Static Transformation of Power Consumption for Software Attestation

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

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

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

***Abstract*—Software attestation seeks to verify the authenticity of a system without the aid of trusted hardware, and has important applications in the field of security. Such attestation schemes are of particular interest in the embedded domain, where simplicity and limited resources constrain more complex security solutions. At the same time, these properties enable attestation approaches that rely on predictable side-effects. Most software attestation schemes aim to verify the integrity of memory using a combination of cryptographic schemes, internal side-effects like TLB misses, and known timing constraints. However, little attention has been paid to leveraging non-traditional side-effects, in particular, externally observable side-effects such as power consumption.**

In this paper we introduce a method for software attestation using power consumption as the side-effect. We show how to circumvent the undecidable nature of program execution for this purpose and present a static compiler transformation which implements the technique. 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.

I. Introduction

Security researchers are increasingly exploring software attestation techniques in the embedded domain. Trusted hardware, such as the Trusted Platform Module [1] or *Palladium* Next Generation Security Computing Base [2], 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 behavior is known [4], [5]. During that routine, those behaviors are used in cryptographic functions to generate memory addresses which are accessed. The results of those accesses and the side-effect behaviors 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 interrupt 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 $1.8\,\mathrm{s}$ [5], and $7.93\,\mathrm{s}$ [4]. 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 paper, 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 meet over all paths (MOP) solution.

The remainder of the paper is organized as follows. We first present background and related work in sections II and III. Section IV states the problem and assumptions we make. We describe our technique in Sections V and VI and its implications in Section VII. We conclude in Section VIII.

II. Related Work

Software attestation has received some 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 [4], while Seshadri et al. (SWATT) used sequential pseudo-random memory reads in a cryptographic checksum to prove the contents of memory were unmodified [5]. Both approaches used challenge-response protocols with time limits on the response to thwart the high presumed cost of side-effect emulation.

These techniques have been heavily criticized. Multiple

published works have shown the method proposed by Kennell and Jamieson [4] to be inherently flawed. Shankar, Chew and Tygar proposed a substitution attack [6] 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 [5]. In [7], Castelluccia et al. presented attacks on SWATT using memory shadowing, and on ICE-based schemes using return oriented programming (ROP) [8].

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.

Recent works have focused on the problem of correlating power consumption with execution path. Eisenbarth et al. [9] proposed a disassembler that used power consumption to reconstruct the executed code, while Moreno et al. [10], [11] proposed a technique that determines executed blocks of source code from power consumption measurements. Clark et al. proposed a technique that uses power consumption to perform malware detection in embedded medical devices [12]. Although effective in the proposed context, their technique is limited by its reliance on the device operation being simple and highly repetitive, and by requiring both nominal and off-nominal labeled training samples.

III. Background

In this section, we review background information regarding power based program tracing and static analysis.

A. Power Consumption

Power consumption in modern microprocessors is a function of the instructions they execute and the data on which they operate [13], [9], [10]. 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.

For a given processor with given operating conditions, a power trace is determined by the sequence of instructions it executes and the data on which those instructions operate. The predominant effect comes from the instructions, with the data introducing small variations over the average power consumption that a given operation produces [13], [10], [9]. Figure 1 shows an example of two measured power traces for two different blocks of code.

For a side-effect to be useful for verification, its *behavior* must be known in advance. For power consumption, we define behavior to mean the power trace observable during execution.

We define a probability density function (PDF) of the power

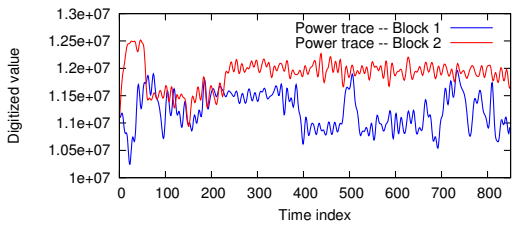

Fig. 1: Example of power traces for two blocks of code

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.

Our proposed method uses some ideas from the field of statistical pattern recognition [14]. 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 [10], [9], [11]. Our technique deals 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.

B. Mapping to Intermediate Representation

As we will discuss in Sections IV and V, our technique requires the ability to manipulate the power trace of a given program through transformations that preserve the semantics of the program. We wrote an optimization pass using the LLVM compiler framework [15] to perform these transformations statically. Our pass is 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 behaviors for the target CPU. Figure 2 shows an example of such a mapping: four IR instructions are shown above a power trace of their execution. For example, 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, so their power traces occur without interruption. The training technique described in [11] demonstrates a method to map IR instructions to power traces. In our case, the variance of each index is also useful for determining an appropriate error threshold (ε), which is discussed in more detail in Sections V and VI.

