Non-Intrusive Runtime Monitoring Through Power Consumption to Enforce Safety and Security Properties in Embedded Systems

Carlos Moreno and Sebastian Fischmeister

Electrical and Computer Engineering, University of Waterloo
{cmoreno,sfischme}@uwaterloo.ca

Abstract. The increasing complexity and connectivity of modern embedded systems highlight the importance of runtime monitoring to ensure correctness and security. This poses a significant challenge, since monitoring tools can break extra-functional requirements such as timing constraints. Non-intrusive program tracing through side-channel analysis techniques have recently appeared in the literature and constitute a promising approach. Existing techniques, however, exhibit important limitations.

In this paper, we present a novel technique for non-intrusive program tracing from power consumption, based on a signals and system analysis approach: we view the power consumption signal as the output of a system with the power consumption of training samples as input. Using spectral analysis, we compute the impulse response to identify the system; the intuition is that for the correct training sample, the system will appear close to a system that outputs a shifted copy of the input signal, for which the impulse response is an impulse at the position corresponding to the shift. We also use the Control Flow Graph (CFG) from the source code to constrain the classifier to valid sequences only, leading to substantial performance improvements over previous works.

Experimental results confirm the effectiveness of our technique and show its applicability to runtime monitoring. The experiments include tracing programs that execute randomly generated sequences of functions as well as tracing a real application developed with SCADE. The experimental evaluation also includes a case-study as evidence of the usability of our technique to detect anomalous execution through runtime monitoring.

Keywords: Program tracing, runtime monitoring, embedded software security, side-channel analysis, power-based program tracing, signal processing, signals and systems analysis.

Manuscript submitted March 2017, revised version submitted July 2017, accepted for publication August 2017.

1 Introduction

Modern embedded devices are rapidly increasing in complexity and connectivity, making it ever more important to incorporate runtime monitoring systems for the purpose of ensuring correctness and security. This introduces an important challenge, as instrumentation added to the system can break extra-functional requirements such as real-time constraints in the operation. Non-intrusive program tracing through side-channel analysis techniques have recently appeared in the literature and constitute a promising approach, as we argue in [20]. These techniques use an external device to measure power consumption and reconstruct the program trace. From the perspective of runtime monitoring, there are several benefits: (i) we obtain the program trace without any instrumentation that could affect the device's functionality; (ii) once the program trace is obtained, additional monitoring (processing/analysis) tools can be introduced without the risk of interfering with the device's functionality or breaking any extra-functional requirements; and (iii) the runtime monitor is *tamper-proof* in the sense that it is not affected by system "crashes" or even deliberate cyber-attacks.

Moreno et al. presented a novel technique for non-intrusive program tracing and debugging through side-channel analysis [21]. In that work, they used power consumption measurements — *power traces* — to determine blocks of source code being executed. An additional novel aspect of that work was the use of a commercial off-the-shelf (COTS) sound card on a PC to capture the power traces. That work was an important step in showing the technical feasibility of these program tracing techniques. However, it exhibits important limitations with respect to both methodology and performance. In particular, it requires a user-assisted training phase where fragments of source code have to be isolated and individually executed. Moreover, the technique in [21] operated at the granularity level of whole functions, which may be too coarse to be practical. Indeed, [21] does not present any case-studies to support the idea of this non-intrusive tracing technique being useful in practice. The work in [22] proposes a technique that can be combined with the approach in [21], and indeed can be combined with our proposed technique, potentially increasing its performance through a compiler-assisted transformation of the generated binary code.

Eisenbarth et al. [9] presented a different approach, introducing the idea of a side-channel disassembler. Without using information about source code, they attempted to obtain the sequence of CPU instructions from power consumption. They obtained statistical models for the power consumption of individual CPU instructions and used that information to match the measurements during execution. However, their results showed a performance far too low to be applicable in practice. Msgna et al. [23] presented a follow-up to Eisenbarth's work, as they use a similar technique. However, their technique is also impractical, as they use a high-end oscilloscope as capture device, sampling the power signal at rates of 5 GB/s. Furthermore, [23] averages multiple traces to reduce the effect of the noise; both aspects make their proposed technique unsuitable for a practical application in runtime monitoring. Liu et al. [16] presented another follow-up to Eisenbarth's work, also impractical for runtime monitoring, as they still use an

oscilloscope with a high sampling rate of 1.25 GB/s. Clark et al. [6] used side-channel analysis to identify execution traces in medical devices for the purpose of tamper-detection. That work is limited in the sense that it only works at the granularity level of the entire execution trace, and relies on the assumption that the device's task is simple and highly repetitive.

Using online trace information, our approach can work within the conceptual scheme of traditional runtime monitoring and verification systems [25], but it exhibits important advantages with respect to their implementation. The main benefits derive from the fact that in our system, the external monitor is a physically isolated subsystem, yet suitable for low-cost microcontrollers that have little or no hardware support for debugging, tracing, or in general runtime monitoring. Both event-triggered [4, 12–14, 28] and time-triggered frameworks [24] typically rely on components or instrumentation that run together with the monitored system, making them vulnerable to security threats and failures involving memory corruption ("system crashes").

1.1 Our Contributions

In this work, we propose and implement a novel technique for non-intrusive program tracing through side-channel analysis, and show its application to online runtime monitoring through anomaly detection. We introduce conceptual changes that improve the effectiveness and efficiency of power-based program tracing, thus addressing most of the limitations in [21], [6], and [9]. Our proposed technique has several aspects that account for these improvements over previous work:

- **Novel use of signal processing for classification in power-based program tracing**. Instead of standard statistical pattern recognition techniques, we propose a novel approach based on signal processing; specifically, a form of system identification. We use a computationally efficient procedure that determines the best match for a trace segment and also the position of the match (without requiring any extra, separate computation). This addresses one of the important limitations in [21]: the system is given a single power trace and has to split it into segments to be classified, maintaining alignment with the correct segments boundaries (of which the system is given no information as input). Our signals and system analysis approach proved to not only work well in terms of the performance of the system, but also contributed to a substantial improvement in processing speed, with a measured speedup of more than 4× attributable to this aspect.
- **Use of code analysis to improve performance**. Using the Control Flow Graph (CFG) obtained from the source code, we assist the classification system by constraining the blocks to those that are part of valid sequences. The intuition is that the probability of misclassification is lower if the classifier counts on additional information that reduces the set of candidates. This is illustrated by Figure 1, where sub-figure (a) represents classification when considering all possible blocks, and sub-figure (b) represents classification where a reduced set of candidates is considered. Our technique builds

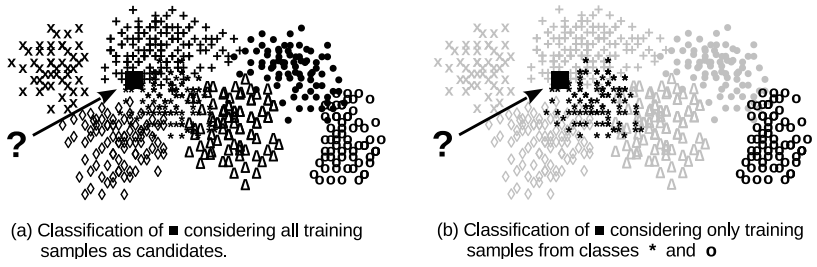


Fig. 1. Reducing the set of candidates for classification

upon this intuition: by expanding the CFG using a dynamic programming approach, we validate sequences of blocks; this can be seen as a mechanism where we obtain fine granularity, but with the equivalent of the classifier working at a coarser granularity so that it reduces the probability of misclassification by working with larger segments.

- **Improved methodology and nearly fully automated work flow**. We instrumented the source code using the CFG, allowing us to achieve nearly full automation of both the training phase and the performance evaluation phases of the system.

In addition to the experimental evaluation where we measure the performance of our system, we include two case-studies presented as evidence of the usability of this technique. The case-studies apply in the context of runtime monitoring as well as in the context of computer security, where our technique may be used as an Intrusion Detection System (IDS) [17] for embedded devices.

