EAI Endorsed Transactions
on Security and Safety
Research Article EAI.EU

Side-channel Programming for Software Integrity Checking

Hong Liu[1] and Eugene Y. Vasserman[2]

[1]Work performed while at Kansas State University hxdc77@163.com
[2]Department of Computer Science, Kansas State University, Manhattan, KS 66506 USA eyv@ksu.edu

Abstract

Verifying software integrity for embedded systems, especially legacy and deployed systems, is very challenging. Ordinary integrity protection and verification methods rely on sophisticated processors or security hardware, and cannot be applied to many embedded systems due to cost, energy consumption, and inability of update. Furthermore, embedded systems are often small computers on a single chip, making it more difficult to verify integrity without invasive access to the hardware.

In this work, we propose "side-channel programming", a novel method to assist with non-intrusive software integrity checking by transforming code in a functionality-preserving manner while making it possible to verify the internal state of a running device via side-channels. To do so, we first need to accurately profile the side-channel emanations of an embedded device. Using new black-box side-channel profiling techniques, we show that it is possible to build accurate side-channel models of a PIC microcontroller with no prior knowledge of the detailed microcontroller architecture. It even allows us to uncover undocumented behavior of the microcontroller. Then we show how to "side-channel program" the target device in a way that we can verify its internal state from simply measuring the passive side-channel emanations.

Received on 23 March 2021; accepted on 27 May 2021; published on 02 June 2021

Keywords: Security; Embedded systems; Software integrity; Side-channel analysis

doi:10.4108/eai.2-6-2021.170013

1. Introduction

Embedded devices are pervasive in our everyday life. From electric thermostats, elevators, automobiles, to insulin pumps, most embedded devices nowadays are cyber-physical systems which are composed of one or more processors and are controlled by software. Since many of these devices are essential to our safety and well-being, verifying their integrity is an important task. Developers traditionally focus on realizing device functionality, while overlooking an attacker who may change the behavior of a device by overwriting its program and/or data remotely or locally.

Enforcing software integrity for embedded systems, especially legacy and deployed systems, is extremely difficult. Modern integrity checking mechanisms rely on specialized trusted hardware and/or sophisticated processors that provide security functionality to detect or prevent software tampering [1–9]. However, many embedded systems do not possess such capabilities due to prohibitive cost, power consumption, space, and inability of update.

Our solution is to utilize "side-channels" – any channels that leak information about the runtime state of a device as a by-product of executing software on a physical device. Internal signal switches of a CMOS circuit will cause current flows to charge or discharge internal node capacitance, which can be measured from energy consumed from the power pin. Researchers have tried using power consumption and electromagnetic radiation to detect abnormal behavior at the level of function blocks or code segments [10–14]. However, attackers can write compact malware as small as one instruction in size (e.g., not setting an important flag) which has minimal impact on side-channel measurements. More importantly, a side-channel-aware attacker can profile the target device and rewrite malware in a fashion that the tampered

code has indistinguishable difference in side-channel measurement from the authentic code.[1]

In such situations, software attestation can be used to detect software tampering down to single instructions [15–18], even for a side-channel-aware attacker. Software attestation utilizes the timing side-channel and does not rely on specialized hardware or sophisticated processors, but requires interruption of normal execution, and is also inapplicable to deployed systems that cannot be updated to support software attestation (while malware tampers with RAM). In addition, software attestation is vulnerable to transient malware in which the memory configuration is tampered and then restored between two attestations.

In this work, we explore the feasibility of using passive side-channel emissions of an SoC for software integrity checking, without modifying the hardware, or interrupting device execution. The idea is to first find the relationships between internal activities of an embedded device and side-channel emanations. Then we design a protection mechanism utilizing the side-channel characteristics. The goal is to determine whether the internal state is an expected one or not by measuring the passive side-channel emanations of a running device. Unexpected state may be a sign of incorrect execution or malware. Compared to side-channel analysis for other purposes (such as extracting cryptographic key materials), this requires inferring internal runtime status of an embedded system at a granularity that is able to detect transient, compact, and side-channel-aware malware from a single side-channel measurement.

Such an integrity checking mechanism is very useful because: (a) measuring passive side-channel emissions without interrupting device execution will not leave a trace of the integrity checking mechanism – the integrity verifier is not visible to attackers who penetrate the target device; (b) the integrity verifier resides outside the target device and does not incur computation, power, or space overhead to the device; (c) it is easy to deploy and update the verifier without modifying the target device; (d) adversaries only get stronger; a verifier that is external to a device is the only possible verifier when all the security mechanisms internal to the device fail, or do not even exist.

The main contributions of this work include:[2]

- Details on constructing side-channel models of a microcontroller instruction set at clock-cycle level accuracy, enabling inference of internal activities given a single passive capture of side-channel measurements, including discovery of several undocumented behaviors of the target device.
- A new integrity checking approach – "side-channel programming" – which allows programs to utilize the side-channel characteristics of a target device. The approach is unique in that any code that satisfies the side-channel constraints is guaranteed to reach the desired final state. Furthermore, the security properties of side-channel programming hold even in the presence of strong adversaries who know they are being monitored.
- The design, pseudo-code, and evaluation of side-channel programming on a microcontroller.

We note that a major differentiating factor of this work from other contributions in the area of side-channels is that we are using side-channels "for good rather than evil". This means that side-channel programming must not expose additional attack surfaces. This aligns with our goal of *code integrity*. Briefly, out attacker model (see Section 3 for details) is a full-control but "hardware-respecting" adversary: they have full control of any running code but cannot modify any of the hardware characteristics of the microcontroller. This attacker would see no benefit from profiling side-channels from the microcontroller as they would know the ground truth of the executing code and runtime characteristics/parameters. The same would hold after the application of side-channel programming. In short, *confidentiality* of the running code is a non-goal.

2. Related Work

A fundamental integrity checking method for embedded devices without hardware modification is to sniff bus signals. This is easy to perform on systems composed of discrete components. For small systems integrated in single chips, however, internal activities cannot be observed easily. And it is not always practical or efficient to perform micro-scoping or micro-probing, due to the invasive access to the ICs, requirement of costly equipment and expertise [21].

An alternative strategy is software and/or firmware modification. For example, the Symbiote [22] integrity checking method does not require security hardware or sophisticated processors, but rather is injected into embedded device firmware in a randomized way and computes checksums on protected regions periodically. Symbiote however cannot detect transient malware or defend attackers who are aware of the existence of Symbiote.

[1] Although side-channel-aware attackers are considered in [12], the authors do not rigorously evaluate the difficulty in creating malware that produces indistinguishable side-channel measurements (c.f. Sections 4.3 and 5).

[2] Our preliminary experiments on power consumption and the idea of side-channel programming were first presented in [19], and [20] briefly mentioned our experiments on electromagnetic radiation.

Our solution is to utilize side-channel information to determine whether the internal state of a device is legitimate or not. It involves two tasks: first, modeling side-channel emanations to find the relationships between side-channel emanations and internal activities, and second, utilizing the side-channel models to design the verifying mechanism.

2.1. Side-channel Profiling

Side-channel models can be built at different levels given different degrees of system configuration knowledge. At the lowest level, power consumption of a CMOS circuit is computed at the transistor level in order to analyze power usage and EMI/EMC properties [23, 24]. Dynamic power consumption, which is of more interest for security analysis, is in general modelled as the aggregation of power consumed by each node in a device [25]. The switching power consumption of an internal node is proportional to the clock frequency, the load capacitance of the node, and the frequency of switches of the node. More accurate power models also consider the cross-talk (interference) between signals of neighboring wires [26]. These power models are often used in combination with SPICE simulation and manufacture parameters to estimate dynamic power of small-scale circuits [23–25, 27–29]. In practice, however, the complexity and obscurity of embedded devices prevent such analysis. Researchers have in turn tried to build empirical ("black-box") models from real measurements, with limited or no knowledge of the chip architecture.

In the simplest form, application developers read the voltage of the battery of a mobile phone from time to time to determine the power consumption of the system so that some power-saving strategy can be applied. At a finer granularity, researchers have studied the average power consumption of instructions of various devices to guide power-efficient processor design or software development [30–32]. Researchers often execute the target instruction (e.g., multiply, branch) for many times and then compute the average power consumption or EM cartography. Such study does not concern runtime side-channel emanations for individual instruction executions. These averaged side-channel models cannot be used to design integrity checking mechanisms.

Empirical profiling of side-channel emanations has also been researched for many other purposes, including device fingerprinting, covert channels, detecting hardware trojans, and breaking cryptographic hardware [33–41]. There are several major differences between side-channel analysis for software integrity and for other purposes:

1. Analysis for software integrity is on a single captured side-channel measurement that represents a one-time execution of some code with data;
2. The analysis is over the entire instruction set instead of a few special instructions;
3. The analysis is on the entire trace in one capture instead of a few special points in the trace;
4. The analysis is on a black-box device the design detail of which is unknown;
5. The analysis must consider compact malware that may be composed of a single instruction, or a change of original instruction only on the operands;
6. The analysis must consider side-channel-aware attackers who actively attempt to evade detection by computing alternative code that has near indistinguishable side-channel measurement from that of the original code.

