Detailed Tracking of Program Control Flow Using Analog Side-Channel Signals: A Promise for IoT Malware Detection and a Threat for Many Cryptographic Implementations

Haider Adnan Khan[a], Monjur Alam[b], Alenka Zajic[a], and Milos Prvulovic[b]

[a]The School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA
[b]The School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA

ABSTRACT

Side-channel signals have long been used in cryptanalysis, and recently they have also been utilized as a way to monitor program execution without involving the monitored system in its own monitoring. Both of these use-cases for side-channel analysis have seen steady improvement, allowing ever-smaller deviations in program behavior to be monitored (to track program behavior and/or identify anomalies) or exploited (to steal sensitive information). However, there is still very little intuition about where the limits for this are, e.g. whether a single-instruction or a single-bit difference can realistically be recovered from the signal.

In this paper, we use a popular open-source cryptographic software package as a test subject to demonstrate that, with enough training data, enough signal bandwidth, and enough signal-to-noise ratio, the decision of branch instructions that cause even single-instruction-differences in program execution can be recovered from the electromagnetic (EM) emanations of an IoT/embedded system. We additionally show that, in cryptographic implementations where branch decisions contain information about the secret key, nearly all such information can be extracted from the signal that corresponds to only a single cryptographic operation (e.g. encryption). Finally, we analyze how the received signal bandwidth, the amount of training, and the signal-to-noise ratio (SNR) affect the accuracy of side-channel-based reconstruction of individual branch decisions that occur during program execution.

Keywords: cryptanalysis, side-channels, RSA, program execution monitoring, control-flow tracking

1. INTRODUCTION

In cryptanalysis, the term *side channel attack* refers to an attack that extracts sensitive information, such as information about the secret cryptographic key, by observing implementation-dependent events and/or signals that are not part of the algorithm's specified input/output behavior. A wide variety of side channels have been demonstrated, using information obtained by measuring execution time,[1–4] monitoring the behavior of caches[5–7] and other performance-oriented features[8] within the processor, and by analyzing analog signals produced (or affected) by the computer system's physical implementation, such as power consumption,[9–14] electromagnetic (EM) emanations,[15–17] acoustic emanations (sound),[18–20] etc.

Recent work also shows that analog side channels can be used for benevolent purposes, as a way to monitor program execution. Examples of this include analysis of the power consumption signal to detect battery-draining attacks in mobile devices,[21–24] integrity assessment of Software Defined Radio (SDR) devices,[25] and detection of malware in medical devices.[26] Even more recently, the EM side channel has also been used to monitor systems for malware detection[27] and for performance analysis (profiling),[28,29] without any changes to or interaction with the monitored system.

Both cryptanalysis and software-monitoring uses of side channels have seen steady improvement, allowing ever-smaller differences in program behavior to be detected during monitoring or exploited during cryptanalysis. However, it is still unclear where the limits for this are, e.g. whether a difference of just a few instructions (or

Send correspondence to Milos Prvulovic; E-mail: milos@cc.gatech.edu

even a single instruction) in program execution can be identified from the side channel signal, and how these limits depend on the rate at which the signal is observed, the amount of training that is available, and the quality of the signal (the signal-to-noise ratio).

In this paper, we experimentally verify that, given enough signal bandwidth, training data, signal-to-noise ratio, and knowledge about the program code itself, even single-instruction differences in program execution can be identified with very high accuracy. The software we use for these experiments is an open-source implementation of the RSA public-key cryptosystem or, more precisely, the modular exponentiation used by the RSA implementation. This choice of software test subject has several key advantages. First, RSA's modular exponentiation has already been subjected to numerous side channel attacks, so each branching decision (a decision to either continue to the next instruction in the program or jump to another instruction in the program) in its program code is extremely well understood. Second, many branching decisions depend on the bits of the secret exponent in RSA decryption, so our experimental results are directly relevant for cryptanalysis. Finally, the implementation already contains mitigation for previously demonstrated side channel attacks, so the individual (secret-key-dependent) differences in program execution are very small and largely independent of each other, so our identification of each small difference in execution is not aided by signal changes caused by other correlated differences.