1.2 Extension/Revisions With Respect to Our Previous Work

This paper is an extended version of our work presented at the 2016 Runtime Verification conference [19]. Below are the new contributions that extend the scope of that work:

- **Added higher-speed capture device**. A custom-made interface and capture device was introduced to the setup. The purpose is twofold: (1) we wanted to address two potential limitations with the sound card capture; and (2) though the use of a sound card demonstrates the possibility of implementing this technique at low-cost, a custom compact capture device is still necessary for an actual practical implementation of our technique.
- **Additional case-study**. A second case-study, an IDS for embedded devices, demonstrates the possibility of our technique operating in real-time (i.e., processing the power trace data at the rate they are produced and detecting anomalous conditions on-the-fly).
- **Evaluated a resynchronization feature**. We propose a mechanism for the system to resynchronize with the power trace and the CFG for the cases where the classifier goes out of sync. The mechanism is also useful for the

initial synchronization when the monitoring system starts operation (at an unknown point of the power trace and unknown point in the execution).

- **Expanded description of the system, analysis, and future work**. We expanded several sections in the description of our proposed system and experimental setup. This includes added textual descriptions, figures, and algorithms. Moreover, some additional aspects suggested as future work are discussed.

1.3 Organization of the Paper

The remaining of this paper proceeds as follows: Section 2 presents a brief review of signals and system analysis tools. Section 3 describes our proposed approach. Our experimental setup is described in Section 4, followed by the results in Section 5, including the case-study. Finally, a discussion and concluding remarks are presented (sections 6 and7).

2 Background – Frequency Domain Analysis of Signals and Systems

A discrete-time linear time-invariant (LTI) system can be fully described by its impulse response, $h(n)$. This impulse response is the output of the system when the input is the impulse signal $\delta(n)$, where $\delta(0) \triangleq 1$ and $\delta(k) \triangleq 0 \; \forall \; k \neq 0$. For an arbitrary input signal $x(n)$, the system's output $y(n)$ is obtained through the *convolution* relationship [27]:

$$y(n) \;=\; \sum_{k=-\infty}^{\infty} h(k)\, x(n-k) \tag{1}$$

A frequency domain representation of a discrete-time signal $x(n)$ can be obtained through the (Discrete-Time) Fourier Transform $\mathcal{F}$, defined as [27]:

$$\mathcal{F}\{x\} \;=\; \mathcal{X}(\omega) \;=\; \sum_{k=-\infty}^{\infty} x(k)\, \mathrm{e}^{-\mathrm{j}\omega k} \tag{2}$$

where ω is the *angular frequency* ($-\pi < \omega < \pi$), and j denotes the imaginary unit (i.e., $\mathrm{j}^2 = -1$).[1]

Given the Fourier Transform $\mathcal{X}(\omega)$, the signal $x(n)$ can be obtained through the inverse Fourier Transform $\mathcal{F}^{-1}$, defined as [27]:

$$\mathcal{F}^{-1}\{\mathcal{X}\} \;=\; x(n) \;=\; \int_{-\pi}^{\pi} \mathcal{X}(\omega) \mathrm{e}^{\mathrm{j}\omega n} d\omega \tag{3}$$

The properties of the Fourier Transform for discrete-time signals regarding convolution in the time domain are the same as those of the Fourier Transform

[1] We adopt the electrical engineering convention of using j to denote the imaginary unit, to avoid ambiguity with the symbol for electrical current or intensity, i.

for continuous-time signals. In particular, if $x(n)$, $y(n)$, and $h(n)$ follow the relationship described in Equation (1), then it holds that:

$$\mathcal{Y}(\omega) \;=\; \mathcal{X}(\omega)\,\mathcal{H}(\omega) \tag{4}$$

where $\mathcal{X}(\omega)$, $\mathcal{Y}(\omega)$, $\mathcal{H}(\omega)$ are the Fourier Transforms of $x(n)$, $y(n)$, $h(n)$, respectively. Thus, given an input signal $x(n)$ and its corresponding output signal $y(n)$, the impulse response $h(n)$ of the system can be obtained as:

$$h(n) \;=\; \mathcal{F}^{-1}\left\{\frac{\mathcal{Y}(\omega)}{\mathcal{X}(\omega)}\right\} \;=\; \mathcal{F}^{-1}\left\{\frac{\mathcal{F}\{y\}}{\mathcal{F}\{x\}}\right\} \tag{5}$$

To apply frequency domain analysis to a segment or a window of a signal of length N (viewed as a signal $x(n)$ with $0 \leqslant n < N$), we use the discrete Fourier Transform (DFT), defined as [27]:

$$\mathcal{DFT}(x) \;=\; \mathcal{X}(k) = \sum_{n=0}^{N-1} x(n)\mathrm{e}^{-\mathrm{j}\frac{2\pi kn}{N}} \tag{6}$$

with $0 \leqslant k < N$. Its inverse operation is given by:

$$\mathcal{DFT}^{-1}(X) \;=\; x(n) \;=\; \frac{1}{N}\sum_{k=0}^{N-1} \mathcal{X}(k)\mathrm{e}^{\mathrm{j}\frac{2\pi kn}{N}} \tag{7}$$

The DFT can be efficiently computed through the Fast Fourier Transform (FFT) algorithm [27]. In our case, we used the FFTW library [10], which efficiently computes both FFT and inverse FFT. The DFT represents the Fourier Transform of a periodic signal with period N where $x(n)$ comprises one period of the signal. The properties shown above hold, with the system's output being given by the *circular convolution* of the input signal and the impulse response — convolution computed with time indexes treated in a modulo N fashion. This allows us to obtain the impulse response of a system when looking at N-samples windows of the related signals:

$$h(n) = \mathcal{DFT}^{-1}\left\{\mathcal{H} = \frac{\mathcal{Y}}{\mathcal{X}}\right\} \tag{8}$$

where the quotient $\mathcal{H}$ is computed through sample-wise division. That is, for each $k \in [0, N)$, $\mathcal{H}(k) = \frac{\mathcal{Y}(k)}{\mathcal{X}(k)}$.

3 Proposed Technique

This section describes the main aspects and novelty of our proposed technique.

3.1 Frequency Analysis: Classifying and Determining the Shift in the Power Trace Segments

The main idea and novel aspect behind our proposed approach for classification is to view the power trace segments as the output of a system whose input is

the power trace of the training samples. For each of the training samples (corresponding to fragments of code) we perform a system identification; in particular, we obtain the impulse response as described in Section 2. The intuition is that for the correct fragment, the input and output are similar, with the possibility of a shift from the position where we evaluate to the position where the signals match. Thus, for the correct fragment, the identified system will appear close to a system that outputs a copy of the input signal shifted by a certain amount of samples. For this time-shift system, we know that the impulse response is a single pulse at the position corresponding to the shift [27].

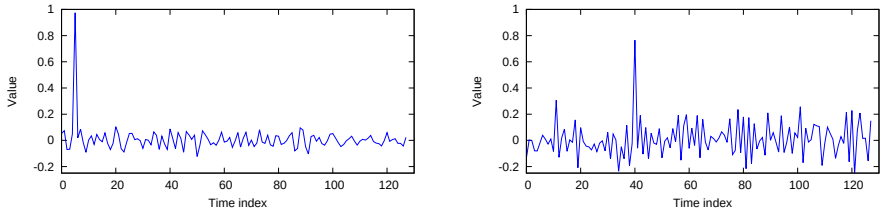
A key detail is that as the system advances through the trace, the exact positions where the trace segments begin (i.e., the position at which the corresponding fragment of code started execution) are not given. One advantage of this system identification approach is that once we determine the best match among the training samples, the shift in the impulse response reveals the position where the match occurs. In terms of execution speed, this represents an important advantage with respect to the technique in [21], where the system needs to attempt classification over a somewhat large range of possible starting positions around the nominal starting point given by the outcome of the previous classification (see [18] for details).

