Static Transformation of Power Consumption for Program Tracing and Software Attestation

by

Sean Kauffman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Parts of this thesis have been adapted from works ([27, 19]) I authored and co-authored with Carlos Moreno, who is a Postdoctoral fellow under the supervision of Associate Professor Sebastian Fischmeister in the Real-time Embedded Software Group at the University of Waterloo.

Chapter 4 and parts of chapters 2 and 3 are adapted from the work entitled "Efficient Program Tracing and Monitoring Through Power Consumption – With A Little Help From The Compiler," published at the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE) [27] in Dresden, Germany.

Chapter 6 and parts of chapters 2 and 3 are adapted from the work entitled "Static Transformation of Power Consumption for Software Attestation," published at the 2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) [19] in Daegu, South Korea.

Both works were completed under the supervision of Dr. Sebastian Fischmeister at the University of Waterloo, who contributed with ideas, discussion, editing and manuscript review.

Abstract

This thesis presents methods to statically modify programs at compile-time to improve the effectiveness of power consumption based program analyses. Two related applications are considered, and algorithms are introduced for both. The first is power consumption based program tracing, and the second is software attestation with power consumption as a side-effect.

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. Our framework maximizes distinguishability between power traces for different code sections. To this end, we introduce a static transformation to reduce the probability of misclassification by reordering intermediate representation (IR) to find the ordering that produces power traces with the highest distances between them. Experimental results confirm the effectiveness of our technique.

We also consider improvements to the algorithm, replacing the naïve, exhaustive permutation algorithm used in the original solution with Monte Carlo permutations. Due to the complexity of the naïve solution, its search space is constrained, making it unlikely to find a good solution when the number of instructions in a program section is too large. Variations on a basic stochastic implementation are described, and their expected results are compared. The Monte Carlo algorithms consistently found better solutions than their exhaustive counterpart while showing improved scalability.

We then introduce a related technique to statically transform programs to use power consumption as the side-effect for software attestation. We show how to circumvent the undecidable nature of program execution for this purpose and present a static compiler transformation which implements the algorithm. Our approach is less intrusive than traditional software attestation because the system does not require interruption to compute a cryptographic checksum. It is particularly well suited to real-time systems where consistent timing is more important than speed.

Acknowledgements

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

Dedication

This thesis is dedicated to Ed and Gay Kauffman, who have always supported my choices; to Kaitlin Lindquist, who has had amazing patience for me the last two years; to Aaron Severance, who has kept me sane through the best and worst times; and to the Minister of Propaganda, who has always made sure I take the time to pet him.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADPCM adaptive differential pulse-code modulation

AES Advanced Encryption Standard

ARM Advanced RISC Machine

CFG control flow graph

CPU central processing unit

DFA data flow analysis

GDFAP global data flow analysis problem

GPIO general purpose input/output

ICE indisputable code execution

IR intermediate representation

MCU microcontroller unit

MOP meet over all paths

PC personal computer

PDF probability density function

PRNG pseudorandom number generator

ROP return oriented programming

SAKE Software Attestation for Key Establishment

SCUBA Secure Code Update By Attestation

SETA Side-Effect Transformation Analysis

SHA Secure Hash Algorithm

SWATT SoftWare-based ATTestation

TLB translation lookaside buffer

Chapter 1

Introduction

Power consumption has long been a topic of interest for embedded systems researchers. Their primary goal has traditionally been to lower the amount of power consumed by devices during their operation. Embedded systems are often used in environments where even small changes in power use can affect the viability of a design. Software research in this area focused on reducing switching costs and execution time.

More recently, researchers have found ways to exploit the inherent connection between the tasks a computer performs and power it consumes. The power consumed by a central processing unit (CPU) is directly correlated with the tasks it performs and the data on which it operates, making it possible to analyze the power of a processor to learn about what it is doing [29, 13, 7]. For example, security researchers have developed side-channel analyses called power-monitoring attacks that use power consumption to determine the secret keys used in cryptographic routines.

Existing power analysis techniques have found some success, but are always limited by the features of the programs they seek to analyze. For example, power-monitoring attacks typically depend on the target datum's use in computations against a predictable value. If bit masks or random noise are introduced, these attacks may fail [17].

This thesis introduces a framework for statically transforming programs at compile-time to improve the accuracy and precision of power analysis techniques. Two specific techniques are addressed, and algorithms are proposed for each. The first algorithm modifies a program so that its use with power-based program tracing is more likely to succeed. The second algorithm alters a program so that its power trace can be used as a side-effect for software attestation.

Power-based program tracing uses statistical pattern recognition to reconstruct control flow from power traces. It is a technique that has attracted significant attention in recent years due to its potential applications for debugging deployed systems [13, 30, 7, 27, 26]. Like other power analysis techniques, power-based program tracing depends on characteristics of the program under analysis. If the program contains sections with power signatures that are too similar, the precision of power-based program tracing will decrease. Our framework seeks to modify the program so that the power traces from different program sections are more distinguishable.

Software attestation attempts to create a trusted environment using only software and hardware side-effects like timing, instead of relying on special-purpose trusted computing hardware. Most software attestation schemes involve a challenge response protocol, where trust is established by computing checksums using memory contents and side-effects like translation lookaside buffer (TLB) misses [20, 32, 37]. Despite interest from the embedded systems community, no technique has yet been found without serious drawbacks due to their reliance on timing guarantees and their inability to check the program while it executes [38, 8]. Our algorithm cirumvents these challenges by modifying a program so that its power consumption can be used as a side-effect for software attestation during execution.

1.1 Contributions

This is the first work to introduce static program transformations to optimize or enable power consumption-based analyses. Prior works have proposed alterations to analysis algorithms to improve their accuracy, but no other work has used compiler-based transformations to modify the program under analysis for the same purpose. This work does not address execution side-effects apart from power-consumption, but its ideas could be applied more generally.

We introduce new techniques and explore their possible improvements and applications. We propose the idea of statically optimizing a program to reduce the misclassification rate of power-based program tracing [27]. We also explore possible improvements to the idea, such as alternative optimization algorithms based on Monte Carlo methods. Finally, we introduce a method to enable the use of power consumption as a side-effect for software attestation [19].

The rest of the thesis is organized as follows: Chapter 2 contains background information and terminology. Chapter 3 covers related work. Chapter 4 introduces the static transformation for improving the accuracy of power-based program tracing. Chapter 5 looks at improvements to the algorithm introduced in the previous chapter. Chapter 6 presents the transformation to allow power consumption to be used as a side-effect for software attestation. The thesis concludes in Chapter 7.

Chapter 2

Background

Throughout the paper, we use $\mathbb{R}$ to mean the set of all real numbers and $\mathbb{N}$ to mean the set of all natural numbers.

2.1 Power Consumption

The energy consumed by a program E is the product of the average power (P) consumed by a processor while executing a program and the execution time (T) of the program [10] [23].

$$E = P * T \tag{2.1}$$

The average power (P) consumed by a processor while executing a program is the product of the switching activity factor (K), the load capacitance (C), the clock frequency (f), and the supply voltage squared V_{dd}^2 [9] [10].

$$P = K * C * f * V_{dd}^2 \tag{2.2}$$

Given a modern microprocessor, the load capacitance, clock frequency, and supply voltage can be assumed to be fixed during operation. The switching activity factor is a function of the instructions executed and the data on which those instructions operate.

Power consumption in modern microprocessors, then, is a function of the instructions they execute and the data on which they operate. The predominant determinant of the power trace is the instructions the processor executes, with the data introducing small variations over the average power consumption that a given operation produces [29, 13, 7].

A *power trace* corresponding to the execution of a program or a fragment of a program is defined as the function of power consumption over the execution time interval. In practice, we sample the power consumption at regular intervals and represent the power trace as a sequence of N samples that we treat as an N-dimensional vector [7], where N is proportional to the execution time of the program. For example, if a program runs for 2 ms and we sample at a rate of 1 MHz, then $N = 2{,}000$. Figure 2.1 shows an example of two measured power traces for two different blocks of code.

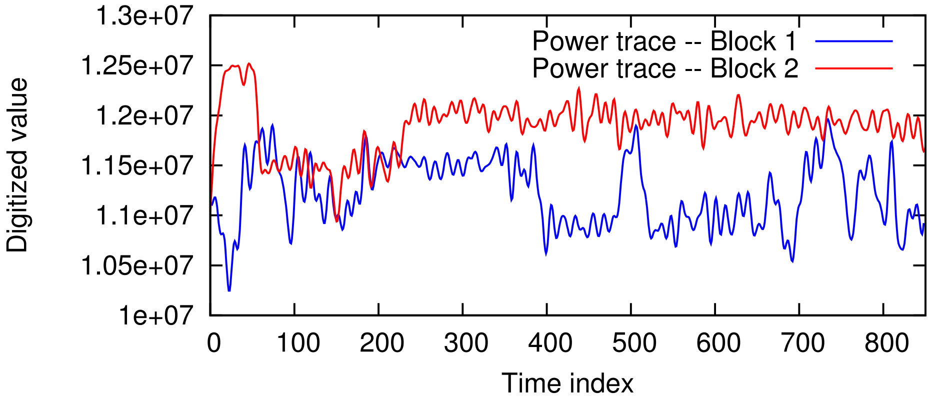


Figure 2.1: Example of power traces for two blocks of code

We define a probability density function (PDF) of the power trace for a given program, where the probabilities are taken over the space of multiple executions with randomly selected input data. This PDF could be given as mean and variance vectors, where each coordinate in the vectors corresponds to a time index at which the sample is taken. In most situations, we work with an approximation of the PDF by experimentally measuring power traces for randomly selected input data.

In a practical implementation, engineers could make use of any available domain knowledge to inform the distribution of input data. This would ensure that the resulting model of the power trace is representative of the values that occur during operation.

2.2 Statistical Pattern Recognition

Our proposed methods relate to ideas from the field of statistical pattern recognition [2]. These techniques aim to maximize the probability over a set of candidate classes, given an observation. They are used in existing power-based tracing methods to find the most likely fragment of code that produced an observed power trace [7, 13, 27]. Our techniques deal with the problem of determining, from power trace observations, whether the execution follows the expected pattern, and thus some of the ideas from this field are applicable.

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. Let $\mathbf{X}$ be a random variable corresponding to a *feature vector* with features extracted from a given element associated with an unknown class $C \in \mathcal{C}$, where $\mathcal{C}$ is the set of q possible classes, $\mathcal{C} = \{C_1, C_2, \cdots, C_q\}$. Class C is determined as the class with highest a posteriori probability given the observation $\mathbf{X}$:

$$C = \arg\max_{C_k \in \mathcal{C}} \{\Pr\{C_k \mid \mathbf{X}\}\} \tag{2.3}$$

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:

$$\overline{\mathbf{X}}_k = \frac{1}{N_k} \sum_{n=1}^{N_k} \mathbf{X}_n \tag{2.4}$$

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 $\mathbf{X}$, with Euclidean distance being the usual metric.

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.

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

2.3 Static Analysis

2.3.1 Control Flow

Control flow is defined as the specific order that control flow statements, like branches and function calls, execute. Determining control flow in advance is undecidable in the general case. We model control flow using a control flow graph (CFG), which is a directed graph (V, E) where the set of vertices (or nodes) V represent sequential sections of code, called *basic blocks*, and the set of edges E represent possible control flow. More precisely, a basic block is a sequence of instructions with only one entry and one exit, so that if the entry is reached then all instructions in the basic block will execute. An edge is a tuple $(\mathit{tail}, \mathit{head}) \in E : \mathit{tail}, \mathit{head} \in V$ where execution of the tail may be followed by execution of the head.

The vertices in a CFG with outgoing edges directed at a given vertex are its *predecessors*. The vertices in a CFG at which the outgoing edges from a given vertex are directed are its *successors*. More precisely, given a vertex $v \in V$, its *predecessors* P are defined as $P \subseteq V : \exists p \in P : \exists (p, v) \in E$ and its *successors* S are defined as $S \subseteq V : \exists s \in S : \exists (v, s) \in E$. A vertices's *siblings* are the successors of its predecessors.

2.3.2 Global Data Flow Analysis Problems

Compile-time program analysis and transformation involves solving a class of problems called global data flow analysis problems (GDFAPs) to determine information found throughout the program. We wish to find the meet over all paths (MOP) solution to a GDFAP, which is the calculation of maximum information relevant to the GDFAP for every statement in a program. By Rice's Theorem, these problems are undecidable in the general case. However, many GDFAPs have been generalized into frameworks which are decidable. If a function space can be shown to be monotone [18] and distributive [21] over a bounded semi-lattice, then the MOP solution can be efficiently obtained by known algorithms.

2.4 Intermediate Representation

Our techniques require the ability to manipulate the power trace of a given program through transformations that preserve the semantics of the program. We utilize the LLVM compiler framework [11] to perform these transformations statically. Our passes are applied after all other optimizations, to ensure that the changes are not undone by performance optimizations that remove or reorder code.

