Efficient Program Tracing And Monitoring Through Power Consumption – With A Little Help From The Compiler

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

Sean Kauffman
School of Computer Science
University of Waterloo, Canada.
Email: skauffma@uwaterloo.ca

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

***Abstract*—Ensuring correctness and enforcing security are growing concerns given the complexity of modern connected devices and safety-critical systems. A promising approach is non-intrusive runtime monitoring through reconstruction of program execution traces from power consumption measurements. This can be used for verification, validation, debugging, and security purposes.**

In this paper, we propose a framework for increasing the effectiveness of power-based program tracing techniques. These systems determine the most likely block of source code that produced an observed power trace (CPU power consumption as a function of time). Our framework maximizes distinguishability between power traces for different code blocks. To this end, we provide a special compiler optimization stage that reorders intermediate representation (IR) and determines the reorderings that lead to power traces with highest distances between each other, thus reducing the probability of misclassification. Our work includes an experimental evaluation, using LLVM for an ARM architecture. Experimental results confirm the effectiveness of our technique.

I. Introduction

Modern connected devices and safety-critical systems are rapidly increasing in complexity and functionality. Consequently, there is growing interest in runtime monitoring for the purpose of ensuring correctness and enforcing security. The complexity of modern systems makes it difficult to incorporate runtime monitoring tools that work alongside the rest of the software without breaking extra-functional requirements such as timing constraints.

A promising approach is non-intrusive monitoring through reconstruction of program execution traces from power consumption measurements. This can be used for verification, validation, debugging, and security purposes. Moreno et al. [1] presented a novel approach where power consumption is used to reconstruct program traces. This is accomplished through the use of statistical pattern recognition techniques, where the system determines the most likely fragment of code that produced an observed power trace (captured sequence of power consumption as a function of time) [2].

Eisenbarth et al. [3] attempted a similar technique, targeting assembly-level instructions. However, at this fine granularity, the reported performance was too low for a practical application. Clark et al. [4] presented a malware detector based on side-channel analysis (power consumption) for medical devices. Their technique, however, is limited in that it operates at the granularity level of the execution of the entire program, and that it relies on the device executing a simple and highly repetitive task.

In this paper, we propose a framework for increasing the effectiveness of power-based program tracing techniques. We focus on the techniques that use classifiers to determine most likely blocks of code that produced the observed power traces. Our framework increases the effectiveness of this classification process by maximizing distinguishability between power traces for different code blocks. To this end, we provide a special compiler optimization stage that affects the code generation and layout with distinguishability as the optimization criterion. This optimization stage reorders intermediate representation (IR) instructions and estimates the resulting power trace for a given reordering. It then determines the reorderings that lead to power traces with highest distances between each other, thus reducing the probability of misclassification.

An additional feature in our framework with respect to the work presented in [1] is the use of the control flow graph (CFG) [5]. Our approach assumes the use of the CFG to constrain the classification process and only consider valid sequences of blocks; given the sequence of blocks that executed in the immediate past, the CFG indicates the set of blocks that can be currently executing. Thus, the classifier only needs to consider those blocks as candidates in the classification process. As a consequence, the compiler optimization stage only needs to maximize distinguishability between blocks corresponding to *sibling* nodes in the CFG (i.e., nodes with a common parent).

Our work includes an experimental evaluation, implemented using LLVM [6] with an Atmel SAM D21 ARM MCU [7]. Results from our experiments confirm the soundness of our approach and the potential for usability in practical scenarios.

The remainder of this paper proceeds as follows: we start with a brief background review in Section II. We describe our proposed approach in Section III, followed by the details of our experiments (sections IV and V). We finish with a discussion in Section VI and some concluding remarks in Section VII.

II. Background – Statistical Pattern Recognition

Our proposed framework focuses on power-based tracing techniques that rely on statistical pattern classification to determine most likely blocks of code given an observed power trace [2].

978-3-9815370-6-2/DATE16/

Specifically, these techniques seek to maximize the conditional or *a posteriori* probability among all candidate fragments given the power trace [1] produced by the execution of the unknown fragment of code.

