Building a Side Channel Based Disassembler

Thomas Eisenbarth[1], Christof Paar[2], and Björn Weghenkel[2]

[1] Department of Mathematical Sciences
Florida Atlantic University, Boca Raton, FL 33431, USA
teisenba@fau.edu
[2] Horst Görtz Institute for IT Security
Ruhr University Bochum, 44780 Bochum, Germany
{christof.paar,bjoern.weghenkel}@rub.de

Abstract. For the last ten years, side channel research has focused on extracting data leakage with the goal of recovering secret keys of embedded cryptographic implementations. For about the same time it has been known that side channel leakage contains information about many other internal processes of a computing device.
In this work we exploit side channel information to recover large parts of the program executed on an embedded processor. We present the first complete methodology to recover the program code of a microcontroller by evaluating its power consumption only. Besides well-studied methods from side channel analysis, we apply Hidden Markov Models to exploit prior knowledge about the program code. In addition to quantifying the potential of the created side channel based disassembler, we highlight its diverse and unique application scenarios.

1 Motivation

Reverse engineering code of embedded devices is often difficult, as the code is stored in secure on-chip memory. Many companies rely on the privacy of their code to secure their intellectual property (IP) and to prevent product counterfeiting. Yet, in some cases reverse engineering is necessary for various reasons. A company might rely on a discontinued product it does not get any information about from its previous vendor. Or no information is available to ensure flawless interoperability of a component. Often, companies are interested in the details of a competitors new product. Finally, companies may want to identify possible copyright or patent infringements by competitors. In most of these cases that are quite common in embedded product design a disassembler for reconstructing an embedded program is necessary or at least helpful. On most embedded processors, access to code sections can be restricted via so-called lock bits. While it has been shown that for many processors the read protection of the on-chip memory can be circumvented with advanced methods [20], we show in this work that code can be reconstructed with strictly passive methods by analyzing side channel information such as the power consumption of the CPU during code execution.

Side channel analysis has changed the way of implementing security critical embedded applications in the last ten years. Many methods for physical cryptanalysis have been proposed, such as differential power/EM analysis, fault attacks and timing analysis [9, 1, 10]. Since then, methods in side channel analysis as well as countermeasures have been greatly improved by a broad research effort in the cryptographic community. Up to now, most efforts in power and EM

analysis have been put into reconstructing data dependencies in the side channel. Yet, all activity within a device leaves a 'fingerprint' in the power trace. When Kocher *et al.* [10] published power based side channel attacks in 1999, they already mentioned the feasibility of reverse engineering code using side channel analysis. Despite this, virtually all previous work in the are of side channel analysis focus on breaking cryptographic implementations.

We want to show that a program running on a microcontroller can be reconstructed by passively monitoring the power consumption or other electromagnetic emanations only.

1.1 Related Work

Although Kocher *et al.* [10] already mentioned the feasibility of reverse engineering algorithms using side channel analysis, only little work following this idea has been performed. Novak [14] presents a method to recover substitution tables of the secret A3/A8 algorithm. For Novak's attack, one of the two substitution tables and the secret key must be known. Clavier [4] improves reverse engineering of A3/A8 by proposing an attack retrieving the values of both permutation tables and the key without any prior knowledge. Yet, both works concentrate on one specific look-up table and do not consider other parts of the algorithm. In [24], Vermoen shows how to acquire information about bytecodes executed on a Java smart card. The method used in his work is based on averaging traces of certain bytecodes in order to correlate them to an averaged trace of an unknown sequence of bytecodes. Further, Quisquater *et al.* [16] present a method that recognizes executed instructions in single traces by means of self-organizing maps, which is a special form of neural network. Both works restate the general feasibility without quantifying success rates.

1.2 Our Approach

Our final goal is the reconstruction of the program flow and program code. In other words, we want to reconstruct the executed instructions and their execution order of the device under test, the microcontroller, from a passive physical measurement (*i.e.*, an EM measurement or a power trace).

The approach we follow is different from the previous ones, since it is the intention to retrieve information of a program running on a microcontroller by means of single measurements. Under this premise, averaging like in Vermoens approach is not (at least not in the general case) practicable. Although [16] states the general feasibility of a side channel based disassembler, no quantified results are presented. Furthermore, the use of self-organizing maps seems to be inadequate since the possibilities to readjust this approach in case of insufficient results is highly limited.

We apply methods from side channel analysis that are known to be optimal for extracting information to reconstruct executed instruction sequences. We further explore methods to utilize prior information we have about the executed instructions, which is a new approach in side channel analysis. Many publications in side channel analysis borrow methods from other disciplines to enhance side channel cryptanalysis. We want to reverse this trend by showing that methods from side channel analysis can be applied to interesting problems outside of the context of cryptology.

The remaining work is structured as follows: In Section 2 we present methods that recover as much information from the physical channel as possible. Here we apply the most advanced models from side channel analysis research. In Section 3 we apply a hidden Markov model to our problem and introduce methods that increase the performance of our disassembler. All methods are applied to a sample microcontroller platform in Section 4. We also describe and compare the performances of all previously introduced methods. Section 5 discusses possible applications of the proposed methods and Section 6 concludes our work.

2 Extracting Information from Side Channel Leakage

Monitoring side channels for gaining information about a non-accessible or not easily-accessible system is a classical engineering problem, e.g., in control engineering. But especially in cryptography, a lot of effort has been put into methods for retrieving information from emanations of a microcontroller. Hence, we explore the state-of-the-art in side channel information extraction in cryptography to find optimal methods for our purposes. Yet, our goal is different as we extract information about the instruction rather than data.

But how does information about an instruction leak via the side channel? For data in processors, we assume that the leakage originates from the buses which move the data, as well as the ALU processing the data and registers storing the data. The physical representation of an instruction in a microcontroller is more subtle. A unique feature of each instruction is the opcode stored in program memory and moved to the instruction decoder before execution. Besides this, an instruction is characterized by a certain behavior of the ALU, the buses, etc., and possibly other components.

When trying to determine which instruction has been executed, we have in a worst case scenario only one observation of the instruction. Even if we are able to repeat the measurement, the behavior of the instruction will remain the same. Hence we are not able to follow a DPA approach, but rather have to do simple power analysis. In order to succeed, we assume that when trying to recover a program from a microcontroller, we have access to an identical microcontroller which we can analyze and profile. We can use this profiling step to train a Bayesian classifier, as is typically done in template attacks [3]. A Bayesian classifier is a better choice than, e.g., stochastic models [19] when the underlying leakage function is not known [22].

Template Construction The first step of template classification is the construction of a template for every class [3]. Classes are in our case equivalent to individual microcontroller instructions. Each template is constructed by estimating the instructions' distribution of the power consumption from the sample data. Later, during the attack phase, the template recognition is then performed by assigning each new observation of power consumption to the most probable class.