LLVM uses a low-level intermediate representation (IR) for optimization, which is a pseudo-assembly language that maps fairly directly to most machine languages. To use LLVM IR, we need to map it to the known power traces for the target CPU. Figure 2.2 shows an example of such a mapping: four IR instructions are shown above a power trace of their execution. In the figure, the portion of the power trace marked IR$_2$ corresponds to the measurement of the power consumption of the processor while executing `%17 = sext i8 %16 to i32`. The four instructions execute sequentially[2], so their power traces occur without interruption.

IR$_1$:	`%16 = load i8* %b, align 1`
IR$_2$:	`%17 = sext i8 %16 to i32`
IR$_3$:	`%18 = icmp ne i32 %17, 0`
IR$_4$:	`br i1 %18, label %19, label %20`

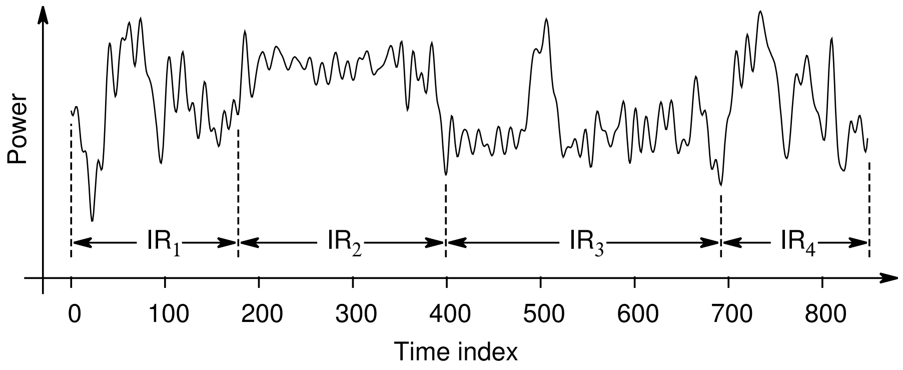


Figure 2.2: Mapping sequential IR instructions to a power trace

[2] As the focus of this work is on application to low-cost, embedded CPUs, out-of-order execution is not considered.

Chapter 3

Related Work

This chapter discusses work from the community related to the topic of the thesis.

3.1 Compiler Transformation of Power Consumption

Much work has been done in the area of modifying the compilation of a program to change its power consumption. 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 [10, 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 [10] 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 [25] or by using a combination of instruction packing, booth multiplier operand reordering [23].

3.2 Power-based Program Tracing

Prior works have introduced the concept of power-based program tracing and shown it to be feasible. Power-based program tracing 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.

Eisenbarth et al. used hidden Markov models to trace assembly-level instructions using power consumption. At this fine granularity, the reported performance was too low for a practical application [13]. Clark et al. presented a malware detector based on side-channel analysis (power consumption) for medical devices [30]. 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.

Msgna et al. used the Viterbi algorithm to find the most probable path through a hidden Markov model of program basic blocks for control flow verification using power consumption [28]. Their technique was effective at recovering control flow on an AT-Mega163 microcontroller unit (MCU) with a 24C256 based smart card. Liu et al. used a revised Viterbi algorithm to recover hidden Markov models using power consumption for program tracing [24]. Their technique was effective at tracing basic blocks on an Intel 8051 MCU and showed an improvement over prior work when targetting assembly instructions.

Moreno et al. presented a novel approach where power consumption is used to reconstruct program traces [7, 26]. 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]. The work in [7] was a feasibility study in that it demonstrated that the approach is viable and potentially effective in practice. The technique was improved upon in [26] with the idea of using the impulse response to identify the execution trace. Our work serves as an extension of the work in [7], making the technique more usable in a practical implementation.

3.3 Software Attestation

Software attestation has received attention in recent literature. Most methods focus on utilizing a combination of cryptographic functions and execution side-effects to produce checksums which are difficult to replicate. Kennell and Jamieson proposed a solution which incorporated side-effect information such as TLB counters into a cryptographic checksum computation to prove system genuinity [20]. Seshadri et al. (SWATT) used sequential pseudo-random memory reads in a cryptographic checksum to prove the contents of memory were unmodified [32]. Shaneck et al. included code obfuscation to hinder analysis of the attestation binary [37]. All of these approaches used challenge-response protocols with time limits on the response to thwart the high presumed cost of side-effect emulation.

Similar work has focused on securing remote embedded systems through guaranteed code execution. Seshadri et al. introduced a system for guaranteed code execution on sensor networks they called indisputable code execution (ICE) [34], based on their previous Pioneer primitive that leveraged Pentium IV Xeon features for guaranteed execution [35]. They went on to use ICE-based approaches for both secure code updates (SCUBA) [34], and key establishment (SAKE) [33] in sensor networks. Like SWATT, ICE based schemes use pseudo-random memory accesses and time limits, but combine these techniques with methods for guaranteeing that execution is not interrupted.

These techniques have been heavily criticized. Multiple published works have shown the method proposed by Kennell and Jamieson [20] to be inherently flawed. Shankar, Chew and Tygar proposed a substitution attack [38] which replaced code on the target system while maintaining the side-effects checked by the genuinity test. Seshadri et al. also demonstrated that such a test will still succeed with 50% probability with a significant amount of memory modification [32]. In [8], Castelluccia et al. presented attacks on SWATT using memory shadowing, and on ICE-based schemes using return oriented programming (ROP) [36].

Armknecht et al. published a method of evaluating software attestation techniques [3], but unfortunately little of it is applicable to this work. They make the assumption that the attestation technique uses a cryptographic challenge-response protocol, which our work does not. They also assume that the technique attempts to verify the contents of memory, which our work does not. We must, therefore, argue our technique without a standard method of evaluation.

Chapter 4

Improving the Classification Rate of Power-based Program Tracing

4.1 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.

In this chapter, we propose a framework for increasing the effectiveness of power-based program tracing techniques. We focus on the techniques that use statistical classifiers to determine the 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 basic 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 IR instructions and estimates the resulting power trace for a given ordering. It then determines the orderings that lead to power traces with highest distances between each other, thus reducing the probability of misclassification.

Our approach assumes the use of the CFG to constrain the classification process and only consider valid sequences of code; given the sequence of basic blocks that executed in the immediate past, the CFG indicates the set of basic blocks that can be currently executing.

The classifier only needs to consider the basic blocks that are feasible as candidates in the classification process. As a consequence, the compiler optimization stage only needs to maximize distinguishability between basic blocks corresponding to *sibling* nodes in the CFG (i.e., nodes with a common predecessor).

Our work includes an experimental evaluation, implemented using LLVM [11] with an Atmel SamD21 Advanced RISC Machine (ARM) MCU [4]. Results from our experiments confirm the soundness of our approach and the potential for its usability in practical scenarios.

The remainder of this chapter proceeds as follows: We describe our proposed approach in Section 4.2, followed by the details of our experiments in sections 4.3 and 4.4. We conclude with a discussion in Section 4.5.

4.2 Proposed Technique

Our technique relies on 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 basic blocks to make the resulting power traces maximally distinguishable.

We also consider that the classifier can take advantage of information about the structure of the program to improve the classification process. The intuition is that if we narrow down the candidate basic 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 basic 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 have the same predecessor in the CFG, we can improve their differentiability and lower the rate of misclassifications for 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 determined from 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.

4.2.1 Profiling the System

Our approach is based on the idea of modifying the order of the instructions. When run after most other optimizations, instruction reordering 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 need more granular information about the power used by instructions.

One consideration when approaching this problem is whether to modify the source language (C) of the program, the IR of the program, or the machine-dependent assembly representation of the program. We quickly ruled out modifying the source language, as it offers too little control over the final machine instructions chosen by the compiler. LLVM offers a powerful IR language which is designed for optimization stages and which represents a machine-independent pseudo-assembly. This has the additional advantage that we can apply the same technique across multiple platforms without modification.

We created a framework for generating machine code corresponding to sequences of a single LLVM IR instruction. Our goal is to modify the IR representation of the program, but the power traces for instructions are 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 1,000 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 machine instructions emitted for different IR instructions.

The possibility for differences between IR instructions with different operand sizes is demonstrated in Table 4.1. The table contains the Thumb assembly instructions generated for the sign-extend (`sext`) IR instruction with different input and output widths. While sign-extending from 16 to 32 or from 8 to 16 bits results in only one assembly instruction, sign-extending from 1 to 8 bits results in 7 instructions including a branch! Because of this variation it was important that we included operand sizes in our model of instruction power traces.

Table 4.1: Comparing emitted Thumb instructions for the sign-extend (sext) IR instruction

Result Width	Operand Width	ASM
32	16	sxth r0, r0
16	8	sxtb r0, r0
8	1	mov r1, r0
		movs r0, #0
		mvns r0, r0
		uxth r1, r1
		cmp r1, #0
		bne .+4
		mov r0, r1

These system profiling programs were then run on the target hardware and their power traces were recorded. To avoid issues with resolution or accuracy in the measurements, each program consisted of 1,000 repetitions of the same sequence of machine instructions corresponding to one IR instruction. We then divided the trace into 1,000 sequences of measurements and took the mean of each index. The resulting vector of real numbers represents the expected power trace of the given IR instruction for the target. A number of examples of these vectors are shown in Figure 4.1.

4.2.2 Optimization Algorithm

This section describes the algorithm to maximimze the distance between the expected power traces of basic blocks and their siblings in the CFG. We model a power trace using

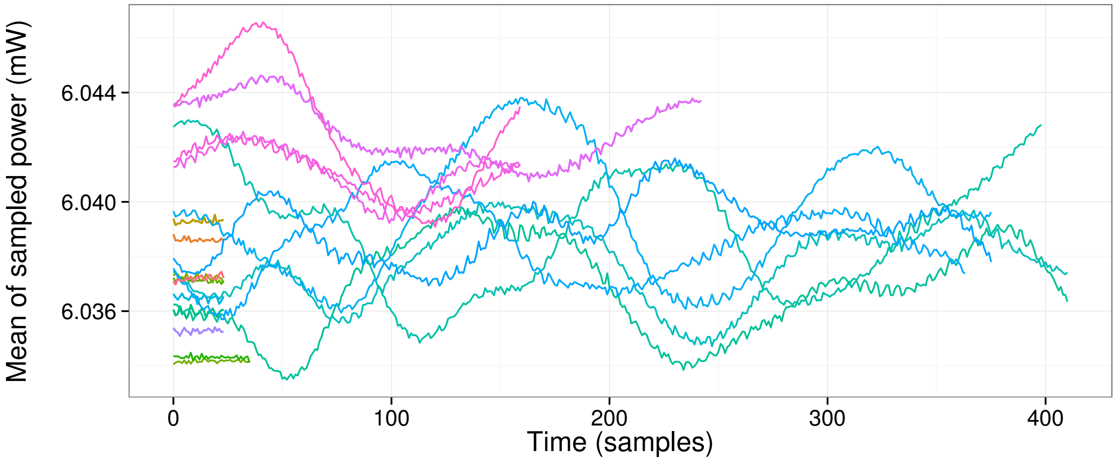


Figure 4.1: Example instruction trace vectors

an N-dimensional vector of real numbers, where N is the number of measurements. The number of dimensions of a vector $\mathbf{v}$ is given by $\dim(\mathbf{v})$.

The distance between two such vectors $\mathbf{a} \in \mathbb{R}^m, \mathbf{b} \in \mathbb{R}^n$ is given by its Euclidean distance:

$$\|\mathbf{a} - \mathbf{b}\| = \left(\sum_{i=0}^{\min(m,n)} (a_i - b_i)^2 \right)^{\frac{1}{2}} \tag{4.1}$$

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.

A vector for a basic block can then be created by concatenating the traces of its instructions. Given two vectors with any positive number of dimensions $\mathbf{a} \in \mathbb{R}^m, \mathbf{b} \in \mathbb{R}^n$, their concatenation is given by:

$$\mathbf{a}^\frown\mathbf{b} = \langle \mathbf{a}_1, \cdots, \mathbf{a}_m, \mathbf{b}_1, \cdots, \mathbf{b}_n \rangle \tag{4.2}$$

We model a *basic block* as a sequence of expected power traces of IR instructions. A basic block β has a length $|\beta| \in \mathbb{N}$ indicating the number of instructions in the sequence. The vector representing the expected power trace of the k-th instruction in the basic block β is given by $\beta[k]$.

We can calculate the differentiability between two basic blocks as the distance between their 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 4.1, which shows how different instructions can have power traces with wildly different lengths.