Several techniques exist and are commonly used to accomplish the above goal. Often, we do not have an explicit (analytic or otherwise) description of the distribution of the elements. In those cases, classification techniques use a training database consisting of a set of S samples $\{\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_S\}$, each of them labelled with the class to which the sample is known to correspond. Classification for a given element is done based on proximity (usually Euclidean distance) to the database samples. The nearest centroid rule also uses a training database of labelled samples; for each class C_k, the system determines the centroid of all training samples with label C_k:

$$\overline{\mathbf{X}}_k = \frac{1}{N_k} \sum_{n=1}^{N_k} \mathbf{X}_n \quad (1)$$

where N_k is the number of training samples with label C_k.

The classification decision for a given element $\mathbf{X}$ consists of selecting the class corresponding to the centroid nearest to $\mathbf{X}$ (also with Euclidean distance being the usual metric).

This is a key notion for our proposed technique: if we consider a two-centroids scenario, the probability of misclassification is related to distance between centroids. This probability is given by the area (or volume, in the multidimensional case) under the curves corresponding to the probability density functions taken from the equidistant point or region with respect to the two centroids; the farther apart the centroids, the smaller this probability should be, since we take a smaller portion of the tails of the probability functions.

III. Our Proposed Technique

We now present the details of our proposed framework as well as our implementation using LLVM with an ARM Cortex-M0 target architecture.

A. Reordering Instructions

The key aspect behind our technique is the relationship between sequences of individual instructions and the resulting power trace. If we reorder the instructions in a program, the power trace produced by the modified program will, in general, be different. Since the classification process is based on distinguishing power traces corresponding to different blocks of code, our approach is based on the idea of reordering instructions corresponding to the various blocks to make the resulting power traces maximally distinguishable.

A trivial solution exists for this problem: we can always introduce new and unnecessary instructions into the program which have wildly different power signatures. Clearly, this would not be an acceptable solution in most cases, due to its detrimental effect on performance. For a solution to have practical value, it should modify the program in such a way that performance is affected the least. One way to ensure that performance is not drastically affected is to avoid introducing new instructions, and instead only reorder the instructions that are part of the original program.

We should also consider the fact that the classifier can take advantage of information about the CFG of the program to improve the classification process. The intuition is that if we narrow down the candidate blocks considered by the classifier, we should reduce the probability of misclassification. Thus, we should focus on maximizing distinguishability between blocks that can be candidates for the same classification instance. This can only happen if the nodes have common predecessors; after execution of a given block corresponding to a node in the CFG, the classifier only considers the successors to the current basic block. If we can increase the distance between power traces corresponding to basic blocks that are the successor nodes to the same block in the CFG, we can improve their differentiability and lower the rate of misclassifications on the program. Additionally, we want to increase the distance for the nodes that are most easily misclassified without adversely affecting our ability to correctly classify the nodes that are already easy to distinguish.

Thus, our optimization task consists of increasing the distance between basic blocks and their siblings in the CFG. We define a sibling to mean all of the successors of all of the predecessors of a basic block in the CFG that are not the block itself. It is important to consider sibling nodes in the above sense rather than considering the successors of each node. This is the case because a block may have more than one predecessor in the CFG. If we only increase the distance between the successors to a single basic block, we may do so by *decreasing* the distance between those blocks and their siblings from other predecessors. During classification, a basic block may be a candidate together with any of its sibling nodes since the set of candidate blocks depends on the specific instance of execution.

To calculate this distance metric, the power trace of each basic block can be broken down into the power traces of its component instructions. Since a basic block consists of a sequence of instructions with a single entry and a single exit point, it can be thought of as a series of instructions which will execute in order. Thus, the power trace of a basic block can be conceived as the series of the power traces of its component instructions, given that the power trace corresponds to power consumption as a function of time. If we know the number of clock cycles for every instruction to complete and we know how much power they require on average, we can determine the average power traces for basic blocks.