We have to be careful, however, with the "circular" nature of the DFT: consider a system that shifts the signal by n_0 samples. If we look at an N-samples window of a periodic signal with period N, the shift occurs circularly within the window — samples being shifted and disappearing into one edge of the window will be identical to those appearing from the other edge, due to the signal's periodicity. However, for the case of a non-periodic signal (as it is our case), shifting the signal and comparing input and output in the same N-samples window corresponds to truncating the signal on one end and introducing an alien fragment on the other end. Thus, the impulse response obtained through DFT analysis within an N-samples window will not be a single pulse.

The key observation is that for small values of n_0 compared to N, the impulse response will be close to a single pulse, since the output corresponds to the linear superposition of a large fraction of the signal shifted and two signals that are nonzero only in a small fraction of the interval. Figure 2 illustrates this intuition, with sub-figure (a) showing the computed impulse response for a shift by a small amount (5 positions in a 128 samples window) and sub-figure (b) showing the response for a larger shift (40 positions). The impulse response for the small shift shows a very prominent pulse at index 5, whereas the response for the larger shift exhibits a higher "noise level" outside the main pulse near index 40, thus making the pulse less prominent. It should be obvious that the response for two unrelated signals should not have any prominent pulses, so we omit any examples.

3.2 Statistical Pattern Recognition

Though the use of pattern recognition as the main classification technique was largely replaced by the signal processing approach, some elements from this field



(a) Shift = 5 (b) Shift = 40

Fig. 2. Examples of impulse responses

are present. In particular, we use a distance metric to quantify how close the impulse response is from a single pulse, and this distance is evaluated for the elements of a database of training samples; we determine the k closest matches from the database and evaluate the average distance — a logic similar to that behind the k nearest neighbors (k-NN) rule [30].

For the distance metric, we used the following heuristics: we quantify how close a given impulse response is from a single pulse based on the following parameters (computed in the same order as listed):

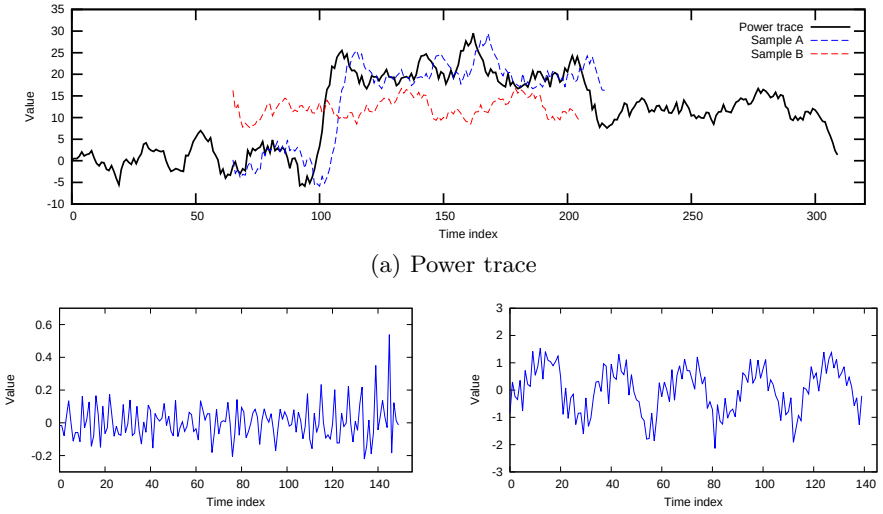
- Highest value of the signal (the "height" of the main pulse; denoted H_p) and position where it occurs (denoted n_0).
- Median of the absolute values of the signal; denoted $\tilde{h}$.
- Width of the main pulse (obtained from the interval around n_0 for which the absolute value of the signal is above $\tilde{h}$; denoted W_p).
- Highest absolute value of the signal outside the interval corresponding to the main pulse (the "noise" level; denoted L_n).

With these parameters, the distance, d (a metric corresponding to the natural notion that the smaller the distance, the closer the match), is given by:

$$d \;=\; W_p \;\times\; \frac{L_n}{H_p} \tag{9}$$

The first term accounts for the effect that the narrower the main pulse, the closer it is to a single pulse. The second term accounts for the effect that the smaller the values outside the main pulse (relative to the height of the main pulse), the closer it is to being a single pulse.

Figure 3 shows an example to illustrate the technique, where training database samples A and B are considered for classification at position $t = 65$. Sample A matches the trace at position $t = 60$; we see that the resulting impulse response has the largest peak at position 146. Sample A is 151 samples long, hence position 146 is equivalent to position -5, which is where the sample matches, relative to the position where we apply the classification procedure. Sample B does not match; thus, the impulse response is clearly dissimilar to a single impulse.



(a) Power trace

(b) Impulse response for sample A

(c) Impulse response for sample B

Fig. 3. Example of classification at position $t = 65$

3.3 Static Analysis: Using the Control Flow Graph

The second important aspect introduced in this work is the addition of static analysis tools to assist the classifier by restricting the classification choices to blocks that constitute allowed sequences. In particular, use of the CFG allows us to constrain the choice of best match to those that are part of valid sequences. To this end, we used a dynamic programming approach [7]: at each point in the classification, we expand the CFG to determine the set of possible paths up to a given depth (given as a configuration parameter). For each of the nodes in this expanded/unrolled CFG, we evaluate the distance (as described in Section 3.2). We choose the path $\mathcal{P}$ with lowest sum of distances, and the classifier's decision corresponds to the first node in $\mathcal{P}$.

This can be seen as a mechanism where we obtain fine granularity in the execution trace, but with the equivalent of using a coarse granularity for the classification, reducing the probability of misclassification by working with longer traces. The dynamic programming implementation improves computational efficiency: we advance through the tree, discarding the subtrees of the sibling nodes to the selected one, but keeping the subtree of the selected node so that we avoid redundant calculations when expanding the CFG at the new node. Algorithm 1 shows the details of this procedure. In the algorithm, the expression $\mathrm{Suc}(\cdot)$ denotes the set of successors of the argument $\cdot$, and G_n denotes the CFG G with a state indicating that it is currently at node n.

Notice that this "recursion forward" is possible because we have the complete trace for analysis; in an actual implementation where the system has to operate online (i.e., classify traces on-the-fly), this simply means that we have to allow for a small delay in the classification process, so that at block n of the trace, the

Algorithm 1: Classification Procedure.

Input: G (CFG), P_T (Power Trace), D (Depth)
Output: T (Program Trace) Expressed as sequence of blocks

```
begin
    R ← RootNode;
    repeat D times;
        for each leaf node n ∈ R do
            n.child_nodes ← Suc(G_n);
            Compute distance and start pos. (shift) for added nodes
    end
    while R leaf nodes not at end of P_T do
        P ← Path to leaf with lowest sum of distances;
        T ← T ‖ P(1);
        R ← Subtree with root P(1);
        for each leaf node n ∈ R do
            n.child_nodes ← Suc(G_n);
            Compute distance and shift for added nodes
        end
    end
end
```

classifier is making the decision for block $n - D$, where D is the depth of the expanded CFG.

We also highlight the aspect that this dynamic programming approach of expanding the CFG can be combined with other classification techniques, since it relies on a distance metric that quantifies how close given samples are from training samples. Though our signals and system analysis approach proved effective, other techniques may be suitable under different conditions, and could exhibit better results in terms of classifier's performance. Being able to combine any such techniques with the CFG expansion approach ensures that one can improve the classifier's performance while targeting a fine granularity regardless of the classification technique being used.

3.4 Resynchronizing

A mechanism for synchronizing (including synchronizing with the CFG) with the trace at an unknown point of execution is necessary for a practical system, since it addresses the start of operation of the tracing system, when the classifier has no information about where in the trace or in the CFG the execution is.

This mechanism also has potential use to address out-of-sync conditions that the system is unable to fix — we simply disregard recent past classifications that show evidence of difficulties on the classifier, and proceed as though we were just starting to capture the trace. Though this condition never occurred during

our experiments, it is definitely a situation that can occur in practice, even if it happens with extremely low probability.