Given a basic block $\beta : |\beta| = \ell$, we define its expected power trace $\exp(\beta) \in \mathbb{R}^k$ to be:

$$\exp(\beta) = \beta[1]^\frown \cdots ^\frown \beta[\ell] : k = \sum_{\mathbf{i} \in \beta} \dim(\mathbf{i}) \tag{4.3}$$

The distance between a basic block β and the set of its sibling nodes in the CFG S is given by:

$$\text{distance}(\beta, S) = \frac{1}{|S|} \sum_{\sigma \in S} \|\exp(\beta) - \exp(\sigma)\| \tag{4.4}$$

We iteratively improve the total distance of a program until a threshold is reached. Algorithm 1 shows the details of this optimization procedure. The algorithm starts in lines 3 through 6 by iterating over each basic block in the CFG and finding its sibling nodes, using the edge set E to find all the successors of its predecessors. It then begins the main loop (lines 7 through 13), which starts by placing all of the basic blocks into a list sorted by their distance on Line 8. The basic block with the lowest distance to its siblings is then removed (popped) from the list on Line 10, and its instructions are reordered to maximize this distance. Algorithm 2, used to reorder the instructions in a basic block, is described in the next paragraph and discussed at length in Chapter 5. Once the block is reordered, its siblings are removed from the list on Line 12. The block with the next lowest distance in the list is then chosen to repeat the process. Once the list becomes empty, it is repopulated and the loop repeats until the change in total distance of the program drops below the specified threshold, shown as the end of the loop on Line 13.

To optimize the instructions in a basic block, we use Algorithm 2 to exhaustively test every valid permutation of its first instructions. We focus on optimizing the power trace of the beginning of the basic block because the classification algorithm we are targetting is disproportionately affected by these instructions. Chapter 5 discusses alternative reordering algorithms based on Monte Carlo methods, but experimental evaluation was performed using Algorithm 2. The algorithm takes two parameters, shown in the overloaded function on Line 1 which then calls the five parameter recursive version that begins on Line 2. Algorithm 2 is a standard recursive permutation algorithm [39], modified to find the optimal permutation and to skip unnecessary work. Line 4 tests if the ordering is valid,

Algorithm 1 Optimization Algorithm

```
procedure OPTIMIZE($CFG = (V, E)$, $threshold$)
    $SiblingsMap \leftarrow \varnothing$
    for $(predecessor, bb) \in E$ do
        for $(tail, successor) \in E$ do
            if $tail = predecessor \wedge successor \neq bb$ then
                $SiblingsMap[bb] \leftarrow SiblingsMap[bb] \cup \{successor\}$
    do
        $BBList \leftarrow V$, sorted by distance to siblings
        while $|BBList| > 0$ do
            $bb \leftarrow$pop($BBList$)
            $bb \leftarrow$reorder($bb$, $SiblingsMap[bb]$)
            $BBList \leftarrow BBList \setminus SiblingsMap[bb]$
    while $\Delta \sum$ distance($bb \in BBList$, $SiblingsMap[bb]$) $< threshold$
```

by checking the data dependencies of the instructions through the last changed instruction. If the ordering is invalid, then all permutations that contain that invalid sequence are skipped. Line 5 tests for the null condition, which is when the permutation is checked. The number of instructions considered for reordering is limited to a maximum of 13 on Line 3 because of the $\mathcal{O}(n!)$ complexity of the permutation algorithm. The distance is tested and improvements are saved on lines 6 and 7.

Algorithm 2 Exhaustive Permutation Reordering Algorithm

```
 1: procedure REORDER($bb$, $Siblings$) return reorder($bb$, $Siblings$, 1, $bb$)
 2: procedure REORDER($bb$, $Siblings$, $n$, $best$)
 3:     MAXINDEX $\leftarrow$ min($|bb|$, 13)
 4:     if $n = 1 \vee bb$ from index 1 until index $n$ does not violate data dependencies then
 5:         if $n =$MAXINDEX then
 6:             if distance($bb$, $Siblings$) $>$ distance($best$, $Siblings$) then
 7:                 $best \leftarrow bb$
 8:         else
 9:             for $i : i \in \mathbb{N}, n \leqslant i \leqslant$MAXINDEX do
10:                 swap($bb[n]$, $bb[i]$)
11:                 $best \leftarrow$reorder($bb$, $Siblings$, $n + 1$, $best$)
12:                 swap($bb[n]$, $bb[i]$)
13: return $best$
```

4.2.3 Example

As a contrived example see the C code in Figure 4.2 which is part of a program that we would like to classify. It contains two basic blocks which are siblings in the CFG. Figure 4.2 also shows the equivalent LLVM IR generated by the program `clang`.

```
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
```

Figure 4.2: 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 basic blocks are identical. When Algorithm 1 is run on this program, one of these two blocks is sorted to the top of the list to be reordered first, because the distance to its siblings is zero. When different permutations of the basic block are tried by Algorithm 2, 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.

4.3 Experimental Setup

A challenge to the conduct of 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. 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 general purpose input/output (GPIO) pin to signal transitions between basic blocks by toggling the pin. The instrumentation simply places a pin-toggle statement at the beginning of each basic block. We capture power traces through the Line-In input of a standard personal computer (PC) sound card. This idea was introduced in [7]. 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 4.3 shows a simplified diagram of this setup. The current sensing shunt resistors

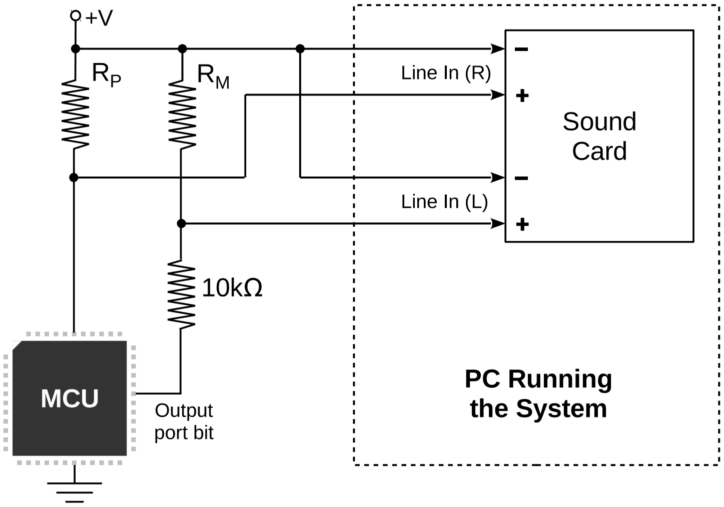


Figure 4.3: Simplified diagram of the power consumption capture system

R_P and R_M (to measure Power and Markers, respectively) are adjusted to produce signals in the order of a few millivolts. This is critical for the power-in line to the MCU, since a resistor producing a large voltage drop would disrupt the internal functionality of the MCU. The 10 kΩ resistor acts as a voltage divider and also keeps the current consumption low. Notice that the current consumption through the markers pin does not affect the

MCU's current consumption since the circuit is closed directly through the +V terminal and not through the power-in connection at the MCU's pin.

Figure 4.4 shows an example of a captured trace. The system automatically detects the positions of the markers by identifying pairs of nearby local maxima and minima, and determines the position of the inflection point between these two extremes. We used the standard numerical approximations of the derivatives to determine the instant at which the second derivative changes sign [41].

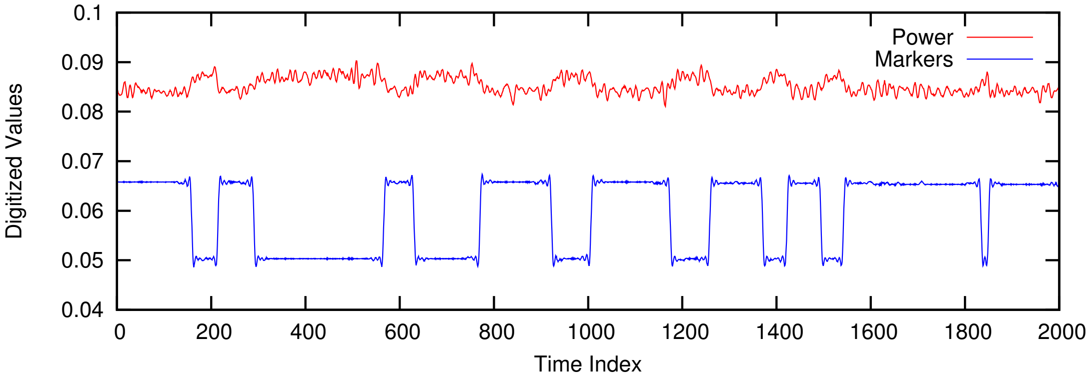


Figure 4.4: Example of captured power trace with markers

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 basic block labels that are output by the print-instrumented version, and thus we are able to label each of the power trace segments with the basic block 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 [12].

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. 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.

4.4 Experimental Results

The focus of our experiments is to demonstrate the difference in the classifier's precision as a consequence of the modifications performed by the compiler through the optimizing stage that reorders 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.

$$P \triangleq \frac{\#CC}{\#CC + \#IC} \tag{4.5}$$

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 with 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 several MiBench functions 1,000 times, obtaining a total number of segments (corresponding to individual executions of basic blocks) 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 basic block into two sets. We used one of the sets to construct the training database — essentially, to obtain the centroids for each class (each basic block) — and the other set to run the classifier and obtain the precision. We used a process similar to bootstrapping [6] 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 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 4.2 summarizes the results, including execution of Advanced Encryption Standard (AES) encryption, the Secure Hash Algorithm (SHA) update function, part of the Security section of MiBench [16], and the adaptive differential pulse-code modulation (ADPCM) algorithm from the Telecommunication section. The ± figures indicate the 95% confidence interval for the given parameter.

Table 4.2: Summary of classifier results

	Precision	Dist. between centroids
AES encrypt		
Unoptimized	56.53% ± 0.26	118753
Optimized	63.76% ± 0.3	259506
SHA update		
Unoptimized	99.0% ± 0.04	1855802
Optimized	98.83% ± 0.05	1839577
ADPCM coder		
Unoptimized	58.11% ± 0.22	130781
Optimized	59.19% ± 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 to minimize the total distance, generating the least distinguishable version of a program. To calculate these distances, we only consider the instructions in each basic block that Algorithm 2 considers during reordering. That is, we only consider up to the first 13 instructions in each basic block.

Table 4.3 shows these computed improvements. 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 4.3 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.

4.5 Discussion and Suggested Work

Perhaps the most interesting observation from our experiments is the effect of the code structure and size on the effectiveness of our technique. The precisions obtained for different functions exhibit large variations. The results, combined with inspection of the various

Table 4.3: 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

fragments of code being considered, suggest that it is mostly the structure and size of the code that impacts 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 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. Additional research could uncover patterns or relationships between characteristics of the source code and the effectiveness of the technique.

Additional use of the CFG for program trace classification 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 a means to increase the performance, the computational efficiency, or both.

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 [29], on devices with binaries created by such compiler. Embedded systems security engineers should be aware of this possibility, even if its likelyhood, in practice, may be remote.

Chapter 5

Power Trace Distance Maximization Using Monte Carlo Permutations

5.1 Introduction

In this chapter we explore the use of Monte Carlo methods to find the optimal permutation of instructions in a basic block to improve the classification rate of power-based program tracing as introduced in Chapter 4. Optimization problems often involve search spaces that cannot be exhaustively explored due to their size. Finding the optimal permutation of a sequence is such a problem, with $n!$ possible solutions where n is the number of items in the sequence.

Monte Carlo methods offer multiple advantages over exhaustive search. First, the asymptotic complexity of Monte Carlo algorithms is usually constant in some parameter. This allows the run time of Monte Carlo methods to be carefully controlled to trade off results for speed. Second, because of the run time is decoupled from the size of the search space, the search space can be arbitrarily large.

We present an exploration of the use of a number of variations on a basic Monte Carlo permutation algorithm. We compare each variation to the others and to the exhaustive search algorithm they serve to replace.

The measurements presented in this chapter were obtained from an implementation written using the LLVM compiler framework [11] version 3.8.0, compiled from source for Linux for the x86-64 architecture. The expected power traces we used were recorded from an Atmel SamD21 [4], and all programs were compiled for this processor.

The rest of the chapter is organized as follows. In Section 5.2 we define notation used throughout the chapter. In Section 5.3 we outline the problem. Section 5.4 describes our solutions and Section 5.5 presents a discussion of the results.

5.2 Preliminary Notation

We model a power trace using an N-dimensional vector of real numbers, where N is the number of measurements. The number of dimensions of a vector $\mathbf{v}$ is given by $\dim(\mathbf{v})$.

The distance between two such vectors $\mathbf{a} \in \mathbb{R}^m, \mathbf{b} \in \mathbb{R}^n$ is given by:

$$\|\mathbf{a} - \mathbf{b}\| = \left(\sum_{i=0}^{\min(m,n)} (a_i - b_i)^2 \right)^{\frac{1}{2}} \tag{5.1}$$

Given two vectors with any positive number of dimensions $\mathbf{a} \in \mathbb{R}^m, \mathbf{b} \in \mathbb{R}^n$, their concatenation is given by:

$$\mathbf{a}^\frown \mathbf{b} = \langle \mathbf{a}_1, \cdots, \mathbf{a}_m, \mathbf{b}_1, \cdots, \mathbf{b}_n \rangle \tag{5.2}$$

We model a *basic block* as a sequence of expected power traces of IR instructions. A basic block β has a length $|\beta| \in \mathbb{N}$ indicating the number of instructions in the sequence. The vector representing the expected power trace of the k-th instruction in the basic block β is given by $\beta[k]$.