B. A Little Help from the Compiler

The process of reordering instructions described in the previous section was implemented in a compiler optimization stage. We chose LLVM for its modularity and ease of modification, and to make our modifications to the intermediate representation (IR) of the program.

Much work has been done in the area of modifying the compilation of a program to change its power consumption.

[1] More specifically, given a *noisy* measurement of the power trace

Typically, the goal is to reduce the power consumption of the program. There are several approaches to this problem, but the most typical is to try to reduce transition activity in the instruction bus by reducing the hamming distance between instructions in the program [8], [9]. This is an effective way to approach the problem, since it provides a simple heuristic for which the code can be optimized and for which one approach can be judged to be objectively better than another. However, the problem is NP-Hard [8] and tends to discount the many other factors in a processor. Other research has been done into methods that require customized hardware, such as optimizing the use of way-specific registers in multiple issue digital signal processors [10] or by using a combination of instruction packing, booth multiplier operand reordering [11].

Our approach is based on the idea of modifying the order of the instructions. When run after most other optimizations, instruction ordering gives us control over the power trace of a basic block with low performance impact. Although some basic blocks contain only instructions with data dependencies between them, most contain some instructions which can be reordered without changing the semantics of the program. To do this, we needed more granular information about the power used by instructions.

We created a framework for generating machine code corresponding to sequences of a single LLVM IR instruction. Our goal was to modify the IR representation of the program, but the power traces for instructions were CPU/MCU specific. To be able to associate a power trace with an IR instruction on a target, we need to profile the system: determine what machine instructions are emitted by the LLVM backend for a particular IR instruction on the particular target, and then measure the power trace for that sequence of assembly-level instructions. We generated assembly files representing 1000 executions of an IR instruction for most of the instructions in the LLVM language and for most valid combinations of operand sizes. Depending on the target, there could be large variations between the number and type of instructions emitted for different IR instructions.

These files were then run on the target hardware and their power traces were recorded. To avoid issues with resolution or accuracy in the measurements, each file consisted of 1000 repetitions of the same sequence of machine instructions corresponding of one IR. We then divided the trace into 1000 sequences of measurements and took the mean of each index. The resulting vector represents the given IR instruction for the target. A number of examples of these vectors are given in Figure 1. We will present a more detailed description in Section IV, when describing the experimental setup.

A vector for a basic block can then be created by concatenating the traces of its instructions. We can calculate the differentiability between two basic blocks as the distance between these two vectors. Because the length of each instruction vector varies, it is not enough to simply add up the distance between subsequent instruction traces. This is demonstrated in Figure 1, which shows how different instructions can have power traces with wildly different lengths.

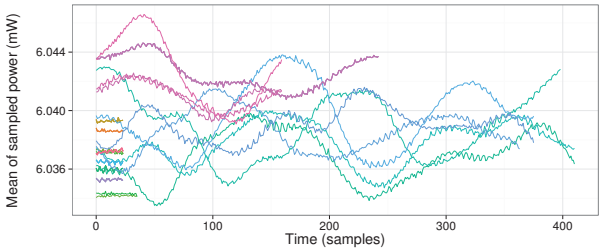

Fig. 1. Instruction Trace Vectors

We define the distance of a basic block to be the average of the square Euclidean distance between its power trace and the power traces of its siblings in the CFG.[2] When evaluating the distance between two traces, we consider only the initial portion of the traces that overlaps, since this is the only meaningful way to compare traces when the classifier is making a decision during normal operation. Given a basic block b and the set of its sibling basic blocks S_b, where a basic block consists of a vector representing its power trace over time, the distance is given by:

$$\text{distance}(b) = \frac{1}{|S_b|} \sum_{s \in S_b} \sum_{t=0}^{M_s} (b[t] - s[t])^2 \qquad (2)$$

where $M_s \triangleq \min(|b|, |s|)$.