2.2. Side-channel-based Protection Mechanisms

Previous research on side-channel analysis for integrity of general programs tries to detect anomalous behaviors and/or malware from passive system-wide power measurements of functions and code segments [10–14, 42–44]. Often pattern matching and machine learning techniques are used to build the reference side-channel emanations for code blocks, and new side-channel measurements are matched against the reference to detect anomalous behaviors. Because the learned side-channel profiles cannot predict side-channel emanations of arbitrary code, these methods rely on malware being sufficiently long/causing peculiar side-channel emanations, and not written to adapt side-channel profiles. Some works [12, 13, 42] only used very few examples of tampering to test the effectiveness of their approach. To use side-channels for rigorous integrity checking, however, it is important to secure against arbitrary, compact, and side-channel-aware malware, given any original code to protect.

Software attestation [15–18] is capable of detecting malware at such precision and does not require modification of hardware. It however requires software update and interruption of the device execution, a particular drawback for deployed systems. Software attestation is also unable to detect transient faults and attacks such as self-delete malware and data-only attacks in which memory is tampered and then restored between two attestations.

Researchers have tried to build side-channel profiles of the instruction set for certain smart cards, programmable logic controllers, and microcontrollers [12, 14, 42, 45–51]. The purpose is to recognize instruction

operations from passive side-channel measurements (e.g., `MOVLW`). For integrity checking, in contrast, the entire instruction (e.g., `MOVLW 0xAA`), as well as internal states such as the content of registers and memory must be verified (e.g., data-only attacks). In addition, we will show in later sections that profiling side-channels based on instruction operations is unlikely to be very successful at least for our target device.

3. Threat Model

Device Under Test The device under test (DUT) is a PIC16F687 microcontroller (μC). PIC16F687 has a 8-bit RISC processor in Harvard architecture [52, 53]. The instruction set has 35 operations. The processor, program flash, RAM, on-chip oscillator, and peripherals, such as the programming interface, timers, and a A/D Converter (ADC), are integrated into a single chip. Though a simple device compared to many more modern microcontrollers, most related research has been performed on this IC [47–51], making this device ideal for comparison of different techniques.

Adversary We assume a powerful attacker, able to modify the software of the DUT, for instance by modifying RAM through buffer overflows, code-reuse, data-only attacks, etc., or by reprogramming the device, or by inserting trojans into the CAD software. Furthermore, the attacker is able to profile the side-channel emissions of the DUT and to modify the software in a fashion that minimizes side-channel deviations from original code. The attacker is however unable to inject faults or modify the hardware, including the IC design and the Printed Circuit Board (PCB) on which the DUT is mounted – in essence, the attacker cannot modify the inherent side-channel emissions of the target device. We also assume that the attacker is not an insider of the IC manufacturer and is unable to tamper with the CAD tools upon which we based the ground truth.

Verifier We assume that the verifier is trusted and able to profile side-channel emissions of the DUT. The verifier cannot modify the hardware, including the IC and the PCB to amplify side-channel emissions. The verifier does not have more knowledge about the IC design besides public documents. The verifier does not need to reverse-engineer the IC, although doing so may facilitate side-channel profiling (c.f. [21]). In particular, the verifier can only passively measure the DUT with measurement equipment that incurs minimal impact on the Electromagnetic Compatibility (EMC) of the DUT. The verifier for example may remove the shielding enclosure for measurement, but should not remove the noise decoupling circuits, which may cause errors in device execution.

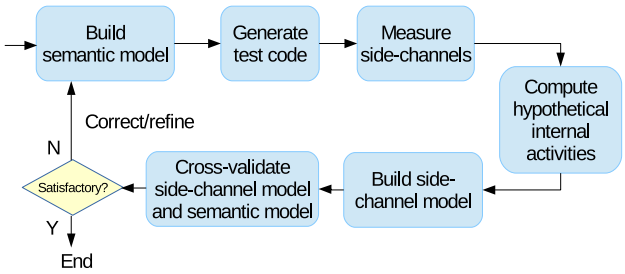

Figure 1. Side-channel profiling a black-box device.

4. Side-channel Profiling of the Instruction Set

To design a side-channel-based protection mechanism such as "side-channel programming", we must first discover how side-channel information is linked to internal activities. As introduced in Section 2.1, dynamic power consumption of a CMOS circuit can be approximated as a sum of power consumed by all the internal nodes, proportional to the load capacitance and the frequency of signal switches of each node. For our black-box DUT, however, we do not know the detailed structure of the IC or how internal signals change as instructions are executed. Public documents of the DUT only describe the chip skeleton and functional behaviors of the instruction set [52, 53].

Previous work in profiling side-channels of a "black-box" device (without knowing the IC design or reverse-engineering the IC), often make the simplifying assumption that side-channel leakage is related to instruction operations (e.g., `ADD` has different side-channel patterns with `XOR`), and use pattern matching/classification or machine learning techniques to classify side-channel measurements according to operations [12, 14, 48, 49, 51]. In our experiments we find that, at least for our DUT, such methods cannot lead to accurate side-channel models or reliable integrity checking schemes, as side-channel leakage is only related to instruction operations indirectly, as a macroscopic effect of internal signal switches.

Our approach to accurately profile side-channels is to build semantic model of the instruction set, and then to relate the side-channel patterns with the semantic model. Repeat the process until we get satisfactory semantic model and side-channel profile simultaneously, as shown in Figure 1:

1. Build semantic model of the instruction set, using known architecture information;
2. Generate test code and collect side-channel measurements;
3. Calculate runtime activities according to the semantic model;
4. Build side-channel model and cross-validate the side-channel model with the semantic model;

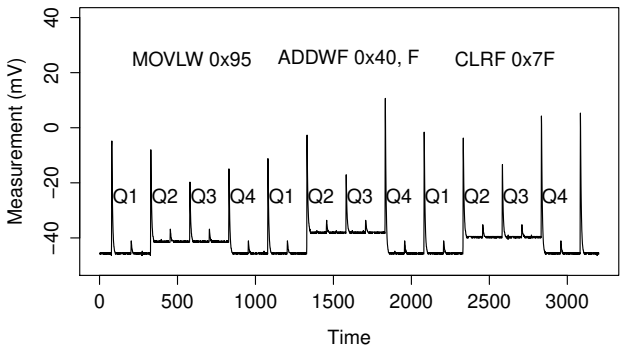

Figure 2. A single captured power measurement executing {`MOVLW 0x95; ADDWF 0x40, F; CLRF 0x7F`}. The x-axis is time and the y-axis is voltage drop. The oscilloscope offset to -60 mV to span the vertical range.

5. Correct or refine the semantic model and repeat the procedure.

To build a semantic model of the instruction set, we enumerate internal activities, such as fetch, decode, and data read/write, that may occur during instruction execution. Because detailed architecture information is unknown, our semantic model is only a conjecture to be tested against side-channel measurements. Next, we generate code that tests properties of the semantic model and perform side-channel measurements. Meanwhile, we compute the potential runtime activities of the test code according to the semantic model. We then try to find the linkage between internal activities and side-channel measurements. A highly accurate side-channel model that relates internal activities to side-channel information proves the validity of the semantic model as well.

Initially we only have a coarse semantic model and generate code to test very specific properties. We then collect side-channel measurements and obtain an initial side-channel model. Next, we analyze the outliers and try to correct the semantic model. If no outliers exist, we repeat the procedure to test more generic properties or to find higher-order relationships, until we obtain satisfactory semantic model and side-channel model simultaneously. For example, we first generate test code that only uses a portion of general-purpose registers in order to execute each operation many times with different data, and obtain a side-channel model for these GPRs. Then we generate code that uses all GPRs to see if the side-channel model still holds for other GPRs.

A final remark is that we do not try to profile side-channels of code that involves system-level behaviors such as I/O events, putting device to sleep, or accessing ADC or other peripherals. We only model side-channels of "computation" and leave other profiling tasks for further work.

4.1. Semantic Model

Although the DUT is a simple μC, there are many factors that may cause internal signal switches and thus affect the side-channel leakage. The instruction set consists of 35 different operations. The processor has a two-stage pipeline. Each instruction execution is overlapped with the next instruction fetch. Most instructions execute in single instruction cycle, except branches. Unconditional and conditional branches take two instruction cycles if a branch is taken, and a `NOP` operation will be executed instead of the original next instruction in the second instruction cycle. Otherwise a branch instruction takes one instruction cycle. Each instruction cycle is composed of four clock cycles, denoted as Q1 to Q4, as shown in Figure 2. The working register is one of the two operands of the ALU. There is a 128-byte register file including general-purpose registers (GPRs) and special function registers (SFRs). GPRs are composed of registers indexed `0x40` to `0x7f`.

Based on the architecture description in the PIC16F687 datasheet [53], we deduce that potential data that may appear on buses, and therefore are likely to cause major side-channel emissions, include the program counter (PC), the operands and coding of instructions, the working register, the selected file register, and the `STATUS` SFR. Formally, we represent the internal state of the device as $T = (W, C, F, PC, I_{prev}, I_{curr}, I_{next}, \texttt{Type}, \texttt{OPRD1}, \texttt{OPRD2}, B, D)$ where

- W is the working register,
- C is the `STATUS` register,
- F is the GPRs (indexed from `0x40` to `0x7f`),
- PC is the program counter,
- I_{prev} is the previous instruction (including operation and operands),
- I_{curr} is the current instruction,
- I_{next} is the next instruction,
- `Type` is the type of the operation of the current instruction,
- `OPRD1` is the *content* of first operand,
- `OPRD2` is the *content* of second operand,
- $B \in$ {True,False} is whether a branch is executing,
- D is the result of the current instruction.