C. Control Flow

We need to be able to verify programs where *control flow statements* like branches and function calls exist. The specific order that those control flow statements execute is called the

```
IR1:    %16 = load i8* %b, align 1
IR2:    %17 = sext i8 %16 to i32
IR3:    %18 = icmp ne i32 %17, 0
IR4:    br i1 %18, label %19, label %20
```

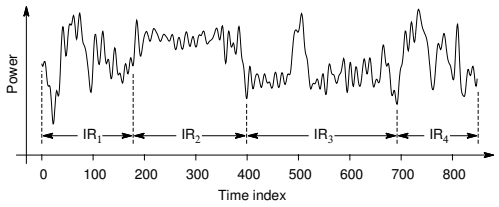

Fig. 2: Mapping sequential IR instructions to power consumption behavior

control flow, and determining it in advance is undecidable in the general case. We model it using a Control Flow Graph (CFG), which is a directed graph (V, E) where the set of vertices V represent sequential sections of code, called *basic blocks*, and the set of edges E represent possible control flow.

D. 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 [16] and distributive [17] over a bounded semi-lattice, then the MOP solution can be efficiently obtained by known algorithms.

As part of this paper, we will show how to map our problem to a gdfap to find the MOP solution. We will formalize a Data Flow Analysis (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*.

IV. Problem Statement and Assumptions

This work 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 behavior of the power consumption on the prover during its 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 [9], [10], [11], [12]. Like with any side-effect used for software attestation, this requires building a model of the power consumption behavior 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 [18]).

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.

V. 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, the system designer first profiles their target processor as described in Section III-B. This finds the predicted power consumption behavior for each IR instruction. Profiling is only done once for a processor model, as each unit has equivalent power characteristics.

The release engineer then chooses a secret power consumption behavior, β, and an error threshold ε. More details on selecting these parameters are included in Sections V-A and VI.

During compilation, the static transformation is applied to the program, with β and ε used as parameters.

During operation, an external device monitors power consumption to detect deviations from the secret power signature embedded in the program. Figure 3 sketches this functionality. If the verifier observes a power trace that is within the error threshold ε of the target power consumption behavior β, 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.

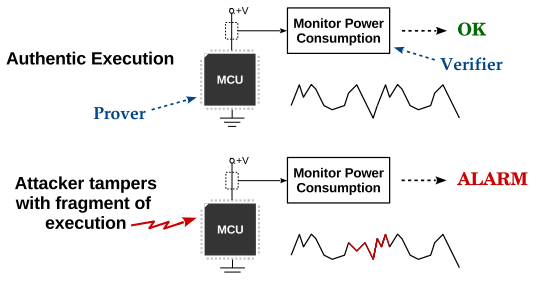

Fig. 3: Operation Phase

The following sections describe the static transformation in detail. Section V-A establishes the mechanisms by which we map the predicted power consumption behaviors for IR instructions to the target power consumption behavior $\boldsymbol{\beta}$. Section V-B describes the DFA that uses those mechanisms to find the possible alignments in the target behavior for each part of the program. Section V-C explains the selection of the alignments that minimize the overhead, and Section V-D describes the strategy to fill in the gaps left by those alignments.

A. Relating Power Consumption Behaviors

We will modify the program so that its power consumption behavior is identical, regardless of the control flow. We choose a desired power consumption behavior 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 consumption behavior and then modify the program to repeat it until termination.

The power consumption behavior of a program fragment is given by its corresponding power trace. We represent such behaviors 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* $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} \stackrel{\text{def}}{\equiv} \langle x_1, x_2, \cdots, x_n \rangle$; *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:*

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

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

To evaluate the proximity of two power consumption behaviors, 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:

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

This is a commonly used metric in pattern recognition techniques [14], and has been successfully used in existing power-based tracing works [10], [9].

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 behavior $\mathbf{b} \in \mathbb{R}^m$ that we predict to occur during any execution of the program and the desired behavior $\boldsymbol{\beta} \in \mathbb{R}^n$, 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 behavior to the behavior from a portion of the program. For this we can simply use an offset from the beginning of the desired behavior. In our previous example, if we only want to compare the behavior $\mathbf{b} = \langle b_{70} \cdots b_{120} \rangle$, we would use a temporary vector $\boldsymbol{\beta}_{70,51} = \langle \beta_{70} \cdots \beta_{100}, \beta_1 \cdots \beta_{20} \rangle$.