We use an iterative, greedy algorithm to improve the total distance of a program until a threshold is reached. The algorithm starts by iterating over each basic block in the CFG and finding its sibling nodes. It then begins the main loop, which starts by placing all of the basic blocks into a list sorted by their distance. The block with the lowest distance is then chosen and all valid permutations of its instructions are checked and its instruction order is changed to the permutation with the highest distance. A valid permutation means that the semantics of the basic block are not changed, so the order of instructions with data dependencies is preserved. Because this validity restriction drastically reduces the number of possibilities, it becomes practical to search all permutations for a moderately large basic block. Once the block is reordered and its siblings are removed from the list, the block with the next lowest distance in the list is chosen to repeat the process. Once the list becomes empty, it is repopulated and sorted until the change in total distance of the program drops below some specified threshold. Algorithm 1 shows the details of this optimization procedure.

C. Example

As a contrived example let us look at the C code in Figure 2 which is part of a program that we would like to classify. It contains two basic blocks which are siblings in the CFG. Figure 2 also shows the equivalent LLVM IR generated by the LLVM program `clang`.

[2] This is a common optimization in pattern recognition techniques, where comparing square distances yields the same result as comparing distances without incurring a square root computation.

Algorithm 1 Optimization Algorithm

```
1: procedure REORDER(CFG, threshold)
2:     for block ∈ CFG do
3:         for pred ∈ block.predecessors do
4:             block.siblings ← pred.successors \ block
5:     do
6:         BBList ← CFG.blocks
7:         Sort BBList, ranked by distance
8:         while BBList ≠ ∅ do
9:             block ← BBList[0]
10:            for p ∈valid permutations of block do
11:                if distance(p) > distance(block) then
12:                    block ← p
13:            BBList ← BBList \ {block}
14:            BBList ← BBList \ block.siblings
15:    while Δ ∑ distance(block ∈ BBList) < threshold
```

```
if (x < y)
{                       if.then:
  x = x + z;              %add = add nsw i16 %z, %x  ← will be
  y = y << z;             %shl = shl i16 %y, %z      ← reordered
  x = x & y;              %and = and i16 %add, %shl
}                         br label %if.end
else
{                       if.else:
  y = y + z;              %add2 = add nsw i16 %z, %y
  x = x << z;             %shl2 = shl i16 %x, %z
  x = x & y;              %and2 = and i16 %shl2, %add2
}                         br label %if.end
```

Fig. 2. C code and equivalent LLVM IR

Even without knowing the power traces for any of the instructions, we can see that the two basic blocks are similar, so they will be difficult to differentiate. In fact, looking only at the type of instruction and the sizes of its operands, these two blocks are identical. When the algorithm is run on this program, one of these two blocks is sorted to the top of the list to be rearranged first, because the distance to its siblings is 0. Then when different permutations of the block are tried, they are limited because of data dependencies. The `br` instruction cannot be moved, because it is the terminating instruction for each basic block and must come at the end, and the `and` instruction before it depends on the results of the other two instructions so it cannot come before either of them. Only the `add` and `shl` instructions can be moved, so the algorithm chooses to change their order in the first basic block.

IV. Experimental Setup

The key aspect in our experiments was the capture of power traces corresponding to the execution of basic blocks. This introduced an important difficulty, since we needed to run the fragments of code in the natural sequence as they occurred in the program to obtain good accuracy in the measurements. This is due to the fact that power consumption may be affected by low-level hardware features such as pipelines and internal state transitions; thus, if we execute fragments of code in isolation, the power consumption may not reflect the actual power consumption during operation.

