Non-Intrusive Program Tracing and Debugging of Deployed Embedded Systems Through Side-Channel Analysis

Carlos Moreno
University of Waterloo
cmoreno@uwaterloo.ca

Sebastian Fischmeister
University of Waterloo
sfischme@uwaterloo.ca

M. Anwar Hasan
University of Waterloo
ahasan@uwaterloo.ca

Abstract

One of the hardest aspects of embedded software development is that of debugging, especially when faulty behavior is observed at the production or deployment stage. Non-intrusive observation of the system's behavior is often insufficient to infer the cause of the problem and identify and fix the bug. In this work, we present a novel approach for non-intrusive program tracing aimed at assisting developers in the task of debugging embedded systems at deployment or production stage, where standard debugging tools are usually no longer available. The technique is rooted in cryptography, in particular the area of side-channel attacks. Our proposed technique expands the scope of these cryptographic techniques so that we recover the sequence of operations from power consumption observations (power traces). To this end, we use digital signal processing techniques (in particular, spectral analysis) combined with pattern recognition techniques to determine blocks of source code being executed given the observed power trace. One of the important highlights of our contribution is the fact that the system works on a standard PC, capturing the power traces through the recording input of the sound card. Experimental results are presented and confirm that the approach is viable.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Tracing

General Terms Theory, Algorithms, Experimentation

Keywords Embedded systems; debugging; tracing; side-channel analysis; simple power analysis

1. Introduction

Debugging is one of the hardest aspects of embedded software development. The task is especially hard when the faulty behavior is observed at the production or deployment stage, when the software no longer has any auxiliary components dedicated to assist in the debugging task [3]. For systems at this stage of the development cycle, non-intrusive observation of the system's behavior is likely the only available technique — developers are no longer allowed to modify the source code, or even re-compile to include or activate the debugging tools. Moreover, if we need to restart the device to enable any available debugging techniques, we may not be able to reproduce the faulty behavior that the device was exhibiting. Without these debugging tools usually available in earlier phases of development, developers may be limited to non-intrusive observation, which often provides insufficient information to infer the cause of the problem and identify and fix the bug.

In this work, we present a novel approach for non-intrusive debugging of deployed embedded systems. The system can be observed and an output indicating the sequence of executed code is produced, without having to modify anything in the target system or even restart it. The technique is rooted in cryptography, in particular the area of side-channel attacks [11, 12]. These types of cryptographic attacks take advantage of the relationship between the instructions that a processor is executing and the data it is working with, and observable side-effects such as timing of computations (timing attacks), power consumption (power analysis), or electromagnetic emissions (EM analysis) to obtain the secret data (typically an encryption key). The techniques, as they exist, are not directly applicable to the debugging of embedded software, since they focus on obtaining specific pieces of secret data embedded in the device (and inaccessible through "legitimate" means), and they typically require interaction and direct control over what the target device is executing.

On the other hand, the goal when tracing and debugging a deployed embedded system is to analyze an operating device for which we have observed a faulty behavior, and obtain information allowing us to identify and fix the bug. It may be essential that we allow the device to continue its operation without restarting it or in any way exerting control over what the device is doing; otherwise, we could lead the device to a state where we may not be able to reproduce the faulty behavior.

As an additional aspect in terms of motivation, this tracing system could be used for monitoring as an *intrusion detection system* (IDS) [14]. In the wake of threats like Stuxnet [13], one should consider adapting tools like IDSs, classically viewed as applicable to servers and networks, to embedded systems as well. Unlike a software-based IDS embedded in the device, our approach could lead to a tamper-proof IDS, given that the monitoring system is physically independent of the device and the software running in it; thus, any malware that tampers with the functionality of the device will not be able to tamper with the IDS and as a consequence, any anomaly in the device's behavior will most likely be detected.

Our proposed technique focuses on power consumption (though the underlying techniques are in principle applicable to EM emissions). A current sensing shunt resistor is placed in series with the Power-In signal going to the Microcontroller Unit (MCU),[1] producing a voltage proportional to the current being consumed. The

LCTES'13, June 20–21, 2013, Seattle, Washington, USA.

[1] Though the technique is applicable to both CPUs and MCUs, we use MCU throughout the paper, to simplify the text, and also since it is the more likely target for our technique.

resistor is selected to produce a voltage in the range of a few millivolts, thus not affecting the operation of the device. Our technique expands the scope of the cryptographic techniques so that we recover the sequence of operations executed by a processor, as opposed to simply one piece of data accessed during a particular operation of the device. To this end, we use digital signal processing techniques (in particular, spectral analysis) to extract *features* of the signal (the power trace) that allow us to match sections of the power trace against fragments of the source code through the use of statistical pattern recognition techniques [21]. It is reasonably likely that this information would be valuable for the purpose of identifying and fixing the bug. We observe that in the context of embedded systems, this relationship between operations being executed and power consumption has been used for the purpose of estimating or minimizing power consumption, obtaining power consumption as a function of the executed instructions. Going in the other direction may be seen as a far bigger challenge, for at least two reasons: (1) the operation being executed is not uniquely determined as a function of the power consumption; thus, information about the progression of power consumption through an interval of time may be needed, combined with statistical processing; and (2) we need to get around the "polluting" effect of the data the processor is working with (i.e., the same operation with different data produces a different amount of power consumption) and the measurement noise.

One of the important highlights of our contribution is the fact that the system works on a standard personal computer (PC), capturing the power traces through the recording input of the sound card — side-channel analysis techniques usually rely on digital oscilloscopes or other expensive and bulky pieces of equipment. A standard, reasonably high-quality sound card (24-bits, 192kHz sampling rate, nowadays available at prices below $200) suffices to make the system work on a wide range of microprocessors and microcontroller units. Given the typical computing power of today's mainstream PCs, with this setup, we claim that on-the-fly processing is within reach for a wide range of target devices. Of course, the technique is suitable for use with a digital oscilloscope; indeed, for processors with high clock frequencies, higher sampling rates will be required for the system to work, most likely without on-the-fly processing, depending on the clock frequency and architectural aspects of the target device such as pipeline depth, memory management unit (MMU), and cache memory [9].

Experimental results confirm the validity of our technique and its practical aspects: we used an Atmel MCU, AVR Atmega2560 [1] as the target device (8-bit MCU running at 1MHz) and we included a subset of the MiBench suite [6] as a set of tasks representative of typical embedded software (at least typical for certain applications areas). The technique was implemented on a mainstream PC with an HT Omega Claro+ sound card [10], which at the time of purchase the cost was around $150, and as the results show, the setup is viable for on-the-fly processing obtaining accurate results.

The remaining of this paper is organized as follows: in Section 2 we briefly review the notions from side-channel analysis as well as the digital signal processing techniques and the statistical pattern recognition techniques used. Section 3 presents our proposed technique. Section 4 describes the experimental setup used to verify the validity and applicability of our approach, with Section 5 presenting the results. A brief discussion, future work and concluding remarks follow, in sections 6 and 7.

2. Background

Our work draws upon background from different areas: side-channel analysis, pattern recognition, and digital signal processing. We briefly review some of the basic notions and mathematical background from these areas.

2.1 Side-Channel Attacks and Simple Power Analysis

Side-channel analysis plays an important role in the area of embedded systems security. Mobile, hand-held, and many other types of embedded devices increasingly make use of cryptographic techniques, in particular public-key cryptography and elliptic-curve cryptography (ECC), where exponentiation with large *secret* exponents is one of the central operations on which the security of the system relies (we refer the reader to [16] or [8] for more details).

