Non-Intrusive Program Tracing of Non-Preemptive Multitasking Systems Using Power Consumption

Kamal Lamichhane
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
klamichh@uwaterloo.ca

Carlos Moreno
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
cmoreno@uwaterloo.ca

Sebastian Fischmeister
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
sfischme@uwaterloo.ca

***Abstract*—System tracing, runtime monitoring, execution reconstruction are useful techniques for protecting the safety and integrity of systems. Furthermore, with time-aware or overhead-aware techniques being available, these techniques can also be used to monitor and secure production systems. As operating systems gain in popularity, even in deeply embedded systems, these techniques face the challenge to support multitasking.**

In this paper, we propose a novel non-intrusive technique, which efficiently reconstructs the execution trace of non-preemptive multitasking system by observing power consumption characteristics. Our technique uses the control-flow graph (CFG) of the application program to identify the most likely block of code that the system is executing at any given point in time. For the purpose of the experimental evaluation, we first instrument the source code to obtain power consumption information for each basic block, which is used as the training data for our Dynamic Time Warping and k-Nearest Neighbours (k-NN) classifier. Once the system is trained, this technique is used to identify live code-block execution (LCBE). We show that the technique can reconstruct the execution flow of programs in a multi-tasking environment with high accuracy.

***Index Terms*—Power tracing, System tracing, Runtime monitoring, Scheduling, Operating system, Embedded software**

I. Introduction

Microelectronic embedded systems are an important part of our daily lives, as they have been an important factor in many technological advances in recent decades. Such an extensive outreach has demanded an increase in research devoted to the development and security of these systems. Embedded systems are deployed to run independently in a self-sufficient manner. Once deployed, these systems behave as a black box with no capacity for runtime debugging. This opacity of internal code execution offers the benefit of increased security. However, it also makes system tracing, runtime monitoring, and execution reconstruction challenging.

If the faulty behaviour is observed in the production and/or deployment phase of product development cycle where developers are no longer able to make use of debugging tools, non-intrusive tracing is possibly the only available technique. Due to the dynamic nature of multitasking system, some runtime anomalies are non-recurrent in consecutive system executions and may be difficult to reproduce after recompiling or restarting the device. One popular workaround is to hard-code code snippets to aid in runtime debugging. However, it is often challenging to think ahead and exhaust all possible error scenarios prior to deployment. Further, due to memory constraints in deployed products, developer prefer omitting debugging information. Added instrumentation to obtain the runtime tracing could break extra-functional requirements of the system. These memory limitations and non-reproducible nature of run-time errors advocate the necessity of a non-intrusive run-time monitoring system to safeguard the integrity of modern electronic devices.

In this paper, we propose a novel non-intrusive technique, which efficiently reconstructs the execution trace of non-preemptive multitasking system by observing power consumption characteristics. The proposed technique can be extended to a broad range of microcontroller (MCU) platforms. For this work, we chose the AVR ATMega2560, which is extensively used in cyber-physical systems these days.

In [1], [2], Moreno et al. showed a technique for non-intrusive program tracing and debugging through power side channel analysis where they use the power consumption characteristics of the MCU to identify blocks of code being executed. In [3], they improve performance through a compiler-assisted stage that maximizes distinguishability of traces for different blocks of code. All of these works lack multitasking support, which limits the practical applicability of the techniques. In [4], Eisenbarth et al. have presented side-channel disassembler technique to obtain the sequence of CPU instructions without using the source code information. Accuracy is the main limitation of this work which is too low for any practical environment. Msgna et al. [5] presented the idea of side channel control flow security in the embedded system where they collect several traces and calculate the mean of traces to minimize the inherent and ambient noise introduced by the measurement setup. Calculating the mean of all traces may work in the dedicated environment, but it is not suitable for a practical application where the processing must be done on-the-fly to ensure the security and reconstruct the execution trace of the program. Clark et al. [6] uses the behaviour monitoring system of medical devices to model permissible behaviour and detect deviation. That work, however, is limited to simple and highly repetitive operation of the device.

A. Our Contribution

We address some important limitations of the current state-of-the-art by adding support for co-operative multitasking scheduling. Our technique uses the control flow graph (CFG) of the application program along with the power trace to identify the most likely block of code that the system is executing at any given point in time. To this end, we use a statistical classification approach, with dynamic time warping (DTW) as the distance metric used by the nearest neighbours (NN) technique. We show that our proposed method can accurately reconstruct the execution trace of programs in the multi-tasking environment.