We use the EM side channel for these experiments primarily because it can provide high signal bandwidth, and that bandwidth can be changed at the receiver, which means that the rate at which the signal is sampled can be meaningfully varied. The ability to meaningfully vary signal bandwidth is shared by other analog side channels, whereas the timing side channel provides only one measurement for an entire decryption, and cache and other processor-feature side channels rely on detecting discrete events, so their "sampling rate" is much more difficult to meaningfully vary. Among analog side channels, we selected the EM channel because it has a high usable bandwidth and a non-intrusive way to obtain the signal. Indeed, in our experiments we use up to 160 MHz of bandwidth, which is at least an order of magnitude more than the acoustic channel can provide, and we collect the signal by simply placing the EM probe close to the monitored system, rather than inserting current measurement probes into the power supply circuitry around the monitored system's processor.

Our experimental results show that fine-grained differences in program execution (only a few instructions, or even one instruction when the surrounding code exhibits very stable behavior) can be identified with >90% accuracy, even when receiving the signal at a rate that is only 4% of the monitored system's clock cycle rate, as long as the signal quality is relatively good (a signal-to-noise ratio of 20 dB or better). We also find that accuracy further improves as bandwidth is increased and when the signal-to-noise ratio is improved. Another finding is that larger differences (tens of instructions) in program execution can be identified with >99% accuracy even when using bandwidth that is only 2% of the processor's clock cycle rate, and even with relatively poor signal quality (a signal-to-noise ratio of only 5 dB). Finally, we find that accuracy improves rapidly with more training up to a point and then mostly saturates once about one thousand training examples are available for possible execution of each small fragment of the monitored code.

Overall, we find that each branch decision in program execution can be recovered through the EM side channel with high accuracy, using bandwidth that is easily provided by sub-$1,000 equipment, but with training data that uses several orders of magnitude more memory than the monitored program's code (instructions). For cryptographic implementations, our results imply that secret-key-dependent branching should not be used - the training data is easy to obtain by simply performing cryptographic operations with known keys and, because the key parts of the program code are compact, the amount of training data is still small enough to fit in memory of most laptop/desktop systems. For program monitoring, our results imply that highly accurate tracking of program execution at basic-block and single-instruction granularity should be possible even with relatively compact and low-cost equipment, and that the likely main obstacles will be how to obtain sufficient training for each part of the code and how to represent the training data for large programs in a more compact form.

The rest of this paper is organized as follows. Section 2 describes the modular exponentiation implementation that is used as a monitoring subject in our experiments, Section 3 describes our method for side-channel-based branch decision identification, Section 4 details obtained results, and Section 5 provides some concluding remarks.

2. MONITORED SOFTWARE - MODULAR EXPONENTIATION IN OPENSSL'S

The program code whose branching decisions we will try to recover through side channel analysis is OpenSSL's implementation of sliding window modular exponentiation (function `BN_mod_exp_simple` from `bn_exp.c` in the OpenSSL source code). The listing below shows the main part of this code, with some redaction (mainly removal of error checking) to enhance clarity:

```
1  wvalue = 0;                    /* The 'value' of the window */
2  wstart = bits - 1;             /* The top bit of the window */
3  wend = 0;                      /* The bottom bit of the window */
4
5  BN_one(r);
6
7  for (;;) {
8    if (BN_is_bit_set(p, wstart) == 0) {
9      BN_mod_mul(r, r, r, m, ctx);
10     if (wstart == 0)
11             break;
12     wstart--;
13     continue;
14   }
15
16   /*
17   * We now have wstart on a 'set' bit, we now need to work out how big
18   * of a window to do.  To do this we need to scan forward until the last
19   * set bit before the end of the window
20   */
21
22   j = wstart;
23   wvalue = 1;
24   wend = 0;
25   for (i = 1; i < window; i++) {
26     if (wstart - i < 0)
27       break;
28     if (BN_is_bit_set(p, wstart - i)) {
29       wvalue <<= (i - wend);
30       wvalue |= 1;
31       wend = i;
32     }
33   }
34
35   /* wend is the size of the current window */
36   j = wend + 1;
37   for (i = 0; i < j; i++)
38     BN_mod_mul(r, r, r, m, ctx);
39
40   /* wvalue will be an odd number < 2^window */
41   BN_mod_mul(r, r, val[wvalue >> 1], m, ctx);
42
43   /* move the 'window' down further */
44   wstart -= wend + 1;
45   wvalue = 0;
46   if (wstart < 0)
47     break;
48 }
```