Given a basic block $\beta : |\beta| = \ell$, we define its expected power trace $\exp(\beta) \in \mathbb{R}^k$ to be:

$$\exp(\beta) = \beta[1]^\frown \cdots {}^\frown \beta[\ell] : k = \sum_{\mathbf{i} \in \beta} \dim(\mathbf{i}) \tag{5.3}$$

The distance between a basic block β and the set of its sibling nodes in the CFG S is given by:

$$\text{distance}(\beta, S) = \frac{1}{|S|} \sum_{\sigma \in S} \|\exp(\beta) - \exp(\sigma)\| \tag{5.4}$$

An *ordering* is a function $\Omega : \mathbb{N} \to \mathbb{N}$ that maps the indices of a basic block to new indices. We use the shorthand of applying an ordering to a basic block to mean that it is applied to each index of the basic block. Given an ordering $o \in \Omega$ and a basic block $\beta : |\beta| = \ell$,

$$o(\beta) = \beta[o(1)], \cdots, \beta[o(\ell)] \tag{5.5}$$

5.3 Problem Statement

Given a basic block, and the set of its siblings in the CFG, find an ordering of the instructions in the basic block that maximizes the distance between its expected power trace and that of its siblings. More formally, given a basic block β, and the set of its siblings in the CFG S, find

$$\arg\max_{o \in \Omega} \ \text{distance}(o(\beta), S) \tag{5.6}$$

In Chapter 4, we used a static program transformation to maximize the distance between the expected power traces of the sibling basic blocks in a program. Algorithm 1, introduced in that chapter, puts all of the basic blocks in a program into a list, ordered by distance to their sibling basic blocks, so that the first element is the basic block with the lowest distance between its predicted power trace and those of its siblings. For each element in the list, Algorithm 2 is used to test every possible permutation of its IR instructions to find the distance between its predicted power trace and those of its siblings, and the ordering with the highest distance is returned. Algorithm 1 then reorders each basic block in the list, repeating until the increase in expected distance falls below a given threshold.

While Algorithm 1 is nominally effective, Algorithm 2 exhaustively tests every permutation of the instructions. This algorithm has an asymptotic complexity of $\mathcal{O}(n!)$ in the number of IR instructions in the basic block. Despite some optimization to avoid testing invalid permutations, the practical consequence is that no more than 13 IR instructions may be considered in an ordering.

Table 5.1 shows example results from running this exhaustive permutation algorithm on a benchmark. The results are from running Algorithm 1 from Chapter 4 on the `aes.c` program from the MiBench benchmark suite [16]. In the table, the Size column indicates the number of instructions considered for reordering, the Time column indicates the amount of time it took to run in seconds, and the % Improvement column shows the percent improvement of the total distance of all basic blocks in the program from their siblings. Unlike in Section 4.4, the % improvement column includes all instructions in a basic block, not just the instructions considered for reordering by Algorithm 2. For example, the first line shows that when the number of instructions included was 6, the algorithm took 0.851 seconds to run and resulted in a 0.26% improvement in the total distance.

Figure 5.1 plots the data from Table 5.1. Figure 5.1a plots the % improvement of the exhaustive algorithm against the number of instructions considered for reordering, while Figure 5.1b plots the algorithm's time cost against the same factor. Figure 5.1a

Size	Time (s)	% Improvement
6	0.851	0.26
7	1.415	0.33
8	2.850	0.44
9	19.510	0.49
10	85.578	1.04
11	308.859	1.04
12	628.629	1.04
13	7,948.829	1.54

Table 5.1: Results from using the exhaustive permutation algorithm (Algorithm 2)

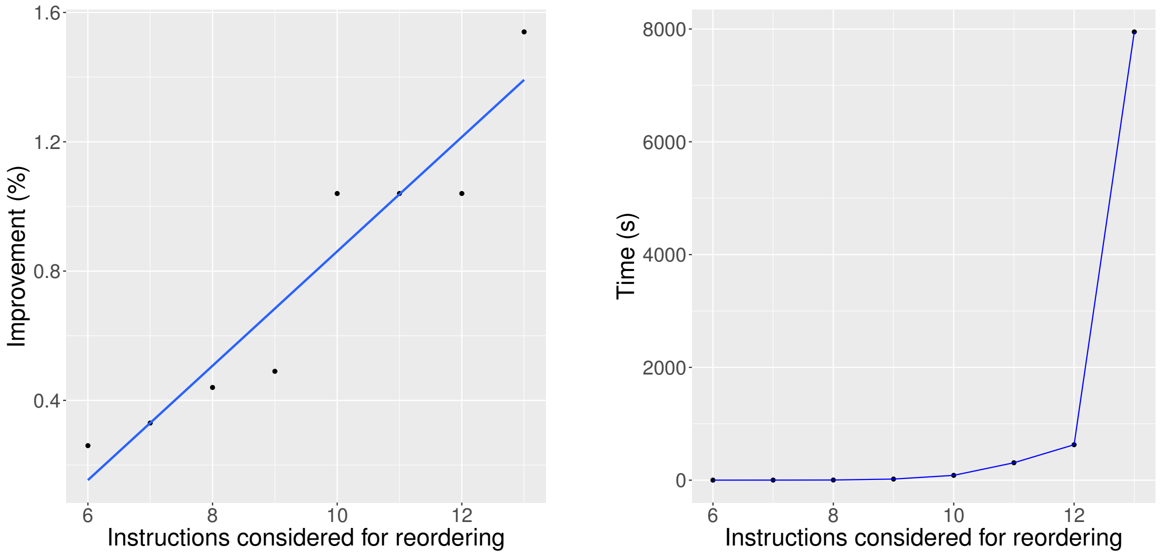
shows a linear relationship between the % improvement and the number of instructions considered, and the line in the chart is a linear approximation of the data. Figure 5.1b shows an exponential relationship between the time cost of the algorithm and the number of instructions considered. These figures demonstrate that the exhaustive algorithm becomes prohibitively time consuming as the number of instructions considered increases while producing only modest improvements in total distance.

5.4 Solution

By replacing the exhaustive implementation with a stochastic permutation algorithm, we can control the runtime complexity of the compiler transformation without compromising its effectiveness. Even better, since a Monte Carlo solution may consider all the instructions in a basic block in its reordering, the upper bound of its solution may be higher than that of the exhaustive algorithm.

5.4.1 Basic Monte Carlo Permutations

Algorithm 3 shows the basic procedure for using Monte Carlo permutations to find an instruction ordering that results in the highest possible distance. In the algorithm, *bb* is the basic block, *Siblings* is the set of siblings in the CFG, and τ is an iteration limit. Line 2 initializes the best known ordering. Line 3 creates a set of natural numbers representing the indices of instructions in *bb*. The algorithm then loops τ times. Line 5 shows two random variables being assigned uniformly distributed natural numbers that represent the indices of instructions. Line 6 modifies the ordering of *bb* so that the instructions at the



(a)

(b)

Figure 5.1: The % improvement and time cost of the exhaustive permutation algorithm plotted against the number of instructions considered for reordering

two indices are swapped in place. Line 7 tests to see if the ordering has invalidated the basic block. If not, Line 8 checks to see if the distance of the new ordering is greater than that of the best known ordering, and Line 9 replaces the best ordering if so. If swapping the instructions in Line 6 has invalidated *bb*, then Line 11 reverts the change. After the iteration limit is reached, Line 12 returns the best ordering found.

Algorithm 3 Basic MC Permutation Algorithm

```
 1: procedure REORDER($bb, Siblings, \tau$)
 2:     $best \leftarrow bb$
 3:     $Indices \leftarrow \{x : x \in \mathbb{N}, 1 \leqslant x \leqslant |bb|\}$
 4:     while $\tau$-- $> 0$ do
 5:         $\xi_1, \xi_2 \leftarrow$ random number uniformly distributed $\in Indices$
 6:         swap($bb[\xi_1], bb[\xi_2]$)
 7:         if $bb$ does not violate data dependencies then
 8:             if distance($bb, Siblings$) $>$ distance($best, Siblings$) then
 9:                 $best \leftarrow bb$
10:         else
11:             swap($bb[\xi_1], bb[\xi_2]$)
12: return $best$
```

Table 5.2 shows the results of running Algorithm 3 on the same benchmark used in Table 5.1. The Time and % Improvement columns in the table are the same as from Table 5.1, but Table 5.2 has an Iteration column, which corresponds to the value of τ in Algorithm 3. Eleven experiments were performed for each row and the values in the Time and % Improvement columns reflect their mean ± standard deviation.

Iterations	Time (s)	% Improvement
1,000	18.5 ± 7.1	4.7 ± 0.75
10,000	121.5 ± 79.4	5.9 ± 0.90
100,000	950.1 ± 603.7	6.7 ± 0.75

Table 5.2: Results from the basic Monte Carlo permutation algorithm

Algorithm 3 is simple, yet effective. Using only 1,000 iterations, the improvement in total distance in the program is more than three times as much as from the exhaustive algorithm, and it completes in less than 30 seconds.

5.4.2 Modifications to the Basic Algorithm

Hybrid between Monte Carlo and exhaustive

A Monte Carlo solution makes sense when the number of instructions in a basic block is large, but when it is small the exhaustive algorithm will find the globally optimal value in a shorter time than running many iterations of a stochastic algorithm. Since optimizing an entire program means reordering both long and short basic blocks, we combined the two algorithms by choosing which one to run based on the length of the basic block we want to reorder.

This *hybrid* solution runs the exhuastive algorithm (Algorithm 2) if the number of instructions in the basic block is six or less. If the number of instructions is higher, then it runs the Monte Carlo algorithm (Algorithm 3).

Table 5.3 shows the results from running the hybrid algorithm on the same `aes.c` benchmark. Just like in Table 5.2, the Iterations column shows the value of τ when running the Monte Carlo algorithm. The Time and % Improvement columns are the same as before. Eleven experiments were performed for each row.

The hybrid solution performs similarly to the basic Monte Carlo algorithm, but with slightly raised average % Improvement and slightly lower average Time. This meets expectations, since it is identical to the basic Monte Carlo algorithm for basic blocks with

Iterations	Time (s)	% Improvement
1,000	16.0 ± 5.7	4.7 ± 1.00
10,000	114.2 ± 72.2	6.4 ± 1.02
100,000	875.7 ± 743.2	6.8 ± 0.85

Table 5.3: Results from the hybrid Monte Carlo / exhaustive algorithm

more than six instructions, but for shorter basic blocks it will find the global optimum in less time.

Only Retain Improvements

We looked for a way to improve the rate at which the Monte Carlo permutations algorithm converged on a good solution. Many of the orderings tested during an execution of the routine will reduce the distance between the basic block and its siblings in the CFG. We modified the algorithm to only retain changes to the ordering of the basic block when the change improved the distance.

Only retaining improvements means that instructions may only be swapped in the ordering that is the current best known solution. Implementation only requires a small modification to Algorithm 3. Algorithm 4 shows the addition. On Line 10, an else clause is added to the distance comparison. If the distance of the new ordering is not higher than the best known ordering, the instructions are swapped back to their prior positions on Line 11.

Algorithm 4 Only Retain Improvements

```
 8: if distance($bb, Siblings$) > distance($best, Siblings$) then
 9:     $best \leftarrow bb$
10: else
11:     swap($bb[\xi_1], bb[\xi_2]$)
```

Table 5.4 shows the results from using the hybrid technique described in Section 5.4.2 with the Monte Carlo permutations algorithm modified to only retain improvements as in Algorithm 4. The results are, again, from running on the `aes.c` program and the Iterations, Time, and % Improvements columns have the same meanings as in previous tables. Eleven experiments were performed for each row.

The modification to only retain improvements performs similarly to the basic Monte Carlo or hybrid algorithm with 1,000 iterations, but worse with 10,000 or 100,000 iterations.

Iterations	Time (s)	% Improvement
1,000	12.2 ± 5.3	4.9 ± 0.78
10,000	35.0 ± 1.1	5.2 ± 0.39
100,000	352.3 ± 14.1	4.7 ± 0.59

Table 5.4: Results from the hybrid algorithm, modified to only retain improvements

The average % improvement after 1,000 came out slightly higher than the average after 100,000 iterations, and the % improvement after 10,000 iterations was even higher. These values are within the standard deviations, but it is likely that more experiments are needed to narrow the results. The modification also resulted in a lower time cost, which can be attributed to Algorithm 1 finishing more quickly.

Non-uniform PDF

The random variables used in Algorithm 3 are chosen with a uniform distribution over the natural numbers representing the indices of the IR instructions that make up the basic block. Choosing these indices uniformly means that no prior knowledge of the instructions or their expected power traces is used. We looked for a way to include some prior knowledge of the expected power traces of the instruction in the distribution of random numbers.

We hypothesized that, since the algorithm only explores a small subset of the search space, it might be helpful to assign higher weights to the likelyhood of choosing instruction pairs which represent a larger change in the expected power consumption of the basic block. Doing so might increase the probability of finding globally optimal solutions which differ greatly from the original ordering.

