Non-Intrusive Runtime Monitoring Through Power Consumption: A Signals and System Analysis Approach to Reconstruct the Trace

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.

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. 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 [19]. In that work, they used power consumption measurements — *power traces* — to determine blocks of source code being executed. 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 [19] operated at the granularity level of whole functions, which may be too coarse to be practical. Indeed, [19] does not present any case-studies to support the idea of this non-intrusive tracing technique being useful in practice. The work in [20] proposes a technique that can be combined with the approach in [19], 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. However, their results showed a performance far too low to be applicable in practice. Clark et al. [5] 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 [22], 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, 25] and time-triggered frameworks [21] 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 [19], [5], 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 [19]: 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

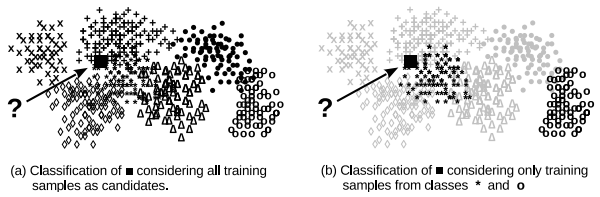


Fig. 1. Reducing the set of candidates for classification

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 a case-study presented as evidence of the usability of this technique. This case-study applies 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. The case-study involves introducing a buffer-overflow bug/vulnerability, exploited in two distinct ways: (i) overflowing the stack to make execution return to a random address (a "bug" in the conventional sense); and (ii) through a buffer-overflow attack [1, 7], where the stack is overwritten in a controlled way to hijack the device's execution. Results from the case-study confirm our approach's potential and usability in these two contexts.

1.2 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 [24]:

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

A frequency domain representation of a discrete-time signal $x(n)$ can be obtained through the (Discrete-Time) Fourier Transform $\mathcal{F}$, defined as [24]:

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

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

[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.

Given the Fourier Transform $\mathcal{X}(\omega)$, the signal $x(n)$ can be obtained through the inverse Fourier Transform $\mathcal{F}^{-1}$, defined as [24]:

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

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 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 [24]:

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

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 [24]. 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.

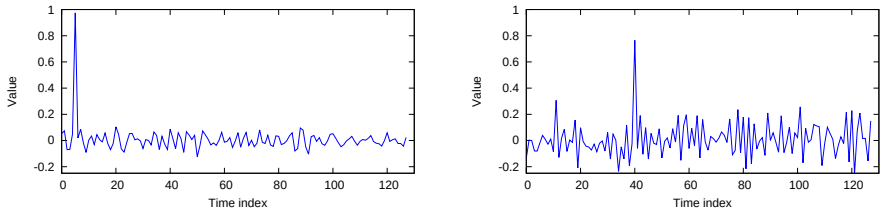
3.1 Frequency Analysis: Classifying and Determining the Shift in the Power Trace Segments

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 identified system will correspond 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 [24].

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 [19], 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-based analysis: consider a system that shifts the signal by n_0 samples, with impulse response $h(n) = \delta(n - n_0)$. If we look at an N-samples window of a periodic signal, the shift occurs circularly within the window. 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.



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

Fig. 2. Examples of impulse responses

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 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 [26].

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.

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 [6]: 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.

