q_i. The "between-class" covariance σ_B is computed as

$$\sigma_B = \sum_{i=1}^{S} (\mu_{q_i} - \mu)(\mu_{q_i} - \mu)^T \tag{6}$$

where μ_{q_i} is the individual class's mean as defined in equation (1) and μ is the mean of the entire observation which is computed as shown in equation (7).

$$\mu = \frac{1}{N}\sum_{\forall x} x = \frac{1}{N}\sum_{i=1}^{S} N_{q_i}\mu_{q_i} \tag{7}$$

Now, let us consider a D-dimensional unit vector $\overrightarrow{u_1}$ onto which the data is projected. This time the objective is to maximise both the projected "between-class" and the projected "within-class" covariance:

$$\mathcal{J}(\overrightarrow{u_1}) = \frac{\overrightarrow{u_1}^T \sigma_B \overrightarrow{u_1}}{\overrightarrow{u_1}^T \sigma_W \overrightarrow{u_1}} \tag{8}$$

The projected $\mathcal{J}$ is maximised if $\overrightarrow{u_1}$ is the *eigenvector* of $\sigma_W^{-1}\sigma_B$. The D-dimensional subspace is created by the first D orthogonal directions that maximise the projected $\mathcal{J}$. These are given by the D *eigenvectors* $\overrightarrow{u_1}, \cdots, \overrightarrow{u_D}$ of $\sigma_W^{-1}\sigma_B$ with the largest *eigenvalues* $\lambda_1, \cdots, \lambda_D$.

Calculating the Most Probable State Sequence. The probability distribution matrices $\mathbf{E}$, $\mathbf{T}$ and $\boldsymbol{\pi}$ can be constructed ahead of time using an identical reference device and the target application. Now let us consider we observe emissions (power consumption traces) $\mathcal{O} = \{\mathcal{O}_t, \mathcal{O}_{t+1}, \mathcal{O}_{t+2}, \cdots, \mathcal{O}_{t+n}\}$, where n is the length of the state sequence. These emissions were recorded while the device was executing the target application. The most likely sequence of states that produces the observations $\mathcal{O}$ is calculated using the *Viterbi Algorithm* [43] as shown in equations (9) and (10). This state sequence is regarded as the control flow that the device has followed when executing the program.

$$\mathcal{V}_{1,j} = \mathcal{P}(\mathcal{O}_1 \mid s_1 = q_j) \cdot \pi_j \tag{9}$$

$$\mathcal{V}_{t,j} = \mathcal{P}(\mathcal{O}_t \mid s_t = q_j) \cdot max_{i \in S}(\tau_{ij} \cdot \mathcal{V}_{t-1,j}) \tag{10}$$

In equation (10), S is the state space of the *Markov Process*, π_j the probability of state q_j being the initial state and τ_{ij} probability of transitioning from state q_i to state q_j. The $\mathcal{V}_{t,j}$ is the probability of the most probable state sequence responsible for the first t emissions that has q_j as its final state. The state sequence that resulted in highest probability, according to equation (10), from all possible state sequences of the same length as the emission is regarded as the most probable state sequence that generated the emissions.

3.2 Verifying the Reconstructed State Sequence

As described in section 3, a program is a combination of basic blocks. Before loading the program into the target device, a list of valid transitions between the states (basic blocks) are extracted using a code analysis tool. This list of valid transitions is known as the *Control Flow Graph (CFG)*. A CFG, $G = (I, P)$, is represented by the program's states identity, I, and control flow path, P. For instance, for the program illustrated in Fig. 1, the CFG is given as $G = (I, P)$,

where $I = \{A, B, C, D, E\}$ and $P = \{(A, B), (A, C), (B, B), (B, C), (C, D), (C, E), (D, C), (D, E)\}$. The CFG is then installed into the verifying device (i.e the terminal in the case of a smart card application).