We use a piecewise constant PDF [15] to assign weights to different indices. The first random variable is chosen with a uniform distribution. Once the first index is known, each index is given a weight according to the distance from the expected power trace of the instruction at that index.

Given a basic block bb and a random variable ξ_1 uniformly distributed over the natural numbers that form the indices of bb, the probability of choosing an index is given by

$$\forall_{i:i\in\mathbb{N},1\leqslant i\leqslant|bb|}\,\mathcal{P}(i)=\frac{\|bb[i]-bb[\xi_1]\|}{\left(\sum_{i\in bb}\|bb[i]-bb[\xi_1]\|\right)} \tag{5.7}$$

Equation 5.7 means that the probability of choosing an index as the second random number is proportional to the distance from its expected power trace to the expected power trace at the index given by the first random number ξ_1. In practice, these probability values are precomputed into a weight matrix and only need to be reordered to correspond to the changes in the instruction ordering.

Table 5.5 shows the results from running the hybrid algorithm from Section 5.4.2, modified with a non-uniform PDF, on the `aes.c` benchmark. The Iterations, Time and % Improvement columns have the same meaning as in previous tables. Eleven tests were run for each row.

Iterations	Time (s)	% Improvement
1,000	22.4 ± 6.9	3.3 ± 0.67
10,000	117.4 ± 58.0	4.8 ± 1.40
100,000	1,386.7 ± 286.3	7.2 ± 0.49

Table 5.5: Results from the hybrid algorithm, modified to use a non-uniform PDF

Modifying the Monte Carlo part of the hybrid algorithm to use this non-uniform PDF causes the results to improve more slowly, but to eventually find a better outcome. The amount of time taken is also increased, both from more iterations in the outer algorithm and from the overhead caused by keeping the weights up-to-date.

5.5 Discussion

The simple Monte Carlo permutations algorithm, described in Section 5.4.1, may obtain better results than the exhaustive algorithm because it is able to consider a larger search space. Although it does not explore the search space completely, its ability to consider every instruction in the basic block for reordering means that it may have a higher upper bound on its result.

Figure 5.2 shows a comparison of the basic algorithm and its modifications from a single run on a single basic block in the `aes.c` benchmark. The x-axis shows the number of iterations in log scale and the y-axis shows the % improvement in distance between the basic block and its siblings in the CFG. Each method is represented by a differently colored line. The figure shows the results from a single run, so it is subject to some variability, but it demonstrates the relative behavior of the different modifications. The basic and hybrid algorithms are, in this instance, the same, since the basic block is longer than six instructions.

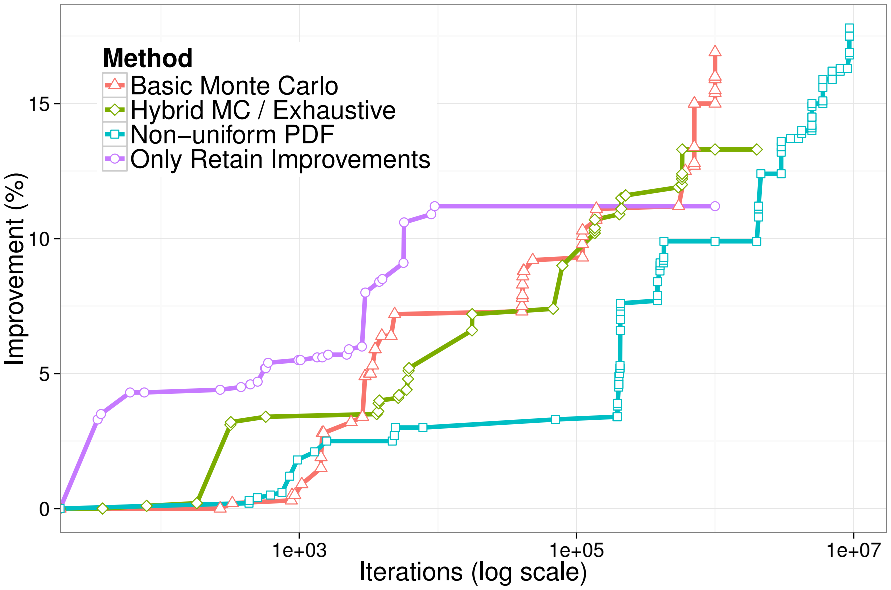


Figure 5.2: The % improvement of the different Monte Carlo algorithms on a single basic block, plotted against the number of iterations required

The modification from Section 5.4.2 to only retain improvements causes a faster increase in the best known ordering, but it does not reach as optimal of a solution as the other methods. The modification more efficiently improves the result, but causes the search to be confined to local optima near the original ordering.

The modification from Section 5.4.2 to use a non-uniform PDF causes a slower increase in the best known ordering, but it eventually finds a better solution than the other methods. The modification causes the search space to be more fully explored, but it does this at the expense of speed.

We have demonstrated the viability of using Monte Carlo methods to optimize the ordering of a basic block for the distance between its expected power trace and those of its siblings in the CFG. We found that even a simple implementation of a Monte Carlo permutations algorithm is faster and finds better solutions than a naïve exhaustive method. We explored some modifications of the basic algorithm and found different performance characteristics that make them useful in different circumstances.

It is our hope that ongoing experimentation will demonstrate the effectiveness of these algorithmic improvements in practice.

Chapter 6

Adapting Power Consumption for Use as a Side Effect in Software Attestation

6.1 Introduction

Security researchers are increasingly exploring software attestation techniques in the embedded domain. Trusted hardware, such as the Trusted Platform Module [22] or *Palladium* Next Generation Security Computing Base [31], are effective but often too expensive for use in embedded devices. The limited resources, simple design, and strict requirements of most embedded systems require a different approach to software security. Software attestation allows an external *verifier* to check that an embedded system, the *prover*, has not been compromised, without requiring specialized hardware [3]. This is accomplished by leveraging the very characteristics of the embedded system that make it difficult to secure. Because of their simple nature, embedded devices exhibit side-effects with low variance. Software attestation uses foreknowledge of these side-effects to check that the system is operating as expected and is uncompromised.

Prior efforts at software attestation have focused on using a variety of side-effects to verify the contents of memory. Typically, the verifier asks the prover to execute a special routine with an input for which the side-effect's behavior is known [20, 32]. During that routine, those side-effects are used in cryptographic functions to generate memory addresses which are accessed. The results of those accesses and the side-effects are used to compute a checksum that is returned to the verifier, which then confirms that the execution time bounds were not exceeded. These methods rely on the physical security of the hardware and the difficulty of precomputing the routine, and are only able to check static memory regions that may be known in advance.

Time bounds are used to ensure that no extra code may be executed, but this requires that the routine interrupts regular operation, that the routine itself be uninterruptible, and that the code be repeated enough times for small variations to be noticeable. Existing frameworks have shown examples of their verification routines taking up to 1.8 seconds [32], and 7.93 seconds [20]. These requirements mean that memory can only be verified at system start up, or on systems where a several second break in operation is acceptable.

In this chapter, we present a novel software attestation technique based on the power consumption of a device. By embedding a known power signature into the software, we enable this side-effect to be used to verify that the system is authentic and uncompromised. Unlike prior works, our method is applicable to real-time embedded systems, as it does not require interrupting the software to perform computations that are part of the attestation process. Instead, we use a compiler assisted transformation to modify the program so that it can be effectively monitored during operation. As part of this work, we prove that the algorithm that analyzes and modifies the program will converge for all input programs and that we can find the MOP solution.

The remainder of the chapter is organized as follows. Section 6.2 states the problem and assumptions we make. We describe our technique in Sections 6.3 and 6.4 and its implications in Section 6.5.

6.2 Problem Statement and Assumptions

This chapter addresses the following problem: given an embedded system with real-time constraints, show with high probability that its running software is genuine and uncompromised, using only execution side-effects and without interrupting normal operation.

To attest to the authenticity of a program we will use the power trace of the prover during the program's execution. Recent literature has demonstrated the feasibility of using power consumption on an embedded device to recover the execution trace or detect deviations with respect to the expected patterns in that trace [13, 7, 27, 30]. Like with any side-effect used for software attestation, this requires building a model of the power consumption of the device when executing different sequences of instructions.

This work assumes that a device will prove its authenticity to its operator. This facilitates enforcing security on the device, and is in contrast to other contexts, such as privacy related technologies. In those scenarios, a device may need to prove its authenticity to a third party, even when the device's operator is an untrusted entity (see for example [43]).

We also assume that attackers do not have physical access to the device, and thus cannot measure the power consumption profile for the purpose of designing malware that mimics it. Our technique is therefore well suited for safety-critical systems such as industrial control systems, medical equipment, and ground support equipment in the aerospace industry.

We do not address the effect of interrupts, task switching, or shared libraries in this work. Ignoring these effects does not greatly reduce the scope of our technique, as hard real-time requirements often preclude their inclusion. In particular, we are not targeting battery powered systems where interrupt-driven operation is commonplace.

6.3 Technique

Our proposed technique uses a compiler assisted static transformation to embed a secret power signature in a given program. The resulting executable produces a power trace of our choice, regardless of the program's original source code and without changing its semantics.

To implement the technique, a system profiling technique is first used to map IR instructions to power traces[1]. For each IR instruction, a program of 1,000 sequential instances of the instruction is created. Each program runs on the target in a loop synchronized with the power capturing device. In each iteration of the loop, new argument values from the random input distribution are used. After capturing a sufficient number of power traces, each trace is divided into 1,000 sequences of measurements. The expected trace for the IR instruction is made up of the mean of all of the sequences at each time index.

For each type of IR instruction, we distinguish between different argument widths, as they typically lead to changes in the generated machine-level instructions. For example, we treat `load i8` as distinct from `load i16`. Other variations in the machine code for a single IR instruction are sufficiently small that it may be safely assumed that the differences between them are handled by the profiling technique.

The release engineer then chooses a secret power trace, $\boldsymbol{\beta}$, and an error threshold ε. More details on selecting these parameters are included in Sections 6.3.1 and 6.4.

During compilation, the static transformation is applied to the program, with $\boldsymbol{\beta}$ and ε used as parameters. Figure 6.1 illustrates the effect of this transformation. An unmodified

[1] A more detailed explanation of system profiling can be found in Section 4.2.1.

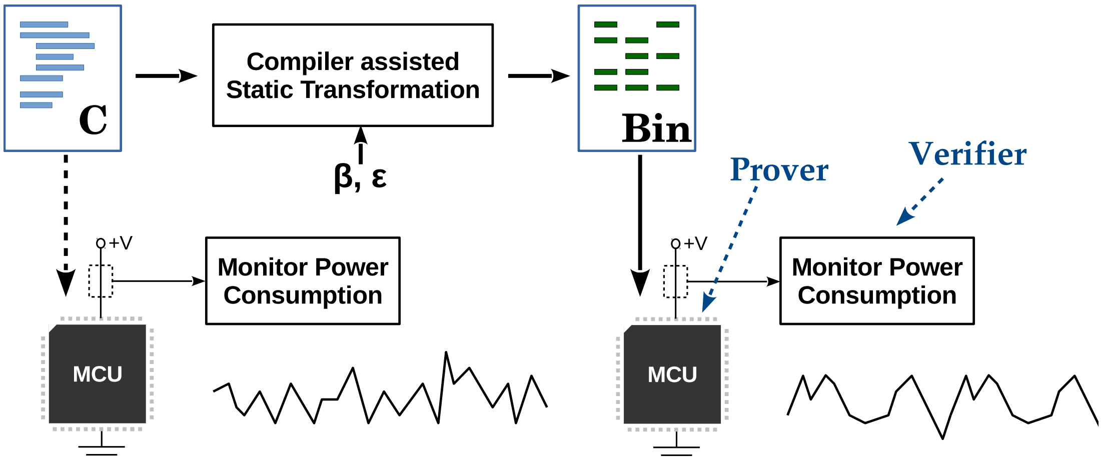


Figure 6.1: Setup phase of attestation using power consumption

program, represented in the figure by a source file, could be compiled and run on the prover MCU but the verifier would see an unpredictable power trace. After modification using our static transformation, with target power trace $\boldsymbol{\beta}$ and error threshold ε, the power trace that the verifier observes is predictable.

During operation, an external device monitors power consumption to detect deviations from the secret power signature embedded in the program. Figure 6.2 sketches this functionality. If the verifier observes a power trace that is within the error threshold ε of the target power trace $\boldsymbol{\beta}$, it does not report an anomaly. If an attacker tampers with the prover to execute unexpected code, the power trace violates the error threshold and the verifier raises an alarm.

The following sections describe the static transformation in detail. Section 6.3.1 establishes the mechanisms by which we map the expected power traces for IR instructions to the target power trace $\boldsymbol{\beta}$. Section 6.3.2 describes the data flow analysis (DFA) that uses those mechanisms to find the possible alignments in the target power trace for each part of the program. Section 6.3.3 explains the selection of the alignments that minimize the overhead, and Section 6.3.4 describes the strategy to fill in the gaps left by those alignments.