`Type` is one of the instruction categories we summarize from the instruction set document [52, 53], as shown in Table 1. We then build the hypothetical

Table 1. Instruction summary

Description	Type	Example(s)
byte-oriented file register operation with the working register as destination	wfw	ADDWF f,W
byte-oriented file register operation with GPR as destination	wff	{ADDWF f,F}, {CLRF f}, {MOVWF f}
bit-oriented increment/decrement branch operation with the working register as destination	fszw	{INCFSZ f,W}, {DECFSZ f,W}
bit-oriented increment/decrement branch operation with GPR as destination	fszf	{INCFSZ f,F}, {DECFSZ f,F}
bit-oriented test branch operation	btfs	BTFSC, BTFSS
bit-oriented set/clear operation	bxf	BCF, BSF
literal operation	lw	ADDLW k
literal clear operation	clrw	CLRW
goto operation	goto	GOTO
call operation	call	CALL
return operation	ret	RETURN, RETLW, RETFIE
no operation	nop	NOP
inserted NOP when branch is taken	brnop	

semantic model for each instruction, e.g.,

$$T_0 = (W, C, F, PC, I_{prev}, \{\texttt{ADDWF f,W}\}, I_{next}, \texttt{Type0}, \texttt{OPRD1}, \texttt{OPRD2}, \text{False}, D) \xrightarrow{\texttt{ADDWF f,W}}$$
$$T_1 = (W' = mod(W + (f), 256), C', F, PC + 1, \{\texttt{ADDWF f,W}\}, I_{next}, I_{next\_next}, \texttt{wfw}, W, (f), \text{False}, W')$$

which means, if a branch is not executing (i.e., current instruction is not replaced with NOP), after executing ADDWF f,W, the internal state of the device will change from T_0 to T_1, with (1) the working register is updated with the truncated value of $(W + (f))$, where (f) is the content of the GPR f; (2) the STATUS register is updated according to the result of $(W + (f))$; (3) the GPRs remain the same; (4) the program counter increments; (5) next instruction is read; (6) the type of the operation is updated to wfw; (7) the first operand is the working register; (8) the second operand is the content of the GPR f; (9) no branch will happen; and (10) the result of the instruction is W'. The state transition of ADDWF f,F can be defined similarly, with the exception that the result is stored back to f instead of W. In this way we build the hypothetical semantic models for ADDWF f,d, ANDWF f,d, COMF f,d, DECF f,d, INCF f,d, IORWF f,d, RLF f,d, RRF f,d, SUBWF f,d, SWAPF f,d, XORWF f,d, where d is W or F indicating whether the result is stored to the working register or the GPR. Note that for SUBWF f,d, the result is the truncated value of $((f) - W)$. One of the operands can be both W or two's complement of W. We have tried both cases in cross-validation.

For conditional branch instructions,take the example of INCFSZ f,W:

$$T_0 = (W, C, F, PC, I_{prev}, \{\texttt{INCFSZ f,W}\}, I_{next}, \texttt{Type0}, \texttt{OPRD1}, \texttt{OPRD2}, \text{False}, D) \xrightarrow{\texttt{INCFSZ f,W}}$$
$$T_1 = (W' = mod((f) + 1, 256), C, F, PC + 1, \{\texttt{INCFSZ f,W}\}, I_{next}, I_{next\_next}, \texttt{fszw}, (f), \text{NA}, \begin{cases}\text{False if } W' \neq 0\\ \text{True if } W' = 0,\end{cases} W')$$

which means, if a branch is not executing, after executing INCFSZ f,W, the internal state of the device will change from T_0 to T_1, with (1) the working register is updated with the truncated value of $((f) + 1)$; (2) the STATUS register and GPRs are not affected; (3) the program counter increments (see below); (4) next instruction is read (see below); (5) the type of the operation is updated to fszw; (6) the first operand is the content of the GPR f; (7) the second operand is not related; (8) branch (skip) will happen if $W' = 0$; and (9) the result of the operation is W'.

If a branch is taken after INCFSZ f,W:

$$T_0 = (W, C, F, PC, \{\texttt{INCFSZ f,W}\}, I_{curr}, I_{next}, \texttt{fszw}, (f), \text{NA}, \text{True}, W) \xrightarrow{I_{curr}}$$
$$T_1 = (W, C, F, PC + 1, \{\texttt{NOP}\}, I_{next}, I_{next\_next}, \texttt{brnop}, W, 0, \text{False}, W)$$

which means, if a branch is executing, for any I_{curr}, the internal state of the device will change from T_0 to T_1, with (1) the actual instruction executed is NOP; (2) the program counter increments; (3) next instruction is

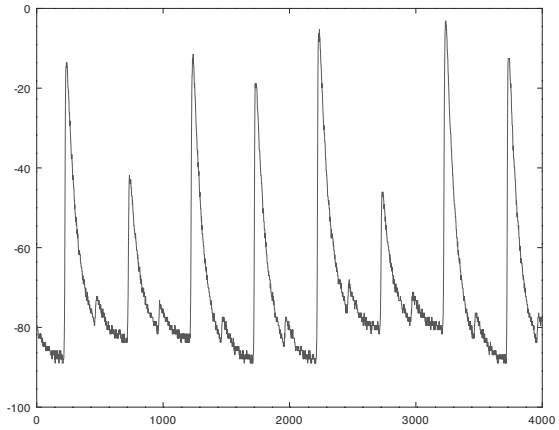

Figure 3. A single captured power measurement. The amplitude has a -100 mV offset.

read; (4) the type of the operation is updated to `brnop`; (5) the first operand is W, the second operand is 0, and the result of the operation is W (c.f. Section 4.3); (6) branch will not happen. The semantic model of each instruction operation is defined similarly.

4.2. Experimental Setup

We perform passive measurements of power consumption and electromagnetic (EM) radiation when running the test code. The power consumption is measured by inserting a 82Ω shunt resistor to the ground pin of the DUT. The voltage drop cross the shunt resistor is sampled by a PicoScope 5244B oscilloscope, which has a 200 MHz bandwidth and a maximum per-channel sampling rate of 500 MS/s. We use the oscilloscope's 20 MHz integrated hardware filter to avoid aliasing. Because the side-channel signal is of frequency much higher than the processor's main clock, the clock frequency should be much lower than 20 MHz. We set the clock to 125 kHz, generated by the internal clock generator of the device. A typical single-captured waveform is shown in Figure 2. It exhibits sharp peaks at clock rising edges and much smaller peaks at falling edges. A plateau can be seen through Q2 to the peak of Q4.

We build side-channel models at 125 kHz and then repeat the experiment at 1 MHz clock (using a 100Ω shunt resistor). A typical single-captured waveform is shown in Figure 3, where the low-pass filtering effect becomes stronger due to the higher clock and also to the 20 MHz hardware filter. The side-channel model for the peak amplitudes is essentially identical between the 125 KHz and 1 MHz experiments, with the exception that at 1 MHz, SNR is much lower and the plateaus following the peaks in Q2 and Q3 is no longer visible.

We also repeat the experiment using EM measurement at 125 kHz. It is a proof-of-concept to test an alternative to insertion of a shunt resistor for power measurement, where no modification of the original

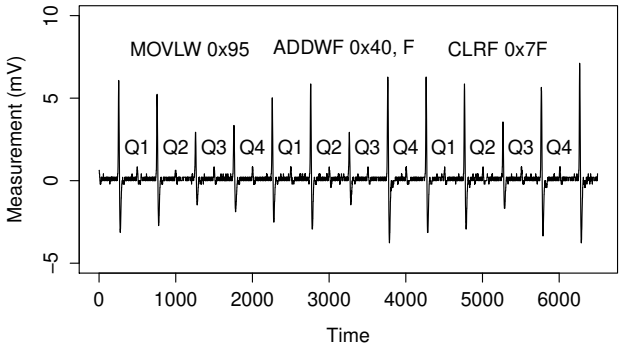

Figure 4. A single captured EM measurement.

hardware is necessary. EM emission is measured by a hand-made probe following the design of EMC probe in [54]. The loop probe is placed over the power line from the V_{SS} pin which also forms a loop (with an opening) to the power source. The perimeter of the probe is 25 mm. The magnetic field generated by the power line will cause voltage drop in the loop probe, which is amplified by a 20 dB amplifier. The probe therefore essentially collects global current. The amplifier output is sampled by the same oscilloscope. The 20 MHz integrated hardware filter of the oscilloscope is also turned on. A typical single-captured waveform of EM radiation of the DUT is shown in Figure 4. Although the waveform of EM radiation appears to be different, the side-channel models are similar to those of power consumption as shown in Section 4.3.

At each iteration, we generate test code traces and calculate internal activities from the hypothetical semantic model. Operation for each instruction is chosen from a portion of the instruction set, and the operands are selected to compose a legal instruction (file register access is limited to GPRs and the `STATUS` SFR). `CALL`, `RETFIE`, `RETLW`, and `RETURN` are manually inserted in multiple places so that the program can execute normally. `SLEEP` (put device to standby mode) and `CLRWDT` (clear watchdog timer) are not profiled. The target μC has 2 kB memory and around 1400 instructions are programmed each time. We then cross-validate the measurements with the hypothetical internal activities to find their relationships.