Algorithm 2 shows a sketch of this mechanism; the idea is that classifications done without the help of the CFG are filtered based on consistency with allowed sequences as per the CFG. When detecting one block, execution could be at many places in the CFG (we recall that blocks are part of functions, and the same function can be invoked from different places in the code). As we continue to detect additional blocks in sequence, the CFG will disallow the sequence for some of those places, reducing the number of possibilities. When there is just one remaining feasible sequence, the procedure completes. If all sequences are discarded, the procedure failed, and we just start over. Clearly, when no feasible

Algorithm 2: (Re)synchronization with the Trace and the CFG.

```
Input:    CFGG  (CFG of a function)
          Input Power Trace
Comments: FP denotes the set of Feasible Paths
          Classifier does not use CFG

begin
    while FP = ∅ do
        C ← Classifier's output;
        FP ← {Nodes + CFG Pos} that contain C;
        while | FP | > 1 do
            C ← Classifier's output;
            for each path p ∈ FP do
                if Can not advance through p to node C then
                    Remove p from FP;
                end
                else Advance;
            end
        end
        if | FP | = 1 then exit (success);
    end
end
```

sequences are left, our only choice is to start over; yet, the procedure should, under reasonable assumptions, successfully synchronize with the trace and the CFG. A simple probabilities argument may be used to support this intuition: if we have a probability of correct classification $P > 0$, independent for different classifications (a reasonable approximation when not using the CFG), the probability of L consecutive classifications is P^{L}. This means that the probability that resynchronization has not succeeded after N attempts at sequences of L classifications is $\Pr\{\text{Fail}\} = (1 - P^{\mathrm{L}})^N$, which approaches 0 as N grows.

3.5 Segmentation of Traces and Fragments of Source Code

One important limitation in the approach proposed in [21] relates to the difficulty in training the system. For the training phase, fragments of code (whole functions, in that work) had to be run in isolation and surrounded by markers. In our proposed approach, during the training phase we run the fragments of code in the natural sequence as they occur in the source code. An instrumented version of the source code allows us to segment the trace into the sections that correspond to the fragments in the source code by flipping a port bit at the boundaries between fragments.

As we will discuss in Section 4.1, we use a PC sound card to capture power traces, similarly to the approach proposed in [21]. However, one important aspect that allowed us to improve the methodology with respect to [21] is the fact that we sample two signals, taking advantage of the stereo input of the sound card; we use one of the channels to capture power consumption and the other channel to capture the markers given by this port bit signal. Through visual observation, we verified that the instrumentation needed to flip the port bit causes a negligible effect on the traces (between one and two audio samples). We remark that this is possible in a more general setup that uses a two-channel digital oscilloscope or any sampling device with at least two input channels.

For the training phase, where we require a priori knowledge of the fragment of code being executed, an additional instrumented version is created with print statements at the boundaries between segments. This instrumented instance is run outside the target, in "offline" mode; both instrumented versions produce the same execution trace, since the source code is the same for both cases and the input data is the same (it is chosen at random, but once chosen it is "hard coded" into the programs — Section 4.1 includes a more detailed description). Thus, the system can automatically determine the fragment of code corresponding to each segment of the trace, as marked by the edges in the port bit signal. Appendix A shows an example of the two instrumented versions for the `adpcm_coder` function.

During the training phase, the system loops generating random sequences of code with random input data. For each instance of generated code, the system flashes the target device and captures and processes the traces without user intervention (other than getting the process started).

3.6 Instrumenting the Source Code

We used LLVM [5] to extract a CFG from the source code. However, for our setup — with an AVR Atmega2560 [2] operating at 1MHz — basic blocks produce trace segments that are too short for the classifier to operate successfully. We devised a procedure to merge CFG nodes into nodes representing larger blocks of source code, yet maintaining a valid CFG structure [2] where the beginning of execution of each block can be marked in the source code.

[2] Technically, the resulting graph is not a CFG, since the blocks can contain conditionals; however, it maintains the aspect that is relevant to our application: edges indicate the possible sequences during execution.

Since we require markers between segment boundaries, and segments correspond directly with blocks of code associated to CFG nodes, the important aspect to maintain is preserving the beginning of the block by merging nodes corresponding to short blocks into their predecessor nodes. As an example, consider the subgraph of a CFG shown at the left in Figure 4, where block B is too short. If we duplicate node B to merge it into nodes D and E, then marking the

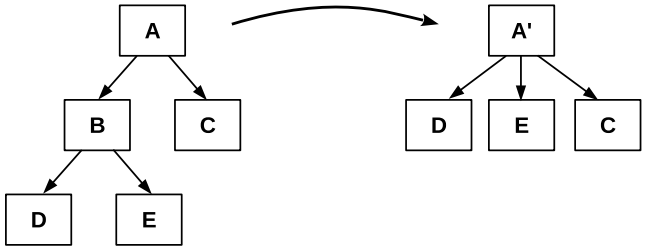


Fig. 4. Example of merging CFG nodes

beginning of the new resulting blocks would become difficult, unless we actually duplicate the actual source code corresponding to B; but this would introduce important difficulties. Instead, we merge node B into its predecessor, node A to create node A′. The result is consistent with the initial CFG: the meaning of this new CFG subgraph is that if we enter node A′, then the possible successors are node C (if block B does not get executed) or nodes D or E (if B does execute). The beginning of block A′ (the line in the source code) remains the same as the beginning of block *A*, and there is no ambiguity. Block B no longer needs its beginning marked, since block B is no longer being considered, and instead, it is part of block A′. When executing, marks are correctly applied at the beginning of each block. Blocks with multiple possible internal paths are not a problem; we enter block A′ and its starting point is marked. The next mark will occur at the beginning of one of its successors, and execution of any instance of block A′ will be enclosed between the mark at its beginning and the next mark that appears.

Algorithm 3 shows the procedure used to manipulate the CFG to eliminate blocks that are shorter than a specified threshold (expressed as number of lines of source code), with Suc($\cdot$) denoting the set of successors of the argument $\cdot$, and Pred($\cdot$) the set of predecessors of the argument $\cdot$.

This process of creating the two instrumented versions of the program was, for the most part, automated. However, we had to manually handle some corner cases and some quirks in the CFG extraction software. We still claim that our proposed technique can be fully automated in that for all of these corner cases in which we fixed the situation manually, it was possible to automatically handle it without user intervention; it was just a matter that we chose to fix it manually given that it represented less work and creating an automated solution seemed overkill for our purpose. As an example, we noticed the issue that the starting

Algorithm 3: Merge CFG Nodes for Short Blocks.

Input: G (CFG), T (Threshold – minimum block size)

On exit: G is modified to represent the same source code with larger/merged blocks.

```
begin
    while G ∋ block b : |b| < T do
        for each node N ∈ G do
            if N.size < T then
                S ← Suc(N);
                P ← Pred(N);
                for each node p ∈ P do
                    Suc(p) ← Suc(p) ∪ S;
                    p.size ← p.size + N.size;
                end
                G ← G \ N;
            end
        end
    end
end
```

line of the initial BB in the extracted CFG of a function is always marked as beginning at the line where the function name and parameter is given. For example, in the code below:

```
Line 1:  void polling_loop (int sleep_time)
Line 2:  {
Line 3:      sleeping = 0;
...
```

The beginning of the entry block in the extracted CFG is line 1, instead of line 3. Though it is certainly feasible to automatically fix these cases, it was simpler for the purpose of our research project to manually edit the instrumented code.

Another aspect where we adjusted manually is related to actual lengths of the segments of the captured traces corresponding to CFG blocks. Here, we not only tried to avoid short traces (for which we merged with successor nodes as needed), but also wide variations in trace lengths. Our experiments showed that the classifier's performance is negatively affected by having to classify a trace when candidates have wide variations in length.