Paul Kocher pioneered the field of side-channel analysis, showing that even though the cryptographic algorithms are secure from a mathematical standpoint, *implementations* of such algorithms may be vulnerable to attacks that use side-effects of the computation — side-effects that can be observed by an attacker with physical access to the device [11, 12].

Of interest to us is power analysis, in particular simple power analysis (SPA), where the relationship between what the processor is executing at a given time and the power consumption at that time is exploited to recover the secret exponent with a single power trace during the execution of the exponentiation. Many countermeasures, as well as different encryption techniques and different attacks exist; we omit a more detailed description since this is beyond the scope of this work. The interested reader may consult [18] for a more in-depth description as well as several of the existing countermeasures.

We borrow and expand upon this idea of using the relationship between computation and power consumption to determine what the processor is doing given an observation of power consumption (a power trace). Our technique is more general (and thus, requires additional, novel approaches to processing the power trace) in that side-channel attacks focus on recovering specific pieces of data, and some of the more advanced side-channel attack techniques require that the attacker exert control over what the device is doing. In our case, however, it is essential that we observe what a faulty device is doing without disrupting it, to avoid the possibility of leading the device to a state where we are unable to reproduce the faulty behavior.

Though template attacks [2] use a technique that is somewhat closer to our approach, there is still a fundamental difference in that they use pattern matching for the noise characteristics, relying on the *diffusion* property of encryption techniques, making it a cryptography-specific approach.

2.2 Statistical Pattern Recognition

At the heart of our proposed technique are elements from the field of pattern recognition, since our goal is to *classify* a given segment of execution as an instance of one of the possible fragments of source code according to a database, given noisy observations that in principle provide enough statistical information to determine the most likely fragment of code that produced such observation [21]. We notice that the set of all possible fragments of code being executed under normal conditions is known with certainty — since we use our technique for tracing and debugging, we can safely assume that the source will be accessible.

Since we do not count on an analytic model for the probability distribution, we can resort to techniques based on databases of *training samples*, for which the classification is known with certainty. These training samples are in principle a set of values drawn from the probability distribution for the process in question. Thus, they should be representative of the probability distribution of the process. The task of the classification system is described as follows: Let $\mathbf{X}$ be a random variable corresponding to a *feature vector* with features from a given sample associated with an unknown

class C from a set of Q possible classes $\mathcal{C} = \{C_1, C_2, \cdots, C_Q\}$. The task of the pattern recognition system is that of determining the class C to which the feature vector $\mathbf{X}$ corresponds with highest a posteriori probability:

$$C = \arg \max_{C_k \in \mathcal{C}} \{\Pr\{C_k \mid \mathbf{X}\}\} \quad (1)$$

Among the common techniques used to achieve this goal are Linear Discriminant Functions (LDF) and Nearest Neighbors. With LDFs, the training phase of the system collects a database of S *labelled* samples $\{\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_S\}$ of feature vectors (the label being the class C to which the sample is *known* to correspond). For each class C_k, we compute the sample average or centroid $\overline{\mathbf{C}}_k$ as

$$\overline{\mathbf{C}}_k = \frac{1}{S_k} \sum_{\mathbf{X}_i \in C_k} \mathbf{X}_i \quad (2)$$

where S_k is the number of training samples labelled as C_k.

In the detection or classification phase, a given feature vector $\mathbf{X}$ is associated to the class C that corresponds to the nearest centroid (usually Euclidean distance in the multi-dimensional feature space is used):

$$C = \arg \min_{C_k \in \mathcal{C}} \{ \|\mathbf{X} - \overline{\mathbf{C}}_k\| \} \quad (3)$$

The LDF corresponds to a hyperplane orthogonal to the line between the two centroids and intersecting that line at the point equidistant from the centroids, providing an efficient implementation mechanism.

For the Nearest Neighbor (NN) rule, the classification phase associates a given feature vector $\mathbf{X}$ to the class of its nearest neighbor among all training samples:

$$C = C_I \text{ with } I = \arg \min_{1 \leqslant i \leqslant S} \{ \|\mathbf{X} - \mathbf{X}_i\| \} \quad (4)$$

The k-Nearest Neighbors (k-NN) rule [21] provides a higher level of robustness with respect to noise in the measured features. Given a feature vector $\mathbf{X}$, we obtain the k nearest neighbors among all training samples, and the classification is done by majority vote among the k labels of these nearest neighbors. That is, if the k nearest training samples have labels $\{C_{n_1}, C_{n_2}, \cdots, C_{n_k}\}$, then feature vector $\mathbf{X}$ is associated to class C given by

$$C = C_I \text{ with } I = \arg \max_n \left\{ \sum_{\substack{i=1 \\ n_i = n}}^{k} 1 \right\} \quad (5)$$

2.3 Spectral Analysis of Digital Signals

One of the fundamental concepts when applying spectral analysis to digital signals is that of the Discrete Fourier Transform (DFT). Given a discrete-time signal $\mathbf{x}$ of finite duration, represented by a sequence of N real values $\mathbf{x} = \{x_0, x_1, \cdots, x_{N-1}\}$, its DFT $\boldsymbol{\mathcal{X}}$ [19] is given by the sequence of N complex values $\boldsymbol{\mathcal{X}} = \{\mathcal{X}_0, \mathcal{X}_1, \cdots, \mathcal{X}_{N-1}\}$, where each $\mathcal{X}_k$ is given by

$$\mathcal{X}_k = \sum_{n=0}^{N-1} x_n \, \mathrm{e}^{-\mathrm{j} \frac{2\pi k n}{N}} \quad (6)$$

where j denotes the imaginary unit [2] (i.e., $\mathrm{j}^2 = -1$)

A straightforward implementation clearly takes $\mathrm{O}\left(N^2\right)$ time to compute the DFT of a sequence of N values. In practice, Fast Fourier Transform (FFT) is normally used, being an efficient algorithm to compute the DFT. FFT exploits the symmetry in the DFT to implement an in-place divide-and-conquer [4] algorithm and obtain the DFT in $\mathrm{O}\,(N \log N)$ time.

For our system, we used the libfftw [5] implementation, as it is, to the best of our knowledge, a correct and very efficient FFT implementation.

[2] We use the standard "electrical engineering" notation j for the imaginary unit, to avoid the confusion of i with the standard symbol used to denote electrical current or intensity.

3. Our Proposed Technique

As briefly described in Section 1, our proposed technique is centered around the idea of non-intrusively measuring power consumption as a function of time (i.e., capturing power traces), and use the relationship between what the processor is executing and the power consumption, combined with statistical processing to determine the sequence of instructions that were executed, thus assisting in the debugging process. To this end, a current-sensing shunt resistor is placed in series with the Power-In line going to the MCU, so that a voltage proportional to the power consumption is produced. This shunt resistor is selected to produce a voltage in the order of a few millivolts, thus not disrupting the functionality of the MCU. This voltage is then captured through the Line input of a sound card, as shown in Figure 1.

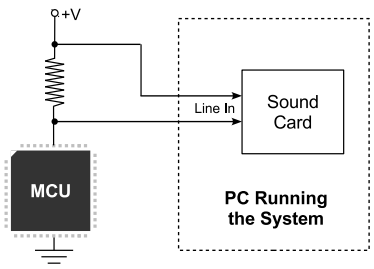

Figure 1. Simplified Diagram of our System.