This program code computes $v^p mod m$, where v is the(encrypted) message, p is the (secret) exponent, and

m is the modulus. Conceptually, it follows the grade-school exponentiation approach, where the exponent is examined starting from the most significant bit, squaring the result for each bit of the exponent, and multiplying the result with the message when the bit of the exponent has a value of one. However, rather than process one bit of the exponent at a time, the sliding window algorithm splits the exponent into multi-bit chinks called *windows* and performs multiplication with the (appropriately pre-exponentiated) message an entire window at a time. In preparation for this, for each possible value of the window (*wvalue*), the value of v^{wvalue} *mod* m has been pre-computed and placed in the table *val* at an index that corresponds to *wvalue*.

Initially, the very first window starts with a value of 0 (line 1), its most significant bit is the most significant bit of the exponent (line 2), and the window contains one bit from the exponent (line 3, note that the value of *wend* is always one less than the number of bits in the window). Like the message and the exponent, the result r is a very large number (>1,000 bits) that is kept in a data structure that contains an array of 32-bit integers, so function *BN_one* sets the large number to a value of 1. The main loop of the exponentiation begins (line 7), and in each iteration of this a window is formed and the result is updated for that window. In this algorithm, a multi-bit window must begin and end with a non-zero bit. Therefore, line 8 examines the exponent's bit at the starting position of the window. If that bit is zero, that bit alone will be a (zero-valued) window. This means that the result should be squared (line 9) but no multiplication with the message is needed, so a new window will begin at the next bit position (line 12) and a new iteration of the main loop is begun (line 13). Note that, if the bit that was just examined was the last (least significant) bit of the exponent (line 10), the entire exponentiation is now complete (line 11).

If the starting bit of the window is not zero, the code at lines 22 through 33 attempts to form a multi-bit window. Since the non-zero bit begins the window, the value of the window is now 1 (line 23), and the loop at line 25 iterates over the bits that are examined for potential inclusion into the window. Variable i is the number of bits that have been examined, and it starts at 1 because the very first (most significant) bit of the window has already been examined (and found to be 1). Line 26 checks if the would-be window would go past the last (least significant) bit of the exponent, in which case the window cannot be expanded further (line 27). Line 28 examines the next candidate bit. Since a window cannot end with a 0-valued bit, a 0-valued bit is simply skipped without expanding the window. However, a 1-valued bit causes the window to be expanded to include that bit, as well as all the zero-valued bits between it and the rest of the window. This is done by shifting the *wvalue* by enough bit positions to make room for the bits that are being included (line 29), setting the least significant bit of *wvalue* to 1 (line 30) because the new least significant bit of the window is 1 – note that the other bits that are being included into the window are all zero-valued, otherwise they would have been included when they were encountered – and setting *wend* (line 31) to the new size of the window. The loop at line 25 ends when the number of bits examined for the current window reaches the maximum size allowed for a window. Variable *window* stores this maximum size, and it is relatively small (5 or 6 for typical RSA key sizes) so that the table *val* (that contains a large number for every possible value of the window) is still small enough to fit in the processor's data cache (which is good for performance). Not every window has the maximum size - the window's size depends on when a one-valued bit was last included. For example, if the maximum winodw size is 6 (i.e. when $window = 6$), when the bits examined for the window are 100000, the window only has one bit (the initial 1), when the examined bits are 110000 the window has only two bits (11), when the examined bits 111000 or 101000 the resulting window has 3 bits (111 or 101), etc.

Once the window is formed, the result is squared for each bit in the window (lines 36-38) and then multiplied (line 41) with the value from table *val* that was pre-computed by exponentiating the message with the value of the window. Finally, *wstart* and *wvalue* are updated to begin a new window (lines 44 and 45), and another iteration of the main loop begins to form another window. The exception to this is when the just-processed window included the last bit of the exponent, in which case the entire exponentiation is complete (lines 46 and 47).