As sample data we consider N D-dimensional observations of the processor's power consumption $\{\boldsymbol{x}_n\}$, where $\boldsymbol{x}_n \in \mathbb{R}^D$, $n = 1, \ldots, N$. Each observation belongs to exactly one of K classes $\mathcal{C}_k$, representing the instructions modeled by a finite set of instruction states q_k, $k = 1, \ldots, K$. Each class $\mathcal{C}_k$ contains $N_k = |\mathcal{C}_k|$ elements. We assume that for each class our samples are drawn from

a multivariate normal distribution

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \mathbf{S}_k) = \frac{1}{(2\pi)^{D/2}\left|\mathbf{S}_k\right|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^T\mathbf{S}_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k)\right). \quad (1)$$

Given the sample data $\{\boldsymbol{x}_n\}$, the maximum-likelihood estimations for the class mean $\boldsymbol{\mu}_k$ and the class covariance $\mathbf{S}_k$ are given by

$$\boldsymbol{\mu}_k = \frac{1}{N_k}\sum_{\boldsymbol{x}_n \in \mathcal{C}_k} \boldsymbol{x}_n \quad (2)$$

and

$$\mathbf{S}_k = \frac{1}{N_k}\sum_{\boldsymbol{x}_n \in \mathcal{C}_k} (\boldsymbol{x}_n - \boldsymbol{\mu}_k)(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^T. \quad (3)$$

Thus, the template for each class is defined by $(\boldsymbol{\mu}_k, \mathbf{S}_k)$.

Template Classification During the classification phase, a new observation of power consumption $\boldsymbol{x}$ is assigned to one of the possible instruction states q_k. This is done by evaluating every template and determining the class state $\tilde{q}$ with the highest posterior probability. Considering the Bayes rule, we get:

$$\tilde{q} = \arg\max_{q_k} p\left(q_k|\boldsymbol{x}\right) = \arg\max_{q_k} p\left(\boldsymbol{x}|q_k\right)\Pr\left(q_k\right), \quad (4)$$

where $p\left(\boldsymbol{x}|q_k\right) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \mathbf{S}_k)$ and $\Pr(q_k)$ is the prior probability of instruction state q_k.

In practice, the observations $\boldsymbol{x}_n$ available for training a template are too high dimensional and too closely correlated to generate a well-conditioned covariance matrix $\mathbf{S}_k$, making its inversion impossible [18]. Building the templates in a suitable subspace can solve these problems. In the subspace, less observations $\boldsymbol{x}_n$ are necessary to create a regular covariance matrix and the estimated class distributions become more reliable.

Several methods for the reduction of the size of the observations $\boldsymbol{x}_n$ have been proposed in the context of side channel analysis [18, 21]. Even more are available in the standard literature [2]. We tried Principal Component Analysis and Fisher's Linear Discriminant Analysis.

Principal Component Analysis Principal Component Analysis (PCA) is a technique to reduce the dimensionality of our data while keeping as much of its variance as possible. This is achieved by orthogonally projecting the data onto a lower dimensional subspace.

Consider again the N observations of power consumption $\{\boldsymbol{x}_n\}$, $n = 1, \ldots, N$, and their global covariance matrix $\mathbf{S}$ which is built in analogy to (3). A one-dimensional subspace in this Euclidean space can be defined by a D-dimensional unit vector $\mathbf{u}_1$. The projection of each data point $\boldsymbol{x}_n$ onto that subspace is given by $\mathbf{u}_1^T\boldsymbol{x}_n$. It can be shown that the direction that maximizes the projected variance $\mathbf{u}_1^T\mathbf{S}\mathbf{u}_1$ with respect to $\mathbf{u}_1$ corresponds to the eigenvector of $\mathbf{S}$ with the largest eigenvalue λ_1 [2]. Analogous, an M-dimensional subspace, $M < D$, that maximizes the projected variance is given by the M eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_M$ of $\mathbf{S}$ corresponding to the M largest eigenvalues $\lambda_1, \ldots, \lambda_M$.

Since our goal is the reliable distinction of many different instructions it seems reasonable not only to maximize the overall variance of the data but alternatively

to maximize the variance of the different class means $\boldsymbol{\mu_k}$. If moving the class means away from each other also results in less overlapping, the classification will be easier. We apply PCA in both ways, *i.e.*, for the whole data and for class means.

Fisher's Linear Discriminant Analysis Similar to PCA, with Fisher's Linear Discriminant Analysis (or Fisher LDA) we have another method for dimensionality reduction. But instead of just maximizing the variance of the projected data, information about the different classes and their covariances is taken into consideration.

Again, we have our N observations $\{\boldsymbol{x}_n\}$ in a D-dimensional Euclidean space. Each observation belongs to one of K different classes $\mathcal{C}_k$, $k = 1, \ldots, K$, of size $N_k = |\mathcal{C}_k|$.

Then, the within-class covariance $\mathbf{S}_W$ for all classes is given by

$$\mathbf{S}_W = \sum_{k=1}^{K} N_k \mathbf{S}_k \tag{5}$$

and the covariance of the class means, the between-class covariance $\mathbf{S}_B$, given by

$$\mathbf{S}_B = \sum_{k=1}^{K} N_k (\boldsymbol{\mu}_k - \boldsymbol{\mu})(\boldsymbol{\mu}_k - \boldsymbol{\mu})^T, \tag{6}$$

where $\boldsymbol{\mu}$ is the mean of the total data set and $\boldsymbol{\mu}_k$ and $\mathbf{S}_k$ are the individual class mean and covariance as defined in (2) and (3).

Now consider again a D-dimensional unit vector $\mathbf{u}_1$ defining a one-dimensional subspace onto which the data is projected. This time, the objective used to be maximized in the subspace is the ratio of the projected between-class variance to the projected within-class variance:

$$J(\mathbf{u}_1) = (\mathbf{u}_1^T \mathbf{S}_W \mathbf{u}_1)^{-1} (\mathbf{u}_1^T \mathbf{S}_B \mathbf{u}_1). \tag{7}$$

As for PCA it can be shown that this objective is maximized when $\mathbf{u}_1$ corresponds to the eigenvector of $\mathbf{S}_W^{-1}\mathbf{S}_B$ with the largest eigenvalue λ_1, leading to a one-dimensional subspace in which the class means are wide-spread and the average class variance will be small [2]. Again, the M-dimensional subspace, $M \leq K - 1$, created by the first M orthogonal directions that maximize the objective J are given by the M eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_M$ of $\mathbf{S}_W^{-1}\mathbf{S}_B$ with the largest eigenvalues $\lambda_1, \ldots, \lambda_M$.

The PCA approach of maximizing the variance will not always lead to good separability of the classes. In these cases Fisher LDA can be clearly superior. On the other hand it is more prone to overfitting since more model parameters have to be estimated.

In addition to the described template recognition we also tried different multi-class Support Vector Machines implemented in the Shark machine learning library [8]. Unfortunately, with 41 classes and 2000 training examples per class (cf. Section 4) the computational costs were too high for a thorough search for parameters. Furthermore, the first results we received were not very promising. Therefore we did not further pursue this approach.

3 How to Include Code Properties

In this section we extend the model of a microcontroller's power consumption by a second level. In the previous section we modeled the power consumption of single instruction states. We expand our approach by additionally exploiting general knowledge about microcontroller code.