4.3. Side-channel Model

We first profile the power measurement with respect to instruction operations, as done in previous research [12, 14, 47–50]. The problem can be described as: given a single trace of power samples of four clock cycles, the verifier tries to recognize one out of 33 instruction operations based on the profiling model[3] – a typical pattern recognition/classification problem. We have applied

[3]Recall that `SLEEP` and `CLRWDT` are excluded

various general classifiers, including naive Bayes, k-nearest neighbor, support vector machine (SVM), multilayer perceptron, and template analysis. Power samples are preprocessed with/without feature selection by principal component analysis, mutual information, and linear discriminant analysis. The highest recognition rate is obtained by using template analysis, in which the power consumption is approximated as multi-variate Gaussian signals, similar with [48, 55]. The average recognition rate is 45.6%, which is comparable to the unoptimized results of [48, 55] and the single-location result of [50]. While some operations have acceptable recognition rates, such as `CALL` (94.3% recognition rate), `GOTO` (97.8%), `CLRW` (99.0%), and `COMF f,F` (95.7%), some operations, such as `CLRF`, `DECFSZ f,W` and `IORWF f,F`, are almost always misclassified.

Inspired by previous discoveries of linear relationship between power consumption and some data operation [47] and the success of linear regression analysis in cryptographic hardware [56, 57],[4] we apply regression analysis to profile power consumption of arbitrary code for the target device. Leveraging on the hypothetical semantic model and iterative correction, we succeed to obtain a very accurate side-channel model.

Let runtime activities at time t be a vector of variables $\vec{x}_t$, the power measurement at t be a random variable Y_t, we assume Y_t depends on $\vec{x}_t$:

$$Y_t = f(\vec{x}_t) + N_t$$

where N_t encloses remaining components in the power consumption at time t including time-dependent components and noise, as in [58]; N_t and Y_t are random variables; $\vec{x}_t$ is a result of executing instructions and is therefore controllable.

We include the internal state T of the hypothetical semantic model in the regressor $\vec{x}_t$, as well as the Hamming distance (HD) and the Hamming weight (HW) of the variables of T.[5] We have intentionally added more variables in the semantic model than necessary (not all shown in Section 4.1). This does not affect regression since adding more regressors will always give smaller mean square errors, as long as no multicollinearity exists among regressors [59]. Unnecessary regressors are pruned afterwards, using t-test, Partial F-test, and confidence interval of the regression coefficients.

It turns out that there are strong linear relationships between internal activities and power consumption measurements. For any instruction cycle t, there are four regression models, corresponding to four clock cycles Q1 to Q4:

$$Y_{q_i} = \vec{x}_t' \vec{\beta}_{q_i} + b_{q_i} + N$$

where $i \in [1, 4]$ indicates the clock cycle, $\vec{\beta}_{q_i}$ is a vector of weights (regression coefficients) and b_{q_i} is a constant. The noise component N is assumed to be time-independent.

The actual regression models are built for individual instruction categories (Type). We use the Pearson's correlation coefficient r and the Spearman's correlation coefficient ρ to show the performance of the regression models. For two random variables $X = \vec{x}' \vec{\beta}$ and Y, the Pearson correlation coefficient is a measure of linear dependence between X and Y:

$$r = \frac{\sigma_{XY}}{\sigma_X \sigma_Y} = \frac{\sigma_X}{\sqrt{\sigma_X^2 + \sigma_b^2}}$$

where the covariance of the two random variables are estimated by using the empirical sample variance. r tends to ± 1 as σ_b^2 tends to 0. Spearman's rank correlation is the Pearson correlation between weakly-ordered values. Spearman's correlation is able to find non-linear dependence between X and Y. The two correlations are identical for values which are linearly related. Spearman's correlation is more sensitive to outliers.

Q1. We find that the HD of PC and (PC+1) influences the peak in Q1, *regardless of instruction operations* due to the fact that each instruction execution is overlapped with fetching of the next instruction, and PC is incremented in Q1 for instruction fetch.

Q2. The result of previous the instruction (which is likely the data already on some internal bus) and the operand loaded for current instruction influence the peak amplitude in Q2. In Q2, different categories of operations will load different types of operands. And the peak amplitude in Q2 is linear to the HD of previous result and the content of the operand (e.g., (f) instead of f):

$$Y_{q2} = \beta_{q2} \cdot HD(previous\_result, data\_for\_current) + b_{q2}$$

where Y_{q2} is in millivolt (mV, same unit for the following Y). Table 2 shows the regression models.[6]

While performing the measurements described, we encountered some interesting and surprising findings. Contrary to the functional descriptions, some instructions loaded register contents whether or not they were needed. For instance, the content of the

[4] Note the difference between side-channel profiling for breaking cryptographic hardware vs. software integrity checking (c.f. Section 2)

[5] HD counts the number of bit differences between two binary values; HW counts the number of 1s of a binary value.

[6] The file register operations in Table 2 include all the operations not listed in the following entries, e.g., `ADDWF`, `CLRF`, `CLRW`, `MOVWF`, `BTFSC`, etc. The literal operations include `ADDLW`, `ANDLW`, `IORLW`, `XORLW`, and `MOVLW`. The constants b_{q_i} are negative because the oscilloscope is set to have a -60 mV offset.

Table 2. Regression analysis of power consumption in Q2

	[2][c]File register operations	
Predictors	$HD(previous\_result, (f))$	Constant
Coefficients	2.88	-15.30
r	0.97	
ρ	0.97	
	[2][c]Literal operations	
Predictors	$HD(previous\_result, literal)$	Constant
Coefficients	2.86	-19.34
r	0.92	
ρ	0.92	
	[2][c]SUBLW	
Predictors	$HD(previous\_result, literal)$	Constant
Coefficients	1.73	-17.99
r	0.95	
ρ	0.88	
	[2][c]NOP	
Predictors	$HD(previous\_result, 0)$	Constant
Coefficients	2.49	-19.63
r	0.90	
ρ	0.90	
	[2][c]GOTO	
Predictors	$HD(previous\_result, address)$	Constant
Coefficients	2.38	-22.09
r	0.92	
ρ	0.89	

file register was unnecessarily loaded for CLRF, CLRW, MOVWF.The content of the file register is loaded for bit-oriented and byte-oriented file register operations (e.g., ADDWF, INCF, BSF, INCFSZ, and BTFSC) regardless of whether the instruction is a conditional branch or not. The operand (e.g., f and b in BSF f,d) does not affect the peak amplitude in Q2.

Through our iterative profiling approach, we are able to uncover the actual data loaded in Q2:

- The content of GPR 0x7f is loaded for CLRW – we discover that CLRW is actually implemented as CLRF 0x7f,W;
- The target address is loaded for GOTO;
- The operand is loaded for literal operations such as ADDLW (moreover, the operand of SUBLW, not its two's complement, is loaded);
- Zero is loaded for NOP since NOP is implemented as ADDLW 0, except that it has no effect on the STATUS flags.

The Plateaus. The plateaus following the peaks in Q2 and Q3, are linear to the HW of next instruction, *regardless of instruction operations*:

$$Y_{plateaus} = 0.836 \cdot HW(I_{next}) - 45.71$$

$r = 1.00$ and $\rho = 0.99$. The amplitude of the plateaus is the mean of 150 samples between rising/falling edges of Q2 and Q3 respectively. The amplitude of plateau of Q2 is nearly the same with that of Q3, as shown in Figure 2.

We find that the next instruction is still the instruction immediately following the current instruction even if a branch will be taken after executing current instruction (e.g., current instruction is GOTO or BTFSS f,b and bit b of GPR f is 1).

Table 3. Regression analysis of power consumption in Q4

	[3][c]**W** as destination		
Predictors	HD $(data\_loaded\_in\_q2, result)$	$HW(I_{next})$	Constant
Coefficients	2.93	2.15	-25.09
r	0.99		
ρ	0.99		
	[3][c]**f** as destination		
Predictors	HD $(data\_loaded\_in\_q2, result)$	$HW(I_{next})$	Constant
Coefficients	3.60	2.15	-23.78
r	1.00		
ρ	1.00		

Q3. The peak amplitude in Q3 is linear to the HW of next instruction and the HW of current instruction, *regardless of instruction operations*:

$$Y_{q3} = 1.32 \cdot HW(I_{curr}) + 0.828 \cdot HW(I_{next}) - 31.57$$

$r = 1.00$ and $\rho = 1.00$. We find that the current instruction is NOP if current instruction cycle is an inserted cycle after a branch is taken. The $HW(I_{curr})$ is therefore zero, regardless of neighboring instructions. So I_{curr} of current instruction cycle is not necessarily equal to I_{next} of the previous instruction cycle.

Q4. The peak amplitude in Q4 is linear to the HD between data loaded in Q2 and result of current instruction and the HW of next instruction:

$$Y_{q4} = \beta_{q4,1} \cdot HD(data\_for\_current, result) + \beta_{q4,2} \cdot HW(I_{next}) + b_{q4}$$

Table 3 shows the detailed regression models.[7] We find that the linear relationship between internal activities and power consumption measurement is very strong. Operations with file register as destination consume more power than those with the working register as destination.

We are able to uncover the actual data stored in Q4:

- The result of MOVLW is the new W, which has the same value with the operand of the instruction. The HD is therefore always zero.
- The result of GOTO is still the operand (i.e., branch address). The HD is therefore always zero. The next instruction of GOTO is the immediate instruction followed, and is not the instruction at the goto address.

[7] The operations with the working register as destination include instruction types of lw, nop, brnop, goto, fszw, btfs, clrw, wfw. The operations with the GPR as destination include fszf, wff, bxf.