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
```

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

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 Segmentation of Traces and Fragments of Source Code

One important limitation in the approach proposed in [19] 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. This was done in a way such that the effect on the power traces is negligible (Section 4.1 describes this setup in more detail).

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.

3.5 Instrumenting the Source Code

We used LLVM [16] 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.

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

[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.

corresponding to short blocks into their predecessor nodes. As an example, consider the subgraph of a CFG shown at the left in Figure 3, where block B is too short. We merge node B into node A to create node A′. The result is consistent

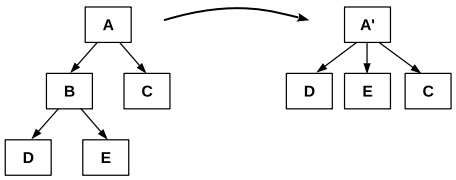


Fig. 3. Example of merging CFG nodes

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.

4 Experimental Evaluation

The experimental evaluation includes two parts:

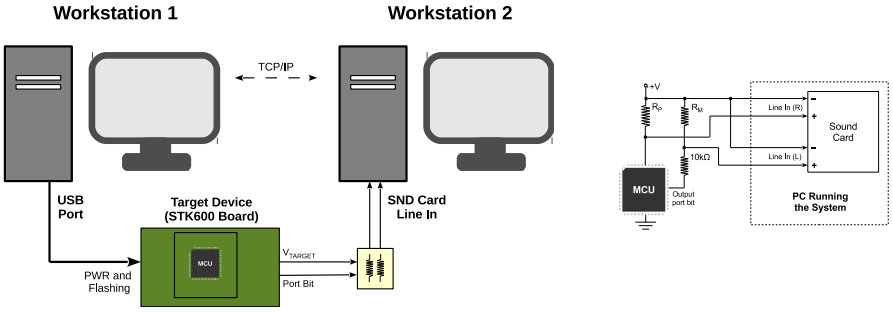
- **Random sequence of functions.** We evaluate our system against a target executing randomly generated sequences of MiBench [11] functions, with a random choice of two functions to execute next at each step in the sequence. The experiment is run multiple times, and we randomly generate a different 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 [19]; 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.

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

4.1 Workflow

Figure 4 shows the hardware setup, including the use of two workstations to automate the experimentation (Figure 4(a)) and the interface subsystem to capture the power trace and markers through the sound card (Figure 4(b)). The



(a) Setup for automated experimentation (b) Power trace capture

Fig. 4. Experimental setup

workflow itself does not require two workstations; but the connections for the signals capture forced us to electrically isolate the flashing from the capture.

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. 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 [23], with interpolation to find the position of the inflection point with sub-sample resolution.

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`.

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 5 and 6 show the experimental procedures for the training phase and the performance evaluation phase, respectively.

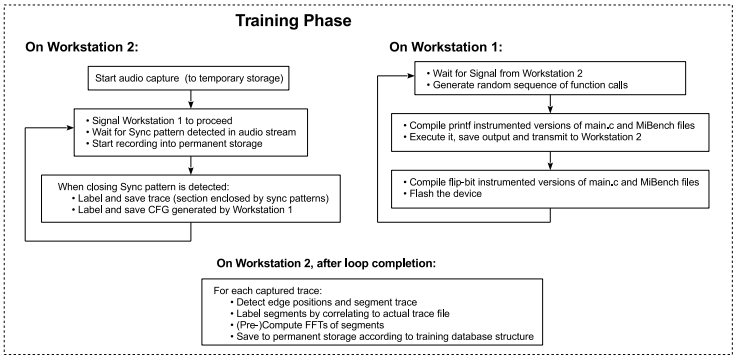


Fig. 5. Procedure for the training phase

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.

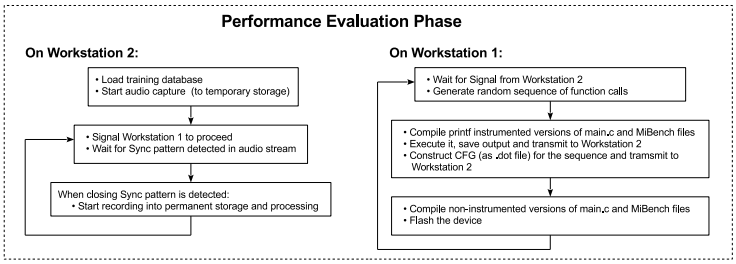


Fig. 6. Operation phase and performance evaluation

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):

$$P \triangleq \frac{\sum |I_{T_P}|}{\sum |I_{T_P}| + \sum |I_{F_P}|} \tag{10}$$

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 [19], 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. With the use of the dynamic programming / CFG expansion approach, 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.

	Random Sequence	**Cruise Control Application**
Raw	97.1% ± 0.3%	––
With CFG Expansion	86.25% ± 3.4%	**95.68% ± 0.01%**

Table 1. Classifier Precision.

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 [19]. Working at this substantially finer granularity, the precisions that we obtain are similar to those in [19]: 97.1% precision for classification of individual blocks; close to the 98% reported in [19] 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 [19]. For the SCADE 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.

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

5.2 A 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). 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 7 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.

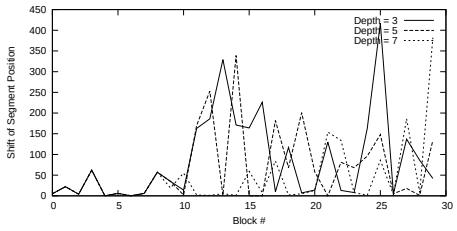


Fig. 7. Effect of a buffer overflow bug/attack on the classifier's shifts

Though we did not incorporate any formal anomaly detection techniques [3] to automate the reporting of these unrecognized segments, the results represent encouraging evidence to the usability of our technique in the context of either monitoring to detect faulty behavior or as an IDS.

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 [20] 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.

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. A case-study confirmed 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. 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)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Third edn. (2009)
7. Designer, S.: "return-to-libc" Attack. Bugtraq (Aug 1997)
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. Lattner, C., the LLVM Developer Group: The LLVM Compiler Infrastructure – online documentation, `http://llvm.org`
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., Hasan, M.A.: Non-intrusive Program Tracing and Debugging of Deployed Embedded Systems Through Side-Channel Analysis. Conference on Languages, Compilers and Tools for Embedded Systems pp. 77–88 (2013)
20. 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)

21. 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)
22. Pnueli, A., Zacks, A.: PSL Model Checking and Run-Time Verification via Testers. 14th International Symposium on Formal Methods (2006)
23. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical Recipes in C. Cambridge University Press, Second edn. (1992)
24. Proakis, J.G., Manolakis, D.G.: Digital Signal Processing: Principles, Algorithms, and Applications. Prentice Hall, Fourth edn. (2006)
25. 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)
26. Webb, A.R., Copsey, K.D.: Statistical Pattern Recognition, 3rd ed. Wiley (2011)