Up to now we did not consider *a priori* knowledge we have about the code we want to reverse engineer. Even in a scenario where we do not know anything specific about the target code, we have prior knowledge about source code in general. For instance, some instructions occur more often than others. As an example we can focus on the PIC microcontroller we analyze in Section 4.2. Since one of the operands of two-operand instructions must be stored in the accumulator, move commands to and from the accu are frequent. Other instructions such as `NOP` (no operation) are quite rare in most programs. By performing an instruction frequency analysis we can provide meaningful prior probabilities to the instruction distinguisher from Section 2. In particular, the performance of the template recognition can be boosted by including the prior probabilities in Equation (4).

For many microprocessor architectures instruction frequency analyses have been performed, mainly for optimizing instruction sets. Unfortunately, for microcontrollers and especially the PIC, no major previous work has been performed. The analysis we performed is described in Section 4.2.

Besides a simple instruction frequency analysis, additional information can be gained by looking at tuples of instructions that usually are executed subsequently. One example are the few two-cycle instructions such as `CALL` and `GOTO`, which are always followed by their second instruction part. But it is also true for conditional commands such as `BTFSS` (bit test a register, skip next instruction if zero), which is commonly used to build a conditional branch, hence followed by a `GOTO` (when branching) or by a virtual `NOP` (when skipping the `GOTO`). The Microchip Assembler itself supports 13 built-in macros which virtually extend the instruction set and are replaced by more than one physical instruction at compile time [13]. Their use will consequently influence the occurrence probability of the corresponding tuples. Similar effects occur if a compiler like a certain C compiler has been used for code generation. Tuple frequency analysis is also a classical method for doing cryptanalysis of historic ciphers.

Other information about the code can also be helpful. A crypto implementation uses different instructions than a communication application or a control algorithm. Additional information can be gained by exact knowledge about certain compiler behavior if the code was compiled, e.g., from C source code. Different compilers can generate assembly code with different properties. Hence, prior knowledge about the application or the compiler can be exploited to improve recognition results.

Hidden Markov Model The microprocessor can be considered as a state machine, for which we want to reconstruct the sequence of taken states. Each state corresponds to an instruction, or, more precisely, to a part of an instruction if the instruction needs several cycles to be executed. We cannot directly observe the state. Instead, the only information we have is the side channel information provided by the power measurement of a full instruction cycle. Yet, we assume that the physical information depends on the state, *i.e.*, the executed instruction of the microcontroller.

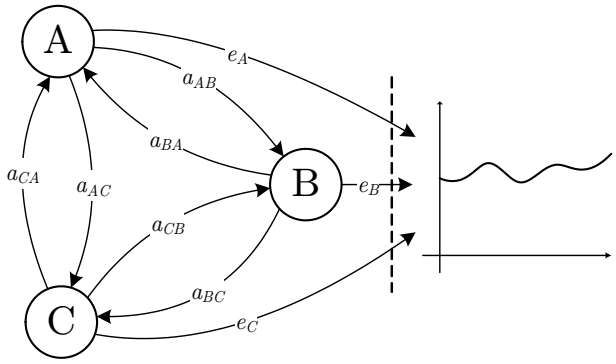

Fig. 1. HMM with three hidden states A, B and C. Only the output on the right of the dashed line is observable.

We define our system to be described by a hidden Markov model (HMM). At each discrete time instance i we make one observation $\boldsymbol{x}_i$, resulting in a sequence of observations $\hat{\boldsymbol{x}}$. These observations are generated by a hidden Markov chain passing through a state sequence $\boldsymbol{\pi}$, with $\pi_i = q_k$ being the state the model is in at time instance i. Each state q_k is followed by a new state q_l with a probability of $a_{kl} = \Pr(\pi_i = q_l | \pi_{i-1} = q_k)$. We implicitly assume that the probability of the new state q_l depends only on the preceding state q_k, but not on any earlier states. The Markov process cannot directly be observed, instead we observe certain emissions of that process. We expect to see an observation $\boldsymbol{x}_i$ with a certain probability $e_k(\boldsymbol{x}_i) = p(\boldsymbol{x}_i | \pi_i = q_k)$, depending on the actual state q_k of the processor.

A simple Markov model with three states A, B and C is given in Figure 1. Unlike for classical HMMs, for which the observations are drawn from a discrete set of symbols, our observations $\boldsymbol{x}_i$ are continuous distributions over $\mathbb{R}^D$ and our emission probabilities are consequently described by the continuous probability density functions $e_k(\boldsymbol{x}_i) = p(\boldsymbol{x}_i | \pi_i = q_k)$.

Our system can completely be described as a hidden Markov model (HMM) consisting of the state transition probability distribution $\boldsymbol{A} = \{a_{kl}\}$, the emission probability distribution $\boldsymbol{E} = e_k(\boldsymbol{x}_i)$, and an initial state distribution $\boldsymbol{\kappa} = \{\kappa_k | \kappa_k = \Pr(q_k)\}$. We will use tuple analysis of executed instruction sequences to derive the transition probabilities $\boldsymbol{A}$ of the hidden Markov chain. The instruction probabilities derived from the frequency analysis can also serve as an initial state distribution $\boldsymbol{\kappa}$ for the HMM. Finally, the emission probability distribution $\boldsymbol{E}$ is provided by the templates described in Section 2. The process of the actual parameter derivation for our model ($\boldsymbol{A}$,$\boldsymbol{E}$,$\boldsymbol{\kappa}$) is described in Section 4. Having the model and a set of observations $\boldsymbol{x}$, several methods for optimal reconstruction of the state sequence $\boldsymbol{\pi}$ exist.

3.1 Optimal Instruction Reconstruction

Assuming that we have reconstructed all parameters of our HMM, namely $\boldsymbol{A}$, $\boldsymbol{E}$ and $\boldsymbol{\kappa}$, we assume a sequence of observations $\boldsymbol{x}$ for which we want to reconstruct the state sequence $\boldsymbol{\pi}$ of the hidden Markov process, namely the instructions executed on the microprocessor. Given our model ($\boldsymbol{A}$,$\boldsymbol{E}$,$\boldsymbol{\kappa}$), we are able to reconstruct either

– the state sequence that was executed most likely, or
– the most probable state executed at a certain time instance, given the set of observations.

Though similar, the solutions are not always the same and are derived using two different algorithms. We evaluate both algorithms, the Viterbi algorithm and the Forward-Backward algorithm.

Viterbi Algorithm The Viterbi algorithm determines the most probable state path $\boldsymbol{\pi} = \{\pi_i\}$ that might have caused the observations $\hat{\boldsymbol{x}} = \{\boldsymbol{x}_i\}$ [17, 6]. The path with the highest probability is given by

$$\boldsymbol{\pi}^* = \text{argmax}_{\boldsymbol{\pi}} p(\boldsymbol{\pi}|\hat{\boldsymbol{x}}) = \text{argmax}_{\boldsymbol{\pi}} \frac{p(\hat{\boldsymbol{x}}, \boldsymbol{\pi})}{p(\hat{\boldsymbol{x}})} = \text{argmax}_{\boldsymbol{\pi}} p(\hat{\boldsymbol{x}}, \boldsymbol{\pi})$$

and can be determined recursively by $v_l(i+1) = e_l(\boldsymbol{x}_{i+1})\text{max}_k(v_k(i)a_{kl})$ and $v_k(1) = \boldsymbol{\kappa}_k e_k(\boldsymbol{x}_1)$, where $v_k(i)$ is the probability of the most probable path ending in state q_k. Hence we drop all transition probabilities leading to state q_k, except for the one with the highest probability. Usually for every $v_l(i+1)$ a pointer to the preceding state probability $v_k(i)$ is stored together with the probability itself. After reaching the last observation, the path yielding the optimal state sequence is computed by simple back tracking from the final winning state.