- The results of conditional branches `BTFSC f,b` and `BTFSS f,b` is always zero, regardless of whether branch is to happen or not. (The HD is therefore $HD((f), 0) = HW((f))$.)
- The results of conditional branches `DECFSZ f,d` and `INCFSZ f,d`, in contrast, are the incremented or decremented (f) value. (The HD is therefore $HD((f), mod((f) + 1, 256))$.)
- The result of `brnop` is W, as if a normal `NOP` is executed. The next instruction of `brnop` is the branch destination.
- The results of instructions of types `lw`, `clrw`, `wfw`, `fszw`, `nop` are the new W. The HD for `NOP` is therefore equal to $HW(W)$.
- The file register operations `COMF f,d` and `MOVF f,d` always have constant HDs – the HD for `COMF f,d` is always eight, and for `MOVF` is always zero.

Implications. The side-channel models have several implications. First, they reveal that side-channel measurements have strong dependencies on data and weak dependencies on instruction operations. This explains the result of previous template analyses: while most logic and arithmetic operations are not distinguishable by classification techniques, `COMF f,F` has a 95.7% recognition rate and `CLRW` has a 99.0% recognition rate. `COMF f,F` has the highest power consumption in Q4 because the HD is always eight and the destination is a file register. For `CLRW`, the HW of its instruction is always one, which is unique when file register access is limited to GPRs. Other operations in contrast have mostly overlapping measurements due to internal activities.

Second, they imply that some internal bus, which loads instructions and data, is the main source of power consumption of the μC. Although theoretically every internal node consumes energy when signal switches, the effect of other nodes is negligible.

Third, the dependency in I_{curr} and I_{next} through Q2 to Q4 leaks information about the control flow. While not directly revealing arbitrary neighboring instructions, this already allows us to identify special instructions, such as `NOP` and `brnop` (whose instruction coding is the unique 0).

Fourth, the regression coefficient is in mV per bit, and is large enough to be resilient to measurement noise.

Profiling Electromagnetic Radiation. Although the waveform of EM radiation appears to be different from that of power consumption, it is interesting to observe that regression analysis still works for profiling the EM radiation. The peak amplitude at each clock rising edge is linearly related with the internal activity of the DUT. The resulting regression models for individual clock cycles are only slightly different from those of power consumption. One major difference is the missing of the plateaus following the peaks in Q2 and Q3, which is not surprising since EM measurement will filter out near DC components.

Table 4. Regression analysis of EM radiation in Q2

	[2][c]File register operations	
Predictors	$HD(previous\_result, (f))$	Constant
Coefficients	0.346	3.708
r	0.96	
ρ	0.96	
	[2][c]Literal operations	
Predictors	$HD(previous\_result, literal)$	Constant
Coefficients	0.362	3.209
r	0.91	
ρ	0.91	
	[2][c]`SUBLW`	
Predictors	$HD(previous\_result, literal)$	Constant
Coefficients	0.224	3.385
r	0.94	
ρ	0.88	
	[2][c]`NOP`	
Predictors	$HD(previous\_result, 0)$	Constant
Coefficients	0.312	3.180
r	0.89	
ρ	0.89	
	[2][c]`GOTO`	
Predictors	$HD(previous\_result, address)$	Constant
Coefficients	0.309	2.769
r	0.91	
ρ	0.88	

Note that the EM measurement is not a "localized" one as in [21, 50, 60], in which the probe is focused on only a portion of the chip area to collect particular signals. With current setup, we actually collect global current through the probe coil. The EM waveform is therefore very similar to the power waveform.

We are not trying to extract more information by using EM measurement. Indeed, the goal of this paper is not to develop any new side-channel measurement techniques or to use sophisticated measurement equipment in order to discover new side-channel features. We instead develop new techniques to build more accurate side-channel models which enable us to discover hidden behavior of the DUT, and new protection mechanisms using the side-channel models.

Table 4 shows that in Q2, the measurement of EM radiation is still linear to the HD of previous result and data loaded for current instruction. The only difference is the value of regression coefficients and the model performance r and ρ of the EM models are slightly smaller than those of the power consumption models. It is not surprising since EM measurement often has lower SNR than power measurement.

The peak amplitude of EM measurement in Q3 is linear to the HW of current instruction, *regardless of instruction operations*:

$$Y_{q3} = 0.148 \cdot HW(I_{curr}) + 1.832$$

Table 5. Regression analysis of EM radiation in Q4

	[3][c]**W** as destination		
Predictors	HD $(data\_loaded\_in\_q2, result)$	$HW(I_{next})$	Constant
Coefficients	0.333	0.090	2.563
r	0.91		
ρ	0.90		
	[3][c]**f** as destination		
Predictors	HD $(data\_loaded\_in\_q2, result)$	$HW(I_{next})$	Constant
Coefficients	0.442	0.087	2.738
r	0.99		
ρ	0.99		

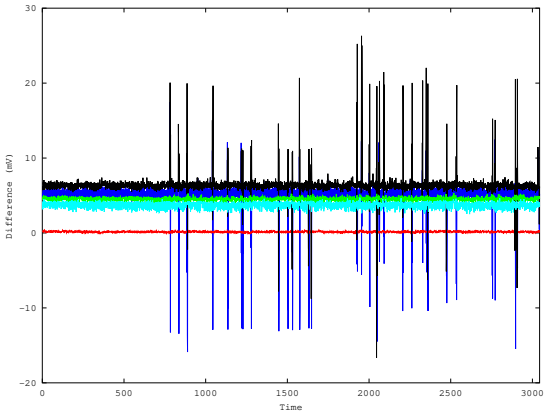

Figure 5. Difference in averaged peak amplitude among chips of the same type.

$r = 0.97$ and $\rho = 0.96$. Note that the EM measurement in Q3 is not related with the HW of next instruction, unlike the case in power consumption.

Table 5 shows again that in Q4, the EM measurement is linear to the HD of data loaded for current instruction and result, and to $HW(I_{next})$.

Profiling and Testing on Different Devices. Up to now we have assumed that the device we use to profile side-channels is also used for testing. Sometimes we may want to profile one device and use the model for testing other devices of the same type. To examine whether the side-channel model can be generalized to other devices, we collect power measurements of six chips running at 1MHz clock. We randomly pick a chip and take its averaged peak amplitude of each clock cycle as the reference and compare it with the power traces of other devices. For clarity, only the averaged difference is shown in Figure 5. A single captured peak is estimated as the averaged amplitude plus a Gaussian noise with standard deviation around 0.84 mV.

It is interesting to observe that among the five testing chips, three of them have similar measurement with the reference plus a constant bias. Two of them have similar measurement except at clock cycles executing `CLRW`. To use a side-channel model on another chip, at least some calibration is needed and clock cycles executing `CLRW` cannot be used for testing.

5. Side-channel Programming

After obtaining an accurate side-channel model, we design the integrity checking scheme. Now we can predict side-channel leakage of arbitrary programs and evaluate the effects of tampered code and/or data on side-channel emanations. For integrity checking, however, the side-channel characteristics do not reveal a direct reverse mapping from side-channel measurements to code and data – it is the Hamming distance/weight rather than the exact data that is leaked from side-channels. For a side-channel-aware attacker, it is possible to compute alternative instructions that have the same HD with previous result in Q2, go through different operations, and again have the same HD with the current result in Q4, since the instruction set coding is quite compact.

The good news is that a change in instruction and/or data may have cascading effects. Tampering with one instruction affects neighboring instructions and/or data, which may be reflected from subsequent side-channel measurements. Given an initial state and a series of instructions, we are able to derive the side-channel measurements, as well as the number of all the possible alternative instructions and final states. If the number of all the possible final states is one, then the device must function as desired. We design the integrity checking scheme following this idea.

5.1. Side-channel Constrained Transitions

From the side-channel model, we derive that the *side-channel-distinguishable* runtime state S is $S = (W, C, FS, D, B)$, where

- W is the working register,
- C is the `STATUS` register,
- $FS = \{fs_w | w = 1, \ldots, 7\}$ is a set of *sets* of GPRs grouped by HW, i.e., the 64 GPRs (indexed f from `0x40` to `0x7f`) are divided into seven sets fs_w according to the HW of index $w = HW(f)$; the GPRs of the same HW are in one set and therefore do not distinguish index order. This is because we cannot distinguish file registers of the same HW (e.g., `0x41` and `0x50`) from side-channel measurements.
- D is the result of current instruction,
- $B \in$ {True,False} is whether a branch is to execute.

S is valid for both power and EM measurements. Note the difference in the definition here and the hypothetical internal state used for side-channel profiling in Section 4.1.

We only need to consider three values for one instruction cycle:

1. $q2 = HD(previous\_result, data\_loaded\_for\_current\_operation)$,
2. $q3 = HW(I_{curr})$, and
3. $q4 = HD(data\_loaded\_for\_current\_operation, current\_result)$.

$q2 \in [0, 8]$ corresponds to the power measurement in Q2 of current instruction cycle, $q3 \in [0, 14]$ corresponds to the power measurement both in Q3 of current instruction cycle and Q2 through Q4 of the previous instruction cycle, and $q4 \in [0, 10]$ corresponds to the power measurement in Q4 of current instruction cycle, which is adjusted from $[0, 8]$ as instructions with the file register as destination consumes more power than instructions with the working register as destination (c.f. Table 3). $q2$, $q3$, $q4$ are also valid for EM measurement.

Given side-channel constraints $(q2, q3, q4)$ and an initial state, we can exhaustively compute all the possible instructions and the resulting states. Since we can identify the unconditional branch operations `call`, `ret`, and `goto` separately by template analysis and also by the `NOP` instruction that always follows, we only consider the literal operations, the file-register operations that perform arithmetic and logic computation, and the conditional branch operations. This includes 28 operations. Both branch and no branch are considered.