Now the task is verifying if the reconstructed state sequence is among the valid transitions in the CFG. However, since the reconstruction of the state sequence (explained in section 3.1) from the power consumption is a probabilistic process, we have to first confirm that the reconstructed state sequence is the actual state sequence that the device followed when executing the program. This can be acheived by comparing a hash value generated over the identity of actually executed states (H^*) with a hash value generated over the identity of reconstructed states from the power trace ($H^{'}$). In equation (11), H^* is generated by the device that executes the program and sent to the verifying device that generates $H^{'}$.

$$f(H^*, H^{'}) = \begin{cases} 1, & \text{if } H^* = H^{'} \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

If it is a match, the reconstructed sequence is what the processor went through when executing the program. Otherwise, the reconstructed sequence is not the path that was followed by the device. Equation (11) can only verify that the execute state sequence and the extracted state sequence are the same. Unfortunately, this does not verify if the executed state sequence (control flow) is valid. Therefore, the validity of the control flow is verified by comparing it against the pre-calculated paths, P, in CFG. If the reconstructed state sequence is not among the valid paths in CFG, the device/program is regarded as compromised.

4 Experimental Results

To implement the techniques discussed above we chose *ATMega163 + 24C256* based smart card. ATMega163 is an 8-bit microcontroller based on AVR architecture. Note that this smart card does not have any countermeasure against power analysis attacks. To construct a more reliable template for the states of the test program (see Fig (3)), we removed all other factors that influence the power consumption of the device. Such factors can be the intrinsic and ambient noise introduced by the measurement setup. To minimise the influence of the ambient noise, we have properly warmed up the measurement equipment beforehand so that it is all running at a uniform temperature during the power trace collection phase. This requires running few test measurements to be discarded before the actual power trace collection starts. The intrinsic noise introduced into the measurement can be minimised by collecting several traces for each state and calculating the mean. This reduces the standard deviation of the noise by a factor of $\sqrt{n}$, given that n is the number of power traces involved in calculating the mean.

The power consumption is measured as a voltage drop across a resistor connecting the ground pin of the smart card and the ground pin of the voltage

source. The smart card is running at a 4 MHz clock cycle and is powered up by a +5V supply from the reader. The measurements are performed using a *LeCroy WaveRunner 6100A* [44] oscilloscope capable of measuring traces at a rate of 5 billion samples per second (5GS/s). The shunt resistor is connected with the oscilloscope using a *Pomona 6069A* [45] *probe*, a 1.2m co-axial cable with a 250MHz bandwidth, 10MΩ input resistance and 10pf input capacitance. All measurements are sampled at a rate of 500 MS/s and the same setup is used throughout the experiment.

4.1 Control Flow Reconstruction

For our experiment we implemented a test application with five basic blocks (states). Each state accomplishes certain task within the program. The processor follows different control flow paths to execute the application depending on a value "V_{reader}" sent from a terminal. The state machine diagram of the test application is presented in the Fig. 2.

Fig. 2. Test program's control flow diagram

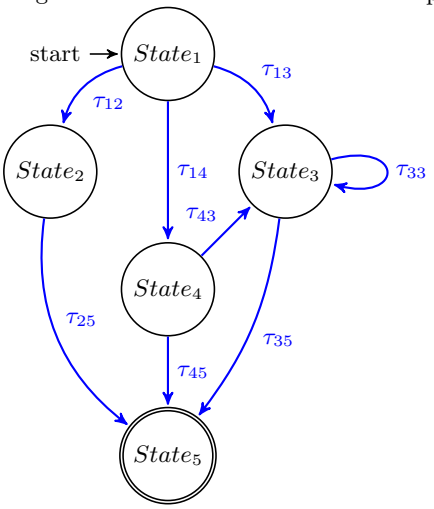

Fig. 3. High-level description of the test program

```
State1: Par = receive()
        Vreader = receive()
        Vnvm = read(nvm)
        if(Vreader == Vnvm)
State2:    par = (par)^2
           goto State5
        end
        else if(Vreader > Vnvm)
State4:    par = par + 216
           par = par/5
           Vreader = Vreader - 2
           if(Vreader < Vnvm)
              goto State3
           end
           else
              goto State5
           end
        end
        else if(Vreader < Vnvm)
State3:    par = par * 2
           par = par - 129
           Vreader = Vreader + 1
           if (Vreader < Vnvm)
              goto State3
           end
           else
              goto State5
           end
        end
State5: clear_registers
        clear_memory
```

Invoking the test program requires passing two arguments: "V_{reader}" ($0 \leq V_{reader} \leq 9$) and "Par" ($0 \leq Par \leq 255$). The "V_{reader}" is compared with a reference value "V_{nvm}" ($0 \leq V_{nvm} \leq 9$) (stored in the non-volatile memory of the

smart card) before changing a state. For our experiment the V_{nvm} is initialised to "4" and the arguments Par and V_{reader} are randomly generated and passed to the program through the smart card reader.