When viewing the states as nodes and the transitions as edges of a trellis, as is typically done in (de-) coding theory, the algorithm becomes more ostensive [11].

The Forward-Backward Algorithm The forward-backward algorithm maximizes the posterior probability that an observation $\boldsymbol{x}_i$ came from state q_k, given the full observed sequence $\hat{\boldsymbol{x}}$, *i.e.*, the algorithm optimizes $p(\pi_i = q_k|\hat{\boldsymbol{x}})$ for every i [17, 6]. In contrast to the Viterbi algorithm, it includes the probabilities of all transitions leading to one state, and browses all transitions twice, once in the forward direction like the Viterbi, and once in the backward direction.

For the *forward* direction we define $\alpha_k(i) = p(\boldsymbol{x}_1\boldsymbol{x}_2 \ldots \boldsymbol{x}_i, \pi_i = q_k)$, α being the probability of the observed sequence up to $\boldsymbol{x}_t$, and $\pi_i = q_k$. The forward algorithm is performed recursively by evaluating all $\alpha_k(i)$. The *backward* algorithm is performed in the same way, just backwards, *i.e.*, $\beta_k(i) = p(\boldsymbol{x}_{i+1}\boldsymbol{x}_{i+2} \ldots \boldsymbol{x}_L|\pi_i = q_k)$. The computation of $\beta_k(i)$ and $\alpha_k(i)$ is performed recursively by evaluating

$$\alpha_l(i+1) = e_l(\boldsymbol{x}_{i+1}) \sum_k \alpha_k(i)a_{kl} \quad \text{and}$$

$$\beta_k(i) = \sum_l \beta_l(i+1)a_{kl}e_l(\boldsymbol{x}_{i+1}).$$

The initial values for the recursions are $\alpha_k(1) = \boldsymbol{\kappa}_k e_k(\boldsymbol{x}_1)$ and $\beta_k(T) = 1$, respectively. By simply multiplying $\alpha_k(i)$ and $\beta_k(i)$ we gain the production probability of the observed sequence with the ith symbol being produced by state q_k:

$$p(\boldsymbol{x}, \pi_i = k) = \alpha_k(i)\beta_k(i) = p(\boldsymbol{x}_1\boldsymbol{x}_2 \ldots \boldsymbol{x}_i, \pi_i = k)p(\boldsymbol{x}_{i+1}\boldsymbol{x}_{i+2} \ldots \boldsymbol{x}_L|\pi_i = k)$$

We can now easily derive the posterior probability $\gamma_k(i) = p(\pi_i = k|\hat{\boldsymbol{x}})$ by simply dividing $p(\hat{\boldsymbol{x}}, \pi_i = k)$ by $p(\hat{\boldsymbol{x}})$:

$$\gamma_k(i) = \frac{\alpha_k(i)\beta_k(i)}{p(\hat{\boldsymbol{x}})} = \frac{\alpha_k(i)\beta_k(i)}{\sum_k \alpha_k(i)\beta_k(i)}$$

The forward-backward algorithm consequently calculates the maximum *a-posteriori* probability for each observation, hence minimizes the number of state errors. This can sometimes cause problems as the resulting state sequence might not be an executable one. The forward-backward algorithm is also known as 'MAP algorithm', or 'BCJR algorithm'.

For a complete description of both algorithms, refer to [17, 5, 11]. The Viterbi algorithm used to be more popular for decoding of convolutional codes (at least until the advent of Turbo codes) due to its lower computational complexity and almost equally good results. It is also easier to take care of numerical difficulties that often occur for both algorithms.

4 Reconstructing a Program from Side Channel Leakage

This section presents the practical results of the code reconstruction from actual power measurements. The methods and models we introduced in the previous two sections are applied to a PIC microcontroller. The PIC microcontroller makes a good choice for a proof-of-concept implementation of the side-channel disassembler, since it features a comparably small instruction set, a short pipeline and is a well-understood target for side-channel attacks [21]. We present the results of every step taken and compare alternative methods where available.

All measurements were done on a PIC16F687 microcontroller mounted on a printed circuit board. The board enables easy measurement of the power consumption of the running microcontroller. The power consumption is measured via the voltage drop over a shunt resistor connecting the PIC's ground pin to the ground of the power supply. The PIC is clocked at 1 MHz using its internal clock generator. Measurements are performed using an Agilent Infiniium 54832D digital sampling oscilloscope featuring a maximum sampling rate of 4 GS/s at 1 GHz bandwidth. All measurements have been sampled at 1 GS/s. The same measurement setup is used for the generation of sample measurements for template generation, template verification, and the measurement of sample programs we used to verify our final choice of methods.

The analyzed PIC16F687 microcontroller features an instruction set of 35 different instructions. We excluded instructions like SLEEP that will not occur in the program flow. Most of the instructions are one-cycle instructions. Yet some instructions, especially branching instructions, can last two instruction cycles. In those cases we created two different templates for each instruction cycle, resulting in a set of 41 different templates or instruction classes, respectively.

Each instruction cycle of the PIC lasts four clock cycles. The power consumption of each peak depends on different properties, of which we can only assume a limited number to be known or predictable. Two typical power traces of the PIC are shown in Figure 2. Each trace depicts the power consumption during the execution of three instructions. Every instruction lasts four clock cycles, each clock cycle being indicated by a peak in the power trace. The first instruction, executed during the first four clock cycles Q1 through Q4, is the same in both cases, *i.e.*, a `NOP` instruction. The second executed instruction is either an `ADDLW` or a `MOVWF`, as indicated. As can be easily seen, the power consumption of two different instructions differs even before the execution cycle of the instruction itself. The PIC features a pipeline of one instruction, hence an instruction is prefetched while the previous instruction is being executed. The different Hamming weights (for `ADDLW` and `MOVWF` the difference is 6 of 14 bit)

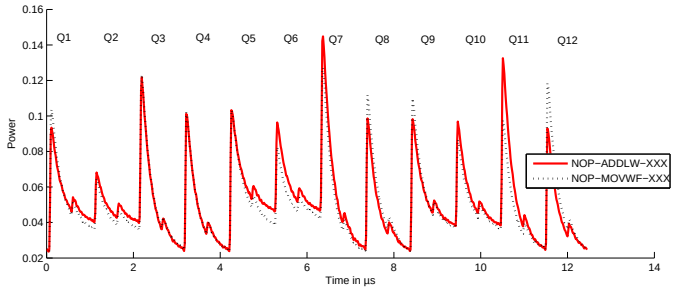

Fig. 2. Power trace showing three examples for the execution of `NOP` and `ADDLW` versus the execution of `NOP` and `MOVWF`.