In prior cryptanalysis work, a cache-based attack[30,31] was used to identify the sequence of squaring and multiplications, while for analog side channels the squaring and multiplications were identified in the signal by using especially chosen messages[32,33] that create a large difference in their side channel signals. Since each squaring corresponds to moving one bit toward the least significant bit of the exponent, and the multiplication corresponds to the end of a non-zero window, the sequence of squarings and multiplications reveals the position of

each 1-valued bit that ends a window. Furthermore, when the number of squarings between two multiplications is less than the maximum allowed size of the window, that indicates the number of zero-valued bits that follow the last bit that was included in the window. By iteratively applying such exponent-reconstruction rules to the sequence of squarings and mulptiplications, a significant percentage of the exponent's bits were recovered - 49% of the bits when the maximum window size is 4, and this percentage of recovered bits declines somewhat when the maximum window size is larger.

To avoid these attacks that rely on exponent-dependent sequence of squarings and multiplications, OpenSSL has switched to a *fixed-window* exponentiation, where all windows are of the same size and all possible values of the window (including the all-zeros window) are represented in the pre-computed table *val*. This results in multiplications that are always separated by an equal number of squarings, regardless of the exponent. Furthermore, because every bit up to the window size is included in the window, there are no branch decisions that depend on the bits of the exponent - in fact, the sequence of *all* branch decisions is always the same for the enitre exponentiation, regardless of the exponent. The listing below show the main part of the exponentiation code (in function `BN_mod_exp_mont_consttime` within the `bn_exp.c` file in OpenSSL's source code), again with some redaction (mainly removing error-checking) for clarity:

```
1  while (bits >= 0) {
2    wvalue = 0;          /* The 'value' of the window */
3    /* Scan the window, squaring the result as we go */
4    for (i = 0; i < window; i++, bits--) {
5      BN_mod_mul_montgomery(&tmp, &tmp, &tmp, mont, ctx);
6      wvalue = (wvalue << 1) + BN_is_bit_set(p, bits);
7    }
8
9    /* Fetch the appropriate pre-computed value from the pre-buf */
10   MOD_EXP_CTIME_COPY_FROM_PREBUF(&am, top, powerbuf, wvalue,window);
11   /* Multiply the result into the intermediate result */
12   BN_mod_mul_montgomery(&tmp, &tmp, &am, mont, ctx);
13 }
```

Note that this code uses the more efficient Montgomery multiplication algorithm (`BN_mod_mul_montgomery`), and it improves resilience to cache-based side channel attacks using a more complicated organization of the lookup table (the variable that refers to it is named *powerbuf* in this code, rather than *val*) that requires a specialized function `MOD_EXP_CTIME_COPY_FROM_PREBUF` to retrieve the table entry that corresponds to *wvalue* and place it into *am* so it can be multiplied with the result.

Because the existing branch decisions in this code leak no information about the exponent, we change the code so that line 6 is replaced by

```
1      wvalue = (wvalue << 1)
2      if(BN_is_bit_set(p, bits))
3        wvalue|=1;
```

Note that this introduces an exponent-dependent branch decision which creates a single-instruction difference in program execution, so it will allow us to experimentally assess how accurately a single-instruction difference in execution can be identified from the side-channel signal.

3. SIDE-CHANNEL-BASED BRANCH DECISION IDENTIFICATION

The identification method consists of two phases: the training phase and the detection phase. In the training phase, we learn the EM signatures corresponding to each branch decision by executing encryption with known keys. In the detection phase, we use the learned EM signatures from training to identify which branch decision is the best match for the EM signal observed during detection. In both cases, we need to capture and pre-process the signals before running our detection algorithm.