Building the Hidden Markov Model. As discussed in Section 3.1, building a *Hidden Markov Model* requires the initial probability distribution $\boldsymbol{\pi}$, transition probability distribution $\mathbf{T}$ and the emission probability distribution $\mathbf{E}$. As illustrated in Fig. 2, the execution of the test program always starts at $State_1$. Therefore, the probability of $State_1$ being the initial state is "1", and "0" for all other states. If π_i is the probability of $State_i$ being the initial state in the execution of the program, the initial probability distribution vector of our test program is given as:

$$\boldsymbol{\pi} = \{\, \pi_1 = 1, \pi_2 = 0, \pi_3 = 0, \pi_4 = 0, \pi_5 = 0 \,\} \tag{12}$$

To compute the transition probability distribution matrix, $\mathbf{T}$, we invoked the program with a randomly generated "Par" and all possible values (i.e. 0 to 9) of "V_{reader}" and record the control-flow transition of the program. Note that for each different value of "V_{nvm}" the matrix $\mathbf{T}$ is different.

Table 1. Transition probability distribution of the program illustrated in Fig. 2 and 3. The columns represent next states and the rows represent current states.

Transition from	**Transition to** [%]				
	$State_1$	$State_2$	$State_3$	$State_4$	$State_5$
$State_1$	τ_{11}=0	τ_{12}=0.1	τ_{13}=0.4	τ_{14}=0.5	τ_{15}=0
$State_2$	τ_{21}=0	τ_{22}=0	τ_{23}=0	τ_{24}=0	τ_{25}=1
$State_3$	τ_{31}=0	τ_{32}=0	τ_{33}=0.55	τ_{34}=0	τ_{35}=0.45
$State_4$	τ_{41}=0	τ_{42}=0	τ_{43}=0.2	τ_{44}=0	τ_{45}=0.8
$State_5$	τ_{51}=0	τ_{52}=0	τ_{53}=0	τ_{54}=0	τ_{55}=0

To compute the emission probability distribution matrix $\mathbf{E}$, we collected 1000 traces for each state. Using these traces we computed the mean μ_{q_i}, and covariance, σ_{q_i}, for each state as a template.

As shown in figure 4, the states of the test application generate power consumption traces of different dimension. In our experiment $\mathbf{E}$ is computed over the first 250 sample points of the traces. However, a covariance matrix of 250×250 is still too large to compute its inverse. For this purpose we applied the techniques discussed in Section 3.1 (PCA and F-LDA) on the first 250 sample points of the state emission (power consumption) before computing $\mathbf{E}$.

Principal Components Analysis (PCA) is used to find a subspace whose basis vectors corresponding to the maximum variance directions in the original data. In other words PCA searches for those vectors in the underlying data that best describes the data. When applying *PCA* the dimensionality of the