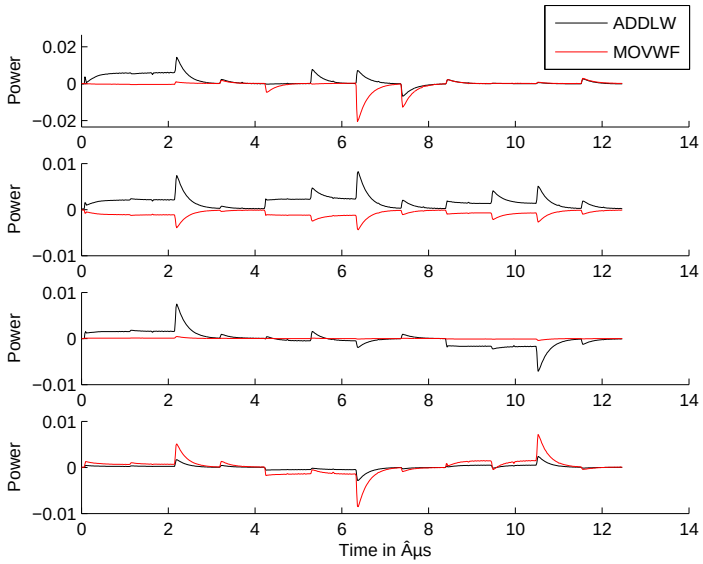

Fig. 3. Sum of the PCA components (top figure) and the three first components of the PCA of the both instructions of Figure 2.

of the prefetched opcodes account in part for the differences in Q1 through Q3. In Q5, at the first execution clock cycle of the monitored instruction, the data is mapped to the ALU, e.g., via the data bus. Hence, the replacement of values on the data bus affects the power consumption in Q5. In Q6, the ALU actually reads the applied data, before processing it and putting the result on the bus in Q7. In Q8, the result is stored at the target register. Of course, these are only some of the effects that show up in the power trace. Again, we saw that data values, especially those written to the bus, have a significant influence on the variation of the power consumption.

Unfortunately, the data dependencies do not help identifying the instruction. They rather obfuscate smaller changes caused by the control logic, the ALU and other instruction-dependent power consumers. Hence, for effective instruction reconstruction, we apply the methods introduced in Section 2 to extract the maximum amount of information from the observed power trace $\boldsymbol{x}$.

4.1 Template Construction

The first step for building templates is the profiling step. To build templates for the instructions, we need several different power measurements of the same instruction. For this purpose we executed specifically generated training code on the training device described above. Since the template must be independent of other factors except the instruction itself, we varied all other variables influencing the power consumption. We generated several code snippets containing the target instruction while varying the processed data, memory location, as well as the instructions before and after the target instruction. The new code is programmed into the microcontroller and executed while the oscilloscope samples the power consumption. The post-processing is explained after the explanation of the training code snippets for the profiling.

To generate a training code set for the profiling of a chosen instruction, this instruction is executed several times. For each execution the data processed by the instruction is initialized to a random value. If the instruction operates on a register, one of the general purpose registers is chosen at random and is initialized with random data prior to being accessed. The accu is always initialized with a random value for every instruction, even if it is not accessed. This is due to the observation in [7] that the Hamming weight of the working registers content has a noticeable effect on the PICs power consumption even while executing a NOP instruction.

Due to the pipeline, we also have to vary the pre-instruction and the post-instruction surrounding the target instruction we want to profile. We also made the pre-instructions and post-instructions operate on random data. Since we included the measurement of the pre- and the post-instruction into the templates, the post-instruction was also followed by another random instruction working on random data. By this we are able to minimize the bias that the surrounding instructions can have on our observations. Finally, we also varied the position of the instructions in program memory, just in case this could influence the power consumption as well.

Target instructions taking two instruction cycles to execute are treated as two consecutive instructions, hence two templates are generated for these instructions. Of course, only the post-instruction or the pre-instruction can be varied in this case. For each of the 41 instruction classes we generated 2500 observations with randomly varying data and surrounding instructions. The raw

power traces including pre- and the post-instruction are then aligned to an arbitrary reference instruction to neutralize small variations in the length of clock cycles.

PCA When performing a template attack in a principal subspace, the dimensionality M of the subspace has to be chosen carefully. On one hand, if M is too low, too much of the variance of the original data gets lost and with it, most likely, important information about the class distribution. If M gets too large, on the other hand, the templates get less reliable again. One reason for this could be the bad conditioning of a large covariance matrix. Another reason is the risk of overfitting model parameters to distributions which we kept random in the template creation process, such as surrounding instructions and processed data.

As the plots of the power consumption profile shows (cf. Figure 2), there are twelve large peaks and another twelve small peaks for three instruction cycles. Thus, we can assume an upper bound of 24 for the number of components containing information. Indeed, as shown in Figure 4, our experiments for PCA show no significant improvements in performance after $M = 16$ dimensions, leading to an average recognition rate of 65.6%.

To find a good subspace, the performance for a given number of dimensions is determined using 5-fold cross validation: 2000 examples per class are split into five parts and, successively, one part is kept as test data, one as training data for the PCA and the remaining three parts as training data for the templates. The PCA is applied to the test data, which is then evaluated using the generated templates. The final result of the cross validation run is the average recognition rate on all five unseen (*i.e.*, not included in any manner in the template building process) test data sets.

After deciding that $M = 16$ is a good choice for the subspace, the 2000 examples per class were taken to compute a new model (750 examples for PCA, 1250 for the templates) which was validated on 41×500 yet unseen examples, resulting again in a recognition rate of 65.2%.

We also tried to normalize the data to $[0 \dots 1]$ and to zero mean and standard deviation $\sigma = 1$, respectively. The normalization steps did not result in better recognition rates.

Following another approach, we used PCA to create a subspace that maximizes the variance between the different class means instead of maximizing the overall variance [21]. This variation of PCA resulted in an improved average recognition rate of 66.5% for $M = 20$. Again, we used 5-fold cross validation to determine the success rate and additional normalization lead to worse results.

Figure 3 shows the sum of all PCA components (upper plot) and the first three PCA components separately (three lower plots) of the PCA-based template means of the `ADDLW` and `MOVWF` instructions. The plots show that the four instruction cycles of the post-instruction contain no information for the instruction recognition. Parts of the pre-instructions, however, contain useful information, due to the instruction prefetch.

Fisher LDA Since the Fisher-LDA, like our second PCA approach, not only takes into consideration the variance of the class means, but also the variance of the different classes, we expect a subspace with less overlaps of the classes and thus better classification results. In accordance to the cross-validation steps above, we reached a recognition rate of 70.1% with $M = 17$ on unseen data.