The technique is centered around the idea of identifying fragments of code, corresponding to segments of the power trace. These fragments need to be sufficiently long so that: (1) there is a large enough amount of actions happening at the circuitry level to create a distinctive profile of power consumption. As an example, assigning a variable is unlikely to be distinguishable from any other operation involving memory, or even from a portion of some other action, such as fetching an instruction, or the additional memory access for instructions with immediate operands; and (2) so that the signal (in our setup, produced by an MCU running at 1MHz) can be sampled by a sound card at lower sampling frequency (in our setup, the sampling rate was 96kHz) and be able to extract meaningful information from it.

In this work, which is the initial phase of a longer project (we discuss future work in Section 6), we decided to use whole functions as the fragments of code to be considered (with only one exception, as will be discussed in Section 4 when describing our experimental setup). This decision was somewhat ad hoc, mainly related to simplifying the system in the context of a study of the feasibility of the technique. We do discuss an alternative approach and its potential benefits in Section 6.

The pattern recognition system then extracts features from these segments of the power trace, and uses one of the classification techniques described in Section 2.2. We tried and evaluated several classification techniques: nearest centroid techniques using LDFs and nearest neighbors techniques. We observed a much better classification performance when using the k-NN technique. We believe that the main factor is the fact that a function can do alternative things depending on the input data — and in general, a given fragment of code could do different things depending on the data it is working with. This leads to different execution times for different

instances of the same function, and in general, it may lead to feature vectors that tend to be spread in the feature space, making the technique based on centroids less effective. For the k-NN rule, we tried values between 3 and 100 for k, obtaining best results with $k = 5$ for individual classification and $k = 21$ for continuous classification (we discuss this distinction in Section 4 as part of the description of our experimental setup). Notice that the use of techniques such as cross-validation [21] was not really necessary in our case, since additionally to the training set, we have the test set to evaluate the performance, and for that set the correct classification for each sample is known a priori.

One of the difficulties in our scenario is that the processing for the classification needs to be done in a continuous way, and it is the system's responsibility to achieve synchronization with the fragments of code to detect. That is, the system is not given a power trace with the assurance that it is the power trace for one of the fragments of code. Instead, the system is given a single power trace that extends indefinitely (in any case, as long as the system is running), and it has to apply the pattern recognition technique for variable starting position and length of the sequences to classify. As an example to illustrate the difficulties arising from this constraint, we can not use the length of the power trace as one of the features to extract — if we could, then this would provide a very relevant piece of information that even alone would give a very high probability of correct classification (since we could always select fragments of code that execute with distinct durations).

The starting point of the fragment is mainly a problem when the system starts up and has to synchronize to the execution; after that, then having recognized/classified a given section of the power trace provides information of where that fragment ends, so the starting point for the next item to be classified is then known, even though adjustments may be necessary to compensate for "noisy" or incorrect outcomes from the previous segments (e.g., a segment that was in reality L samples long may have been detected as being L' samples long). This will be discussed in more detail in Section 4. The system has to try various lengths and see which one gives the closest match with training samples from the database.

When deciding what parameters to use as features to be extracted from the entities (in our case, the power traces), there is often a bit of heuristics and intuition involved, especially when there is no analytic or otherwise simple description of a PDF with "nice" characteristics. To evaluate how good this feature set is, the rate of correct classifications does provide a quantitative measure. In particular, it does implicitly account for the distance between elements of different classes and how spread the PDF is. Misclassifications occur when elements are not far apart given the spread nature of the PDF; thus, the rate of correct classifications tells us how good this feature set is for the purpose of the classification process.

In our case, we decided to use spectral information — logarithmic magnitude and phase — as the feature vector. The intuition on why spectral information may give useful *and robust* information to identify the power trace as corresponding to one of the given fragments of code is based mainly on the following two aspects:

- Getting around issues of alignment — spectral contents are similar even when the signal or portions of the signal are shifted. Thus, for different instances of the same function, prominent portions of the code may still be common to all other traces, but located at different points in the trace (as the result of conditional execution affected by the input data).
- Variations due to "disrupting" factors in the system (such as noise or artefacts that occur due to the mechanism of leakage to the side-channel or the measurement) tend to produce higher deviations in the signal than in its spectrum, making the latter a more robust tool to identify a given power trace. In any case, the deviations in the spectrum tend to have simpler patterns, making it easier to extract the identifying features from spectral information than directly from the signal.

One additional difficulty, for which we resorted to a heuristic approach as our adopted solution, comes from the fact that we use the DFT of the trace directly as the feature vector; that is, each of the N elements of the DFT (more precisely, its complex logarithm, which directly provides logarithmic magnitude and phase) corresponds to one of the coordinates (or one of the dimensions) in the N-dimensional feature space. However, since different traces have different lengths, then we do not have a fixed value of N. That is, computing and comparing Euclidean distances in the feature space poses a challenge. This was an additional issue that contributed to our decision to use k-NN instead of the nearest centroid classification technique, which is not directly suitable for variable-size fragments. We attempted to solve this by computing spectra at a fixed size through zero-padding to obtain an interpolated version of the spectrum at this higher resolution; this proved to be computationally expensive, in addition to exhibiting poor performance compared to the k-NN rule, as already mentioned.

Our heuristic includes two aspects: First, when given a trace and a starting point, we try all of the lengths present in the training database. That is, when looking for the nearest neighbors among the training samples, for each sample from the database, we take its length and consider the segment of the trace that matches that length, so that the distance can be evaluated. This is also consistent with the idea that we need to try different lengths, since we are only given the starting point, but the system needs to determine the length of the fragment as part of the task of identifying it.

The second aspect is that, given the detail mentioned above, it is clear that comparing distances for pairs of traces of one length with distances for pairs of traces of a different length becomes an issue. To get around this, we used the notion of a *normalized* distance, where we normalize with respect to the number of dimensions. As an example, if we have two tridimensional vectors, say

$$\begin{aligned} \mathbf{u_1} &= (x_1,\, x_2,\, x_3) \\ \mathbf{u_2} &= (x_1+\delta,\, x_2+\delta,\, x_3+\delta) \end{aligned}$$

then we get [3]

$$|\mathbf{u_2} - \mathbf{u_1}|^2 = 3\delta^2$$

Our intuition is that for two, say, 5-dimensional vectors

$$\begin{aligned} \mathbf{v_1} &= (y_1,\, y_2,\, y_3,\, y_4,\, y_5) \\ \mathbf{v_2} &= (y_1+\delta,\, y_2+\delta,\, y_3+\delta,\, y_4+\delta,\, y_5+\delta) \end{aligned}$$

the distance, in the context of comparing which of the two pairs are closer, should be the same, since each of the coordinates, corresponding to one descriptive feature, are equally apart. However, a direct Euclidean distance computation for this case gives us $|\mathbf{v_2} - \mathbf{v_1}|^2 = 5\,\delta^2$

Thus, to avoid the nearest neighbors selection to be biased towards the shorter traces, we need to normalize by computing the *square distance per dimension*. Also related to this issue of traces with different lengths: a longer trace may be at a disadvantage if sub-sections of it provide a sufficiently good match to other, shorter traces. We observed that this was the case for the set of MiBench functions that we used. Two different approaches were considered: (1) using an adjustment factor to favor longer traces when otherwise approximately equally close matches; and (2) using an adjustment factor to favor matches at the "nominal" position as determined by the classification at the previous iteration. We tried both approaches, and (2) had the severe adverse effect

[3] We use square distance since this is the common approach used when implementing NN rules.

of reducing the ability to maintain synchronization with the trace, especially resynchronizing after a misclassification.