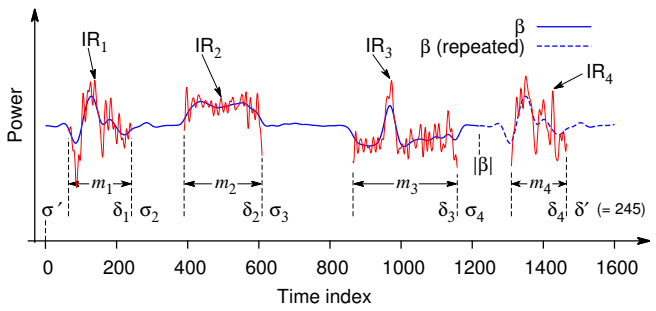

Fig. 4: Applying τ'_ε to a power consumption behavior sequence

Definition 4 *$\tau_\varepsilon : \mathbb{R}^m \times \mathbb{N} \to \mathbb{N}$ maps a power consumption behavior $\mathbf{b} \in \mathbb{R}^m$ 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 a power consumption behavior $\mathbf{b} \in \mathbb{R}^m$, a desired behavior $\boldsymbol{\beta} \in \mathbb{R}^n$, a starting offset σ, and a distance threshold ε, then:

$$\delta = m + \min \left\{ i \in [0, n) : \left\| \mathbf{b} - \boldsymbol{\beta}_{\sigma+i,m} \right\| < \varepsilon \right\} \tag{2}$$

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

Definition 5 *$\tau'_\varepsilon : (\mathbb{R}^m)^i \times \mathbb{N} \to \mathbb{N}$ applies τ to a sequence of i power consumption behaviors, 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 power consumption behaviors $\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_i \in \mathbb{R}^m$, a desired behavior $\boldsymbol{\beta} \in \mathbb{R}^n$, a starting offset σ, and a distance threshold ε, then:

$$\delta' = \tau_\varepsilon(\mathbf{b}_i, \tau_\varepsilon(\cdots \tau_\varepsilon(\mathbf{b}_2, \tau_\varepsilon(\mathbf{b}_1, \sigma')))) \tag{3}$$

τ'_ε applies τ_ε to each of the power consumption behaviors 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 behavior in the sequence. τ' is parameterized by ε, which is used for every τ in the sequence of applications.

Figure 4 shows an example of applying τ'_ε to the behaviors for the four IR instructions from Figure 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 power consumption behavior, and each m_k signifies the length of the behavior. 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 behavior 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 $\sigma_4 = 1158$, the position is found by traversing $\boldsymbol{\beta}$ circularly, since $|\boldsymbol{\beta}|$ is only 1220; in the figure, $\boldsymbol{\beta}$ is repeated and δ_4 is found to be 245.

B. 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 [19]. 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.

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

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

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

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 consumption behavior sequence $\mathbf{b}$.

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

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

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* [20], since L is a power set and $\sqcup = \cup$. □

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

Proof: Suppose, by way of contradiction, that

$$\exists f_{\text{bad}} \in \mathcal{F}, \exists a, b \in L : a \preceq b \text{ and } f_{\text{bad}}(a) \not\preceq f_{\text{bad}}(b)$$

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 $f(a \sqcup b) \not\preceq f(a) \sqcup f(b)$ and second, where $f(a) \sqcup f(b) \not\preceq f(a \sqcup b)$.

Case 1: $(f(a \sqcup b) \not\preceq f(a) \sqcup f(b))$

Given the set of all sequences of power consumption behaviors, $B^* = 2^{(\mathbb{R}^m)^i}$, 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. □

C. Index Selection Pass

The index selection pass identifies the minimum overhead necessary to align each section of the program with the target power consumption behavior. 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 $n = |\boldsymbol{\beta}|$:

$$\hat{\sigma} = \underset{(\sigma,\delta)\in\mathcal{S}}{\arg\min}\ (\delta - \sigma)\ + \sum_{(\sigma_b,\delta_b)\in\mathcal{S}} (\sigma - \sigma_b + n) \bmod n \quad (4)$$

Figure 5 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 V-D.

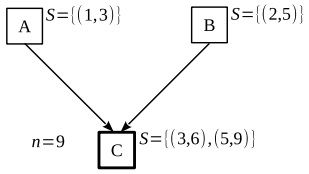

Fig. 5: Selecting start and end indices

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:

$$\hat{\sigma} = \underset{(\sigma,\delta)\in\mathcal{S}}{\arg\min}\ P_{(\delta,\sigma)} \left((\delta-\sigma)\ + \sum P_{(\sigma_b,\delta_b)}((\sigma-\sigma_b+n) \bmod n) \right) \quad (5)$$

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.

D. 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 consumption behavior and the beginning of the next. Where gaps occur in the basic block, we create and insert instructions which most closely match the behavior of that gap.