To this end, we created two instrumented versions of the programs; one of them executes on the target and uses a GPIO pin to signal transitions between BBs by toggling the pin. The instrumentation simply places a pin-toggle statement at the beginning of each BB. We capture power traces through the Line-In input of a standard PC sound card. This idea was introduced in [1]. Unlike in that work, we used the two channels of the *stereo* sound card input so that one of the stereo channels captures power consumption and the other channel captures the markers. Figure 3 shows an example of a captured trace. The system automatically detects the positions of the

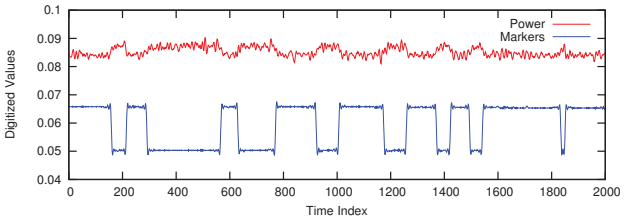

Fig. 3. Example of Captured Power Trace with Markers

markers by identifying pairs of nearby local maximum and minimum, and finally determine the position of the inflection point between these two extreme points. We used the standard numerical approximations of the derivatives to determine the point at which the second derivative changes sign [12].

The second instrumented version contains print statements and it is executed offline on a workstation. We used a pseudorandom number generator (PRNG) to generate inputs for the functions. By seeding the PRNG with the same seed value in both instrumented versions, we ensure that the execution trace will be the same, since the input data is the same in both cases. This allows us to match the segments of the trace (separated by the markers) against the BB labels that are output by the print-instrumented version, and thus we are able to label each of the power trace segments with the BB to which they correspond. To guarantee that this was the case, we coded a custom PRNG, thus avoiding the risk that the standard library random facilities could vary between compilers. We used a linear congruential generator with 64-bit state following the standard conditions to maximize the period [13].

The benefit of obtaining this set of labeled power traces is twofold: (1) we use these labeled samples for the training database. And (2) we can determine the precision of the system (the rate of correct classifications): the experiment runs the classifier feeding the sample *without* labeling and can compare the output of the classifier against the known label associated to the sample. We emphasize the aspect that the experiments always use different samples for the training database and for obtaining the classifier's precision. In particular, different samples of the power traces always correspond to executions with different input data.

V. Experimental Results

The main goal and focus of our experiments is to demonstrate the difference in the classifier's precision as a consequence of the modifications done by the compiler through the optimizing stage that reorders the IR instructions. The precision P is defined as the rate of true positives. This parameter fully

describes the performance of the classifier, since we do not have false negatives (the classifier always outputs something) and thus the notion of recall is not applicable to our case.

$$P \triangleq \frac{\#CC}{\#CC + \#IC} \quad (3)$$

where $\#CC$ denotes the number of correct classifications (true positives), and $\#IC$ denotes the number of incorrect classifications (false positives).

As auxiliary measurements that are directly associated to the manipulations that the compiler performs, we also report distances between centroids, both synthetic (distances between the traces constructed by the compiler while evaluating the reorderings) and obtained from actual measurements on the target. We also created an additional version of the compiler that chooses the estimated worst reorderings; thus, minimizing distances between power traces instead of maximizing them. The purpose of this is to demonstrate the potential effect of these changes in the distances between traces.

We ran each of the MiBench functions 1000 times, obtaining a total number of segments (corresponding to individual executions of BBs) of a little above 1.3 million. Many of the traces, however, correspond to blocks that are too small, and thus we omitted them from the reported results, as they are far too many and add little value since the compiler is limited in how much it can reorder small blocks. To evaluate the precision, we partitioned the sets of power traces corresponding to each BB into two sets. We used one of the sets to construct the training database — essentially, to obtain the centroids for each class (each BB) — and the other set to run the classifier and obtain the precision. We used a process similar to bootstrapping [14] to obtain a good statistical representation of the parameters, including the obtained averages as well as confidence intervals (we report 95% confidence intervals). The process consisted of sampling with replacement, where the partitioning is done multiple times, randomly splitting the set into the two partitions. This corresponds to taking random samples of the population of power traces to run them through the classifier with the rest of the power traces being used to construct the training database.

Table I summarizes the results, including execution of AES encryption and the SHA update function, part of the Security section of MiBench [15]. The $\pm$ figures indicate the 95% confidence interval for the given parameter.