Putting all the pieces together, we define our distance metric as follows: given a trace $\mathbf{x}$ of length N, with DFT $\boldsymbol{\mathcal{X}}$, the associated feature vector is given by

$$\mathbf{X} \;=\; \{\mathrm{Log}\;\mathcal{X}_0, \mathrm{Log}\;\mathcal{X}_1,\; \cdots\;, \mathrm{Log}\;\mathcal{X}_{N-1}\} \tag{7}$$

where $\mathrm{Log}(\cdot)$ denotes the complex logarithm function. With this, the distance between N-dimensional feature vectors $\mathbf{X}$ and $\mathbf{Y}$ is given by

$$\|\mathbf{X} - \mathbf{Y}\| \;=\; \frac{1}{N} \sum_{k=0}^{N-1} |\, X_k - Y_k \,|^2 \tag{8}$$

where X_k and Y_k are the entries in the feature vectors, corresponding to the complex log of the DFT entries.

4. Experimental Setup

This phase of our work consists of two experiments. In the first experiment we evaluate the effectiveness of the pattern recognition system by classifying power traces of known fragments of code and determining the success rate or *precision* (the fraction of power traces that were classified correctly). That is, we test detection of the various fragments of code in isolation, and evaluate the performance of the classification system. To isolate the trace corresponding to the exact time interval of execution, we use markers which are actions known to have high power consumption and thus produce a prominent pulse in the trace. This is one of many possible approaches, and we chose it for our experiments due to its simplicity. Given our STK600 setup, we used the LEDs for this purpose. Figure 2 illustrates these steps.[4]

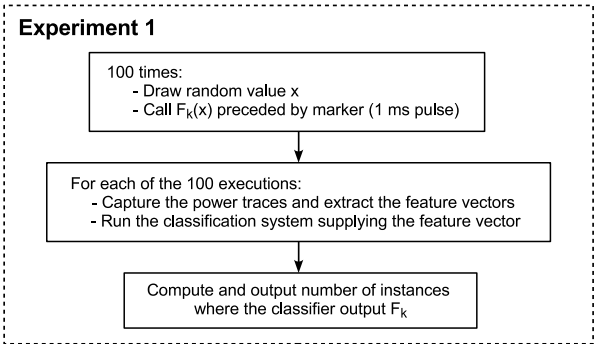

Figure 2. Experiment 1 – Classifier Performance.

In the second experiment, we execute a sequence of function calls (each function being one of the fragments of code for the pattern recognition system) and have the system determine the sequence of functions that was executed, with the power trace as its only input. Figure 3 shows the details. The first experiment tests the building blocks, the basic operations of the system, while the second experiment aims at modeling the operation of an eventual practical implementation of our proposed technique.

For the second experiment, the classification system tries adjusting the starting position to help resynchronizing in the cases where an incorrect classification occurs. That is, given a classification at iteration n, the starting position for the trace segment at iteration $n+1$ is determined by the length of the classified trace. However, if the classification at iteration n was incorrect, the starting position for iteration $n+1$ may be incorrect. Trying starting positions in an interval around this "nominal" starting position allows the system to reestablish synchronization after a misclassification occurs. This will be shown in Section 5.2, where we show the experimental results. For more details, the reader can consult [17] (Chapter 7 and appendices, where source code for the classification programs is shown).

[4] For some of the functions, the input is a graph or a tree. For these cases, we inserted random values in the data structure.

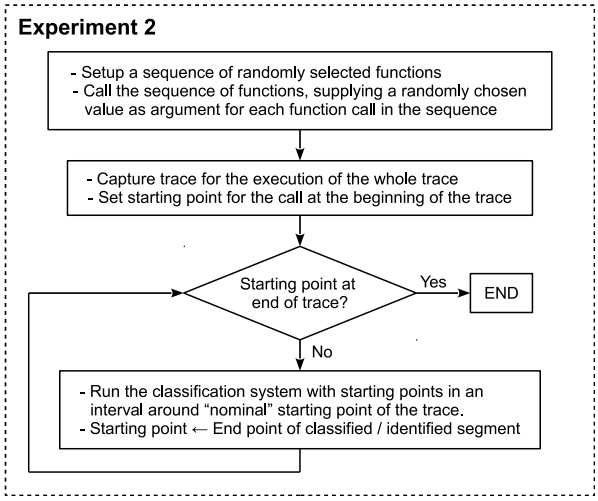

Figure 3. Experiment 2 – "Online" operation.

Notice that for continuous operation, the system could consider the fact that the sequence of fragments is not arbitrary, and classification at a certain point can be narrowed down according to the control-flow graph (CFG) obtained from the source code. In this initial phase of the project, we did not include this aspect, and we will discuss it more in detail in sections 5 and 6.

Both experiments rely on the training phase of the classification system — thus, in this initial step we execute each of the functions S_c times, where S_c corresponds to the number of training samples per class (per fragment of code to be detected); we decided to use $S_c = 1000$ as a reasonably large number to be used as a starting point (we present a more detailed discussion in the next section). The fragments of code, denoted $\mathrm{F_k}$ ($1 \leqslant k \leqslant |\mathcal{C}|$), are either functions or fragments of a function, and for each call we supply a randomly selected value as input argument for the function, as illustrated in Figure 4. For Experiment 2, the training samples

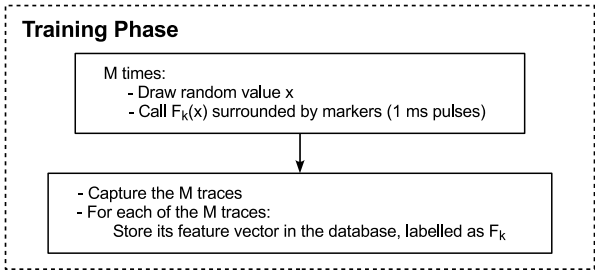

Figure 4. Training Phase.

were not marked by surrounding them with pulses, but rather, surrounding them with some other (randomly selected) function, since this is how they would appear in the classification phase when operating in continuous mode. We emphasize the detail that these traces used for the training database are different from those being identified/classified, since "fresh" random values are chosen at every instance as parameter values — the fragments of code are

part of the same set of possible classes, but each instance of a trace being classified is different from every trace in the training database.

In a real-life application, this training step should consider, if available, the probability distribution for the arguments to each function. For example, if a given function receives as input parameter the measured temperature, we will draw values from a normal distribution with mean 25 °C and relatively small variance.[5] For our experiments, we used uniformly distributed random variables in a reasonable range (the ranges were consistent between the training phase and the classification phase). This does not take into account the possibility of a function being called with unreasonable parameters due to a defect in the software; however, one can easily compensate for this aspect by including a small fraction of training samples using parameter values outside the reasonable range.