3.1 Acquisition of Modulated EM Signals

Unintentional EM emanations occur at various frequencies, but of particular importance is the frequency band centered around the clock frequency of the device's processor and memory. This frequency band contains signals that are primarily a function of the instruction sequence executed by the CPU. Each processor cycle, the CPU draws a current which is a direct result of the instruction(s) being executed. Much of this instruction-dependent current is drawn by the CPU clock circuitry and by circuitry which does new computations (i.e. switches on and off) every CPU clock cycle. This creates a strong current at the CPU clock frequency, which acts as a carrier modulated by the clock-cycle-to-clock-cycle variations in program activity (i.e. executed instructions). When its EM signal is observed this way, the computer system has much in common with a communications system: the processor is a transmitter which (inefficiently and unintentionally) transmits a carrier (i.e. the clock signal) that is amplitude-modulated by a program's activity as it causes activity in the processor's circuitry. We can then receive and demodulate this signal using wireless communications techniques. All EM signals, both in the training phase and in the monitoring phase, are first AM demodulated at the processor clock frequency, low-pass filtered, and sampled before being sent for signal processing.

3.2 Signal Processing

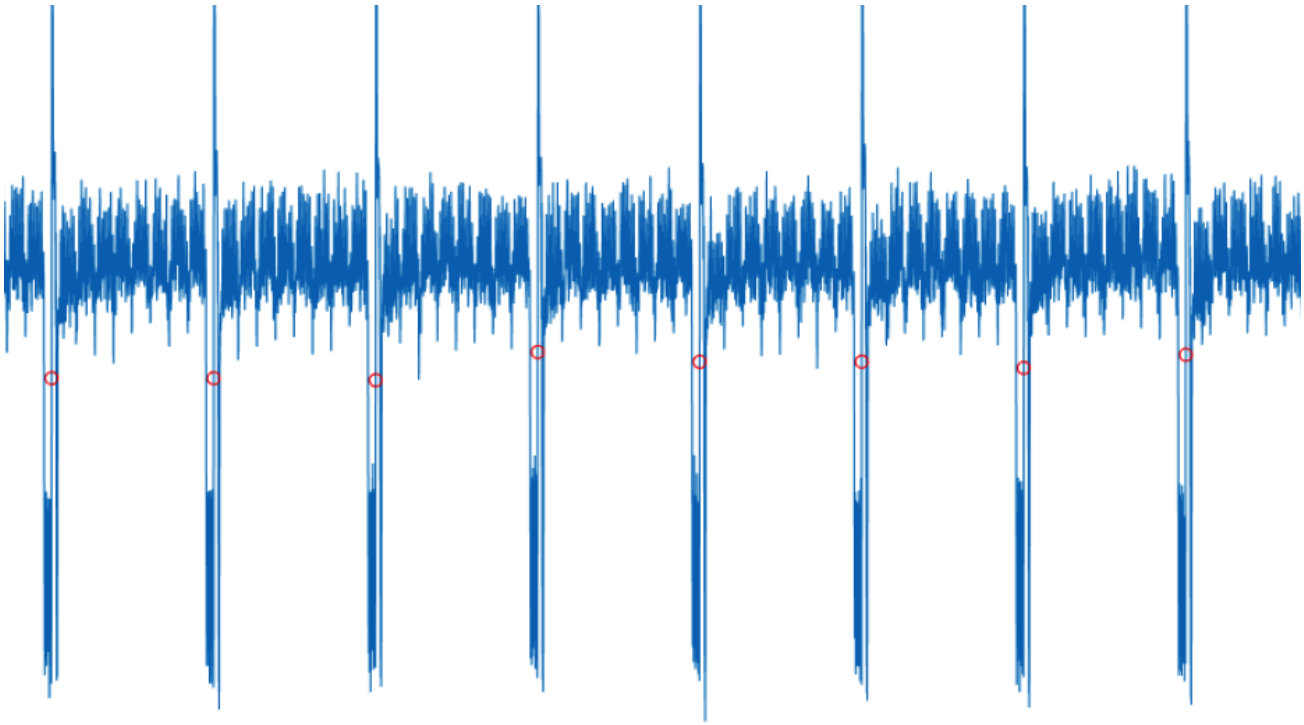

Figure 1. The EM signature of *BN_mod_mul_montgomery* function.