TABLE I
CLASSIFIER PARAMETERS

	Precision	Dist. between centroids
AES encrypt		
Unoptimized	56.53% $\pm$ 0.26	118753
Optimized	63.76% $\pm$ 0.3	259506
SHA update		
Unoptimized	99.0% $\pm$ 0.04	1855802
Optimized	98.83% $\pm$ 0.05	1839577
ADPCM coder		
Unoptimized	58.11% $\pm$ 0.22	130781
Optimized	59.19% $\pm$ 0.19	144209

We can draw some important insights from these results. We observe that the effect of the technique varies for different functions. Perhaps surprisingly, for the SHA blocks, the technique *reduced* the precision. However, the ranges are so close that the difference may be due to measurement or experimental artifacts. Also, with the code being so highly distinguishable in its normal form, we can suspect that the reorderings that the optimizer did were in a sense "driven by noise".

The results for AES, on the other hand, show a remarkable and favorable aspect: the fragment of code below (from MiBench's `aes.c`) shows the two sibling nodes `for.body` and `for.end` corresponding to the body of the loop and the statement that follows the loop:

```
for(rnd = 0; rnd < cx->Nrnd - 1; ++rnd)
{
    round(fwd_rnd, b1, b0, kp);
    l_copy(b0, b1); kp += nc;
}
round(fwd_lrnd, b0, b1, kp);
```

We observe that the two blocks are essentially identical (`fwd_rnd` and `fwd_lrnd` are two macros that expand to the same code, using different data). It is expected that the unoptimized code produced by the compiler would be essentially identical, and the precision greater than 50% could be a result of low-level hardware features that cause a difference between the trace when execution remains within the loop vs. when it leaves the loop.

The results show that the reordering of the instructions corresponding to those blocks plays an important role in the distinguishability between those two otherwise identical blocks. Even though 63.76% is not a particularly high precision, it is still a remarkable result considering that we achieve a reasonable level of distinguishability between two blocks that are normally almost indistinguishable. We should also remark the fact the these blocks are short and thus obtaining a high precision is challenging. This could be compensated for by using CFG expansion to classify based on distances for longer sequences of blocks. This is, however, beyond the scope of this work and is an area that would benefit from future research.

Our results also include computed values corresponding to parameters obtained by the compiler during the optimization stage. Specifically, we computed the total distance for some programs in the MiBench suite before and after optimization by our compiler stage. We also reversed the optimization in order to minimize the total distance, generating the least distinguishable version of a program. The increase against the *default* case represents the improvement in distance over instruction ordering that the compiler generated with no intervention, while the increase against the *worst* case represents the improvement in distance over the least distinguishable version. In the case of `adpcm.c` for ARM, the default case that the compiler generated was indeed the worst case, which is why the increase in distance over them is identical. Table II shows these values. For example, the first row shows that, for the benchmark `adpcm.c` the improvement for ARM was 21.4% above both the default and worst case versions, while

the improvement for AVR was 6.15% over the default and 16% over the worst case.

TABLE II
IMPROVEMENT IN DISTANCE METRIC FROM OPTIMIZATION

Bench	% Increase (default)		% Increase (worst)	
	ARM	AVR	ARM	AVR
adpcm.c	21.4	6.15	21.4	16.0
aes.c (unrolled)	94.2	120.3	105.9	123.2
aes.c	159.4	272.0	186.4	275.6
crc_32.c	8.27	18.97	120.6	60.2
fftmisc.c	8.24	2.37	18.6	15.9
sha.c	58.2	13.14	121.2	27.2

VI. DISCUSSION AND SUGGESTED WORK

Many interesting aspects were observed during the design and implementation of the experiments, as well as from the results obtained. Perhaps the most interesting aspect to discuss is the effect of the code structure and size on the effectiveness of our technique. The precisions obtained for the various functions exhibit large variations. The results, combined with inspection of the various fragments of code being considered, suggest that it is mostly the structure and size of the code that can have a bigger impact on the potential effectiveness of the method. We still claim that the technique is valuable and has a tremendous potential for applicability in practice. One can reasonably expect actual code in real-life applications to include fragments with varying characteristics and structure. Thus, for practical applications, we believe that the technique is bound to work well for a fraction of the blocks being considered, and have little or no effect for the rest, leading to a net increase in the overall performance. Further research could be valuable in getting a more definitive answer to these claims. Additional research could also uncover patterns or relationships between characteristics of the source code and the effectiveness of the technique.