However, the really long blocks only need to be split into one starting block of reasonable size, followed by the remaining of the block. This second block is long, but this is not a problem, since after detection of the first block (non-problematic, as it is of reasonable length), then the *only* candidate successor will be the long block. The important detail is that the classification of the

initial segment is done with candidate blocks that are all within the same range of lengths, and the longer block is never mixed with other candidate blocks for classification. This was notably the case for the SHA function (more specifically, the `sha_transform` section), which takes almost 10 times longer to execute than any of the other functions considered, and the implementation is fully unrolled, so the CFG consists of a single block. The close to 9000 samples segment was split into an initial one of approx. 800 samples, and the remaining 8000+ samples.

4 Experimental Evaluation

The experimental evaluation includes three parts:

- **Random sequence of functions.** We evaluate our system against a target executing randomly generated sequences of MiBench [11] functions, with a random choice of two functions to execute next at each step in the sequence. The experiment is run multiple times, and we randomly generate a different sequence for each execution. The rationale for this choice is twofold: (i) it allows us to compare the performance against previous works, especially against the results reported in [21]; and (ii), a sequence of code with a "random CFG" constitutes a highly demanding task for our classifier, and this has two important consequences: the results obtained are not "helped" by any particular structure of specific software that one may choose for this purpose; and also, the results are more statistically meaningful.
- **Cruise Control application.** The target device executes a SCADE 6 [8] Cruise Control application. This application follows the periodic, real-time tick based scheme where execution alternates between an interval of computations and idle. The rationale for using a concrete, real-world application is also clear: as much as the execution of random sequences of functions has important advantages, we still want to demonstrate the effectiveness of our technique on real applications. Not surprisingly, the performance of our system was substantially better for this case, given the simpler structure of the software and the more systematic patterns in the execution.
- **Case-Studies.** We implemented two case-studies, both in the context of enforcing security properties. The case-studies demonstrate the applicability in practice of our technique. The first case-study shows the efficacy of our approach to detect out-of-sync conditions resulting from buffer overflow conditions (in principle malicious, but applicable to crashes caused by defects/bugs in the software). The second case-study is an intrusion detection system that operates on the fly, detecting anomalous conditions as they happen, with low latency and capacity to process power traces at the throughput that they are produced.

Many aspects in the experimental setup are common for both parts. The following section describes the setup.

4.1 Workflow

Figure 5 shows the hardware setup, specifically the workflow using two workstations to automate the experimentation. The workflow itself does not require two

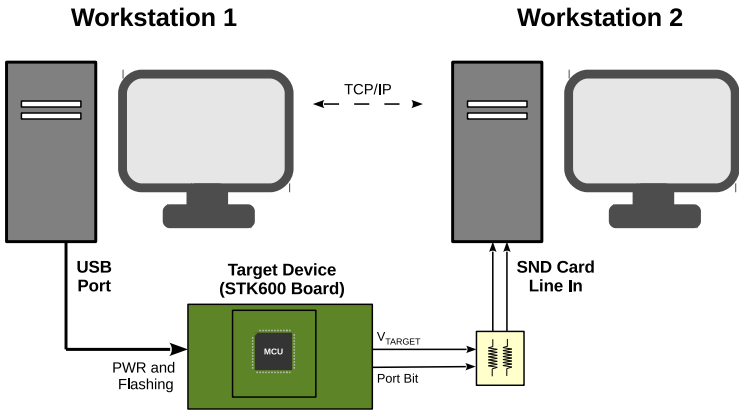


Fig. 5. Experimental Setup for Automated Experimentation.

workstations; but the connections for the signals capture forced us to electrically isolate the flashing from the capture (see below, text related to Figure 6).

The workstations communicate via TCP/IP to synchronize the required actions: Workstation 2 is the "master" in that it instructs Workstation 1 to generate an instance of the software and flash the target device; for the experiments with random sequences of functions, this involves generating a random sequence and generating random input data; for the cruise control application, it involves generating random input data. Workstation 2 then captures the trace, and signals Workstation 1 that it is done with the capture, so that the loop repeats. The software running on Workstation 2 captures and processes the traces. It detects the bit flips (markers at the boundaries between trace segments) by looking for inflection points between neighboring minima and maxima. We used the standard numeric approximations for the derivatives [26], with interpolation to find the position of the inflection point with sub-sample resolution.

As briefly mentioned in Section 3.5, we use both channels of the stereo input of the sound card. This allows us to capture the power trace and additionally a signal with markers at the boundaries between segments of executed code. Figure 6 shows a simplified diagram of this setup. Notice that the $(-)$ terminals in the sound card are connected to the workstation's GND reference, which means that we must connect $+$V to it, since $+$V is the terminal that is common in both differential measurements. This in turn required the use of two workstations with isolated grounds, to be able to isolate the USB ground from the sound card ground.

The $10\,\mathrm{k}\Omega$ resistor provides a voltage divider with $\mathrm{R_M}$ for the port bit signal, and its reasonably high value keeps the current low.

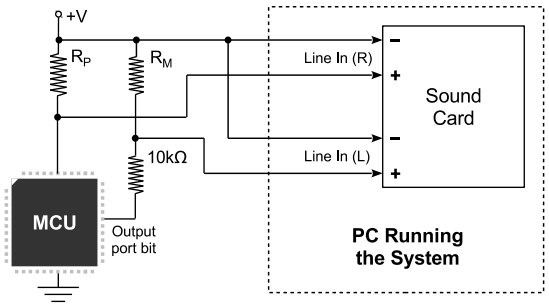


Fig. 6. Power Trace Capture System – Sound Card.

The audio capture does not require any user intervention, as it is performed by the processing program through command-line utilities. Moreover, the section of interest of the trace is surrounded by distinctive "sync" patterns (a sequence of pulses of 1 ms, 2 ms, and 3 ms repeated ten times) to allow the program to automatically isolate the relevant fragments. Figure 7 shows an example of the

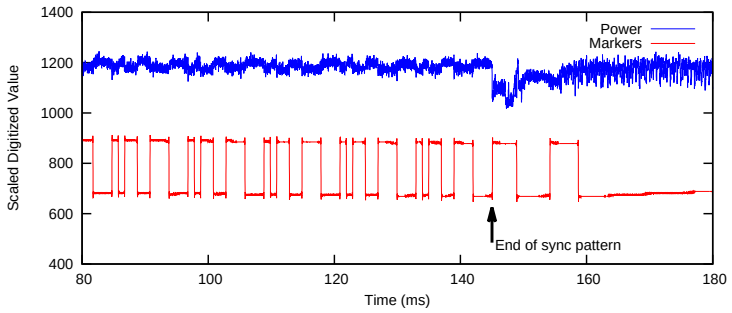


Fig. 7. Example of a Stereo Capture of Trace with Markers.

beginning of one of these traces, showing the power trace and the markers trace, which includes the sync pattern and the edges marking the boundaries between segments.

We also evaluated the technique with a custom-made capture device, including an input instrumentation (differential) amplifier and the analog-to-digital (A/D) converter. In addition to taking steps towards an actual practical implementation of the technique, we wanted to address two potential limitations of the capture with a standard PC sound card:

- Sound cards block DC (in our case, we observed a first-order high-pass filter with 5 Hz cutoff frequency), and lack of short-term DC information may have a negative impact on the classifier's performance.
- Sampling at higher frequency may capture important spectral information that could improve the classifier's performance.

Figure 8 shows a diagram of this higher-precision capture setup. We designed a custom board with these components — as opposed to using a digital oscilloscope and other bench top components — since our short-term objective is an actual practical implementation of our technique. For simplicity, a single block is

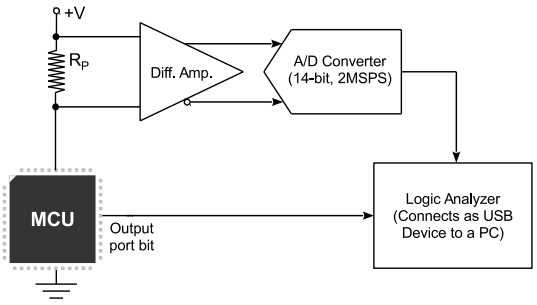


Fig. 8. Power Trace Capture System – Higher-Precision A/D Converter.

shown as differential amplifier. However, this includes an input instrumentation amplifier, anti-alias filter, and a single-ended-to-differential amplifier to drive the ADC. The chain as a whole provides the required amplification to bring the signal from a few millivolts across R_P up to the range of the ADC. The port bit from the MCU is connected to one of the inputs of the logic analyzer, and the positions of the markers are used in a similar manner with respect to the sound card setup, with the exception of the sync patterns: since the logic analyzer has configurable triggers, the use of these sync patterns was not necessary with this setup.