The first step in both training and testing is to identify location where either control-flow branching or window value calculation takes place in the received signal. Since the modular multiplication of large numbers (functions `BN_mod_mul` and `BN_mod_mul_montgomery`) has a fixed sequence of branch decisions and is thus of no interest for our experiments, we find the signature of this multiplication and identify the part of its signal that has a prominent and abrupt change in the signal (see Figure 1). Because modular multiplication was designed to have almost no timing variation, the end of the signal that corresponds to it can be found at a fixed time offset from the point of this match. The signal that we use for training and for detection of exponent-dependent branches consists of the snippets of signal between each ending of a modular multiplication and the beginning of the next one.

3.3 Training Phase

In the training phase, we execute encryption with known keys, select the snippets of signal between modular multiplications, and use them to create a "dictionary" of reference EM patterns for each possible combination of branch outcomes within the code the snippet corresponds to. We first label each snippet according to which modular multiplications it has at its beginning and its end, and treat these snippets as transitions in a state machine whose states are the modular multiplications as shown in Figure 2. In addition to this, the transition from state A to state B (which corresponds to examining the candidate bits for a non-zero window) has 32 different sub-labels that correspond to all possible combinations of outcomes for the five branch outcomes that examine on the candidate bits. Since training is performed with known exponents, the labels for all the snippets collected during training are also known, so the output of the training is a "dictionary" that contains a large number of labeled signal snippets.

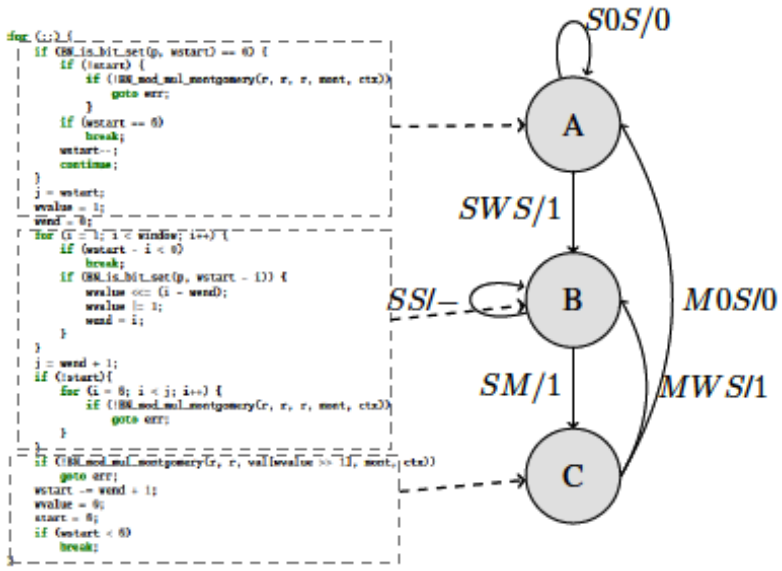

Figure 2. Labeling of the signal snippets for sliding-window exponentiation.

3.4 Prediction Phase

During the detection phase, just like in training, we first identify and extract the snippets of the signal that correspond to execution between modular multiplications. Then, for each snippet encountered in the detection phase, we use the 1-Nearest Neighbor (1-NN) algorithm, with Euclidean distance as the distance metric, to find which snippet in the dictionary is the closest match for that detection-phase snippet, and use the label of that dictionary snippet to label the detection-time snippet.

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

We run the OpenSSL's RSA decryption on an embedded device (A13-OLinuXino board[34]). The A13- OLinuXino board is a single-board computer that has an in-order 2-issue Cortex A8 ARM processor,[35] and it uses the Debian Linux operating system.

Signals are received using small magnetic probe. We place the probe close to the monitored system as shown in Fig. 4.1. The signals collected by the probe are recorded with Keysight N9020A MXA spectrum analyzer.[36] Our decision to use spectrum analyzer was mainly driven by its existing features such as built-in support for automating measurements, saving and analyzing measured results, visualizing the signals when debugging code, etc. We have observed very similar signals when using less expensive equipment such as Ettus USRP B200-mini receiver.[37] The analysis was implemented in MATLAB and, on a personal computer, takes less than one minute to reconstruct the exponent-dependent branch decisions encountered in one 1024-bit modular exponentiation.