For efficient computation, all the possible runtime states for one instruction cycle are stored as a search tree Tr keyed by the runtime state S. The record V of each node of key S is a set of previous state S_{0i} and instructions P_{0i}: $V = \{(S_{01}, P_{01}), \ldots, (S_{0k}, P_{0k})\}$ that result in S. The skeleton of the algorithm that computes a series of possible resulting states, given a trace of side-channel constraints and an initial state, is shown from Algorithm 1 to 4.

Repeating Algorithm 1 will compute all the possible final states in Tr_n, given the initial state $Tr_0 = \{(S_0, null)\}$ and a series of side-channel constraints $\{q_1, q_2, \ldots, q_n\}$, $q_i = (q2_i, q3_i, q4_i)$ for n instruction cycles. If Tr_n only contains one possible state S_n, then we can guarantee that a unique state is reached, no matter what instructions have executed. This leads to the idea of "side-channel programming", in which a program is rewritten in a way that its internal state can be verified by simply checking the side-channels. Note that verifying the code solely cannot rigorously secure a device, due to the existence of code-reuse attacks and data-only attacks.

Our integrity checking problem can therefore be formalized as: given an initial state S_0 and a final

Algorithm 1: `onecycle`: Compute all the possible instructions and resulting states that satisfy a given side-channel constraint $(q2, q3, q4)$ for one instruction cycle.

```
Data: q2, q3, q4, Tr_0
Result: Tr_1
onecycle(q2, q3, q4, Tr_0)
begin
    Tr_1 ⟵ ∅
    for each node (S, V), S = (W, C, FS, D, B) in Tr_0 do
        if B = True (i.e., a branch nop to be executed) then
            if q2 = HW(R) and q3 = 0 and q4 = HW(W) then
                compute resulting S_1 of executing NOP
                addnode(S_1, S, branch_nop, Tr_1)
            end
        end
        else
            procliteral(q2, q3, q4, S, Tr_1)
            procbyte(q2, q3, q4, S, Tr_1)
            procbit(q2, q3, q4, S, Tr_1)
        end
    end
end
```

Algorithm 2: `procliteral`: Find all literal instructions satisfying $(q2, q3, q4)$. Some optimizations omitted.

```
Data: q2, q3, q4, S = (W, C, FS, D, B), Tr
Result: Tr
procliteral(q2, q3, q4, S, Tr)
begin
    for each literal operation op do
        for each operand opr0 that satisfies HD(opr0, R) = q2
          do
            if q3 = HW(opr0) + HW(op) then
                compute resulting S_1 of executing op on the
                  working register
                if adjusted q4 = HD(opr0, R_1) then
                    addnode(S_1, S, {op opr0}, Tr)
                end
            end
        end
    end
    if q3 = 0 and q2 = HW(R) and q4 = HW(W) then  // NOP
        compute resulting S_1 of executing NOP
        addnode(S_1, S, NOP, Tr)
    end
    if q3 = 1 and q2 = HD(R, (0x7f)) and q4 = HW((0x7f))
      then                                          // CLRW
        compute resulting S_1 of executing CLRW
        addnode(S_1, S, CLRW, Tr)
    end
end
```

Algorithm 3: `procbyte`: Find all byte-oriented file register instructions (including conditional branch `DECFSZ` and `INCFSZ`) that satisfy $(q2, q3, q4)$. Some optimizations omitted.

Data: $q2$, $q3$, $q4$, $S = (W, C, FS, D, B)$, Tr
Result: Tr

```
procbyte(q2, q3, q4, S, Tr)
begin
    for each file-register operation op do
        d ⟵ W
                // result stored in the working register
        for each file-register f in fs_w with w = q3 − HW(op)
          do
            if HD(R, (f)) = q2 then
                compute resulting S_1 of executing op on f
                if q4 = HD((f), R_1) then
                    addnode(S_1, S, {op f,d}, Tr)
                end
            end
        end
        d ⟵ F
                                    // result stored in GPR
        q4' ⟵ adjusted q4 for each file-register f in fs_w with
          w = q3 − 1 − HW(op) do
            if HD(R, (f)) = q2 then
                compute resulting S_1 of executing op on f
                if q4' = HD((f), R_1) then
                    addnode(S_1, S, {op f,d}, Tr)
                end
            end
        end
    end
end
```

state S_n, find a trace of side-channel measurements $\{(q2_i, q3_i, q4_i)\}, i = 1, \ldots, n$ of n steps (instruction cycles), such that the transition from S_0 to S_n is unique. Any tampering with the program can either be detected from side-channel measurements, or leads to the same resulting state. For example, when the initial state is

$$S_0 = (16, 000b, \{fs_w | fs_w = \{f_{ij} | f_{ij} = 0\}\}, 32, \text{False})$$

where $000b$ means three binary zeros.

Given the side-channel constraint $\{(1, 8, 1), (7, 10, 6), (1, 7, 4)\}$ for three instruction cycles, only one final state will be reached, namely

$$\begin{aligned} S_3 =& (7, 001b, \\ & \{fs_4 = \{8, 0, \ldots, 0\}, fs_w | fs_w = \{f_{ij} | f_{ij} = 0\}, w \neq 4\}, \\ & 7, \text{False}) \end{aligned}$$

There are 20 possible three-instruction traces that satisfy the side-channel constraints, e.g., {`BSF 0x47,3; ANDLW 0xE7; DECFSZ 0x47,W`}, {`BSF 0x65,3; ANDLW 0xE7; DECFSZ 0x65,W`}, and {`BSF 0x78,3; ANDLW 0xE7; DECFSZ 0x78,W`}. However all the traces lead to the unique final state S_3.

Algorithm 4: `procbit`: Find all bit-oriented file register instructions (including conditional branch `BTFSC` and `BTFSS`) that satisfy $(q2, q3, q4)$. Some optimizations omitted.

Data: $q2$, $q3$, $q4$, $S = (W, C, FS, D, B)$, Tr
Result: Tr

```
procbit(q2, q3, q4, S, Tr)
begin
    if q4 = 0 or 1 then
        for each BCF or BSF operation op do
            for each file-register f do
                if HD(R, (f)) = q2 and
                  w = q3 − HW(f) − HW(op) ∈ [0, 3] then
                    for each bit b ∈ [0, 7] of weight w do
                        compute resulting S_1 of executing
                          op on bit b of f
                        if HD(R_1, (f)) = q4 then
                            addnode(S_1, S, {op f,b}, Tr)
                        end
                    end
                end
            end
        end
    end
    for each BTFSC or BTFSS operation op do
        for each file-register f do
            if HW((f)) = q4 and HD(R, (f)) = q2 and
              w = q3 − HW(f) − HW(op) ∈ [0, 3] then
                for each bit b ∈ [0, 7] of weight w do
                    compute resulting S_1 of executing op on
                      bit b of f
                    addnode(S_1, S, {op f,b}, Tr)
                end
            end
        end
    end
end
```

5.2. Composing the Programs

Because the instruction set coding of the target μC is compact and many instructions having the same HW perform completely different operations, it is not obvious how to obtain side-channel programs, given arbitrary initial state and final state, or even whether it is possible to side-channel program. We adopt a reductionist method: to reach a desired final state from a given initial state, we allow several intermediate states and every pair of neighboring states can be reached by side-channel programs (i.e., unique transformations). Then the left work is to find enough side-channel programs that compose a path from the initial state to the final state.

We can assume that some heuristic function can generate a suitable q based on current runtime state, and the heuristic function will guarantee (at high probability) that the desired unique final state is reached within a few steps. For example, the heuristics may allow more than one possible states to be reached at intermediate steps while forcing the intermediate

Algorithm 5: `uniquetx`: Compute programs that transform a given initial state to a unique final state

Data: S_0, n
Result: S_1

```
uniquetx(S_0, n)
begin
    Tr_i ⟵ ∅, i = 0, ..., n
    add (S_0, null) to Tr_0
    while i ≤ n do
        q ⟵genq(context)
        Tr_{i+1} ⟵ onecycle(q2, q3, q4, Tr_i)
        if size of Tr_{i+1} is one then
            break
        end
        else if size of Tr_{i+1} < threshold then
            i ⟵ i + 1
        end
    end
end
```

states to converge to a unique desired final state. This algorithm is shown in Algorithm 5, where the heuristics $genq(\cdot)$ generates a q for one instruction cycle based on the context. The simplest $genq(\cdot)$ is to generate a random q at each step ignoring current context. An optional throttle value is chosen to limit the size of the search tree at each instruction cycle.

Repeating Algorithm 5 will give a directed graph of states connected by side-channel programs: each node i of the graph is a state S_i, and an edge exists from S_i to S_j only if a side-channel program (of any instructions) exists to uniquely transform S_i to S_j. Since computing the entire graph that connects all the possible runtime states is impractical (and unnecessary), we test the effectiveness of this method by only computing the graphs for representative values.

Random Initial GPRs. First we consider the initial state

$$S_W = (W, 000b, FS, W, \text{False}) \quad (1)$$

where FS is set to random value (note that the last result $D = W$). We compute the graphs of side-channel programs for 20 different FS with $W \in [0, 255]$. For each W, we execute Algorithm 5 200 times, with $genq(\cdot)$ generates random q and $threshold = 100$. We obtain some interesting results.