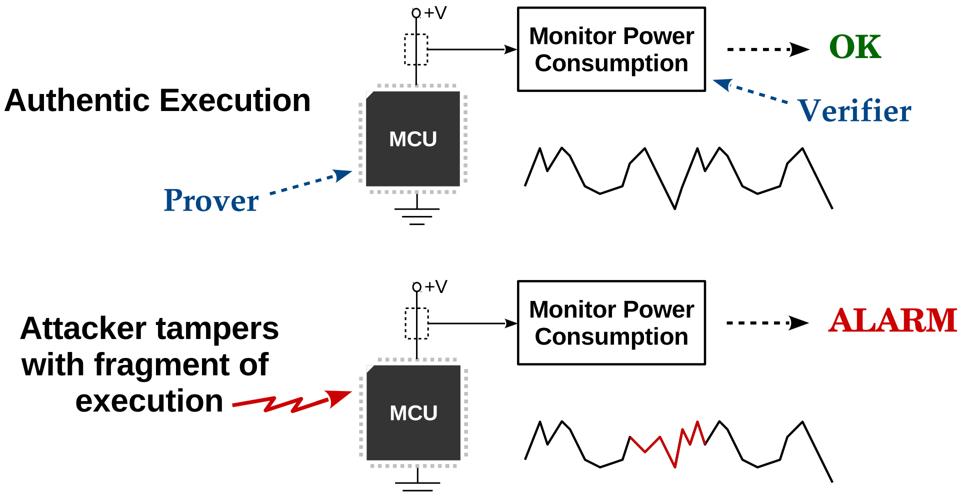

+V
Monitor Power Consumption
OK
Authentic Execution
Verifier
MCU
Prover
+V
Monitor Power Consumption
ALARM
Attacker tampers with fragment of execution
MCU

Figure 6.2: Operation phase of attestation using power consumption

6.3.1 Relating Power Traces

We will modify the program so that its power trace is identical, regardless of the control flow. We choose a desired power trace and apply it to the program using static transformation by a compiler. Since the problem of knowing how long the program will execute is equivalent to the halting problem, which is undecidable, we must choose a finite power trace and then modify the program to repeat it until termination.

The power trace of a program fragment is approximated by its expected power trace. We represent such traces as a finite-dimensional vector of real numbers $\mathbf{b} \in \mathbb{R}^n$, where each dimension in the vector corresponds to a sample of the instantaneous power consumption.

Definition 1 (notational convention) *Given* $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} \overset{\text{def}}{\equiv} \langle x_1, x_2, \cdots, x_n \rangle$; *i.e.,* x_k *denotes the* k*-th coordinate of* $\mathbf{x}$*.*

Definition 2 (notational convention) *Given* $\mathbf{x} \in \mathbb{R}^n$*, then* $\mathbf{x}_{k,m}$ *denotes the* m*-dimensional vector with coordinates given by the coordinates of* $\mathbf{x}$ *starting at index* k *and advancing in circular sequence. Specifically:*

$$\mathbf{x}_{k,m} = \begin{cases} \langle x_k, x_{k+1}, \cdots, x_{k+m-1} \rangle & {\scriptstyle k+m \leqslant n+1} \\ \langle x_k, \cdots, x_n, x_1 \cdots, x_{k+m-n-1} \rangle & {\scriptstyle k+m > n+1,\, m \leqslant n} \\ \langle x_k, \cdots\ \mathbf{x} \text{ repeated as needed } \cdots \rangle & {\scriptstyle m > n} \end{cases}$$

Definition 3 *$\boldsymbol{\beta} \in \mathbb{R}^n$ is a target power trace where each of the n dimensions represents the desired power consumption at the corresponding time index.*

To evaluate the proximity of two power traces, we use the Euclidean distance between the vectors that represent them. More formally, given $\mathbf{p}, \mathbf{q} \in \mathbb{R}^m$ the distance between $\mathbf{p}$ and $\mathbf{q}$, $\|\mathbf{p} - \mathbf{q}\|$, is given by:

$$\|\mathbf{p} - \mathbf{q}\| = \left(\sum_{k=1}^{m} (p_k - q_k)^2 \right)^{\frac{1}{2}} \tag{6.1}$$

This is a commonly used metric in pattern recognition techniques [2], and has been successfully used in existing power-based tracing works [7, 13, 27, 26].

Evaluation of the distance between vectors is only applicable for vectors with the same number of dimensions. The goal of our static transformation, then, is to reduce the distance between the power trace $\mathbf{b} \in \mathbb{R}^m$ that we expect to occur during any execution of the program and the desired power trace $\boldsymbol{\beta} \in \mathbb{R}^n$, repeated and truncated until its number of dimensions is equal to m. For example, if $m = 250$ and $n = 100$, then to find the vector we want to compare to $\mathbf{b}$ we create a new temporary vector $\boldsymbol{\beta}_{1,250} = \langle \beta_1 \cdots \beta_{100}, \beta_1 \cdots \beta_{100}, \beta_1 \cdots \beta_{50} \rangle$. Typically, we need to compare the desired trace to the trace from a portion of the program. For this we can simply use an offset from the beginning of the desired trace. In our previous example, if we only want to compare the trace $\mathbf{b} = \langle b_{70} \cdots b_{120} \rangle$, we would use a temporary vector $\boldsymbol{\beta}_{70,51} = \langle \beta_{70} \cdots \beta_{100}, \beta_1 \cdots \beta_{20} \rangle$.

Definition 4 *$\tau_\varepsilon : \mathbb{R}^m \times \mathbb{N} \to \mathbb{N}$ maps a power trace $\mathbf{b} \in \mathbb{R}^m$ and a starting index $\sigma : 1 \leqslant \sigma \leqslant n$ for $\mathbf{b}$ in $\boldsymbol{\beta} \in \mathbb{R}^n$ to an ending index δ for $\mathbf{b}$ in $\boldsymbol{\beta}$.*

Given an expected power trace $\mathbf{b} \in \mathbb{R}^m$, a desired power trace $\boldsymbol{\beta} \in \mathbb{R}^n$, a starting offset σ, and a distance threshold ε, then:

$$\delta = m + \min \left\{ i \in \mathbb{N}, 1 \leqslant i \leqslant n : \left\| \mathbf{b} - \boldsymbol{\beta}_{\sigma+i,m} \right\| < \varepsilon \right\} \tag{6.2}$$

τ is parameterized by ε, an error threshold, which defines the maximum allowed distance between the expected power trace of the instruction and the corresponding desired power trace in $\boldsymbol{\beta}$. τ finds the closest dimension index to σ in $\boldsymbol{\beta}$ where the expected instruction power trace may start and the distance between the two vectors is below ε, then adds the length of that power trace to find the end index δ.

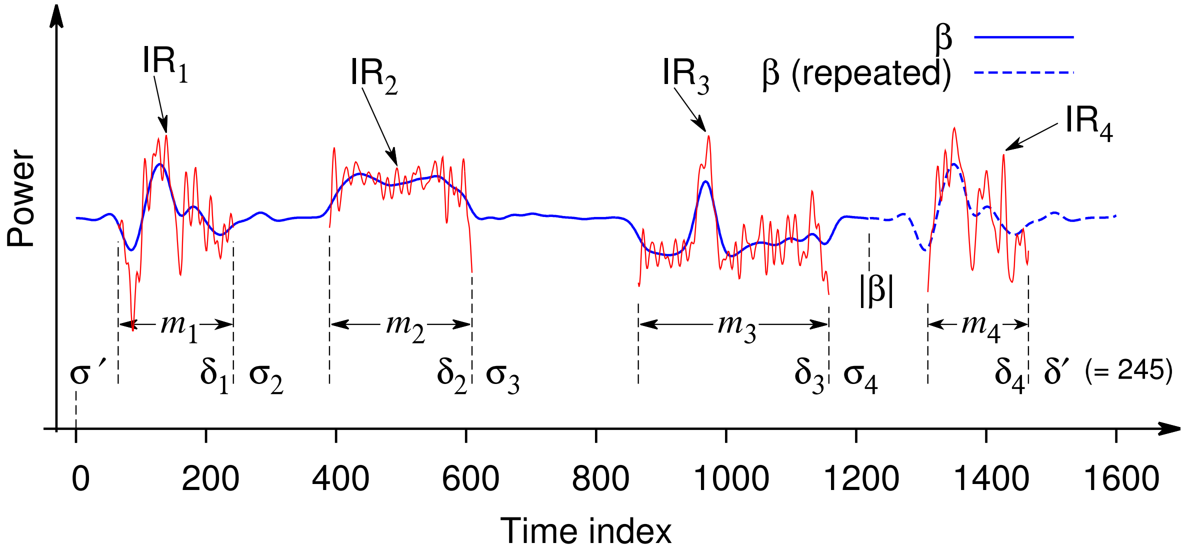


Figure 6.3: Applying τ'_ε to a sequence of power traces

Definition 5 $\tau'_\varepsilon : (\mathbb{R}^m)^i \times \mathbb{N} \to \mathbb{N}$ *applies τ to a sequence of i power traces, finding the end index δ' in $\boldsymbol{\beta}$ of the entire sequence given a start index σ' in $\boldsymbol{\beta}$.*

Given a sequence of i expected power traces $\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_i \in \mathbb{R}^m$, a desired power trace $\boldsymbol{\beta} \in \mathbb{R}^n$, a starting offset σ, and a distance threshold ε, then:

$$\delta' = \tau_\varepsilon(\mathbf{b}_i, \tau_\varepsilon(\cdots \tau_\varepsilon(\mathbf{b}_2, \tau_\varepsilon(\mathbf{b}_1, \sigma')))) \tag{6.3}$$

τ'_ε applies τ_ε to each of the power traces in the sequence, using the δ of each application as the σ of the next. The result δ' is the δ from the final application of τ_ε which corresponds to the end index in $\boldsymbol{\beta}$ of the final trace in the sequence. τ' is parameterized by ε, which is used for every τ in the sequence of applications.

Figure 6.3 shows an example of applying τ'_ε to the expected power traces for the four IR instructions from Figure 2.2. The input $\sigma' = 0$, and the final result $\delta' = 245$. The individual (σ_k, δ_k) pair for each instruction is found by applying τ_ε to its expected power trace, and each m_k signifies the length of the trace. For example, for IR$_2$, σ_2 is 242, and δ_2 is found to be 609 — the first position starting from 242 at which the distance between IR$_2$'s expected power trace and $\boldsymbol{\beta}$ is less than ε. For IR$_3$, σ_3 is therefore 609, the same as δ_2. When τ_ε is applied to IR_4, with $\sigma_4 = 1{,}158$, the position is found by traversing $\boldsymbol{\beta}$ circularly, since $|\boldsymbol{\beta}|$ is only 1,220; in the figure, $\boldsymbol{\beta}$ is repeated and δ_4 is found to be 245.

6.3.2 Data Flow Analysis

We introduce a DFA called Side-Effect Transformation Analysis (SETA) to determine the possible start and end trace indices for each basic block in a program [1]. Once we have found all of the possible start indices we use a single pass through the CFG to select the best one for each basic block, and then a final pass to apply the transformation to each block.

We will map the problem of aligning program instructions with a secret power trace key to a GDFAP to find the MOP solution. We formalize a DFA, which is represented as a tuple $(L, \sqcup, \mathcal{F}, \bot, L_0)$, where L is the set of lattice elements, $\sqcup$ is the join operation, $\mathcal{F}$ is the space of flow functions between CFG vertices, $\bot$ is the bottom element, and L_0 is the initial state of CFG vertices. Functions in $\mathcal{F}$ map the set of lattice elements flowing into the vertex, *IN*, to the set of lattice elements flowing out of the vertex, *OUT*.

Definition 6 *SETA* $= (L, \sqcup, \mathcal{F}, \bot, L_0)$ *is a forward, distributive, monotone DFA framework*

Given a desired power trace $\boldsymbol{\beta} \in \mathbb{R}^n$, a CFG (V, E), and the set of sequences of power traces $B = \{\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_{|V|}\} : \forall \mathbf{b} \in B, \mathbf{b} \in (\mathbb{R}^m)^i$,

$$\begin{aligned}
L &= 2^{[1,n]\times[1,n]} \\
\sqcup &= \cup \\
\mathcal{F} &= \Big\{ f : IN \mapsto \textstyle\bigcup_{(\sigma,\delta)\in IN} \{(\delta, \tau'_\varepsilon(\mathbf{b}, \delta))\}, \forall \mathbf{b} \in B \Big\} \\
\bot &= \varnothing \\
L_0 &= \{(0,0)\}
\end{aligned}$$

The flow functions $\mathcal{F}$ map the end indices δ of elements of the *IN* set to elements in the *OUT* set where the start indices are equal and the end indices are found by applying the τ'_ε function with the given power trace sequence $\mathbf{b}$.