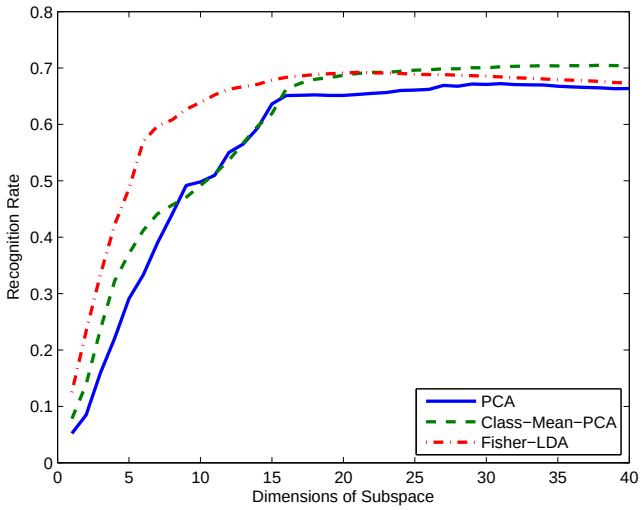

Fig. 4. Results of 5-fold cross validation on generated training data.

Table 1. Percentage of true positives (bold) and false positives during recognition of selected instructions with 17 dimensional Fisher-LDA on unseen test data. The columns indicate the recognized instructions while the line indicates the executed instruction.

Instruction	**Recognized as [%]**								
	ADDWF	ANDWF	BTFSC	BTFSS	CALL	DECF	MOVLW	MOVWF	RETURN
ADDWF	**41**	8	1	5	0	5	0	1	0
ANDWF	4	**38**	3	1	0	11	0	2	0
BTFSC	2	5	**45**	19	0	1	0	0	0
BTFSS	1	2	23	**54**	0	0	0	0	0
CALL	0	0	0	0	**100**	0	0	0	0
DECF	3	9	0	0	0	**30**	0	3	0
MOVLW	0	0	0	0	0	0	**79**	0	0
MOVWF	1	1	0	0	0	3	0	**56**	0
RETURN	0	0	0	0	0	0	0	0	**99**

However, for subspaces with $M < 15$ the performance has been significantly higher than for PCA, as shown in Figure 4. Hence, LDA needs less dimensions resulting in smaller templates to achieve comparable results.

A comparison of the recognition rates for the different instructions reveals large differences between instructions. Table 1 shows a part of the recognition rates for selected instructions. The recognition rates vary from 30% for `DECF` instruction to 100% for `CALL`. Furthermore, one observes *similarities* between certain instructions. For instance, there are many false positives amongst instructions working on file registers, e.g., `ADDWF`, `ANDWF`, `DECF`. Some instructions, like `BTFSC` and `BTFSS`, seem especially hard to distinguish while others, like `RETURN`, show very few false positives or false negatives.

Similar to `BTFSC` and `BTFSS` or the file register instructions, there seem to be several *families* of instructions with a template distribution very close to each other, resulting in a huge cross-error. Therefore we also tried an hierarchical

Table 2. Results for instruction frequency from code analysis.

Instruction	Freq. [%]	Instruction	Freq. [%]
MOVWF	10.72	BSF	6.95
BCF	9.68	MOVF	6.14
MOVLW	8.22	BTFSS	3.69
GOTO	8.12	BTFSC	3.67
CALL	8.06	RETURN	3.48

LDA which performs the recognition in a layered manner, first identifying a family of instructions and then, applying a second set of templates, identifying the actual instruction within the family. However, this approach did not result in an increased recognition rate and was hence not further explored.

Until approximately 16 dimensions, LDA clearly outperforms the two PCA approaches. The class mean PCA shows a better performance to the classical PCA and should hence be preferred. All three sets of templates make a decent choice for the generation of the emission probabilities $\boldsymbol{E}$ for our HMM, as they all achieve a similar recognition rate of almost 65% for a choice of 16 or more dimensions.

4.2 Source Code Analysis

For the instruction frequency analysis and tuple frequency analysis we analyzed the source code of several programs. We built up a code base by using publicly available source code from various web sites, e.g., the microchip PIC sample source code [23, 15, 25]. We also included several implementations of own source code, cryptographic as well as general purpose code.

Due to loops and branches, the instruction frequency of source code is not equal to the instruction frequency of actually executed code. In absence of a reliable simulator platform which is needed to perform an analysis of the executed code, we decided to further process the disassembly listings of the code base. We extracted function calls and loops and unrolled them into the code, just as they would be executed. Also lookup-tables, which are implemented as a list of RETLW (assign literal to accu and return) were reduced to a single RETLW, as would be executed by a program.

Still, actually executed code can deviate from the assessed probabilities for various reasons. One should keep in mind that microcontroller code is often very special-purpose and can deviate strongly from one application to another. We included different kinds of programs in the code frequency and tuple analysis. Classical controller applications such as reading A/D converter info or driving displays and other outputs involves a lot of 'bit-banging', *i.e.*, instructions like BCF or BSF (clear/set bit of register). Other applications that involve more complex data processing such as crypto applications, include more arithmetic.

The result of the instruction frequency analysis is shown in Table 2. Move instructions are the most frequent ones. The PIC increases their general commonness on most microprocessor platforms further by limiting arithmetic to always include the accu and one other register. Table 3 shows the 12 most common instruction tuples. The MOVLW-MOVWF combination is typical for loading values from the code to registers. Also quite common is a conditional skip (BTFSC or

Table 3. Frequency of 12 most frequent instruction tuples.

Instruction Pair		Freq.	Instruction Pair		Freq.
first	second	[%]	first	second	[%]
MOVLW	MOVWF	3.40	MOVWF	BCF	2.09
BCF	BSF	2.36	ANDLW	MOVWF	1.83
MOVLW	CALL	2.35	BSF	BSF	1.78
BTFSS	GOTO	2.31	MOVWF	MOVLW	1.75
MOVWF	MOVF	2.25	CALL	MOVLW	1.70
BTFSC	GOTO	2.11	MOVF	MOVWF	1.65

BTFSS) followed by a GOTO, hence the emulation of a branch instruction. If, as shown here, some tuples are much more common than others (the expected tuple frequency for a uniform distribution of the tuples is 0.08%), the post processing step based on HMM presented in Section 3 will further increase the detection rate considerably.

With the instruction frequency and tuples analyzed, we can now build the HMM of the microprocessor. As mentioned in Section 3, the instruction frequency and tuple frequency are used to construct the initial state distribution $\boldsymbol{\kappa}$ and the state transition probabilities $\boldsymbol{A}$, respectively. We constructed the HMM transition matrix $\boldsymbol{A}$ by performing the following steps:

For non-jumping instructions the post-instructions are directly taken from the tuple analysis. Instructions always lasting two cycles, *i.e.*, CALL and GOTO, consist of two states. While CALL1 (or GOTO1) is always succeeded by CALL2 (GOTO2), the latter is assumed to be succeeded by their target address instruction (due to the jump). For the transition probabilities of conditionally branching instructions, like BTFSC, we analyzed the first succeeding instruction as well as the second succeeding instruction. The second succeeding instruction were counted as post-instruction for the second part, e.g., BTFSC2, while the first successors became the weighted post-instructions for the first part of the instruction. The weight is necessary, because in a certain number of cases the first part of the instruction will be succeeded by its second part rather than the following instruction. In lack of better information we chose the weight to be 50%. Hence, in half the cases the instruction was considered jumping and its second part was counted as post-instruction (e.g., BTFSC-BTFSC2). Finally the resulting transition matrix is normalized row-wise to represent a proper probability distribution and averaged with uniformly distributed transition probabilities. For the latter step, all impossible transitions are excluded, e.g., ADDLW-CALL2.