We used a *custom-made* pseudorandom number generator (PRNG) to randomize the input data and the choice of functions to execute. This ensures that execution on the target and on the print-instrumented version produce the same trace. This is not guaranteed if we use the Standard Library PRNG, since it can potentially vary between compilers. We used a linear congruential generator with 64-bit internal state, as described in [15]. The PRNG is seeded by the code generator software running on Workstation 1, using `/dev/urandom`. Appendix B shows an example of the starting fragment of these random sequences.

We emphasize the aspect that the training phase and the operation phase in our experiments always use different input data, to ensure that the results are meaningful. This is the case since every execution of a function (for either training or operation purposes) operates on randomly selected input data.

Figures 9 and 10 show the experimental procedures for the training phase and the performance evaluation phase, respectively.

The implementations are in fact coded as infinite loops, simply relying on the user to interrupt the program when they estimate that a sufficient amount of data has been collected.

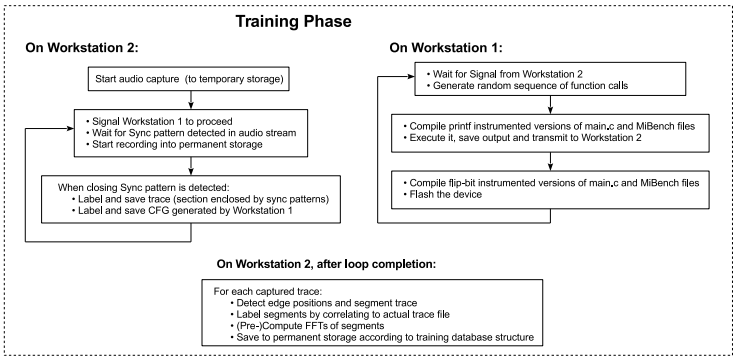


Fig. 9. Procedure for the training phase

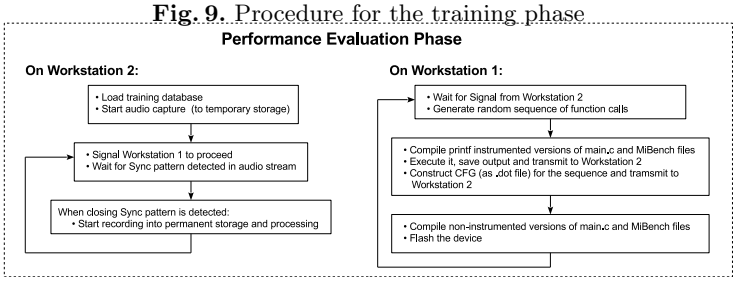


Fig. 10. Operation phase and performance evaluation

5 Experimental Results

In this section we present and briefly discuss the results from our experimental evaluation.

5.1 Classifier's Performance

The metric used to evaluate the performance is the standard notion of precision. In our case, this corresponds to the fraction of the time during which the classifier output corresponds to the correct segment or block (a true positive):

$$P \triangleq \frac{\sum |I_{T_\mathrm{P}}|}{\sum |I_{T_\mathrm{P}}| + \sum |I_{F_\mathrm{P}}|} \tag{10}$$

where P denotes the precision, I_{T_P} are the intervals for which the output of the classifier is a true positive, I_{F_P} are the intervals where the output is a false positive (a misclassification), and $|\cdot|$ denotes the length of the argument $\cdot$ (the length of the interval). The notion of recall is not applicable, since at all times the classifier outputs something — either a true positive or a false positive.

Table 1 shows the measured precision for the various experiments, including 95% confidence intervals. The "Raw" measurement is the precision obtained while the system is in sync with the CFG — roughly speaking, it corresponds to the probability of correct classification when the candidates are restricted to the actual possible options. It was measured by counting misclassifications

but correcting them so that the next classification is done with the correct set of candidates. The purpose of this metric is to isolate the effect of using the CFG to narrow down the set of candidates for the classifier from the issue of having to maintain sync with the CFG. This allows for a more direct comparison against the results in [21], as they report the precision when classifying functions executed in isolation as well as the overall system precision including the task of maintaining sync after misclassifications. The high-precision sampling was only done for the cruise control application, since the random sequences produced traces too long to be processed automatically when using the higher sampling rate.[3] With the use of the dynamic programming / CFG expansion approach,

Table 1. Classifier Precision.

	Random Sequence	**Cruise Control Application**
Raw	97.1% ± 0.3%	––
With CFG Expansion	86.25% ± 3.4%	95.68% ±0.01%
High-precision Sampling	––	**96.06% ±0.08%**

the experiment with random sequence of functions used a depth of 8 for the tree, and with the cruise control application, a depth of 5.

The results show a reasonably good precision, given the granularity at which our system operates — 800 functions correspond to approx. 3000 nodes, giving a granularity close to four times finer than that reported in [21]. Working at this substantially finer granularity, the precisions that we obtain are similar to those in [21]: 97.1% precision for classification of individual blocks; close to the 98% reported in [21] when classifying individual functions in isolation. And 86.25% overall precision, with the classifier never going out of sync; in the same order as the 88% reported in [21]. For the Cruise Control application, the performance was substantially higher, even when working with a lower recursion depth (which also improves execution speed), and the classifier never went out of sync. As expected, the higher-speed sampling lead to a higher precision in the classification, even if the difference was not substantial. We suggest further research to investigate in detail the effect of sampling rate and bandwidth (in principle, relative to the MCU's clock speed), as well as the effect of the DC information on the classifier's performance.

Observation of the classifier's output additionally gave us several interesting insights that will be discussed in Section 6.

[3] Notice that this was an issue for the experimental evaluation, since we used long random sequences that made the matching and alignment too costly.

5.2 Resynchronization with the Trace and the CFG

We implemented, with partial success, the mechanism for resynchronization described in Section 3.4. We tested it only on one of the traces, and it reported successful resynchronization on four occasions. However, the resynchronizations were slow — on one of the instances, it looped four times, failing three times in a row to resynchronize, which would negatively affect the precision if included in a real system. We did not attempt to measure precision or any detailed metrics related to this mechanism, as we are convinced that this is subject for future work.

There are several factors and parameters involved in the operation of the resynchronization mechanism, and some of them are in conflict with the criteria required to make the normal operation (assisted by the CFG) achieve good performance; for that reason, we believe that further research is necessary to determine the best way to implement it. Most notably, the mechanism is based on classification without assistance from the CFG, for which we would expect that we would need to work at a coarser granularity level, and it might require tuning of the database, making it incompatible with the CFG-assisted classifier.

We believe that additional research is necessary before attempting to combine this mechanism with our current classification approach. However, we are convinced that the mechanism is bound to work correctly, and it is a matter of finding the correct parameters and the right way to combine it with the system in its current form.

5.3 Case-Study: Buffer Overflows

As a case-study to assess the usability of our runtime monitoring technique in practice, we repeated the experiments with a deliberately introduced defect that allows buffer overflows. We performed this modified experiment in two distinct ways: overwriting the return address with a random value (a "bug" in the conventional sense); and overwriting the return address with a crafted value to cause execution to return to a different address (a buffer-overflow / code reuse attack [1, 29]). As expected, for both scenarios the system irrecoverably went out of sync with the CFG and misclassified essentially every segment after the buffer overflow occurred.

The shifts in the trace segments (the deviation of the starting point with respect to the "nominal" position, given by the outcome of the previous classification) provide a good indicator of an out-of-sync condition. When the system is operating normally, we expect the shifts to be small, to compensate for minor deviations due to measurement noise. When operating on a trace that is not consistent with the CFG, the matches are found at somewhat random positions, resulting in large values of the shifts. Figure 11 shows the shift values for the case where the buffer overflow occurs at the seventh block; as expected, we observe a noticeable increase in the values after that position.