First, it is surprisingly easy to obtain side-channel programs. Given the maximum number of instruction cycles n to be 2, 4, 8, and 16, the number of side-channel programs generated for any S_W is on average 70, 95, 120, and 136, respectively, for 200 trials. When n is 2, 4, 8, or 16, the side-channel programs result in on average 51, 74, 99, and 115 distinct final state values (as different side-channel programs from the same initial state may reach the same final state). This shows that the side-channel programs compute diverse final states (each by unique transformation) from a given initial state. Among the 51, 74, 99, and 115 distinct final state values S', there are 26, 31, 33, and 33 state values, respectively, that have $FS' = FS$. Others have different FS' that differ for one to several GPR values. There are 21, 26, 28, and 28 state values, respectively, that not only have $FS' = FS$, but also have the resulting data D' equal to the working register W'. There are 10, 12, 12, and 12 state values, respectively, that not only have $FS' = FS$, $D' = W'$, but also have the final STATUS register $C' = 000b$.

We can therefore compose a graph of side-channel programs with each node as $S_W, W \in [0, 255]$. Given n as 2, 4, 8, 16, there are 228, 236, 238, and 238 out of the total 256 S_W that can traverse the entire graph (i.e., transformed to any possible S_W through side-channel programs), by repeating Algorithm 5 for only 200 times per S_W and $genq(\cdot)$ generating random q. Note that the graph is a lower bound of the actual graph with nodes of the form S_W because it does not include the edges that connect two nodes of the form S_W indirectly through nodes not of the form S_W.

Furthermore, since there are 25, 43, 66, and 82 distinct final state values, respectively, that have different FS', the graph of S_W is connected to graphs of different FS' through side-channel programs. Combining the graphs of the same GPRs and of different GPRs will produce a side-channel program that statistically traverses any internal states.

Zero Initial GPRs. Second, we consider special initial states that have all the GPRs of zero values, since after system reboot, all GPRs are initialized with zero. The initial side-channel-distinguishable state is of the form $S_W = (W, 000b, F_0, W, \text{False})$ where $F_0 = \{fs_i | fs_i = \{f_{ij} | f_{ij} = 0\}, i = 1, \ldots, 7\}$, $W \in [0, 255]$, $W = D$. For each W, we execute Algorithm 5 200 times with $n = 8$ and $threshold = 100$. $genq(\cdot)$ generates random q. Then we connect S_i to a final state S_j only if any side-channel program exists resulting in a unique S_j. Note that again the graph does not contain any nodes not of the form S_W. We find that 252 out of the total 256 nodes in the graph can traverse the entire graph, showing that the majority of the runtime states can be reached by side-channel programming, for 200 trials per S_W. The graph of S_W with F_0 is also connected to graphs of GPRs with nonzero values.

5.3. Other Heuristics

The results in Section 5.2 are obtained with the simplest heuristic function: $genq(\cdot)$ uniformly generates q at each step, without considering context. It is free to choose any other heuristic functions. Algorithms 6 and 7 show two other heuristics we have tried.

We consider the initial state to be of the form in Equation 1. Executing Algorithm 5 using Algorithm 6

Algorithm 6: genqec: Generate q by considering context C

Data: C, n
Result: $q2$, $q3$, $q4$

```
genqec(C, n)
begin
    if current step C < n/2 then
        q2 ⟵ rand([3, 5])
        q3 ⟵ rand([5, 9])
    end
    else
        q2 ⟵ rand([0, 2] ∪ [6, 8])
        q3 ⟵ rand([0, 4] ∪ [10, 14])
    end
    q4 ⟵ rand([0, 10])
end
```

Algorithm 7: genqless: Generate q to have fewer possible instructions

Data:
Result: $q2$, $q3$, $q4$

```
genqless()
begin
    q2 ⟵ rand([0, 2] ∪ [6, 8])
    q3 ⟵ rand([0, 4] ∪ [10, 14])
    q4 ⟵ rand([0, 10])
end
```

and 7, respectively, we again compute side-channel-program graphs for 20 different FS with $W \in [0, 255]$. For each S_W, we repeat the computation 200 times, with n set to 8. The numbers of resulting side-channel programs are on average 74 and 180, respectively (i.e., the probability of obtaining a side-channel program of maximally eight instruction cycles is 37% by using Algorithm 6 and 90% by using Algorithm 7). The numbers of distinct final states are on average 66 and 95 (i.e., 33% and 47%), respectively.

Among the 66 distinct final states by using Algorithm 6, on average 6 (actually 5.65) have $FS' = FS$. Others have FS' that differs for one to several GPR values. 5 out of 6 final states not only have $FS' = FS$, but also have $D' = W'$. 2 out of 5 final states further have the final STATUS register $C' = 000b$. The entire side-channel graph for the same FS has on average 25 out of the total 256 states that are connected.

For Algorithm 7, on average 44 out of 95 final states have $FS' = FS$. 35 out of 44 further have $D' = W'$. 13 out of 35 further have $C' = 000b$. The entire side-channel graph for the same FS has on average 204 out of the total 256 states that are connected. Note that above graphs are obtained by considering only the 200 executions of Algorithm 5 with nodes of the same form of S_W.

Table 6. Side-channel programs for transitions among $S_{i,j}$

Target transition	Number of edges	Total number of cycles
$S_{0,0} \rightarrow S_{1,1}$	1	2
$S_{1,1} \rightarrow S_{2,2}$	2	2
$S_{2,2} \rightarrow S_{4,4}$	3	4
$S_{4,4} \rightarrow S_{8,8}$	7	19
$S_{8,8} \rightarrow S_{0x10,0x10}$	6	14
$S_{0x10,0x10} \rightarrow S_{0x20,0x20}$	5	7
$S_{0x20,0x20} \rightarrow S_{0x40,0x40}$	7	17
$S_{0x40,0x40} \rightarrow S_{0x80,0x80}$	3	5

Compared to the results in Section 5.2 which uses random q at each instruction cycle and ignores the context, Algorithm 7 seems to be almost as effective in finding side-channel programs as random q, while Algorithm 6 is surprisingly less effective. Algorithm 7 is more efficient than random q because it takes much less time to compute.

5.4. Example

Here gives an example of side-channel programs. The goal is to output a pattern 0x01, 0x02, ..., 0x80 at a fixed interval. The system initialization includes clearing flags, setting up clock and I/O ports. After initialization, all the GPRs are zero. Writing to the I/O PORTC, for example, with non-zero values uses the code {BCF STATUS,0x5; BCF STATUS,0x6; MOVLW 0xAA; MOVWF PORTC}. The initialization procedure changes W and D. No matter what the values are, W and D can be reset to zero by using CLRW, which can be verified from side-channel measurements since only CLRW satisfies the constraint $q3 = 1$ and $q4 = 0$.[8] The initial state after initialization is therefore

$$S_{0,0} = (0, 000b, F_0, 0, \text{False})$$

To write to PORTC, we only need to change W to each of the target values through side-channel programs. We run Algorithm 5, with Algorithm 7 as $genq(\cdot)$ for 100 times for each $S_{i,j} = (i, 000b, F_0, j, \text{False})$ to obtain edges for the graph of side-channel programs with $FS = F_0$ and STATUS equal to $000b$. The computation takes around 8 hours on a commercial Core i7 computer to run for every $S_{i,j}, i \in [0, 255], j \in [0, 255]$. We are able to find side-channel programs for every transition $S_{0,0} \rightarrow S_{1,1}$, $S_{1,1} \rightarrow S_{2,2}$, ..., $S_{0x40,0x40} \rightarrow S_{0x80,0x80}$. The detailed result is given in Table 6. Column 2 of Table 6 means the number of intermediate nodes $S_{i,j}$ that need to be traversed in order to reach the final state. Column 3 means the total number of instruction cycles required for the target transition.

Because the number of instruction cycles required for each transition is different, padding instructions are needed to output to PORTC at a fixed interval.

[8] Excluding operations on SFRs and unconditional branches

The simplest way is to use NOP, which is the only instruction that satisfies $q3 = 0$. $q2$ and $q4$ can be calculated according to Tables 2 and 3.

6. Conclusion

We have shown that accurate side-channel models of a microcontroller can be built at clock-cycle granularity by iteratively correcting and refining hypothetical semantic models of the instruction set and side-channel models. We even discovered undocumented internal behavior of the device using this approach.

We further proposed a novel software integrity checking method, side-channel programming, in which a program is rewritten in a way that its internal state can be verified by simply checking the passive side-channel emanations. Side-channel information in general cannot be used directly for integrity checking because a side-channel-aware attacker may be able to write malware that has indistinguishable side-channel emission with the legitimate code. The side-channel programming approach shown in this paper has the unique feature that as long as the side-channel constraint is satisfied, any code starting from the given initial state is guaranteed to reach a single final state. We have illustrated in detail on how to generate "side-channel programs" in combination with the side-channel constraints. Compared to existing protection mechanisms, our approach does not interrupt device execution or require hardware modification, and is secure against side-channel-aware attackers.

References

[1] M. Dam, R.G. and Nemati, H. (2013) Machine code verification of a tiny ARM hypervisor. In *TrustED*.

[2] Dam, M., Guanciale, R., Khakpour, N., Nemati, H. and Schwarz, O. (2013) Formal verification of information flow security for a simple ARM-based separation kernel. In *ACM CCS*.

[3] L. Davi, A.-R. Sadeghi, D.L. and Monrose, F. (2014) Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*.

[4] Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K. and Franz, M. (2015) Opaque control-flow integrity. In *NDSS*.