B. Organization of the Paper

The remaining of this paper advances as follows: We discuss the background in Section II and formulate the problem statement in Section III. Section IV presents the proposed technique followed by Evaluation in Section V. Section VI includes discussion and future work followed by some concluding remarks in Section VII.

II. Background

In this section, we discuss the key concepts used in our approach.

A. Power Trace

The power trace $P = \langle k, p_k \rangle$ is the time series representing the power consumption of a microcontroller as a function of time, with power being sampled periodically. p_k is the measured (instantaneous) power consumption at time index $k \in \mathbb{Z}$.

B. Control Flow Graph (CFG)

Control Flow Graph $G = \langle V, E \rangle$ is a directed graph which represents the execution flow of the program. Where V is the set of vertices representing basic blocks, and E is the set of edges — an edge from block $BB1$ to block $BB2$ ($BB1, BB2 \in V$) indicates that execution of block $BB2$ can immediately follow the execution of block $BB1$. A basic block is the sequence of executable instruction with a single entry point at the beginning and exit point at the end.

C. Dynamic Time Wrapping (DTW) Algorithm

DTW is an algorithm to find the optimal match between two-time series which may vary in speed. Since time series are wrapped non-linearly in time dimension independent of the non-linear variation along the series, we may need compression or expansion in time in order to find the best mapping. DTW algorithm is first introduced by [7], and being used in time series data mining and sequence classification [8].

DTW algorithm compares two time series A and B where $A = (a_1, a_2, a_3, \cdots, a_m)$ and $B = (b_1, b_2, b_3, \cdots, b_n)$ of length m and n, respectively. DTW uses local distance matrix between each element of the time series to compare the similarity; elements having a minimum local distance $min(a, b)$ are likely to be more similar. The optimal minimum path can be calculated using the dynamic programming approach.

III. Problem Statement and Assumptions

Our proposed technique addresses the following problem: given the power trace P and the CFG G of a system running a program in non-preemptive scheduling with a known scheduling policy $\mathcal{P}$, determine the correct node $b \in G$ executing at a given time point k.

In the context of this work, we make the following assumptions:

- Known MCU: the system runs on a known processor model. This assumption is necessary since the relationship between power consumption and program execution depends on the processor design and implementation.
- Input Identification: all possible combination of execution contexts are generated using random input initialization and multiple runs.
- Components involved: targeted platform does not have cache, pipeline, or similar micro-architectural features. Additionally, we do not consider multi-core execution in this work. Presence of such component would produce high variation in power consumption depending on the execution context.
- Control flow integrity: we do not consider cases such as random execution due to memory or stack corruption, undefined behaviour deriving from uninitialized pointers or in general invalid pointer operations, "system crashes", etc. We also assume that no active adversary tampers with the control flow using techniques such as code injection or code reuse through buffer overflow attacks [9].

IV. Proposed Technique

As described in Section I, our proposed technique uses the concept of non-intrusive side channel power analysis to determine the code block being executed by the MCU (an ATMega2560 clocked at 1 MHz). We capture the power consumption by adding a shunt resistor in series with the power-in line going to the MCU so that a voltage proportional to the instantaneous power consumption is produced. The shunt resistor is chosen such that the voltage drop that it causes is small enough that the normal functionality of the MCU is not affected. Since the produced voltage is in the order of a few millivolts, we added an analog input stage to amplify it for further processing. The signal is then digitized through an analog to digital converter (ADC), which samples at 14-bit resolution at a frequency of 2 MHz. The converted samples are captured using a Saleae Logic Analyzer. Figure 1 shows the power capture setup of the experiment. Port_Bit_Marker in Figure 1 is useful for the training phase of the classifier: it allows us to segment the power trace into power trace segments corresponding to BBs. To this end, the MCU port_bit (Port_Bit_Marker) toggles the logic level at the beginning of each BB. In our experiments, we use a custom designed non-preemptive multitasking operating system for AVR microcontroller that supports basic multitasking with yield/context switching and Round-Robin scheduling. To show the efficacy of our approach on the industrial real-time system,

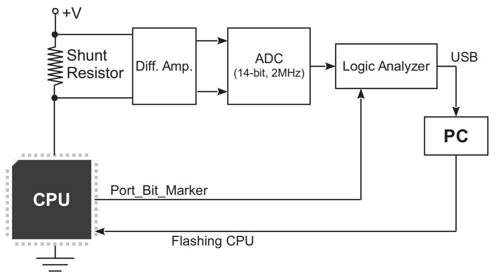

Fig. 1: Capture Setup

we chose four application programs to run each as one task: Cruise Control, Water Level Control, Adaptive Differential Pulse-Code Modulation (ADPCM) encoding, and Cyclic Redundancy Check (CRC32) computation. Cruise Control and Water Level Control are obtained directly from the sample SCADE [10] models and run without any modification in the program. ADPCM and CRC are obtained from MiBench [11].