Though we did not incorporate any formal anomaly detection techniques [3] to automate the reporting of these unrecognized segments, the results represent

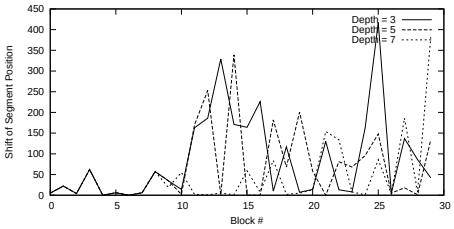


Fig. 11. Effect of a buffer overflow bug/attack on the classifier's shifts

encouraging evidence to the usability of our technique in the context of either monitoring to detect faulty behavior or as an IDS.

5.4 Case-Study: Intrusion Detection System for Vehicle Door Lock

Our second case-study demonstrates the ability of our technique to operate in real-time. By "real-time" we refer to the ability to process power traces at the throughput that they are produced, and detect anomalous conditions as they occur, with low latency.

Figure 12 shows a block diagram of a vulnerable door lock control system and the power-based monitor to detect anomalous behavior. The remote key can

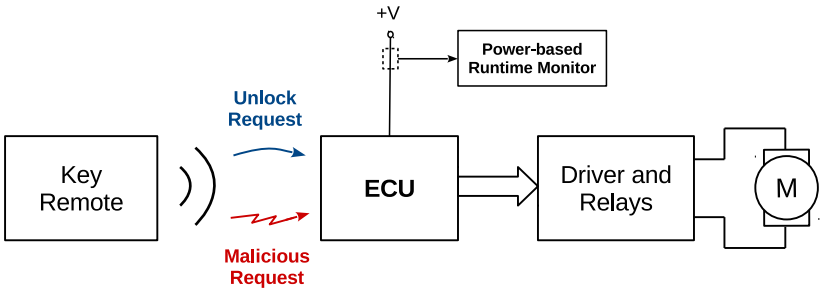


Fig. 12. Vehicle Door Lock Control System and Power-based Monitor.

transmits dual commands, representing the possibility of a maliciously cloned remote key that transmits crafted attack data. This malicious data could for example exploit a buffer overflow vulnerability in the electronic control unit (ECU). In particular, our system demonstrates a code-reuse buffer overflow attack. The ECU implements the algorithm shown below as C-like pseudocode:

```
while (1)
{
    check_incoming_data_available();
    if (data_available)
    {
        char buffer[256+1];     // null-terminated 256-byte blocks
        read_wireless_data (buffer);
        if (authentication_ok (buffer, SECRET_KEY))
            unlock_door();
    }
    sleep (10 milliseconds);
}
```

The maliciously transmitted command implements a buffer overflow that injects the address of the `unlock_door` function to unlock the vehicle while bypassing the authentication.

The system was implemented on the same AVR microcontroller used for the other experiments (as opposed to a real vehicle's ECU). Furthermore we use a donated *Magna Closures* door lock in our demo to showcase the technology and our work on cybersecurity for embedded systems.

The monitoring system is implemented on a workstation with a quad-core processor at 3.4GHz running Ubuntu 14.04 and using gcc/g++ to compile the C++ programs that implement our technique. We split the workload into several processes, in particular to be able to read audio and process it concurrently. To be able to run in real-time, we did not implement the CFG expansion mechanism. The system has four basic blocks: background, receive, authentication, and unlock. The training database contains three background samples, two receive samples, one authentication sample, and one unlock sample.

With this setup, the system operates on the fly, including transmission of the data to a separate server for display/demo purposes. A trace segment of 3000 samples is transmitted every two seconds. Since the operation is manual, we did not extract detailed statistics for the purpose of performance assessment. However, the system has worked successfully as a demo, showing few false alarms, and rarely missing any anomalous conditions.

We acknowledge the fact that the system operates in real time in part because the system being monitored is an aggressively optimized implementation of a simple functionality. Real-time monitoring of real-life systems is still beyond the scope of our technique at the present time. Further research is required to achieve the required processing speed with good performance, including CFG expansion, and reasonable cost of the system. The use of specialized digital signal processing architectures or custom digital hardware may play a favorable role in achieving this objective.

6 Discussion and Future Work

One of the positive aspects to highlight relates to the potential for usability of our system as a runtime monitoring tool in real-world systems; the experimental results confirm this potential for cases where execution follows the CFG but deviating from specifications (e.g., an infinite loop due to lack of validation of input data) and also the cases where execution violates the CFG constraints (e.g., stack corruption, invalid pointer accesses, malware/tampering, etc.). Combining our approach with the technique in [22] is a promising avenue to further improve our system's performance, and is one of the aspects suggested as future work.

The following are some of the interesting insights that we obtained from this work, in particular from analysis of the classifier's output from the experiments:

- **Use of additional static analysis to improve the precision of the classifier**. We could observe that one of the main opportunities for misclassifications arises from segments that are short in length and where the CFG expansion allows a substitution without getting out of sync. Static analysis could reduce the set of paths that can execute (with respect to using the CFG alone). This would also improve speed, as it reduces the size of the expanded CFG in our dynamic programming algorithm in the classifier.
- **Using the shifts to avoid misclassifications**. We could observe several instances where the shifts (the deviation from the nominal starting point of a segment) could help correct misclassifications; indeed, several errors occurred for instances where the correct path was $A \rightarrow B \rightarrow C$ and the classifier output $A \rightarrow C$, with a large positive shift for A and a large negative shift for C, which suggests that the choice $A \rightarrow B \rightarrow C$ was likely the correct one (in any case, the system could confirm this if it verifies that the shifts for the former case are small).
- **Optimizing the choice of CFG blocks**. The choice of CFG blocks could be adjusted to improve the classifier's performance; for example, this could address the aspect mentioned above, where a short segment is incorrectly selected without getting out of sync. By looking at the training samples and estimating probabilities of correct classification, situations prone to errors could be identified and avoided through a different choice of CFG blocks, obtained by merging blocks in different combinations.
- **Use of Digital Signal Processors**. The use of specialized architectures or even custom digital hardware (e.g., FPGA) could play an important role in achieving real-time operation to monitor real-life systems. These could assist with spectral analysis and distance computations, possibly with an additional standard processor for the classification algorithms and CFG expansion.
- **Pruning of the expanded CFG**. An interesting, though possibly more risky, option for improving speed could be the use of the technique known as pruning [31] (§10.5.2); branches of the expanded CFG could be eliminated earlier by recursing down to a smaller depth and discarding the branches

that already at that point exhibit a sum of distances far above the rest of the sibling branches.

The following are some additional aspects that we believe warrant further investigation:

- **Training database size**. This is an important aspect to consider for the technique to be applicable in practice. In addition to the direct implication of the database size, it also relates to the effort required to set up the system, which affects the applicability of our technique in a real-life setup. Future research to quantify this aspect would be valuable for the technique to be used in practical applications.
- **Sampling rate for the power traces**. This is another aspect that affects the applicability of our technique in practice. It would be interesting to determine the required sampling rates for the technique to work reliably. Intuitively, it seems reasonable to expect that the required sampling rate is a function of the clock speed of the processor being monitored. Future research could confirm this intuition, and even quantify the relationship between processor clock speed and required sampling rate for the power trace.

7 Conclusions

In this paper, we presented a non-intrusive program tracing technique and showed its applicability to runtime monitoring. We used a novel signals and system analysis approach, combined with static analysis to further improve both performance and methodology. The proposed technique exhibits substantially better performance compared to previous work on power-based program tracing, as it has comparable precision while working at a granularity level close to four times finer. Two case-studies confirm the potential of our technique either as a runtime monitoring tool or as an IDS for embedded devices.

Acknowledgments

The authors would like to thank Pansy Arafa, Hany Kashif, and Samaneh Navabpour for their valuable assistance with the CFG and instrumentation infrastructure as well as related discussions.

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Ontario Research Fund.