The experiments required a target device and a host platform to compile the programs and "flash" the target device, capture and process the traces, and analyze the results. For the host platform, we used an Ubuntu Linux system, with avr-gcc 4.3.5 and avrdude 5.10. The target device was an Atmel AVR Atmega2560 MCU on an STK600 board [1], and we assembled a quick prototype card to facilitate the connections — a snap-in connector to place the shunt resistor so that different values can be easily tried (in our case, a 10 Ω resistor produced voltage in the correct range), pins to connect oscilloscope probes (for verification purposes, or for future experiments using a digital oscilloscope), and an RCA audio connector to easily connect to the input of the sound card. Figure 5 shows a photograph of this simple prototype card. The

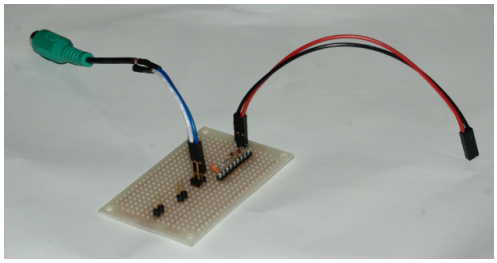

Figure 5. Prototype Card to Facilitate Connections.

red/black cable on the right, ending in a two-pin header connector goes to the VTARGET connector on the STK600 board (so that current to the MCU passes through), and the green connector on the left is the RCA audio connector.

The sound card was an HT Omega Claro+ [10], and we used Audacity [15] to record the power traces. Figure 6 shows a screenshot from one of the power traces in the training phase, showing the two surrounding pulses.

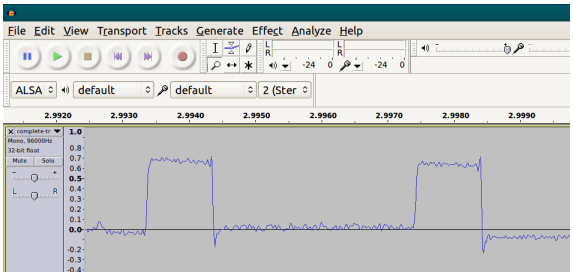

Figure 6. Screenshot – Power Trace in Audio Editor.

[5] Assuming a system intended to work at room temperature.

We used code from MiBench [6] as the source code for the target device in the experiments. That is, the set of fragments $\mathcal{C}$ includes fragments of code from MiBench; in particular, from the telecommunications, network, and security sections of it. We excluded code that required file access or intensive operations as well as code for which we required many modifications for it to compile with avr-gcc. We also excluded redundant items — for example, from the security section, there are several symmetric encryption algorithms and several hash functions; we used only AES (which is the one generally recommended for practical use) for a sample of symmetric encryption, and SHA as a sample of a cryptographic hash function. For simplicity reasons, our work currently operates at the granularity level of entire functions (that is, the fragments of code to be matched are entire functions), with the exception of the SHA algorithm. This exception is due to the fact that SHA executes a large number of rounds repeating the same procedure, thus taking a very long time to execute, making it more reasonable to choose that procedure as the fragment to consider. The exact set of functions used for our experiments is the following: ADPCM encode, ADPCM decode, CRC-32, FFT, SHA (fragment), AES (Rijndael) symmetric encryption, Dijkstra's shortest path algorithm, Patricia Trie (insertion), and pseudo-random number generation (C's `random()` function).

5. Results

We now present and discuss the results for both phases of the experimental setup.

5.1 Experiment 1 – Individual Classification

For Experiment 1, we evaluate and report the precision of the classifier. Since the classifier chooses one of the possible classes (one of the functions being considered), there are no false negatives; that is, there is always an output from the classifier, and it is either a true positive or a false positive. Thus, the precision fully describes the performance of the classifier. The precision P is given by

$$P \triangleq \frac{T_{\mathrm{P}}}{T_{\mathrm{P}} + F_{\mathrm{P}}}$$

where T_{P} denotes the number of true positives (i.e., correct classifications) and F_{P} denotes the number of false positives (i.e., misclassifications, or incorrect classifications). For example, consider a scenario with ten candidate functions, F_1, F_2, $\cdots$, F_{10}, Experiment 1 is run and it executes 100 times function F_1. The classifier outputs 90 times F_1, 4 times F_2, 3 times F_5 and 3 times F_8. Then, the number of true positives is 90, and the number of false positives is $4 + 3 + 3 = 10$. The resulting precision for this example, P_{ex}, would be

$$P_{\mathrm{ex}} = \frac{90}{90 + 10} = 0.9 \quad (90\%)$$

We first adjusted the system's parameters to obtain the best performance. For each of the functions, we initially captured 1000 training samples, but then varied the number of samples effectively used, to determine the optimal value (optimal within the range 1 to 1000, which is the maximum number of available samples). We first maintained the same number of samples for every function and varied the value to obtain the optimal. We then used this as the initial estimate in a simple optimization procedure to determine the optimal number of training samples *for each function*. We omit any additional details or figures, since there is nothing particularly relevant that they would show.

For some of the functions, more than 1000 samples were required for good performance, so we captured additional traces for those. In particular, functions like Dijkstra's shortest path required a larger set, possibly due to the variable nature of the

algorithm — depending on the graph contents (weights, connections, etc.) there may be wide variations in execution time, requiring larger numbers of training samples to compensate for the spread nature of its PDF. For CRC32, we were obtaining a low precision when using 1000 training samples, so we increased the number of samples for this one as well.

With these parameters in place, we started measuring the performance for Experiment 1. Table 1 shows the results for each of the functions being tested. That is, it shows the precision obtained for the classifier when executing each different function.

Function	Precision
adpcm_encode	100%
adpcm_decode	97%
CRC32	92%
FFT	99%
SHA (Fragment)	100%
AES (Rijndael)	97%
Dijkstra's shortest path	98%
Patricia Trie (insertion)	100%
`random()`	99%
Overall (avg.)	**98.0%**

Table 1. Classifier Precision

The results clearly indicate an excellent performance for the classifier, with only one of the functions scoring a 92% precision and no other function scoring below 97% precision. The overall precision is given by the arithmetic mean — every function, being executed the same number of times, has the same overall weight, thus the arithmetic mean is the appropriate averaging mechanism. A figure of 98% for the overall precision is also a solid indication of the excellent performance that our classifier achieves.

5.2 Experiment 2 – Continuous Classification

For Experiment 2, we had to overcome several obstacles. For example, due to the limit of program memory of the MCU, we were unable to simultaneously include all sections of MiBench and make them execute correctly. In particular, Patricia trie insertion and Dijkstra's algorithm fail to run on the target due to insufficient resources. Excluding these two functions, we can run the experiment.

5.2.1 Description of the Experiment

To evaluate the performance of the classifier in continuous operation, we execute a long sequence of randomly chosen functions, with the only constraint being that we always call ADPCM encoding first, to then decode the data. We disregard the distinction between random and pseudo-random, and will refer to random values through the rest of the discussion. In that sense, we used the cryptographic-quality pseudo-random generator `/dev/urandom`, which is, for most practical purposes, "as close as it gets" to true random values [20]. Notice, however, that this random selection is done offline and the sequence is ultimately "hardcoded" in the source code to be compiled and run on the target. This restriction does not affect the random nature of the experiment, yet it is necessary: on-the-fly generation of random values by the target device itself between function calls would introduce artefacts that could skew or possibly even invalidate the results. In particular, the classifier identifies *contiguous* fragments of code; interleaving code to obtain random values for the selection of function and for the parameters would introduce non-negligible deviations from the sequence being tested.

We could not include arbitrarily long sequences of functions, since the entire sequence had to be hardcoded, and the target device imposes a limit on the size of the executable — we observe that the randomly generated data *for each call to a function* had to be stored in a buffer, for the same reason explained above. The longest sequence that we could fit in the target was 500 calls long (close to 100 calls per function on average). Though this number could be considered large enough to claim that the experiment is valid, we repeated the process ten times and collected statistics over a total of 5000 function calls.