To carry out the experimental evaluation of our technique, the source code is instrumented to allow us to obtain the power consumption information of each basic block (Section IV-A discusses this in more detail), which is therefore used as the training data for our DTW and k-NN classifier. Instrumentation done in this phase does not have any effect on the functionality and the execution flow of the program. Moreover, the code instrumentation does not introduce undesirable overhead in the captured power traces. During classification, we use a non-instrumented version of a program where the power trace of a complete program is fed as input to the resulting trained system which gives the real-time execution trace.

The power trace associated with each basic block may exhibit variations due to the context in which it is executed (the input data and state). Thus, we should train the system with power traces of a set of execution instances that is statistically representative of the power traces generation process. To this end, we used randomization for all program inputs and replicate the experiment multiple times.

A. Source Code Instrumentation

We used the LLVM optimizer and analyzer -dot-cfg pass, to extract the CFG of the programs. Some of the basic blocks in the code produce a power trace segment too short for the classifier to work accurately. We merge small CFG nodes (having 1 or 2 lines) to create a large CFG node without deviating from the original control-flow constraints of the program.

We created two instrumented versions; one that executes on the target MCU, and another version that runs offline on a workstation. *Printf instrumented version* is the instrumented code that runs on the workstation, and *flip_port instrumented version* is the instrumented code that runs on the target. In *Printf instrumented version*, we instrument the source code by adding a print statement at the beginning of each BB which prints the ID of the basic block that is currently executing and the yield information at the end of BB. In *flip_port instrumented version*, we instrument the source code by adding the FLIP_PORT_BIT statement in the exact same location as we did for the printf instrumented version. This version executes on the target and uses a general purpose input/output (GPIO) pin to signal transitions between basic blocks including context switch by toggling the port bit. The instrumentation simply places a pin-toggle statement at the beginning of each basic block. This information is captured using the 16^{th} bit of logic analyzer as shown in Figure 1. *Printf instrumented version* gives the actual trace which is processed to get the ground truth for our classifier. we make sure both versions (printf and flip port version) follow the same execution path every time using the same seed for randomized input.

B. Capturing Power Traces, Preprocessing and Classification

We capture the power trace using the logic analyzer whose 16^{th} bit was connected to *port_bit* of MCU and bits 1-15 delivered the power data. We generate traces for each basic block from the entire trace using FLIP_PORT_BIT (each logic level change signifies either the transition of a basic block or a context switching). We replicate the trace capture of the system multiple times. The time complexity of each DTW comparison is $O(n \cdot m)$ where, n and m are the lengths of each time series. Due to this, the time taken to calculate the warping distance of testing time series with *all* the training time series is considerably large. To reduce the number of comparisons, we average the power traces and generate one power trace for each basic block. While this in principle means that we use the nearest centroid rule and lose the benefits of the 1-NN with DTW technique, it has been shown that the performance is similar for these two techniques [8]. Though they use warping as part of the averaging process, this is necessary for the general case, where the average of different time series instances may produce a result completely dissimilar to all the averaged samples. This is not the case in our system, as different instances of execution of a BB still execute the same operations, and it is only the data that varies, which has an effect with the characteristics of added noise.

During classification, we consider all possible sizes from the start point of the trace according to lengths of basic blocks in the training database, and compare them against each trace of training database to determine the best match. The system advances to the next segmentation point, based on the length of the matching segment, and repeat the process.

V. Evaluation

In this section, we describe the experiments and metrics used to evaluate the performance of the implemented technique. To show the efficacy of our technique in the real-time systems, we tested our classifier with multiple number of yields/context switch and multiple number of tasks.

A. Evaluation Metrics

We use the precision to evaluate the performance of the classifier. Since our classifier always outputs one of the options all the time, whether it is true positive or false positive; there is no notion of recall, specificity, and fall-out in our experiment. Precision($P = \frac{TP}{TP+FP}$) is defined as the ratio of total True positive (TP) instances to the sum of True Positive instances and False Positive (FP) instances. In our case, TP are instances where the classifier correctly determines the basic block currently executing in the MCU while FP are instances of incorrect classifications.

Measuring the Precision: To measure the precision, we take the actual execution trace running in offline mode, i.e., printf instrumented version, as the ground truth. We then compare the output of the classifier with the actual trace to determine the true positives (correct classifications) and false positives (misclassifications).