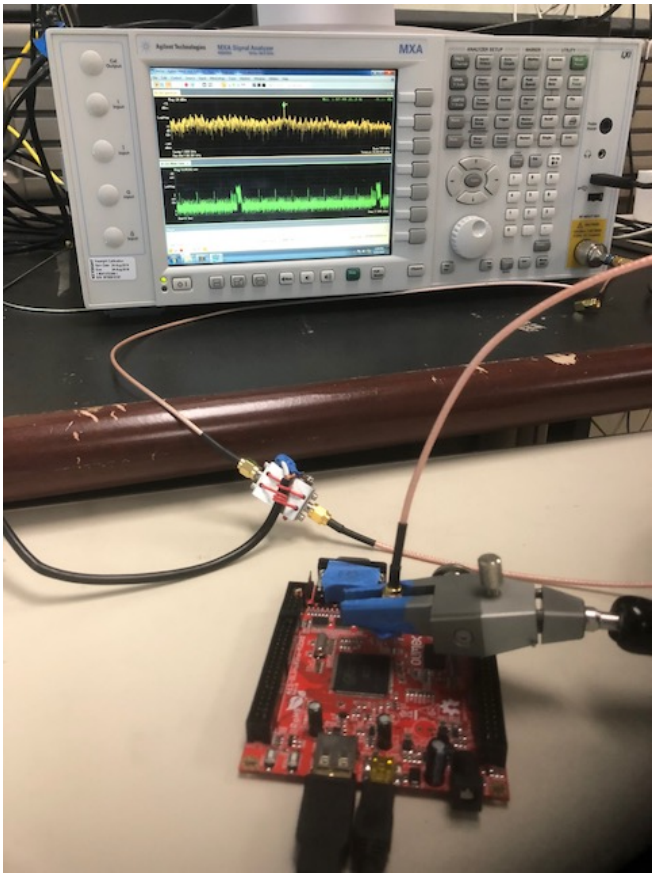

Figure 3. Our measurement setup.

4.2 Impact of Signal Bandwidth on Branch Decision Reconstruction

The branch decisions we are interested in are the exponent-dependent branches in the sliding window implementation and in the modified fixed window implementation. These can be clustered into three categories:

First, branches at lines 8 and 37 in the sliding-window implementation result in relatively large changes in what is executed after them. The branch at line 8 results in either calling `BN_mod_mul` at line 9, or in executing the entire window-forming loop (lines 22-33), followed by lines 36 and 37 before `BN_mod_mul` is entered at line 38. The loop branch at line 37 results in either calling `BN_mod_mul` at line 38, or in exiting the loop, calling `BN_mod_mul` at line 41, and then executing the code at lines 43-47 and entering another iteration of the main exponentiation loop. Compared to the branch at line 8, the outcome of this branch is more difficult to identify using the side channel signal because the long-lasting `BN_mod_mul` is called almost immediately after this branch, regardless of its outcome, and the largest difference that results from its outcome occurs only *after* `BN_mod_mul` returns.

Second, the branch at line 28 in the sliding window implementation results in either executing the code at lines 29-31, which consists of only four processor instructions, or not executing these four instructions. After that the two options converge.

Finally, the exponent-dependent branch in our modified fixed-window implementation creates only a single-instruction (a bitwise OR instruction) difference in what is executed by the processor.

Figure 4 shows the accuracy of identifying (reconstructing) branch decisions for these three kinds of branches, when the signal is received using 20 MHz, 30 MHz, 40 MHz, 80 MHz, and 160 MHz of bandwidth. Note that the processor clock frequency is 1 GHz, so these bandwidth values correspond to only 2%, 3%, 4%, 8%, and 16% of the processor's clock frequency. From these results we can see that branch decisions which create large