We find $\preceq = \subseteq$ because $\sqcup = \cup$ and $\forall a, b \in L$,

$$\begin{aligned}
b \preceq a &\iff b \sqcup a = b \\
b \prec a &\iff b \sqcup a = b \text{ and } a \neq b
\end{aligned}$$

Lemma 1 *$\forall f \in \mathcal{F}, \ell \in L, x \in f(\ell)$, $\exists \mathbf{b} \in B, (\sigma, \delta) \in \ell : (\delta, \tau'_\varepsilon(\mathbf{b}, \delta)) = x$.*

This follows from $\mathcal{F}$, as the union of such pairs.

Theorem 1 *$(L, \sqcup)$ forms a bounded semi-lattice*

Proof: $(L, \sqcup)$ is a special case called a *power set lattice* [42], since L is a power set and $\sqcup = \cup$. □

Theorem 2 *$\mathcal{F}$ is a monotone function space on L. That is, $\forall f \in \mathcal{F}, \forall a, b \in L$, $a \preceq b \implies f(a) \preceq f(b)$*

Proof: Suppose, by way of contradiction, that

$$\exists f_{\text{bad}} \in \mathcal{F}, \exists a, b \in L : a \preceq b \text{ and } f_{\text{bad}}(a) \npreceq f_{\text{bad}}(b)$$

This means that

$$a \subseteq b, \text{ but } \exists(\sigma', \delta') \in f_{\text{bad}}(a) : (\sigma', \delta') \notin f_{\text{bad}}(b)$$

By Lemma 1, this implies that

$$\exists(\sigma_a, \delta_a) \in a : \forall(\sigma_b, \delta_b) \in b, \delta_a \neq \delta_b$$

which means that

$$\nexists(\sigma_b, \delta_b) \in b : (\sigma_b, \delta_b) = (\sigma_a, \delta_a)$$

but $a \subseteq b$, which is a contradiction. □

Theorem 3 *SETA is a distributive DFA framework. That is, each $f \in \mathcal{F}$ is a homomorphism on L, or $\forall f \in \mathcal{F}, \forall a, b \in L, f(a \sqcup b) = f(a) \sqcup f(b)$*

Proof: Suppose, by way of contradiction, that

$$\exists f \in \mathcal{F}, \exists a, b \in L : f(a \sqcup b) \neq f(a) \sqcup f(b)$$

There are two cases: first, where $f(a \sqcup b) \npreceq f(a) \sqcup f(b)$
and second, where $f(a) \sqcup f(b) \npreceq f(a \sqcup b)$.

Case 1: $(f(a \sqcup b) \npreceq f(a) \sqcup f(b))$

Given the set of all sequences of power traces, $B^* = 2^{(\mathbb{R}^m)^i}$, by Lemma 1,

$$\forall(\sigma'_{ab}, \delta'_{ab}) \in f(a \sqcup b), \exists \mathbf{b} \in B^*, \exists(\sigma_{ab}, \delta_{ab}) \in (a \sqcup b) :$$

$$(\delta_{ab}, \tau'_\varepsilon(\mathbf{b}, \delta_{ab})) = (\sigma'_{ab}, \delta'_{ab})$$

Similarly, for $f(a)$,

$$\forall(\sigma'_a, \delta'_a) \in f(a), \exists(\sigma_a, \delta_a) \in a :$$
$$(\delta_a, \tau'_\varepsilon(\mathbf{b}, \delta_a)) = (\sigma'_a, \delta'_a)$$

and for $f(b)$,

$$\forall(\sigma'_b, \delta'_b) \in f(b), \exists(\sigma_b, \delta_b) \in b :$$
$$(\delta_b, \tau'_\varepsilon(\mathbf{b}, \delta_b)) = (\sigma'_b, \delta'_b)$$

By hypothesis

$$\exists(\sigma'_{ab}, \delta'_{ab}) \in f(a \sqcup b) : (\sigma'_{ab}, \delta'_{ab}) \notin f(a) \sqcup f(b)$$

so it must follow that

$$\forall(\sigma_a, \delta_a) \in a, \exists(\sigma_{ab}, \delta_{ab}) \in (a \sqcup b) : \delta_{ab} \neq \delta_a$$

and

$$\forall(\sigma_b, \delta_b) \in b, \exists(\sigma_{ab}, \delta_{ab}) \in (a \sqcup b) : \delta_{ab} \neq \delta_b$$

So it must be that

$$\exists(\sigma_{ab}, \delta_{ab}) \in (a \sqcup b) : (\sigma_{ab}, \delta_{ab}) \notin a, (\sigma_{ab}, \delta_{ab}) \notin b$$

which is a contradiction.

Case 2: $(f(a) \sqcup f(b) \not\preceq f(a \sqcup b))$

By a similar argument to case 1, we can see that

$$\exists(\sigma_a, \delta_a) \in a, \exists(\sigma_b, \delta_b) \in b :$$
$$(\sigma_a, \delta_a) \notin a \sqcup b \vee (\sigma_b, \delta_b) \notin a \sqcup b$$

which is a contradiction. $\square$

6.3.3 Index Selection Pass

The index selection pass identifies the minimum overhead necessary to align each section of the program with the target power trace. When the SETA pass is complete, a single pass through the CFG chooses the start index in $\boldsymbol{\beta}$ of each basic block. The CFG has already been annotated with all of the options for the start and end indices. It is enough to perform a single pass to select the best choice. Each option for a start index corresponds to the possible end of a predecessor in the CFG. We locally minimize the injected instruction overhead by choosing the start index that results in the least number of samples between each basic block and its predecessors, plus the number that must be added within the basic block itself.

Given the set of pairs $\mathcal{S} \subseteq L$ of start and end indices obtained from applying SETA, for each basic block, we find the optimal start index $\hat{\sigma}$ as follows, where $n = |\boldsymbol{\beta}|$:

$$\hat{\sigma} = \operatorname*{arg\,min}_{(\sigma,\delta)\in\mathcal{S}} \ (\delta - \sigma) \ + \sum_{(\sigma_b,\delta_b)\in\mathcal{S}} (\sigma - \sigma_b + n) \bmod n \tag{6.4}$$

Figure 6.4 illustrates this with an example where we find $\hat{\sigma}$ for basic block C. Its $\mathcal{S}$ set contains two elements, both of which originate from the $\mathcal{S}$ sets of its predecessors. The cost for $(\sigma, \delta) = (3, 6)$ is $10 = (6-3)+((3-5+9) \bmod 9)$, while the cost for $(\sigma, \delta) = (5, 9)$ is $6 = (9-5) + ((5-3+9) \bmod 9)$. The first part of the calculation $(\delta - \sigma)$ counts the execution time of C if the (σ, δ) value is chosen. The second part (the sum term) counts the execution time of instructions that will be injected between C and its predecessors where their end indices are not aligned with the start of C. This is further explained in Section 6.3.4.

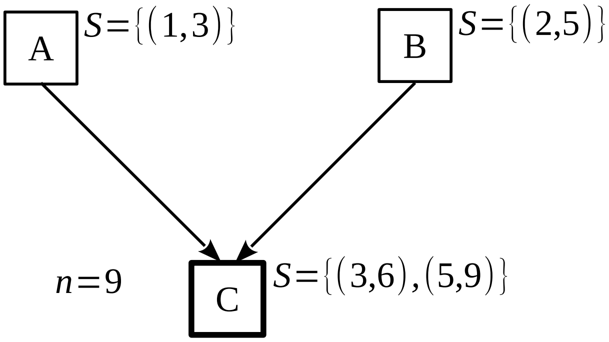


Figure 6.4: Selecting start and end indices

If the probability of execution for the predecessors of the basic block are known, we can include them as weights in the equation to minimize the average overhead:

$$\hat{\sigma} = \operatorname*{arg\,min}_{(\sigma,\delta)\in\mathcal{S}} \ P_{(\delta,\sigma)} \left({\scriptstyle (\delta - \sigma)} \ + \sum {\scriptstyle P_{(\sigma_b,\delta_b)}((\sigma - \sigma_b + n) \bmod n)} \right) \tag{6.5}$$

where $n = |\boldsymbol{\beta}|$, $P_{(\sigma,\delta)}$ is the probability of execution of the predecessor(s) corresponding to the pair $(\sigma, \delta) \in \mathcal{S}$, and the sum is taken over $(\sigma_b, \delta_b) \in \mathcal{S}$.

Each $(\sigma, \delta) \in \mathcal{S}$ may correspond to more than one predecessor, so $P_{(\sigma,\delta)}$ may include the probability of execution of more than one basic block. Conversely, more than one (σ, δ) may correspond to the same predecessor, so $\sum_{(\sigma,\delta)\in\mathcal{S}} P_{(\sigma,\delta)} \geqslant 1$. This is not a problem because we are minimizing a cost and only need to weight each component of that cost by its probability. When more than one (σ, δ) correspond to the same predecessor, their contributions to the summation are added redundantly; however, they are added redundantly for all the choices of (σ, δ), so they cancel out during comparison.

6.3.4 Instruction Injection Pass

After the start and corresponding end indices have been chosen for each basic block, the final step is to inject the instructions. The τ'_ε function for each basic block is applied with σ given by the selected start index from the previous pass. We say that a *gap* occurs when there is a difference between the end index of one power trace and the beginning of the next. Where gaps occur in the basic block, we create and insert instructions which most closely match the trace in that gap.

Definition 7 *A gap, γ, is the difference between the start index in $\boldsymbol{\beta}$ of one power trace, and the end index in $\boldsymbol{\beta}$ of the power trace that precedes it.*

Formally, given a power trace $\mathbf{p} \in \mathbb{R}^m$, a desired power trace $\boldsymbol{\beta} \in \mathbb{R}^n$, and a starting offset σ, we define:

$$\gamma \triangleq (\tau_\varepsilon(\mathbf{p}, \sigma) - m) - \sigma \tag{6.6}$$

We must also take into account gaps between basic blocks. When we choose a start index that does not match the end index of some predecessors, we must add *interstitial* basic blocks. An interstitial basic block contains only injected instructions and is inserted between the terminator instruction of the predecessor and the start instruction that it originally pointed to. Interstitial basic blocks will always use the same terminator instruction as the predecessor that requires no interstitial basic block.

Figure 6.5 shows an example of adding interstitial basic blocks. In Figure 6.5a, the target basic block C has a start index (σ) that corresponds to the end index (δ) of its predecessor B, so there is a gap ($\gamma = 2$) between A and C. In Figure 6.5b, an interstitial basic block A_{int} has been added to fill the gap.

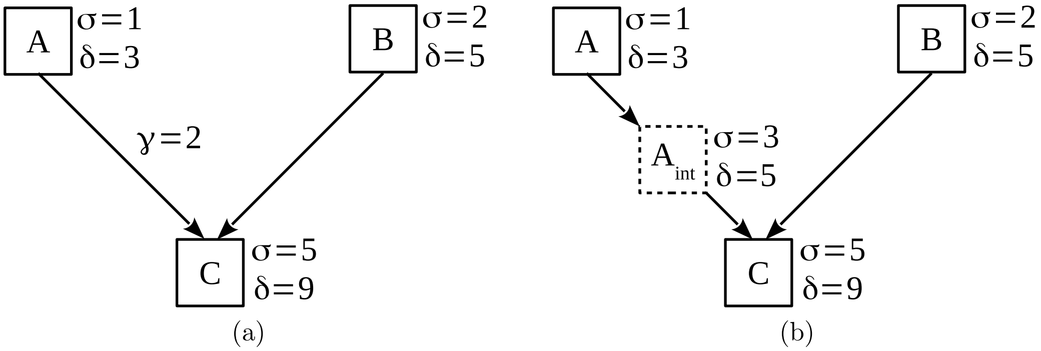


Figure 6.5: Adding interstitial basic blocks

6.4 Implementation

Implementing our technique involves initial training on the prover's hardware, the selection of a secret key $\boldsymbol{\beta}$ and error threshold ε, compilation, and installation of a verifier to monitor power consumption. This corresponds to the process illustrated in Figure 6.2. A more detailed description of the implementation steps follows.

1. Profile the device by capturing power traces for each IR instruction, as discussed in sections 6.3 and 4.2.1. This involves generating and executing binaries that correspond to each IR instruction, and measuring their power traces. This need only be done once per processor model.
2. Define a target power consumption profile $\boldsymbol{\beta}$, as described in Definition 3, which serves as a secret key. A separate $\boldsymbol{\beta}$ should be chosen for each device and deployment of the software.
3. Choose an error threshold ε, as per Definition 4. This threshold should be as low as possible to maximize the efficacy of the technique, and high enough to ensure that the static transformation procedure succeeds. Roughly speaking, ε should be in the same order as the largest values of the variance in the training profile.
4. Compile the program, applying our technique to embed the target power trace and produce an instrumented binary. We created an implementation using the LLVM compiler framework, and others can be easily created using the description in this paper.

5. Install a verifier system to continuously measure power consumption of the prover and compare against $\boldsymbol{\beta}$. This monitoring system should use pattern recognition techniques, described in Section 2.2, to detect deviations in the device's expected power consumption over time. This is beyond the scope of this paper, but there is ample evidence that this is a feasible task [7, 13, 30, 27].