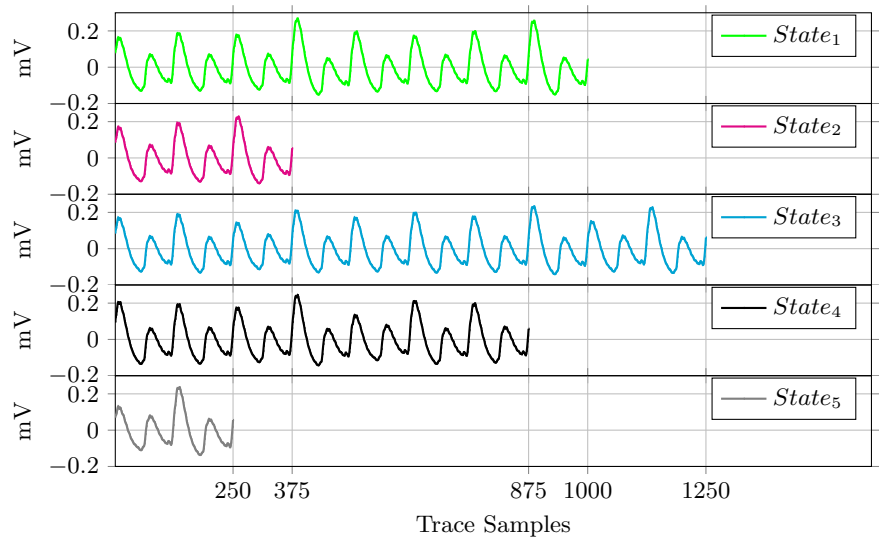

Fig. 4. Mean of the power traces of the states illustrated in Fig. 2

projected data has to be selected carefully. On the one hand, if it is too small, too much of variance of the original data may get lost and with it important information about the state emissions. On the other hand, if it is too large, the state classification becomes less reliable again. This might be because of the bad conditioning of large covariance matrix. Another reason can be, as the dimension increase the class emission cross-correlation increases. Therefore, when choosing the dimensionality for the projected data we have to decide how much of variance of the original data that we can afford to lose.

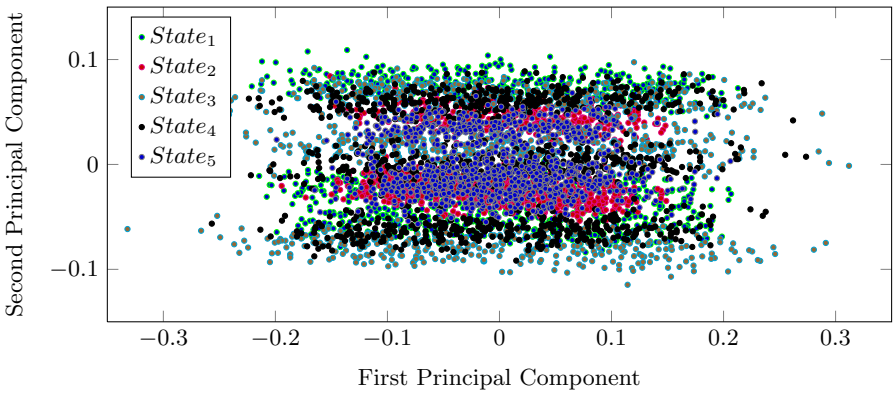

Fig. 5. Original data after PCA

For example, in our experiment the first 100 principal components were accounted for 54.76% of the variance of the original emission. the first 250 principal components are accounted for 80.74% of the variance of the original emission. In Fig.5 we show plots of the first two principal components after PCA.

Fisher's Linear Discriminant Analysis (F-LDA) is a technique used to classify between classes by finding discriminant features of the class data and projecting them onto these discriminant vectors. In other words, F-LDA searches for those vectors in the underlying data that best separates among the classes.

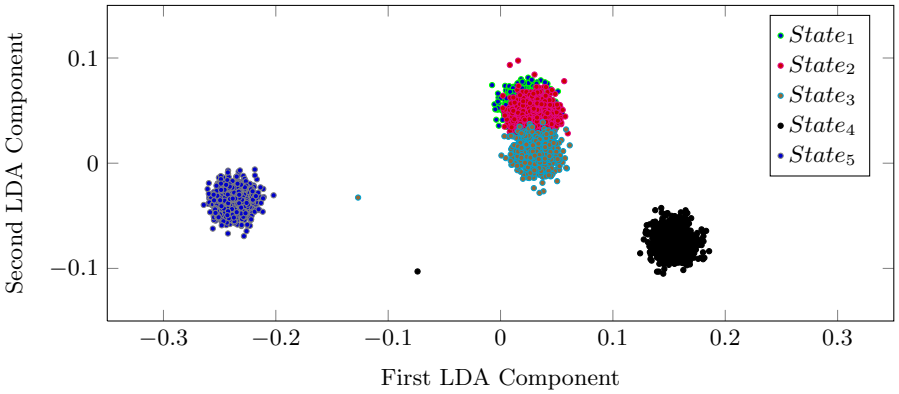

Fig. 6. Original data after F-LDA

In Fig. 6 we present the first two components of the state emissions after F-LDA. As discussed earlier PCA searches for vectors that best describes the original data. However, it does not take the other classes into consideration. For this reason PCA may not produce a satisfactory result when classifying different classes. We can see that in Fig. 5 the principal components of classes emissions overlap. However, as shown in Fig. 6 the classes are better separated after F-LDA.