The offline program that generates this random sequence of calls, as well as the random data, produces two files to be included in the program to be run on the target. One file, `buffer_sizes.h`, defines (through `#define` directives) the sizes of the buffers that contain the parameter values. Below is an example of the contents of this file:

```
#define ADPCM_COUNT 50
#define FFT_COUNT 97
#define AES_COUNT 111
#define CRC_COUNT 101
```

The other file that this offline program generates is the file containing the actual function calls. Each function call receives input data obtained from one of the elements of the buffer, and while generating this sequence, the offline program goes over each element in sequence, hardcoding the value in this output program, as shown in the fragment below, taken from one of the generated files:

```
encrypt (plaintext + 0*AESSIZE, ciphertext, &ctx);
adpcm_coder(pcmdata + 0*PCMSIZE, adpcmdata, PCMSIZE,
            &coder_1_state);
rc = crc32buf (crcdata + 0*CRCSIZE, CRCSIZE);
fft_float (FFTSIZE, 0, real_in + 0*FFTSIZE, imag_in + 0*FFTSIZE,
           real_out, imag_out);
rc = crc32buf (crcdata + 1*CRCSIZE, CRCSIZE);
adpcm_decoder(adpcmdata, pcmdata_2, PCMSIZE, &decoder_state);
nothing = (random() ^ random()) & 0xFFFF;
adpcm_coder(pcmdata + 1*PCMSIZE, adpcmdata, PCMSIZE,
            &coder_1_state);
fft_float (FFTSIZE, 0, real_in + 1*FFTSIZE, imag_in + 1*FFTSIZE,
           real_out, imag_out);
nothing = (random() ^ random()) & 0xFFFF;
```

We can see, for example, for the first two calls to `crc32buf`, the input data coming from the first and second elements of buffer `crcdata` (offsets 0 and 1 hardcoded in the call). Same for `adpcm_coder` and `fft_float`. These hardcoded offsets continue to increase with each subsequent call to each function, until the `XXX_COUNT` value. Since we only show a short fragment of one of the files, the only offsets that we see are 0 and 1.

The declarations for these buffers (in the program that runs in the target device) are shown below:

```
#include "buffer_sizes.h"

short volatile pcmdata[ADPCM_COUNT*PCMSIZE];
char volatile  adpcmdata[PCMSIZE/2];  // Encoder output
short volatile pcmdata_2[PCMSIZE];    // Decoder output

volatile char plaintext[AES_COUNT*AESSIZE];
volatile char ciphertext[AESSIZE];

volatile float real_in[FFT_COUNT*FFTSIZE];
volatile float real_out[FFTSIZE];
volatile float imag_in[FFT_COUNT*FFTSIZE];
volatile float imag_out[FFTSIZE];

char crcdata[CRC_COUNT*CRCSIZE];
```

5.2.2 Performance Evaluation

We used a similar metric to that used for Experiment 1. Since we still have a classifier that always outputs one of the possible candidates, we only have true positives or false positives, which means that the precision still provides a complete picture of the classifier's performance.

However, an important distinction arises from the fact that in continuous classification, the sizes of the traces need to be determined and affect the performance of the process, as they affect the

necessary resynchronization process in the cases of misclassifications. In that sense, a more sensible formula for the precision of the classifier in continuous mode, P_c, is given by the fraction of the time during which the output of the classifier corresponds to a true positive:

$$P_c \triangleq \frac{\sum |I_{T_P}|}{\sum |I_{T_P}| + \sum |I_{F_P}|}$$

where I_{T_P} denotes intervals during which the output of the classifier is a true positive, I_{F_P} denotes intervals during which the output is a false positive or a misclassification, and $|\cdot|$ denotes the length of its argument (the length of the interval). Furthermore, when obtaining the output from the classifier, we do not count on precise a priori knowledge of the lengths of the functions, since the values of the arguments may cause variation in the execution time. This means that we need to estimate these from the output. Section 5.2.4 discusses this aspect.

As an example, consider the scenario with three candidate functions, F_1, F_2, F_3, which take 10 ms, 20 ms, and 30 ms to execute, respectively. If the sequence $F_3 - F_1 - F_2$ is executed and the classifier outputs F_3 at time 0 ms, F_2 at time 20 ms, and F_2 at time 20 ms then, with all units implicitly ms, the intervals with true positive are $I_{T_1} = (0, 20)$ and $I_{T_2} = (40, 60)$, and the interval $I_F = (20, 40)$ is a false positive — the output is F_2, and during the sub-interval $(20, 30)$ the correct class is F_3 and during the sub-interval $(30, 40)$ the correct class is F_1. Thus, in this example, the precision P_{ex_2} would be approximately 67%:

$$P_{ex_2} = \frac{|I_{T_1}| + |I_{T_2}|}{|I_{T_1}| + |I_{T_2}| + |I_F|} = \frac{2}{3}$$

5.2.3 Results

As described in Section 5.2.1, ten sequences of 500 function calls each were executed, and traces were captured for each of them. Figure 7 shows a screenshot of the audio editor displaying the first few milliseconds of one of the traces; in particular, this trace fragment corresponds to the sequence of ten function calls shown in Section 5.2.1. The markers were manually added to the image for illustration purposes, indicating the boundaries between functions.

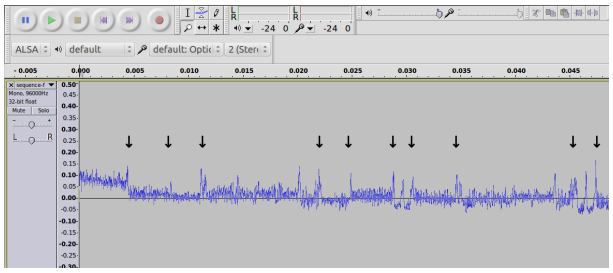

Figure 7. Power Trace for Sequence of Function Calls.

The complete traces were fed to the processing program implementing the classifier as described in the previous sections. An example of the output of this program is shown below (only a fragment, since each trace contains 500 function calls). The reported time uses the unit of audio samples, which is about 10.4 μs:

```
Executed aes at time 18
Executed adpcm-encode at time 428
Executed crc32 at time 810
Executed fft at time 1077
Executed crc32 at time 2127
Executed adpcm-encode at time 2394
Executed random at time 2786
Executed adpcm-encode at time 2942
Executed fft at time 3328
Executed random at time 4407
```

This example again corresponds to the same sequence of ten function calls shown in Section 5.2.1 and corresponding to the trace fragment shown in Figure 7. We observe that in this particular example, there was one misclassification (the sixth function call is reported as `adpcm_encode` when the actual function being executed is `adpcm_decode`).

To measure the performance (the precision) of the classifier in Experiment 2, the output for each of the ten traces was fed, along with the C source code corresponding to each of the ten sequences, to a custom-made program that compares the output and reports statistics allowing us to determine the precision and provide some additional potentially insightful information.

An important reason to use a custom-made program for the analysis of the experimental results was to allow for user intervention in the process of matching sequences that might be hard to properly identify and match algorithmically. Even more importantly, the matching includes timing that may need to be verified against the traces, for the less obvious cases (though only on two occasions we needed to resort to the traces in the audio editor to resolve a mismatch). The program does as much as possible in an automated way to minimize user intervention, and of course does as much validation as possible for the user input, to minimize the effect of human errors and oversights. We omit any additional details or screenshots; however, we do emphasize the aspect that this custom-made program requiring user assistance is necessary only for the purpose of analyzing the experimental results in the context of this study; an actual practical implementation of our system would not require this step.