TABLE I: Overall Precision of the System

Application (Task)	Precision (%)
Water Level Control	97.28
Cruise Control	95.64
ADPCM	97.33
CRC	98.00
Yield	98.26
Average	97.30

Table I shows the precision of a four-task multitasking system. From this table, it is evident that our technique can accurately produce an execution trace of a multitasking system that is running on the embedded device. We also ran our experiment with two, three, and four tasks, and with multiple numbers of yields. Table II presents the precision of an overall system with multiple number of tasks and multiple number of yields. From Table II, it can be seen that the number of context switches in the application does not affect the precision; however, the precision changes by $\pm 1\%$ when varying the number of tasks.

VI. Discussion and Future Work

During our work on power profile classification, we could identify at least one limitation: basic blocks with similar power profiles were the primary cause of misclassification. For example, a block with consecutive additions could be mistaken for a block with one multiplication. This paves the way for future work where finer granularity in the CFG can be exploited for more accurate classification. Though there are faster DTW classifiers available, such as fastdtw instead of the traditional DTW classifier, we intend to undertake it as future work. Another possible area of future research would be to investigate this technique in systems with cache and other microarchitectural elements. Additionally, we could extend this work to support multi-core execution platforms.

TABLE II: Precision with Different Number of Tasks

	Number of Tasks			Number of Yields			
	2	3	4	8	12	15	17
Precision (%)	96.50	96.83	97.30	97.3	97.24	97.14	97.23

Our work can be extended to work in a more general anomaly detection for real-time multitasking systems. With the prior knowledge of CFG and the power trace of all the nodes/basic block on the program, any external interfering program is likely to produce a significantly different control flow path and power traces, hence assisting in anomaly detection.

VII. Conclusion

In this paper, we built upon previous work on non-intrusive program tracing using power side channel analysis, by demonstrating accurate reconstruction of execution trace in the cooperative multitasking environment. We deployed a 1-Nearest Neighbour combined with Dynamic Time Warping and static code analysis to achieve an overall accuracy of 97.3%. To demonstrate usability in real-world applications, we chose cruise control, water level control, ADPCM, and CRC for our experimental evaluation and demonstrated efficient reconstruction. We firmly believe that our method provides a good starting point for future research on non-intrusive runtime tracing of multitasking embedded systems.

References

[1] C. Moreno and S. Fischmeister, "Non-intrusive Runtime Monitoring Through Power Consumption: A Signals and System Analysis Approach to Reconstruct the Trace," in *International Conference on Runtime Verification*. Springer, 2016, pp. 268–284.

[2] C. Moreno, S. Fischmeister, and M. A. Hasan, "Non-intrusive Program Tracing and Debugging of Deployed Embedded Systems Through Side-Channel Analysis," in *Conference on Languages, Compilers and Tools for Embedded Systems*, 2013, pp. 77–88.

[3] C. Moreno, S. Kauffman, and S. Fischmeister, "Efficient Program Tracing and Monitoring Through Power Consumption – With A Little Help From The Compiler," in *Design, Automation, and Test in Europe (DATE)*, 2016.

[4] T. Eisenbarth, C. Paar, and B. Weghenkel, "Building a Side Channel Based Disassembler," in *Transactions on Computational Science X*. Springer Berlin Heidelberg, 2010, pp. 78–99.

[5] M. Msgna, K. Markantonakis, and K. Mayes, "The B-side of Side Channel Leakage: Control Flow Security in Embedded Systems," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2013, pp. 288–304.

[6] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, K. Fu, and W. Xu, "WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices," in *USENIX Workshop on Health Information Technologies*. USENIX, 2013.

[7] R. Bellman and R. Kalaba, "On Adaptive Control Processes," *IRE Transactions on Automatic Control*, vol. 4, no. 2, pp. 1–9, 1959.

[8] F. Petitjean, G. Forestier, G. I. Webb, A. E. Nicholson, Y. Chen, and E. Keogh, "Faster and More Accurate Classification of Time Series by Exploiting a Novel Dynamic Time Warping Averaging Algorithm," *Knowl. Inf. Syst.*, vol. 47, no. 1, pp. 1–26, 2016.

[9] Aleph One, "Smashing the stack for fun and profit," *Phrack magazine*, 1996.

[10] F.-X. Dormoy, "SCADE 6: A Model Based Solution for Safety Critical Software Development," in *European Congress on Embedded Real Time Software*, 2008.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Proc. Workload Charact. 2001. WWC-4. 2001 IEEE Int. Work.*, pp. 3–14, 2001.