Calculating the Most Probable State Sequence. To calculate the most probable state sequence, first we have to implement the *Viterbi Algorithm* discussed in Section 3.1. To do this we have two options: use the MATLAB [46] Statistics Toolbox implementation *hmmviterbi*[47] or create our own implementation of the equations (9) and (10). Although, the MATLAB Statistics Toolbox implementation of *Viterbi Algorithm* might be useful for some statistical calculations we could not use it in our experiment. This was because firstly it does not utilise the initial probability distribution ($\boldsymbol{\pi}$) and secondly the output is not in the format that we want it to be. Therefore, we created our own MATLAB implementation and the source code is available at the end of the paper in Appendix A. As you can see it from the source code, our implementation takes all

three matrices ($\boldsymbol{\pi}$, $\mathbf{E}$ and $\mathbf{T}$) that we discussed in Section 3 and gives us the most likely state sequence as a vector.

Our test program has six valid control-flow paths from the initial state, $state_1$, to the final state, $state_5$. Our implementation of the *Viterbi algorithm* calculates a sequence of states with the highest probability of generating the emission $\mathcal{O}$. We ran the test program for all possible valid paths by varying the argument "V_{reader}" and calculated the most probable state sequence from the smart cards power consumption trace. We ran the test program 1000 times by varying "V_{Reader}", recorded the power trace and calculated the most likely sequence of states for each run.

4.2 Verifying the Reconstructed State Sequence

For all the state sequences that we calculated, we verified them using the 2-step verification system discussed in Section 3.2. Before comparing the reconstructed state sequence against the CFG, we have to make sure that the reconstructed sequence is the actual path that the smart card went through. For that purpose we verified the hash values calculated by the smart card against the hash values calculated over the reconstructed state sequence. Then we compared the reconstructed state sequence against the valid paths in CFG. In our experiment we successfully verified the control flow for all (1000) runs of the test program that we made. In our experiment we calculated the CFG manually; however, for large programs calculating it manually might be difficult and complicated. In such a case the CFG may be extracted using a source code analysis tools, such as MALPAS [48].

5 Conclusion

In the literature several methods have been proposed to counteract a program's control flow violation. In most of them the proposed solutions require either a dedicated hardware module or the main processor to perform extra computations to check the control flow security of the program(s) at runtime. Usually this computation utilises the program's properties which are extracted ahead of time, such as CFG. These properties are then used to check the program's behaviour dynamically. However, these kind of solutions may not be suitable for low-end devices deployed as coprocessors in bigger systems, such as hardware security modules in communication devices.

In this paper we proposed a novel approach into checking a program's control flow integrity by using the side channel leakage of the target device. In our method the device is not required to perform extra computation. However, it requires another device to check for its program's control flow integrity as it executes the program. This method can be used in smart card (or any other embedded device that need to connect to an external device to execute the application) where the terminal (external device) acts as the verifying device.