5.2.4 Estimating the Precision

Measuring the exact value of the precision for the continuous classifier requires us to obtain more information than reasonably feasible given our setup. Computing the exact value of the precision requires that we determine the intervals where the output is correct, and this would require us to have exact timing information for the sequence being executed. However, the actual trace contains calls with random parameter values, so the duration is unknown, and instrumenting the program for the purpose of obtaining that information would affect the measurements, at least for the family of devices that we targeted in our work.

We can, however, obtain a good approximation of the precision if we use the timing that the classifier outputs. Specifically, the skipped elements from the classifier output (the right column in the above example of the processing software) are considered to be false positives and the rest is considered a true positive. As long as the sequences match, we assume that the timing for the matching items is correct and disregard any inaccuracies in the exact positions of the boundaries between functions.

Thus, the difference in the time indexes for the items that have to be skipped to reestablish synchronization reveal the length of the false positive intervals. The estimate is accurate provided that the misclassifications occur with a deviation that is balanced; that is, provided that some errors confuse a function with a longer (in duration) function and some confuse a function with a shorter function, without any imbalance on average. This is a reasonable assumption, and we did not observe any evidence suggesting that there would be any imbalance in the errors for the traces that we processed.

Table 2 summarizes the important parameters describing the performance of the classifier when operating in continuous mode. In addition to the precision, we also determined the average number of items that it took the system to recover from a misclassification. This was done based on the output of the custom-made analysis program mentioned in the previous section, which allows us to determine when the system is back in sync. For more details, the

reader can consult [17]. This metric provides information about the robustness of the system in terms of ability to recover from a misclassification and reestablish synchronization with the trace. It also provides evidence to the quality of the classifier in general, in that short sequences of missed items are certainly easier to compensate for through auxiliary methods, such as doing additional validation of timings, validating feasible program sequences through static analysis, etc. Though we did not use any of these techniques in this initial phase of the project, it still makes sense to claim that smaller values for this parameter correspond to higher quality for the continuous classifier. Since all traces included the same number of fragments, we used arithmetic mean for the average values.

Sequence	Precision	Avg. Recovery
1	88.75%	1.29
2	89.78%	1.30
3	87.65%	1.27
4	88.63%	1.38
5	87.07%	1.34
6	89.03%	1.29
7	89.03%	1.32
8	86.97%	1.48
9	89.59%	1.19
10	87.84%	1.30
Overall (avg.)	**88.74%**	**1.32**

Table 2. Continuous Classifier Performance

For this second experiment we also obtained results indicating a good performance. This is encouraging, since this is clearly the more important of the two experiments, since it models the way an actual practical system would operate. The precision is not as good as that obtained for Experiment 1; this is expected, as this operating mode involves more parameters, more ambiguities and degrees of freedom, and the additional functionality of maintaining synchronization — with or without misclassifications. Also, the fact that traces are now in sequence one after another introduces the possibility that sub-fragments of a trace combined with sub-fragments of another trace could be a good match for some incorrect function. Experiment 1 does not face any of these difficulties. Thus, we believe that a figure of close to 90% precision for continuous classification is a very good result for this initial phase of the project, where the goal is to study the feasibility of our proposed approach.

Another important aspect revealed by these results is that of the robustness of the continuous classifier, in that the system never faced a situation where an error threw it irreversibly out of sync. In all cases, the system reestablished synchronization after a misclassification, and in most cases the misclassification involved just one function replaced with another, and then resynchronization immediately after, as suggested by an average of 1.32 functions skipped before resynchronization.

As suggested before, we can reasonably claim that individual incorrect classifications are essentially irrelevant, as the tool could be extended to make use of the CFG and consider possible execution sequences. Given this information, a match to an incorrect function may have been rejected with high probability, because the CFG would have given indication that it is impossible to reach that particular function within a short period of time after the preceding sequence.

5.2.5 Additional Insights

As additional observations and insights that we gained from the experimental results in this initial phase of the project, we could mention the following:

For some of the functions, long sequences of consecutive calls to the same function showed a higher likelihood of misclassification. For example, sequences of three consecutive calls to CRC32 showed up very often as a prompt for user assistance in the processing software, and in some cases required three or four skipped items from the trace to reestablish synchronization.

Also interestingly, we observed several instances where a sequence of three or more consecutive calls to random() either caused the function immediately following that sequence to be misclassified, or was misclassified as a smaller number of consecutive calls.

These two aspects suggest that it may be a good idea to restrict the fragments of code to sections of the source code with no control structures (conditionals and loops) such that every instance of a given fragment of code exhibits the same execution time. This not only has the potential to increase the precision, but also could play a role in dramatically increasing the computational performance of the system, in that a smaller training database could work well, and possibly the more efficient neareast centroid technique (using LDFs) could be applicable, since it was precisely this aspect of variable execution time within classes what put that approach at a disadvantage with respect to the k-NN technique.

5.3 Interrupts and Interrupt Service Routines

The current experiments were oblivious to interrupts and interrupt service routines (ISR) for two main reasons: (1) the approach to detect them differs due to their asynchronous and usually short-lived nature; and (2) from the evidence we collected and observed, we claim that their detection should be really easy.

We should expect an interrupt request (IRQ) to cause a "power-heavy" reaction by the processor, in that a lot of hardware components need to react and work on processing the IRQ correctly [7]. Consequently, we should see a prominent component in the power trace that would identify the exact moment at which the processor responds to an IRQ.

We collected some experimental evidence supporting this claim; Figure 8 shows a trace of a simple "do nothing" program that runs pointless tasks in the background and uses timer interrupts, with the IRQ firing approximately every 6.5 ms. We observe the prominent peaks that IRQs produce in the trace.

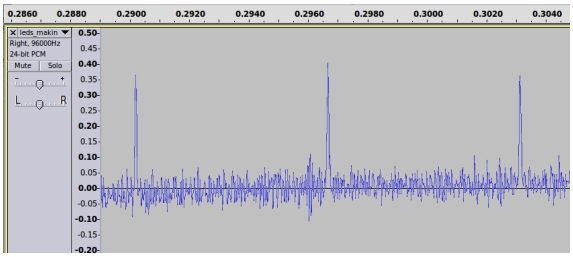

Figure 8. Power Trace Showing the Effect of IRQs.

For an actual practical implementation, however, we must consider interrupts for at least the following two reasons: (1) any fragment of code that the system is attempting to identify can be subject to another (small) fragment corresponding to the interrupt processing and the ISR to be inserted at any arbitrary and unpredictable position of the fragment to identify; and (2) many bugs arise from improper interactions between an ISR and the main/background processing, thus triggering the faulty behavior around the time that an interrupt occurs.

6. Discussion and Future Work

This initial phase of our project presents encouraging results; the effectiveness of the technique was confirmed by the experimental phase, at least in a preliminary way.