References

1. Aleph One: Smashing the stack for fun and profit. Phrack magazine (1996)
2. Atmel Corporation: AVR 8-bit and 32-bit Microcontrollers (2012), http://www.atmel.com/products/microcontrollers/avr
3. Chandola, V., Banerjee, A., Kumar, V.: Anomaly Detection: A Survey. ACM Computing Surveys (CSUR) (2009)

4. Chen, F., Roşu, G.: Java-MOP: A Monitoring Oriented Programming Environment for Java. In: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2005)
5. Chris Lattner and the LLVM Developer Group: The LLVM Compiler Infrastructure – online documentation, `http://llvm.org`
6. Clark, S.S., Ransford, B., Rahmati, A., Guineau, S., Sorber, J., Fu, K., Xu, W.: WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices. In: USENIX Workshop on Health Information Technologies. USENIX (2013)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Third edn. (2009)
8. Dormoy, F.X.: SCADE 6: A Model Based Solution for Safety Critical Software Development. In: Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08) (2008)
9. Eisenbarth, T., Paar, C., Weghenkel, B.: Building a Side Channel Based Disassembler. In: Transactions on Computational Science X, pp. 78–99. Springer Berlin Heidelberg (2010)
10. Frigo, M., Johnson, S.G.: The Design and Implementation of FFTW3. Proceedings of the IEEE (2005), special issue on "Program Generation, Optimization, and Platform Adaptation"
11. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Workload Characterization. IEEE Computer Society (2001)
12. Havelund, K.: Runtime Verification of C Programs. In: International Conference on Testing of Software and Communicating Systems (2008)
13. Havelund, K., Roşu, G.: Monitoring Java Programs with Java PathExplorer. Electronic Notes in Theoretical Computer Science 55(2), 200 – 217 (2001), RV'2001, Runtime Verification
14. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: A Run-Time Assurance Approach for Java Programs. Formal Methods in System Design 24(2), 129–155 (2004)
15. Knuth, D.E.: The Art of Computer Programming. Volume 2: Seminumerical Algorithms. Addison-Wesley, Third edn. (1998)
16. Liu, Yannan and Wei, Lingxiao and Zhou, Zhe and Zhang, Kehuan and Xu, Wenyuan and Xu, Qiang: On Code Execution Tracking via Power Side-Channel. In: ACM Conference on Computer and Communications Security. pp. 1019–1031. ACM (2016)
17. Matt Bishop: Computer Security: Art and Science. Addison-Wesley (2003)
18. Moreno, C.: Side-Channel Analysis: Countermeasures and Application to Embedded Systems Debugging (2013), PhD Thesis (University of Waterloo)
19. Moreno, C., Fischmeister, S.: Non-intrusive Runtime Monitoring Through Power Consumption: A Signals and System Analysis Approach to Reconstruct the Trace. In: International Conference on Runtime Verification. pp. 268–284. Springer (2016)
20. Moreno, C., Fischmeister, S.: On the Security of Safety-critical Embedded Systems: Who Watches the Watchers? Who Reprograms the Watchers? In: 3rd International Conference on Information Systems Security and Privacy (2017)
21. Moreno, C., Fischmeister, S., Hasan, M.A.: Non-intrusive Program Tracing and Debugging of Deployed Embedded Systems Through Side-Channel Analysis. In: Conference on Languages, Compilers and Tools for Embedded Systems. pp. 77–88 (2013)

22. Moreno, C., Kauffman, S., Fischmeister, S.: Efficient Program Tracing and Monitoring Through Power Consumption – With A Little Help From The Compiler. In: Design, Automation, and Test (DATE) (2016)
23. Msgna, M., Markantonakis, K., Mayes, K.: The B-side of side channel leakage: control flow security in embedded systems. In: International Conference on Security and Privacy in Communication Systems. pp. 288–304. Springer (2013)
24. Navabpour, S., Joshi, Y., Wu, W., Berkovich, S., Medhat, R., Bonakdarpour, B., Fischmeister, S.: RiTHM: A Tool for Enabling Time-triggered Runtime Verification for C Programs. In: Foundations of Software Engineering. pp. 603–606. ACM (2013)
25. Pnueli, A., Zacks, A.: PSL Model Checking and Run-Time Verification via Testers. 14th International Symposium on Formal Methods (2006)
26. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical Recipes in C. Cambridge University Press, Second edn. (1992)
27. Proakis, J.G., Manolakis, D.G.: Digital Signal Processing: Principles, Algorithms, and Applications. Prentice Hall, Fourth edn. (2006)
28. Seyster, J., Dixit, K., Huang, X., Grosu, R., Havelund, K., Smolka, S.A., Stoller, S.D., Zadok, E.: Aspect-Oriented Instrumentation with GCC, pp. 405–420 (2010)
29. Solar Designer: "return-to-libc" Attack. Bugtraq (Aug 1997)
30. Webb, A.R., Copsey, K.D.: Statistical Pattern Recognition, 3rd ed. Wiley (2011)
31. Weiss, M.A.: Data Structures and Algorithm Analysis in C++. Addison-Wesley, Third edn. (2006)

Appendix A Instrumentation of the Source Code

Below are examples of the two instrumented versions of the source code for the case of the ADPCM coder.

Print-instrumented version:

```
void adpcm_coder(short indata[],
                 char  outdata[],
                 int   len,
  struct adpcm_state * state)
{
    short *inp;
        /* Input buffer pointer */
    signed char *outp;
        /* output buffer pointer */
    /* ... other declarations */

printf ("Node0x20ccb50\n");
    outp = (signed char *)outdata;
    inp = indata;
    valpred = state->valprev;
    index = state->index;
    step = stepsizeTable[index];
    bufferstep = 1;

    for ( ; len > 0 ; len-- )
    {
printf ("Node0x20ccea0\n");
        val = *inp++;
        diff = val - valpred;
        /* ... */
    }
    /* ... */
}
```

Flip-port-bit-instrumented version:

```
extern char volatile port_bit;
#define FLIP_PORT_BIT \
  {PORTG = (port_bit = !port_bit);}

void adpcm_coder(short indata[],
                 char  outdata[],
                 int   len,
  struct adpcm_state * state)
{
    short *inp;
        /* Input buffer pointer */
    signed char *outp;
        /* output buffer pointer */
    /* ... other declarations */

FLIP_PORT_BIT;
    outp = (signed char *)outdata;
    inp = indata;
    valpred = state->valprev;
    index = state->index;
    step = stepsizeTable[index];
    bufferstep = 1;

    for ( ; len > 0 ; len-- )
    {
FLIP_PORT_BIT;
        val = *inp++;
        diff = val - valpred;
        /* ... */
    }
    /* ... */
}
```

Appendix B Randomized Sequences of Functions

Below is an example of a randomized sequence of functions. The program running on Workstation 1 randomly chooses the 64-bit seed for the `rnd64` PRNG, as well as the choice of functions at each step (for example, `encrypt` and `crc32buf` were randomly chosen for the first step, `sha_update` and `adpcm_coder` for the second step, and so on).

The function `randomize_data` uses `rnd64` to generate pseudorandom input data for the functions. Every eight steps (eight `if` statements) we re-randomize and assign a new random value into `rnd`, since each step consumes one of its eight random bits.

```
srnd64(UINT64_C(8973546545337244988));
uint8_t rnd;

randomize_data();
rnd = ((rnd64() >> 24) & 0xFF);
if (rnd & 0x1)
    encrypt (plaintext, ciphertext, &ctx);
else
    rc = crc32buf (crcdata, CRCSIZE);
rnd >>= 1;


if (rnd & 0x1)
    sha_update (&sha_info, sha_data, SHASIZE);
else
    adpcm_coder (pcmdata, adpcmdata, PCMSIZE,
                 &coder_1_state);
rnd >>= 1;

if (rnd & 0x1)
    fft_float (FFTSIZE, 0, real_in, imag_in,
                           real_out, imag_out);
else
    sha_update (&sha_info, sha_data, SHASIZE);
rnd >>= 1;

...
```