The initial state distribution $\boldsymbol{\kappa}$ is simply directly set to the derived instruction frequency. Here we only have to take care of assigning probabilities to the non-included second parts of the two-cycle instructions which are equal or half the occurrence number of the first part instruction, depending on whether the execution is conditional or static.

Together with the emission probabilities given by the templates, we now have the full HMM specified and can apply it to the measurements of real programs.

4.3 Analyzing Programs

During the generation of test data we paid special attention to randomization of everything that could bias the distribution of the target instruction. It is

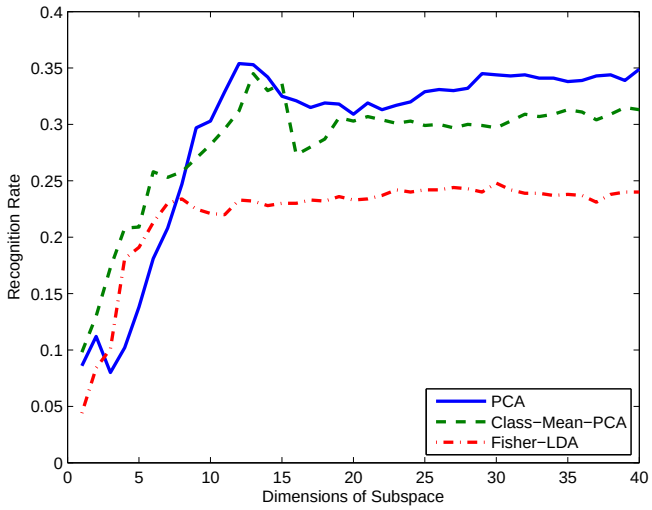

Fig. 5. Recognition rate on 1000 examples of random code. Each time, 750 training examples were used to calculate the subspace and 1750 others for the templates.

unlikely, however, that the same distribution will hold for real code. Indeed we are, as shown in Section 4.2, far away from uniformly distributed instructions or tuples of instructions. The same holds for data: for instance, in real code we can expect to find far more data words like `0x00` or `0xFF` than in uniformly distributed data. As a consequence, the recognition rate of our template attack will drop as soon as the distribution of data values changes. The data dependency can be shown through executing random (in this case non-jumping) instructions without taking care of the content of file registers and the working register. Thus, the main difference to the known distribution is the data the instructions work on. As Figure 5 shows that in this case the best recognition rate reached by our templates is only 35% while, for a similar set of instructions from the test set, we expect a recognition rate of 47%.

As an example for real code we picked an implementation of the KeeLoq crypto algorithm and measured the execution of the first 500 instructions. Now, the highest recognition rate achieved was 40.7% with $M = 19$ and Fisher-LDA, while about 60% could have been expected on similar instructions from the test set. For the template evaluation (cf. Equation (4)) the prior probabilities from the code analysis have been taken into account.

After having modeled the complete Hidden Markov Model with the results from the source code analysis we are now able to apply the Viterbi algorithm and the Forward-Backward-Algorithm to the results from Section 4.1 to calculate the most probable sequence of instruction states as well as the instructions with the highest a posteriori probability. Given the above code example of the KeeLoq crypto algorithm, Viterbi algorithm helps to improve the recognition rate by another 17% points to up to 58% as shown in Figure 7. With recognition rates of up to 52% the Forward-Backward-Algorithm performed slightly worse on this code example.

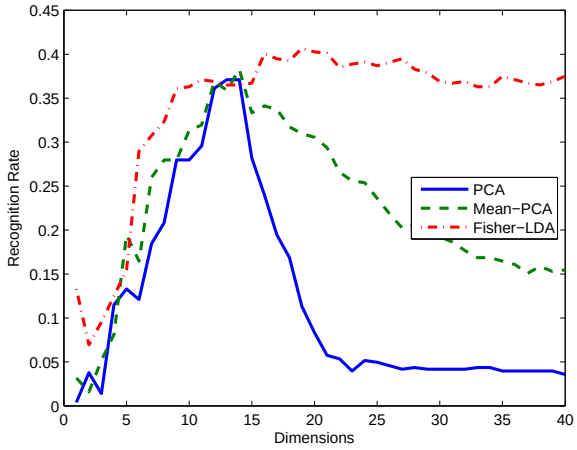

Fig. 6. Recognition rate on the first 500 instructions of the KeeLoq crypto algorithm. Each time, 750 training examples were used to calculate the subspace and 1750 others for the templates.

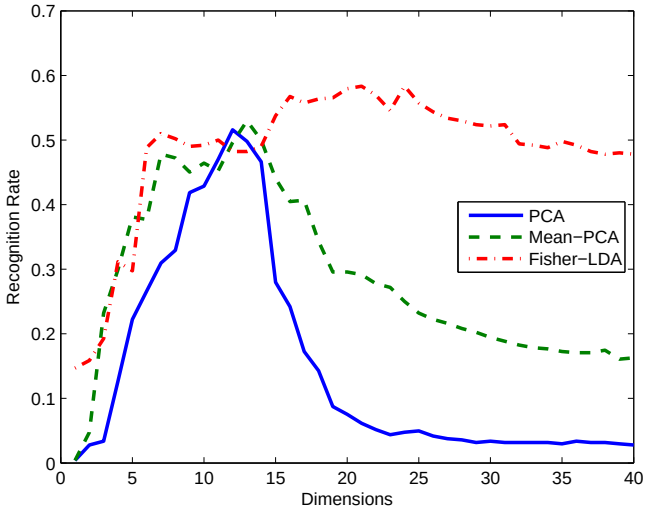

Fig. 7. Recognition rate for the first 500 examples of the Keeloq algorithm in different subspaces after applying the Viterbi algorithm to determine the most probable path

5 Applications and Implications

The most obvious application of the side channel disassembler is reverse engineering of embedded programs. Reverse engineering of software is an established method not only for product counterfeiting, but also for many legal applications. A common application is re-design, where an existing functionality is analyzed and possibly rebuilt. The reasons for this can be manifold, e.g., lost documentation, lost source code, or product discontinuation. But often, reverse engineering does not go that far. Reversing of interfaces is often applied to ensure interoperability with other systems. Debugging is another important application scenario of reverse engineering. It can also be performed for learning purposes, to understand how a code works. Then reverse engineering is very important for security auditing, where a user has to ensure code properties such as the absence of malware.

Many of these applications are highly relevant for embedded applications where access to the code is usually even more difficult than in PC software cases. Usually the program memory is read protected. Often the whole system is proprietary and supplied as-is by a single manufacturer. Many times the user has almost no access to any internal information of the underlying device. In such scenarios the side channel disassembler can be an effective tool for reverse engineering and other applications. Compared to other reverse engineering tools, the side channel disassembler features several advantages:

- it is a low-cost method, as it does not require special equipment, except for a digital sampling oscilloscope which nowadays is available for as little as 1500 $ or can be found in almost every electronics lab.
- it is a non-invasive procedure. Either the power is measured on one of the power supply pins of the target device or, even easier, only the electromagnetic emanation is used for reversing the code. In both cases only limited access to the microprocessor is needed.
- contrary to classical disassembler designs, the side channel disassembler directly gives information on the program flow, as the executed code rather than the code in program memory is analyzed.