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

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

$$\gamma \triangleq (\tau_\varepsilon(\mathbf{p}, \sigma) - m) - \sigma \quad (6)$$

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 shows an example of adding interstitial basic blocks. In Figure 6a, 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 6b, an interstitial basic block A_{int} has been added to fill the gap.

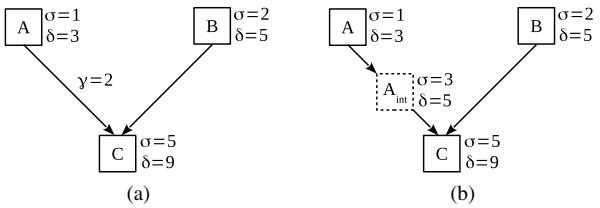

Fig. 6: Adding interstitial basic blocks

VI. 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 3. A more detailed description of the implementation steps follows.

1) Profile the device by capturing power traces for each IR instruction, as discussed in Section III-A. 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. Previous works have shown that it can be automated [11].
2) Define a target power consumption profile $\boldsymbol{\beta}$, as described in Definition 3, which serves as a secret key.

A separate β 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 consumption behavior and produce an instrumented binary. We created an implementation using the LLVM compiler framework [21], 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 β. This monitoring system should use pattern recognition techniques, mentioned in Section III-A, to detect deviations in the expected device's behavior. This is beyond the scope of this paper, but there is ample evidence that this is a feasible task [10], [9], [12], [11].

VII. Discussion and Future Work

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.

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 consumption behavior β 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.

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 β. In a practical implementation the system should assist users with verifying these timing requirements after instrumentation of the binary.

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

There are many additional avenues for extension of our method. Considering systems with cooperative multitasking is a promising research direction, although this may require recompiling parts of the operating system. We would also like to explore the applicability of our approach to systems with interrupts. In the case of periodic interrupts, which is a common paradigm, it may suffice to align the interrupt period to a multiple of the length of β.

VIII. Conclusion

In this paper, we proposed a novel technique to verify the authenticity of the executing software using power consumption. This was made possible through a compiler optimization stage designed to statically modify a program so that its power consumption behavior was known in advance.

Our proposed method is well suited to real-time systems, and we expect that this work will lead to further research into software attestation with non-traditional side-effects.

Acknowledgments

The first author would like to thank Dr. Nomair Naeem for fruitful discussions and ideas. This research has been supported in part by the Natural Sciences and Engineering Research Council of Canada, the Ontario Centres of Excellence, and the Ontario Research Fund.

References

[1] Kinney, Steven L., *Trusted Platform Module Basics: Using TPM in Embedded Systems (Embedded Technology)*. Newnes, 2006.
[2] 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.
[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] Kennell, Rick and Jamieson, Leah H., "Establishing the Genuinity of Remote Computer Systems," in *USENIX Security Symposium*, 2003.
[5] 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.
[6] Shankar, Umesh and Chew, Monica and Tygar, J. D., "Side Effects Are Not Sufficient to Authenticate Software," in *USENIX Security Symposium*, 2004.
[7] 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.
[8] Shacham, Hovav, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM CCS*, 2007.
[9] T. Eisenbarth, C. Paar, and B. Weghenkel, "Building a Side Channel Based Disassembler." Springer Berlin Heidelberg, 2010, pp. 78–99.
[10] C. Moreno, S. Fischmeister, and M. A. Hasan, "Non-intrusive Program Tracing and Debugging of Deployed Embedded Systems Through Side-Channel Analysis," *LCTES*, 2013.
[11] C. Moreno, S. Kauffman, and S. Fischmeister, "Efficient Program Tracing and Monitoring Through Power Consumption – With A Little Help From The Compiler," *DATE*, 2016.
[12] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu, "WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices," 2013.
[13] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," *Advances in Cryptology*, 1999.
[14] Andrew R. Webb and Keith D. Copsey, *Statistical Pattern Recognition*, 3rd ed. Wiley, 2011.
[15] Chris Lattner and the LLVM Developer Group, "The LLVM Compiler Infrastructure – online documentation," http://llvm.org.
[16] Kam, John B and Ullman, Jeffrey D, "Monotone data flow analysis frameworks," *Acta Informatica*, vol. 7, no. 3, pp. 305–317, 1977.
[17] Kildall, Gary A., "A Unified Approach to Global Program Optimization," in *Symposium on Principles of Programming Languages*, 1973.
[18] Williams, Peter and Sion, Radu, "Usable PIR," in *NDSS*, 2008.
[19] Aho, Alfred V and Ullman, Jeffrey D, *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972.
[20] S. Warner, *Modern algebra*. Courier Corporation, 1990.
[21] S. Kauffman, "SETA Source Code," 2016. [Online]. Available: https://bitbucket.org/seanmk/sideeffect-transform