Further research is needed in several areas of the project; we focused on one target device, the Atmega2560, running at 1MHz. In general this is not an important limitation, in that we could extrapolate from the field of side-channel attacks, where SPA has been successfully applied on a wide variety of architectures and target devices. The important aspect for us to consider in this respect is the approach's ability to work with simple hardware, in particular with an off-the-shelf inexpensive sound card. With our Atmega2560 setup, we verified that a standard PC sound card sufficed for the system to run at the granularity level of function calls, producing traces in the order of 200 to 300 samples in length. However, to address either finer granularity, or higher clock speeds for the target device, we may need to investigate the relationship between these: for a given granularity, what is the maximum ratio between the device's clock speed and the sampling rate at which we capture the power trace? Equivalently, for a given clock speed and a given sampling rate, what is the finest granularity at which we can detect fragments of code?

Ideally, we would like our system to detect every possible code segment with fixed execution time (i.e., contiguous blocks of code without conditionals or loops). For example, instead of detecting execution of a loop (as a whole, with variable execution time), we would prefer to individually detect the evaluation of the condition and the body (assuming no nested loops or conditionals inside the body of the loop). This would be beneficial for at least the following three aspects:

- There is a potential increase of precision, since loops executing variable number of passes or other conditional executions can introduce wide variations within classes, potentially limiting the precision.
- There is additionally a potential increase in processing speed; the aspect mentioned in the item above also means that the database of training samples has to be larger to compensate for these variations (that is, to make sure that the training database is representative of the spread PDF when allowing variable length). Additionally, if we have fixed size segments, then the more efficient LDF technique could be suitable.
- The higher level of details in the output of our system will represent a significant improvement in terms of the system's ability to help developers go from program tracing to finding a bug.

We would like to emphasize the fact that these limitations in no way negate or compromise the validity or the value of the reported results. A large class of embedded systems run at low clock frequencies, and for those, the presented approach will be perfectly fine and valuable when assisting in the debugging task during advanced phases of the development cycle. Incidentally, this low-frequency aspect may be correlated with low transistor count MCUs, presumably with simple architectures that may lack any sophisticated debugging tools embedded in the hardware, making our technique particularly valid for this class of target device.

A positive aspect of the results derives from the fact that all of our tests and functions are CPU-bound. Practical systems typically use I/O, which makes a more prominent mark on the power trace and thus helps the classification process. The results of our experiments show a good level of performance even with this disadvantage.

The following are some of the important aspects that we intend to tackle through future research:

- Introduce the notion of *conditional* classification, possibly manually in an initial phase, but with the goal of using a CFG tool when it comes to a practical implementation. The idea is that by looking at the source code, we gain information about the possible fragments of code that could be executed at a particular time, given the previous fragment executed, or even better, the sequence of past fragments. Thus, the classifier can count on additional information, and thus its efficacy should improve.

  In this sense, the fact that our Experiment 1 used only nine fragments should not be seen at all as a number too low to produce valid results — in a practical setup that makes use of the CFG, for most classifications the system may need to consider no more than two or three possible candidate fragments following the most recent classified fragments.
- Reliable detection of a crash condition where the processor ends up executing random code. Detecting such condition, as well as the precise time at which it started, is clearly a valuable piece of information when assisting the developers in the debugging task. This may be related to the option of *reject* in the classifier [21], and would allow us to eliminate the assumption that the execution is restricted to a set of possible fragments of code — an assumption that is reasonable in the sense that the developers always can count on the source code, but less reasonable from the point of view of considering cases such as stack corruption, invalid pointer operations or other situations leading to "random" execution.
- Develop and test strategies to automate the training phase. Having to capture training samples of each fragment of code is perhaps the most severe limitation of our approach. We do believe, however, that this issue can be mitigated and possibly eliminated. One alternative could be instrumenting the program (during the development phase) to interact with an automated system to capture training samples. Another option could be using models of the target devices that would allow an external system to determine exact timings of the execution and thus capture training samples in an automated way by manipulating the data and signals connected to the device to control its execution.
- Considering different architectures; for example, processors with cache memory, deep pipeline or other forms of parallelism, etc. These in principle make our task harder, given that more information is combined together before leaking to the power trace. However, for some of these aspects, the additional complexity in the architecture may go hand-in-hand with additional information being leaked to the power trace, and those could end up making the task easier.

7. Conclusions

In this work, we have proposed a novel approach for non-intrusive debugging of embedded systems, especially useful for debugging faulty behavior observed at advanced phases of the development cycle, such as during production or even after deployment. The idea is based on exploiting the relationship between what a processor is executing and its power consumption to determine the sequence of code executed from observations of power consumption as a function of time (power traces). At the present stage, our approach is applicable to background/foreground programming (superloop structure), multitasking with run-to-completion semantics, and possibly also to co-operative multitasking, depending on whether we can easily identify the `yield` calls.

Our approach and our implementation feature the interesting highlight that the system runs on a standard PC, and the power traces are captured through the recording input of the sound card.

Techniques where power traces are required, such as Power Analysis cryptographic attacks, usually rely on digital oscilloscopes or other expensive or bulky pieces of equipment. Also worth noting, since our experiments produced good results even when using an inexpensive off-the-shelf sound card, we conclude that this technology is perfectly suitable for hobbyists as well as professional developers.

Experimental results confirmed the validity of our approach, showing very good performance when using part of the code base from the MiBench test suite. Several improvements and opportunities for future work were discussed, which we believe will lead to substantial improvements in the performance and the range of target devices for which our technique is suitable.

Acknowledgments

The authors would like to acknowledge the contribution of Summit Sehgal, who offered assistance with the setup and lab equipment for the preliminary tests and experimental phases of the project. The first author would like to thank Dr. Thomas Reidemeister as well, for his valuable assistance and discussions.

This work was supported in part through a grant from the Natural Sciences and Engineering Research Council of Canada, awarded to Dr. Hasan.

This research was supported in part by CFI 20314 and CMC, and the industrial partners associated with these projects.

References

[1] Atmel Corporation. AVR 8- and 32-bit Microcontrollers, 2012. URL `http://www.atmel.com/products/microcontrollers/avr/`.

[2] S. Chari, J. R. Rao, and P. Rohatgi. Template Attacks. *Cryptographic Hardware and Embedded Systems – CHES 2002*, pages 13–28, 2003.

[3] J. Cooling. *Software Engineering for Real-Time Systems*. Addison-Wesley, 2003.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Third edition, 2009.

[5] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14. IEEE Computer Society, 2001.

[7] C. Hamacher, Z. Vranesic, and S. Zaky. *Computer Organization*. McGraw-Hill, Fifth edition, 2002.

[8] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Fourth edition, 2007.

[10] HT Omega. Claro Plus – Online specifications. URL `http://www.htomega.com/claroplus.html`.

[11] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. *Advances in Cryptology*, 1996.

[12] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. *Advances in Cryptology – CRYPTO' 99*, pages 388–397, 1999.

[13] R. Langner. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security & Privacy*, 9(3):49–51, May-June 2011.

[14] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[15] D. Mazzoni. Audacity: Free Audio Editor and Recorder. URL `http://audacity.sourceforge.net`.

[16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. URL `http://www.cacr.math.uwaterloo.ca/hac/`.

[17] C. Moreno. Side-Channel Analysis: Countermeasures and Application to Embedded Systems Debugging, 2013. PhD Thesis (Final version to be submitted May 2013).

[18] C. Moreno and M. A. Hasan. SPA-Resistant Binary Exponentiation with Optimal Execution Time. *Journal of Cryptographic Engineering*, pages 1–13, 2011.

[19] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Prentice Hall, Fourth edition, 2006.

[20] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[21] A. R. Webb and K. D. Copsey. *Statistical Pattern Recognition*. Wiley, third edition, 2011.