Especially the last advantage makes the side channel disassembler unique and hence a useful and innovative addition to other existing reverse engineering methods. Out of the many possible scenarios for usage, we have extracted three application scenarios that highlight the advantages of the side channel disassembler.

Code Recognition One scenario where the side channel disassembler is useful is in cases where known code needs to be recognized. The presented methods can be used to locate certain code segments, e.g. to detect where an error occurred or even to detect a firmware version executed on an embedded system. In a similar application a company might suspect a copyright breach or a potential patent infringement by a similar product. It can then use the side channel disassembler to identify such an infringement or breach by showing that an own known code is executed on the suspicious device, even if the program memory is read-protected.

In general, the disassembler can be operated on a portable device equipped with an antenna. Ideally, holding an antenna close to the processor might be

sufficient to track the program flow executed on the target device[1]. The disassembler is then used to locate the relevant code parts.

Code Flow Analysis A related problem in embedded software analysis is tracking which parts of a code are executed at a certain time. This case occurs when the code is known, but the functionality is not, a typical scenario in reverse engineering. Using the methodology described in this paper allows mapping known code to a certain functionality. The user might also just be interested in learning which parts of the code are used most often, e.g., for evaluating possible beneficiary targets for performance optimization.

Code Reverse Engineering The side channel disassembler can be used to reconstruct unknown executed code. Ideally, if well trained, it can directly reconstruct the program code, identify functions etc. Furthermore, it might be able to identify higher language properties, the used compiler (if the code was generated from a higher language) etc. The latter points can be achieved in combination with other available code reverse engineering tools.

Reverse engineering can also be very interesting for analyzing embedded crypto applications. Especially in cases where the applied cipher is unknown, the disassembler is a great tool to reconstruct unknown routines. It can also be used to align code as a preparatory step for DPA attacks on code with a varying execution time or the shuffling countermeasure.

It might not always be desirable by the implementer that his code can be reverse engineered by a power disassembler. Since the methods of the side channel disassembler are borrowed from side channel research, we can do the same for the countermeasures. One should keep in mind that we are not targeting data, hence masking or shuffling of data will not increase resistance. Instead, the additional variation of data might be used to get better results through repeated measurements. Hence, countermeasures in hardware show better properties to prevent side channel disassembly. Hardware countermeasures have been proposed on different design levels and can be found on smart card processors and also on other secure microcontrollers [12]. All of them are very strong against SPA approaches such as the side channel disassembler. Keep in mind that the countermeasures need only be as good as the protection of the program code itself. So if the code can be easily extracted, simple countermeasures against power analysis suffice.

6 Conclusion

In this work we have presented a methodology for recovering the instruction flow of microcontrollers based on side channel information only. We proved the chosen methods by applying them to a PIC microcontroller. We have shown that subspace based template recognition makes an excellent choice for classifying power leakage of a processor. The employed recognition methods achieve a high average instruction recognition rate of up to 70%. To exploit prior general knowledge about microcontroller code we proposed Markov modeling of the processor to further increase the recognition rate of the template process. Depending on the chosen algorithm, the recognition rate was increased by up to

[1] Standaert et al. [21] showed in a setup similar to ours that the EM side channel can be expected to contain more information than the power side channel.

17% points. The recognition performance is strongly influenced by the assumed distribution of data, resulting in a decreased recognition rate of up to 58% for real programs. Though the recognition rates on real code are leaving space for further improvements, they are more than an order of magnitude higher than the lower boundary given by simply guessing the instructions.

Hence, we can positively assure that side channel based code reverse engineering is more than just a theoretic possibility. Applying the presented methodology allows for building a side channel disassembler that will be a helpful tool in many areas of reverse engineering for embedded systems.

References

1. E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Advances in Cryptology - CRYPTO '97*, volume 1294, pages 513–525. Springer LNCS, 1997.
2. C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, August 2006.
3. S. Chari, J. R. Rao, and P. Rohatgi. Template Attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *LNCS*, pages 13–28. Springer-Verlag, 2002.
4. C. Clavier. Side channel analysis for reverse engineering (scare) - an improved attack against a secret a3/a8 gsm algorithm. Cryptology ePrint Archive, Report 2004/049, 2004. `http://eprint.iacr.org/`.
5. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis: Probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998.
6. G. A. Fink. *Markov Models for Pattern Recognition*. Springer, 2008.
7. M. Goldack. Side-channel based reverse engineering for microcontrollers. Master's thesis, Ruhr-Universität Bochum, Germany, 2008.
8. C. Igel, T. Glasmachers, and V. Heidrich-Meisner. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.
9. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. I. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer Verlag, 1996.
10. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, London, UK, 1999. Springer-Verlag.
11. D. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.
12. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks*. Springer, 2007.
13. Microchip Technology Inc. MPASM Assembler, MPLINK Object Linker, MPLIB Object Librarian User's Guide, 2005. `http://ww1.microchip.com/downloads/en/DeviceDoc/33014J.pdf`.
14. R. Novak. Side-Channel Attack on Substitution Blocks. In *Applied Cryptography and Network Security (ACNS 2003)*, 2003.
15. E. Permadi. The Hardware side of Cryptography. Personal Blog. `http://edipermadi.wordpress.com/`.
16. J.-J. Quisquater and D. Samyde. Automatic Code Recognition for smart cards using a Kohonen neural network. In *CARDIS'02: Proceedings of the 5th Smart Card Research and Advanced Application Conference*, Berkeley, CA, USA, 2002. USENIX Association.
17. L. R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

18. C. Rechberger and E. Oswald. Practical Template Attacks. In *Workshop on Information Security Applications — WISA 2004*, volume 3325 of *LNCS*, pages 440 – 456. Springer, 2004.
19. W. Schindler, K. Lemke, and C. Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *LNCS*, pages 30–46. Springer, 2005.
20. S. P. Skorobogatov. *Semi-invasive attacks – A new approach to hardware security analysis*. PhD thesis, University of Cambridge, April 2005. `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.pdf`.
21. F.-X. Standaert and C. Archambeau. Using Subspace-Based Template Attacks to Compare and Combine Power and Electromagnetic Information Leakages. In *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154, pages 411–425. Springer, 2008.
22. F.-X. Standaert, F. Koeune, and W. Schindler. How to Compare Profiled Side-Channel Attacks. In *ACNS 2009*, volume 5536, pages 485–498. Lecture Notes in Computer Science, 2009.
23. S. Tolmie. PIC Sample Code in C. `http://www.microchipc.com/`.
24. D. Vermoen. Reverse engineering of Java Card applets using power analysis. Master's thesis, TU Delft, 2006. `http://ce.et.tudelft.nl/publicationfiles/1162_634_thesis_Dennis.pdf`.
25. Web site. Program Code for Keeloq Decryption. `http://www.pic16.com/bbs/dispbbs.asp?boardID=27&ID=19437`.