Additional techniques related to the use of the CFG for classification of sequences could help obtain a high precision even at the fine granularity level of basic blocks in the CFG. It would be interesting to study the interaction between our proposed technique and any approaches that the classification system could adopt as means to increase either the performance or the computational efficiency (or both).

Future research is also suggested for the purpose of developing more effective and efficient optimization algorithms that would be applicable in the context of our framework.

As a last remark, it is worth noting that another potential application of our proposed technique is the creation of a rogue (malicious) compiler that could manipulate code generation to facilitate side-channel analysis, in particular power analysis [16], on devices with binaries created by such compiler. Embedded systems security engineers should be aware of this aspect, even if its applicability in practice may be a remote possibility.

VII. CONCLUSIONS

In this work, we presented a novel approach for increasing the effectiveness of power-based program tracing techniques. We showed that by reordering instructions we gain some control over the power traces and can choose reorderings that lead to power traces that are maximally distinguishable. Experimental results confirmed that our approach is viable and has potential applicability in practice.

Important insights were gained from the design of the experiments, implementation, and results. Since this paper introduces a novel idea that can potentially address important problems in the field of embedded systems and in particular embedded systems security, there are many opportunities for future work. As part of this work, we highlighted some of these areas where additional research may prove valuable.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable suggestions and ideas, Brad Lushman and Nomair Naeem for their advice on compiler optimizations, and Aaron Severance for suggesting a comparison against the worst-case.

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

REFERENCES

[1] C. Moreno, S. Fischmeister, and M. A. Hasan, "Non-intrusive Program Tracing and Debugging of Deployed Embedded Systems Through Side-Channel Analysis," *LCTES'13*, pp. 77–88, 2013.

[2] A. R. Webb and K. D. Copsey, *Statistical Pattern Recognition*, 3rd ed. Wiley, 2011.

[3] T. Eisenbarth, C. Paar, and B. Weghenkel, "Building a Side Channel Based Disassembler." Springer Berlin Heidelberg, 2010, pp. 78–99.

[4] S. S. Clark et al., "WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices," *USENIX Workshop on Health Information Technologies*, 2013.

[5] C. N. Fischer, R. K. Cytron, and R. J. L. Jr., *Crafting a Compiler*. Addison-Wesley, 2009.

[6] C. Lattner and the LLVM Developer Group, "The LLVM Compiler Infrastructure – online documentation," http://llvm.org.

[7] Atmel Corporation, "SAM D ARM Cortex-M0+ Microcontrollers," 2015, http://www.atmel.com/products/microcontrollers/arm/sam-d.aspx.

[8] C. Lee, J. K. Lee, and T. Hwang, "Compiler Optimization on Instruction Scheduling for Low Power," in *International Symposium on System Synthesis*, 2000, pp. 55–60.

[9] N. Chabini and M. Wolf, "Reordering the assembly instructions in basic blocks to reduce switching activities on the instruction bus," *Computers Digital Techniques, IET*, vol. 5, no. 5, pp. 386–392, September 2011.

[10] Y.-C. Ma, T.-A. Liu, and W.-S. Chao, "Energy-Aware Compiler Optimization for VLIW-DSP Cores," in *Advances in Intelligent Systems and Applications - Volume 2*. Springer Berlin Heidelberg, 2013, vol. 21.

[11] M.-C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power Analysis and Minimization Techniques for Embedded DSP Software," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1997.

[12] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C*, Second ed. Cambridge University Press, 1992.

[13] D. E. Knuth, *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Third ed. Addison-Wesley, 1998.

[14] A. J. Canty, "Resampling methods in R: the boot package," *R News*, vol. 2, no. 3, pp. 2–7, 2002.

[15] 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." IEEE Computer Society, 2001.

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