References

1. Feng, A., Knieser, M., Rizkalla, M.E., King, B., Salama, P., Bowen, F.: Embedded system for sensor communication and security. IET Information Security 6(2), 111–121 (2012)
2. Shoufan, A.: A hardware security module for quadrotor communication. In: International Conference on Field-Programmable Technology (FPT), December 10-12, 2012, pp. 253–256. IEEE (2012)
3. Jaeger, A., Stuebing, H., Huss, S.: A dedicated hardware security module for field operational tests of Car-to-X communication. In: 4th ACM Conference on Wireless Network Security (WiSec 2011) (June 2011)
4. Parameswaran, S., Wolf, T.: Embedded systems security - an overview. Design Autom. for Emb. Sys. 12(3), 173–183 (2008)
5. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.-L.: Combined software and hardware attacks on the Java Card control flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011)
6. Francillon, A., Castelluccia, C.: Code injection attacks on Harvard-architecture devices. In: ACM Conference on Computer and Communications Security, October 27-31, pp. 15–26. ACM (2008)
7. Delalleau, G.: Large memory management vulnerabilities: System, compiler and application issues, `http://cansecwest.com/core05/memory_vulns_delalleau.pdf` (visited April 2013)
8. Arora, D., Ravi, S., Raghunathan, A., Jha, N.K.: Hardware-assisted run-time monitoring for secure program execution on embedded processors. IEEE Trans. VLSI Syst. 14(12), 1295–1308 (2006)
9. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. 13(1) (2009)
10. Frantzen, M., Shuey, M.: StackGhost: Hardware facilitated stack protection. In: Wallach, D.S. (ed.) 10th USENIX Security Symposium, August 13-17. USENIX (2001)
11. Francillon, A., Perito, D., Castelluccia, C.: Defending embedded systems against control flow attacks. In: Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code, SecuCode 2009, pp. 19–26. ACM, New York (2009)
12. Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P.: Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In: Annual Computer Security Applications Conference (ACSAC), December 11-15, pp. 339–348. IEEE Computer Society (2006)
13. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium, vol. 81, pp. 346–355 (1998)
14. Edward Suh, G., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 7-13, pp. 85–96. ACM (2004)
15. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
16. Dhem, J.-F., Koeune, F., Leroux, P.-A., Mestré, P., Quisquater, J.-J., Willems, J.-L.: A practical implementation of the timing attack. In: Quisquater, J.-J., Schneier, B. (eds.) CARDIS 1998. LNCS, vol. 1820, pp. 167–182. Springer, Heidelberg (2000)

17. Arnaud, C., Fouque, P.-A.: Timing attack against protected RSA-CRT implementation used in PolarSSL. In: Dawson, E. (ed.) CT-RSA 2013. LNCS, vol. 7779, pp. 18–33. Springer, Heidelberg (2013)
18. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
19. Popp, T., Mangard, S., Oswald, E.: Power analysis attacks and countermeasures. IEEE Design & Test of Computers 24(6), 535–543 (2007)
20. Oswald, D., Paar, C.: Breaking Mifare DESFire MF3ICD40: Power analysis and templates in the real world. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 207–222. Springer, Heidelberg (2011)
21. Heyszl, J., Mangard, S., Heinz, B., Stumpf, F., Sigl, G.: Localized electromagnetic analysis of cryptographic implementations. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 231–244. Springer, Heidelberg (2012)
22. Gu, K., Wu, L., Li, X., Zhang, X.: Design and implementation of an electromagnetic analysis system for smart cards. In: Wang, Y., Cheung, Y.M., Guo, P., Wei, Y. (eds.) CIS, Sanya, Hainan, China, December 3-4, pp. 653–656. IEEE (2011)
23. Van Eck, W., Laborato, N.: Electromagnetic radiation from video display units: An eavesdropping risk? Computers & Security 4, 269–286 (1985)
24. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography. Springer (2002)
25. Tuchman, W.: A brief history of the data encryption standard. In: Denning, D., Denning, P. (eds.) Internet Besieged, pp. 275–280. ACM Press/Addison-Wesley Publishing Co., New York (1998)
26. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21(2), 120–126 (1978)
27. Vermoen, D., Witteman, M., Gaydadjiev, G.N.: Reverse engineering Java Card applets using power analysis. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 138–149. Springer, Heidelberg (2007)
28. Eisenbarth, T., Paar, C., Weghenkel, B.: Building a side channel based disassembler. In: Gavrilova, M.L., Tan, C.J.K., Moreno, E.D. (eds.) Transactions on Computational Science X. LNCS, vol. 6340, pp. 78–99. Springer, Heidelberg (2010)
29. Clavier, C.: Side channel analysis for reverse engineering (SCARE) - an improved attack against a secret A3/A8 GSM algorithm. IACR Cryptology ePrint Archive 2004, 49 (2004)
30. Lee, S., Ermedahl, A., Min, S.L., Chang, N.: An accurate instruction-level energy consumption model for embedded RISC processors. In: Hong, S., Pande, S. (eds.) LCTES/OM, Snowbird, Utah, USA, June 22-23, pp. 1–10. ACM (2001)
31. Kavvadias, N., Neofotistos, P., Nikolaidis, S., Kosmatopoulos, C.A., Laopoulos, T.: Measurements analysis of the software-related power consumption in microprocessors. IEEE T. Instrumentation and Measurement 53(4), 1106–1112 (2004)
32. Mayes, K., Markantonakis, K., Chen, C.: Smart card platform-fingerprinting. In: Advanced Card Technology, pp. 78–82 (October 2006)
33. Allen, F.: Control flow analysis. In: Proceedings of a Symposium on Compiler Optimization, pp. 1–19. ACM, New York (1970)
34. Popp, T., Mangard, S., Oswald, E.: Power analysis attacks and countermeasures. IEEE Design & Test of Computers 24(6), 535–543 (2007)
35. Fink, A.: Markov Models for Pattern Recognition. Springer (2008)
36. Rabiner, L.: A tutorial on Hidden Markov Models and selected applications in speech recognition. Proceedings of the IEEE 77(2), 257–286 (1989)