[5] Gu, L., Ding, X., Deng, R., Xie, B. and Mei, H. (2008) Remote attestation on program execution. In *ACM STC*.

[6] Francillon, A., Nguyen, Q., Rasmussen, K. and Tsudik, G. (2014) A minimalist approach to remote attestation. In *DATE*.

[7] Asokan, N., Brasser, F., Ibrahim, A., Sadeghi, A.R., Schunter, M., Tsudik, G. and Wachsmann, C. (2015) SEDA: Scalable embedded device attestation. In *ACM CCS*.

[8] Butterworth, J., Kallenberg, C., Kovah, X. and Herzog, A. (2013) BIOS chronomancy: Fixing the core root of trust for measurement. In *ACM CCS*.

[9] Mahmood, A. and McCluskey, E. (1988) Concurrent error detection using watchdog processors – a survey. *Trans. on Computers* **37**(2).

[10] Clark, S., Ransford, B., Rahmati, A., Guineau, S., Sorber, J., Fu, K. and Xu, W. (2013) WattsUpDoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In *USENIX HealthTech*.

[11] Moreno, C., Fischmeister, S. and Hasan, M. (2013) Non-intrusive program tracing and debugging of deployed embedded systems through side-channel analysis. In *ACM LCTES*.

[12] Liu, Y., Wei, L., Zhou, Z., Zhang, K., Xu, W. and Xu, Q. (2016) On code execution tracking via power side-channel. In *ACM CCS*.

[13] Nazari, A., Sehatbakhsh, N., Alam, M., Zajic, A. and Prvulovic, M. (2017) EDDIE: EM-based detection of deviations in program execution. In *ISCA*.

[14] Han, Y., Etigowni, S., Liu, H., Zonouz, S. and Petropulu, A. (2017) Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations. In *ACM CCS*.

[15] Seshadri, A., Perrig, A., Doorn, L.V. and Khosla, P. (2004) SWATT: SoftWare-based ATTestation for embedded devices. In *IEEE Symposium on Security and Privacy*.

[16] Li, Y., McCune, J. and Perrig, A. (2011) VIPER: Verifying the Integrity of PERipherals' firmware. In *ACM CCS*.

[17] Armknecht, F., Sadeghi, A.R., Schulz, S. and Wachsmann, C. (2013) A security framework for the analysis and design of software attestation. In *ACM CCS*.

[18] Zhang, F., Wang, H., Leach, K. and Stavrou, A. (2014) A framework to secure peripherals at runtime. In *ESORICS*.

[19] Liu, H., Li, H. and Vasserman, E. (2015) Practicality of using side-channel analysis for software integrity checking of embedded systems. In *SecureComm*, *LNICST* **164**.

[20] Liu, H. and Vasserman, E. (2017) Gray-box software integrity checking via side-channels. In *SecureComm*, *LNICST* **238**.

[21] Skorobogatov, S. *Semi-invasive attacks – A new approach to hardware security analysis*. Tech. rep., University of Cambridge.

[22] Cui, A., Kataria, J. and Stofo, S. (2011) From prey to hunter: Transforming legacy embedded devices into exploitation sensor grids. In *ACSAC*.

[23] Muralimanohar, N., Balasubramonian, R. and Jouppi, N. (2009), CACTI 6.0: A tool to model large caches.

[24] Goeders, J. and Wilton, S. (2012) VersaPower: Power estimation for diverse FPGA architectures. In *FPT*.

[25] Li, F., Lin, Y., He, L., Chen, D. and Cong, J. (2005) Power modeling and characteristics of field programmable gate arrays. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **24**(11).

[26] Duan, C., Cordero, V. and Khatri, S. (2009) Efficient on-chip crosstalk avoidance CODEC design. *IEEE VLSI* .

[27] Poon, K., Wilton, S. and Yan, A. (2005) A detailed power model for field-programmable gate arrays. *ACM Trans. on Design Automation of Electronic Systems* **10**(2).

[28] Anderson, J. and Najm, F. (2004) Power estimation techniques for FPGAs. *IEEE Trans. on Very Large Scale Integration Systems* **12**(10).

[29] Kadric, E., Lakata, D. and DeHon, A. (2015) Impact of memory architecture on FPGA energy consumption. In *ACM/SIGDA FPGA*.

[30] Tiwari, V., Malik, S., Wolfe, A. and Lee, M.C. (1996) Instruction level power analysis and optimization of software. In *International Conference on VLSI Design*.

[31] Zipf, P., Hinkelmann, H., Deng, L., Glesner, M., Blume, H. and Noll, T. (2007) A power estimation model for an FPGA-based softcore processor. In *FPL*.

[32] Senn, L., Senn, E. and Samoyeau, C. (2012) Modelling the power and energy consumption of NIOS II softcores on FPGA. In *IEEE International Conference on Cluster Computing Workshops*.

[33] Unknown Author (1972) TEMPEST: A signal problem. *NSA Cryptologic Spectrum* .

[34] Agrawal, D., Baktir, S., Karakoyunlu, D., Rohatgi, P. and Sunar, B. (2007) Trojan detection using IC fingerprinting. In *IEEE Symposium on Security and Privacy*.

[35] Jin, Y. and Makris, Y. (2008) Hardware trojan detection using path delay fingerprint. In *IEEE HOST*.

[36] Song, P., Stellari, F., Pfeiffer, D., Culp, J., Weger, A., Bonnoit, A., Wisnieff, B. *et al.* (2011) MARVEL: Malicious alteration recognition and verification by emission of light. In *IEEE HOST*.

[37] Skorobogatov, S. and Woods, C. (2012) Breakthrough silicon scanning discovers backdoor in military chip. In *CHES*.

[38] Soll, O., Korak, T., Muehlberghuber, M. and Hutter, M. (2014) EM-based detection of hardware trojans on FPGAs. In *IEEE HOST*.

[39] Mangard, S., Oswald, E. and Popp, T. (2010) *Power Analysis Attacks: Revealing the Secrets of Smart Cards* (Springer).

[40] Kocher, P., Jaffe, J., Jun, B. and Rohatgi, P. (2011) Introduction to differential power analysis. *Journal of Cryptographic Engineering* **1**(1).

[41] Whitnall, C. and Oswald, E. (2015) Robust profiling for DPA-style attacks. In *CHES, LNCS* **9293**.

[42] Gonzalez, C. (2011) *Power Fingerprinting for Integrity Assessment of Embedded Systems*. Ph.D. thesis, Virginia Polytechnic Institute and State University.

[43] Yang, Y., Su, L., Khan, M., Lemay, M., Abdelzaher, T. and Han, J. (2014) Power-based diagnosis of node silence in remote high-end sensing systems. *ACM Trans. on Sensor Networks* **11**(2).

[44] Stone, S., Temple, M. and Baldwin, R. (2015) Detecting anomalous programmable logic controller behavior using RF-based Hilbert transform features and a correlation-based verification process. *International Journal of Critical Infrastructure Protection* **9**.

[45] Quisquater, J.J. and Samyde, D. (2002) Automatic code recognition for smart cards using a Kohonen neural network. In *Smart Card Research and Advanced Application Conference*.

[46] Vermoen, D., Witteman, M. and Gaydadjiev, G. (2007) Reverse engineering Java card applets using power analysis. In *Smart Cards, Mobile and Ubiquitous Computing Systems*, WISTP.

[47] Goldack, M. (2008) *Side-Channel Based Reverse Engineering for Microcontrollers*. Master's thesis, Ruhr-Universität Bochum, Germany.

[48] Eisenbarth, T., Paar, C. and Weghenkel, B. (2010) Building a side channel based disassembler. In *Trans. on Computational Science X*.

[49] Msgna, M., Markantonakis, K., Naccache, D. and Mayes, K. (2014) Verifying software integrity in embedded systems: A side channel approach. In *Constructive Side-Channel Analysis and Secure Design*, LNCS.

[50] Strobel, D., Oswald, D., Richter, B., Schellenberg, F. and Paar, C. (2014) Microcontrollers as (in)security devices for pervasive computing applications. *Proc. IEEE* **102**(8).

[51] Strobel, D., Bache, F., Oswald, D., Schellenberg, F. and Paar, C. (2015) Scandalee: A side-channel-based disassembler using local electromagnetic emanations. In *DATE*.

[52] Microchip Technology Inc. (2008), PIC16F631/677/685/687/689/690 data sheet.

[53] Microchip Technology Inc. (1997), PICmicro mid-range MCU family – reference manual.

[54] Ott, H. (2009) *Electromagnetic Compatibility Engineering* (Wiley).

[55] Chari, S., Rao, J. and Rohatgi, P. (2003) Template attacks. In *CHES, LNCS* **2523**.

[56] Schindler, W., Lemke, K. and Paar, C. (2005) A stochastic model for differential side channel cryptanalysis. In *CHES, LNCS* **3659**.

[57] Kasper, M., Schindler, W. and Stöttinger, M. (2010) A stochastic method for security evaluation of cryptographic FPGA implementations. In *FPT*.

[58] Brier, E., Clavier, C. and Olivier, F. (2004) Correlation power analysis with a leakage model. In *CHES, LNCS* **3156**.

[59] Montgomery, D., Peck, E. and Vining, G. (2012) *Introduction to Linear Regression Analysis* (Wiley), 5th ed.

[60] Sauvage, L., Guilley, S. and Mathieu, Y. (2009) Electromagnetic radiations of FPGAs: High spatial resolution cartography and attack on a cryptographic module. *ACM Trans. on Reconfigurable Technology and Systems* **2**(1).