6.4.1 Barriers to Evaluation

Evaluation of our technique was not possible at the time of writing. We are able to transform the power consumption of a real program compiled for the Atmel SamD21 ARM MCU and observe the altered power trace. However, some work remains before a quantitative evaluation is possible.

Although a program's power consumption can be altered using our static transformation, we have not developed the verifier system or the machine learning algorithms necessary to detect deviation from the target power consumption profile $\boldsymbol{\beta}$. Other works have proposed power-based monitoring and debugging systems [7, 13, 30, 27], so there is good reason to believe that developing a verifier is possible.

Improved device profiling techniques are also needed to improve expected power trace accuracy. We utilized profiling techniques developed for power-based program tracing [27], but the precision of that technique suffers from the difficulty of capturing the power traces of some instructions. For example, to profile the `ret` IR instruction, we needed to construct a recursive program and begin capturing the power trace when the null condition was reached. Ideas for possible improvements to device profiling are detailed in Section 7.1.

6.5 Discussion

Like all software attestation schemes, our approach offers security assurances without the need for expensive trusted hardware. However, our method verifies running software, instead of trying to verify the contents of memory. Our technique is also less intrusive, making it suitable for use in systems with real-time requirements.

6.5.1 Attack Model

We incorporate a physically independent verifier, which monitors an externally visible side-effect of the prover's operation. This limits attacks to those that affect the prover without disrupting that side-effect. Such attacks are infeasible without access to the target power trace $\boldsymbol{\beta}$ or the instrumented binary. This is an advantage over existing approaches, where the side-effects are monitored on the prover and can be forged by an attacker.

Our idea, particularly if combined with device-specific target traces, follows the rationale behind Kerckhoffs principle [5]. The embedded power trace is analogous to a secret cryptographic key on which the security of a device relies, as opposed to relying on the secrecy of the design and implementation details of the device. By assumption, the attacker has no physical access to the device, so cannot measure the power consumption. Accidental leakage of a target power trace only compromises one device, and can be remedied by recompiling the source code with a new target power trace. Even if an attacker has access to the source code and an exact copy of the hardware on which it runs, they are still unable to determine the power consumption profile of any specific device they want to attack.

6.5.2 Remaining Work

One concern is whether real-time requirements will be met after code injection. In this work, we emphasized the importance of consistent timing over speed, but the amount of extra code injected depends directly on the threshold ε and the choice of $\boldsymbol{\beta}$. In a practical implementation the system should assist users with verifying these timing requirements after instrumentation of the binary.

The choice of ε and $\boldsymbol{\beta}$ also determines whether the algorithm succeeds: an ε too low, or a $\boldsymbol{\beta}$ with no segments close to the IR power traces, make it impossible to find matching positions in $\boldsymbol{\beta}$ for the instructions. An actual implementation could assist the user in determining these parameters.

The design of a monitoring system to work with our technique should be simplified by the fact that the target processor (the prover) exhibits a fixed and repetitive power trace. This allows for simple and efficient classification techniques such as that in [30]. The works in [7, 13, 28, 24, 27, 26] deal with a more complex problem where the monitoring system aims to recover the trace, whereas [30] uses a simple and efficient classification algorithm. In that work, this simple algorithm is only justified by the assumption that the device's task has low variance and is highly repetitive. When applying our technique, such limitations are not present, since the device can run any arbitrary source code. Yet, simple classification algorithms similar to that in [30] are applicable, since the power trace will have a fixed and repetitive pattern after static transformation of the program.

Chapter 7

Conclusion

This thesis presents compiler-based static transformations for modifying the power consumption of programs to improve power-based program tracing and to enable software attestation with power consumption as a side-effect. Different approaches are explored, including methods to rearrange and inject IR instructions, and Monte Carlo approaches to permutation optimization.

These techniques are well suited to real-time embedded systems. Real-time embedded systems emphasize the need for timing consistency, and often avoid the use of features like interrupts or hardware security modules. Our methods use external monitors to record and analyze power consumption, avoiding any interference with the operation of the machine under examination. Because these techniques statically modify a program using compiler transformations, their application only requires recompiling the application with our tools.

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 orderings that lead to power traces that are more distinguishable. Experimental results confirmed that our approach was effective and had practical applicability.

We demonstrated the viability of using Monte Carlo methods to optimize the instruction ordering of a basic block for the distance between its expected power trace and those of its siblings in the CFG. We found that even a simple implementation of a Monte Carlo permutations algorithm is faster and finds better solutions than a naïve exhaustive method.

We proposed a technique to verify the authenticity of executing software using power consumption. This was made possible through a compiler optimization stage designed to statically modify a program so that its power trace was known in advance.

7.1 Future Work

The techniques presented in this thesis open new areas for future research. One area for future work is the process for learning expected power traces. Another is the algorithm for optimizing the instruction ordering, or injected instructions, in a basic block. Yet another is the development of tools to make power-based software attestation practical.

All of the techniques in this work rely on knowledge of the expected power trace for each IR instruction for a given CPU. The training process presented in this work involves constructing synthetic programs that repeat the considered instruction many times, then recording the power trace while it running. This can work well, but is challenging to apply to IR instructions such as `ret` (return), which change the program counter. To overcome this challenge, we are developing other training methods. One idea with promise is to generate random programs and use their traces to form systems of equations which can be solved for individual instructions.

As illustrated in Chapter 5, more work can be done to improve the algorithm for reordering instructions in a basic block to maximize the differentiability of its power trace. We proposed and tested a number of alternatives to the simplest, exhaustive permutations algorithm based on Monte Carlo methods, but the results suggest that further improvements are possible. We are exploring ideas such as relaxing the weights in the non-uniform PDF over time, and using simulated annealing.

This work proposes a technique to modify a program so that its power trace can be used as a side-effect for software attestation, but we have not solved every barrier to its practical application. The secret program trace ($\boldsymbol{\beta}$) and error threshold (ε) that are inputs to our algorithm must be chosen carefully, or the program transformation may fail. We are developing a system to assist users with the choice of these values so that they produce a feasible solution, and so that random seeds may be introduced for better security characteristics.

7.2 Final Thoughts

The work presented in this thesis shows the promise of using static transformations for enabling or improving side-effect based analyses. The techniques we propose modify power consumption for power-based program tracing and software attestation, but the same or similar methods could be used to modify other side-effects for similar purposes. We hope that this work leads to further exploration of the possible uses of execution side-effects that can be achieved with the aid of static optimization.

References

[1] Aho, Alfred V and Ullman, Jeffrey D. *The theory of parsing, translation, and compiling.* Prentice-Hall, Inc., 1972.

[2] Andrew R. Webb and Keith D. Copsey. *Statistical Pattern Recognition.* Wiley, third edition, 2011.

[3] Armknecht, Frederik and Sadeghi, Ahmad-Reza and Schulz, Steffen and Wachsmann, Christian. A security framework for the analysis and design of software attestation. In *ACM CCS*, 2013.

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

[5] Auguste Kerckhoffs. La cryptographie militaire. In *Journal des sciences militaires*, volume IX, pages 5–38. 1883.

[6] Canty, Angelo J. Resampling methods in R: the boot package. *R News*, 2(3):2–7, 2002.

[7] Carlos Moreno and Sebastian Fischmeister and M. Anwar Hasan. Non-intrusive program tracing and debugging of deployed embedded systems through side-channel analysis. *LCTES'13*, pages 77–88, 2013.

[8] Castelluccia, Claude and Francillon, Aurélien and Perito, Daniele and Soriente, Claudio. On the difficulty of software-based attestation of embedded devices. In *ACM CCS*, 2009.

[9] Chabini, N. and Wolf, M.C. Reordering the assembly instructions in basic blocks to reduce switching activities on the instruction bus. *Computers Digital Techniques, IET*, 5(5):386–392, September 2011.

[10] Chingren Lee and Jenq Kuen Lee and TingTing Hwang. Compiler optimization on instruction scheduling for low power. In *International Symposium on System Synthesis*, pages 55–60, 2000.

[11] Chris Lattner and the LLVM Developer Group. The LLVM compiler infrastructure – online documentation. `http://llvm.org`.

[12] Donald E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, Third edition, 1998.

[13] Eisenbarth, Thomas and Paar, Christof and Weghenkel, Björn. Building a side channel based disassembler. pages 78–99. Springer Berlin Heidelberg, 2010.

[14] Charles N. Fischer, Ron K. Cytron, and Richard J. LeBlanc Jr. *Crafting a Compiler*. Addison-Wesley, 2009.

[15] Green, Peter J. Reversible jump markov chain monte carlo computation and bayesian model determination. *Biometrika*, 82(4):711–732, 1995.

[16] Guthaus, M. R. and Ringenberg, J. S. and Ernst, D. and Austin, T. M. and Mudge, T. and Brown, R. B. MiBench: A free, commercially representative embedded benchmark suite. IEEE Computer Society, 2001.

[17] Joye, Marc and Paillier, Pascal and Schoenmakers, Berry. On second-order differential power analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 293–308. Springer, 2005.

[18] Kam, John B and Ullman, Jeffrey D. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[19] Kauffman, Sean and Moreno, Carlos and Fischmeister, Sebastian. Static transformation of power consumption for software attestation. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2016 IEEE 22nd International Conference on*, pages 188–194. IEEE, 2016.

[20] Kennell, Rick and Jamieson, Leah H. Establishing the genuinity of remote computer systems. In *USENIX Security Symposium*, 2003.

[21] Kildall, Gary A. A unified approach to global program optimization. In *Symposium on Principles of Programming Languages*, 1973.

[22] Kinney, Steven L. *Trusted Platform Module Basics: Using TPM in Embedded Systems (Embedded Technology)*. Newnes, 2006.

[23] Lee, M.T.-C. and Tiwari, V. and Malik, S. and Fujita, M. Power analysis and minimization techniques for embedded dsp software. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1997.

[24] Liu, Yannan and Wei, Lingxiao and Zhou, Zhe and Zhang, Kehuan and Xu, Wenyuan and Xu, Qiang. On code execution tracking via power side-channel. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1019–1031. ACM, 2016.

[25] Ma, Yung-Cheng and Liu, Tse-An and Chao, Wen-Shih. Energy-aware compiler optimization for VLIW-DSP cores. In *Advances in Intelligent Systems and Applications - Volume 2*, volume 21. Springer Berlin Heidelberg, 2013.

[26] Moreno, Carlos and Fischmeister, Sebastian. Non-intrusive runtime monitoring through power consumption: A signals and system analysis approach to reconstruct the trace. In *International Conference on Runtime Verification*, pages 268–284. Springer, 2016.

[27] Moreno, Carlos and Kauffman, Sean and Fischmeister, Sebastian. Efficient program tracing and monitoring through power consumption – with a little help from the compiler. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1556–1561. IEEE, 2016.

[28] Msgna, Mehari and Markantonakis, Konstantinos and Mayes, Keith. The B-side of side channel leakage: control flow security in embedded systems. In *International Conference on Security and Privacy in Communication Systems*, pages 288–304. Springer, 2013.

[29] Paul Kocher and Joshua Jaffe and Benjamin Jun. Differential power analysis. *Advances in Cryptology – CRYPTO' 99*, pages 388–397, 1999.

[30] 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.

[31] Sadeghi, Ahmad-Reza and Stüble, Christian. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *Workshop on New Security Paradigms*. ACM, 2004.

[32] Seshadri, A. and Perrig, A. and van Doorn, L. and Khosla, P. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.

[33] Seshadri, Arvind and Luk, Mark and Perrig, Adrian. SAKE: Software attestation for key establishment in sensor networks. In *International Conference on Distributed Computing in Sensor Systems*, pages 372–385. Springer, 2008.

[34] Seshadri, Arvind and Luk, Mark and Perrig, Adrian and van Doorn, Leendert and Khosla, Pradeep. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94. ACM, 2006.

[35] Seshadri, Arvind and Luk, Mark and Shi, Elaine and Perrig, Adrian and van Doorn, Leendert and Khosla, Pradeep. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 1–16. ACM, 2005.

[36] Shacham, Hovav. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM CCS*, 2007.

[37] Shaneck, Mark and Mahadevan, Karthikeyan and Kher, Vishal and Kim, Yongdae. Remote software-based attestation for wireless sensors. In *European Workshop on Security in Ad-hoc and Sensor Networks*, pages 27–41. Springer, 2005.

[38] Shankar, Umesh and Chew, Monica and Tygar, J. D. Side effects are not sufficient to authenticate software. In *USENIX Security Symposium*, 2004.

[39] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Cliffor Stein. *Introduction to Algorithms*. The MIT Press, Third edition, 2009.

[40] Varun Chandola and Arindam Banerjee and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 2009.

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

[42] Warner, Seth. *Modern algebra*. Courier Corporation, 1990.

[43] Williams, Peter and Sion, Radu. Usable PIR. In *NDSS*, 2008.