37. Gut, A.: An Intermediate Course In Probability, 2nd edn. Springer, Department of Mathematics, Uppsala University, Sweden (2009)
38. Berrendero, J.R., Justel, A., Svarc, M.: Principal components for multivariate functional data. Computational Statistics & Data Analysis 55(9), 2619–2634 (2011)
39. Strang, G.: Introduction to Linear Algebra, 3rd edn. Wellesley-Cambridge Press, MA (2003)
40. Fisher, R.A.: The use of multiple measurements in taxonomic problems. Annals of Eugenics 7, 179–188 (1936)
41. Fukumi, M., Mitsukura, Y.: Feature generation by simple-FLDA for pattern recognition. In: CIMCA/IAWTIC, November 28-30, pp. 730–734. IEEE Computer Society (2005)
42. Zhang, L., Wang, D., Gao, S.: Application of improved Fisher Linear Discriminant Analysis approaches. In: International Conference on Management Science and Industrial Engineering (MSIE), pp. 1311–1314 (2011)
43. Forney Jr., D.: The Viterbi Algorithm: A personal history. CoRR, abs/cs/0504020 (2005)
44. Teledyne LeCroy. Teledyne LeCroy website, `http://www.teledynelecroy.com` (visited February 2013)
45. Pomona Electronics. 6069A Scope Probe, `http://www.pomonaelectronics.com/pdf/d4550b-sp150b_6_01.pdf` (visited October 2012)
46. MATLAB. Version 7.10.0.499 (R2010a). The MathWorks, Inc., Natick, Massachusetts (2013), `http://www.mathworks.co.uk/index.html`
47. MATLAB. Hidden Markov Model most probable state path, `http://www.mathworks.co.uk/help/stats/hmmviterbi.html` (visited March 2013)
48. Atkins Limited. MALPAS, `http://www.malpas-global.com/` (visited April 2013)

Appendix

A Viterbi MATLAB Implementation

Listing 1.1. MATLAB implementation of the Viterbi algorithm described in section 3.1

```
% [state_sequence] = viterbi_sequence(initial_probability,
%                                     transition_probability,
%                                     emission_probability)
% initial_probability = initial probability (\pi_{i})
% transition_probability = transition probatility (T)
% emission_probability = emission probability (E)
% state_sequence = most probable state sequence that would
    have resulted to the emission of (O)
% Author: Mehari G. Msgna
% Date: 16 April, 2013

function [state_sequence] = viterbi_sequence(
    initial_probability, transition_probability,
    emission_probability)
    number_of_states = length(initial_probability(1,:));
    number_of_observations = length(emission_probability(1,:)
        );
    state_sequence = zeros(1,number_of_observations);
    sequence_probability = zeros(number_of_observations,
        number_of_states);

    for c = 1:number_of_states
        sequence_probability(1,c) = emission_probability(c,1)
            * initial_probability(1,c);
    end
    for r = 2:number_of_observations
        temp = zeros(1,number_of_states);
        for c = 1:number_of_states
            for c1 = 1:number_of_states
                temp(1,c1) = transition_probability(c1,c) *
                    sequence_probability(r-1,c1);
            end
            mx = max(temp(1,:));
            sequence_probability(r,c) = emission_probability(
                c,r) * mx;
        end
    end
    for j = 1:number_of_observations
        [value, index] = max(sequence_probability(j,:));
        state_sequence(1,j) = index;
    end
end
```