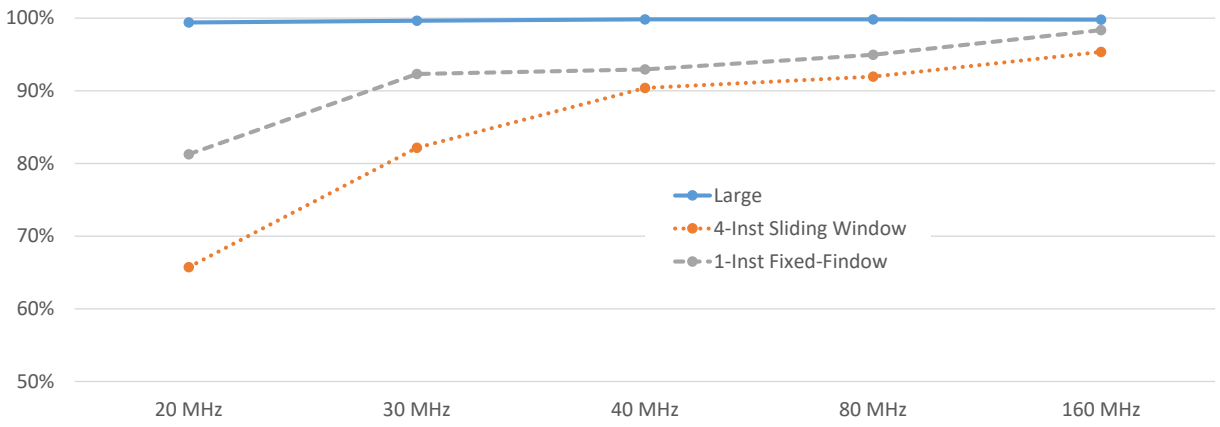

Figure 4. Accuracy of side-channel-based reconstruction of branch decisions (vertical axis) for different values of the received signal bandwidth (horizontal axis).

changes in subsequent program execution can be reconstructed nearly perfectly even when using limited signal bandwidth. For branches that cause 4-instructions and single-instruction changes in program execution, some reconstruction is possible even when using bandwidth that is only 2% of the processor's clock frequency, but reconstruction accuracy significantly improves when bandwidth is increased to 4% of the clock frequency, and then modestly improves as bandwidth is expanded to 8% and then 16% of the processor's clock frequency. However, we found it surprising that reconstructing accuracy for 1-instruction differences in our modified fixed-window implementation was consistently better than the reconstruction accuracy for 4-instruction differences in the sliding-window implementation. Upon further investigation we have found that, in addition to signal bandwidth and the amount of execution change, the accuracy of branch decision reconstruction also depends on how much is known about the branches that belong to the same signal "snippet" during analysis. In the fixed-window implementation, the outcomes of all branch decisions in the signal "snippet" are already known (because the entire exponentiation follows a fixed sequence of branch decisions), so only one bit of information is extracted from the signal "snippet", i.e. the reconstruction decision must choose between only two possibilities. In contrast, in the sliding-window implementation, five exponent-dependent branches occur in the same signal "snippet", so five bits of information must be extracted from the signal "snippet", i.e. the reconstruction decision is a choice among 32 possibilities.

Our results for reconstruction of branch decisions that cause large changes in program execution are an improvement over prior state of the art in analog side-channel cryptanalysis[32,33] in at least two significant ways. That prior work[32,33] relies on specially crafted messages (a chosen-cyphertext attack) that cause signals for a squaring (result-result multiplication) and a (result-message) multiplication to differ significantly, which allows the sequence of squaring and multiplication operations to be recognized in the sliding-window implementation, and then some bits of the key can be recovered from the squaring-multiplication sequence. The first advantage of our approach, when used for cryptanalysis, is that it works for any message, and in fact requires no knowledge of the message at all. The second cryptanalysis advantage of our approach is that, instead of distinguishing between squaring and multiplication operations, it reconstructs the exponent-dependent branch decisions (branches at lines 8 and 37 in the sliding-window code) that decide not only whether a squaring or multiplication should be executed next, but also *which* squaring operation follows - the one used for a single-bit zero-valued window (line 9) or the one for a non-zero window (line 38). This allows recovery of more of the exponent's bits – we discover at which bit-position each non-zero window begins and ends, and also at which bit positions all the single-bit zero-valued windows are, so the only bits left unknown are those between the leading 1 and the trailing 1 in each non-zero window. The effect of this is that, where prior work was able to recover 49% of the exponent's bits when the maximum window size was 4 (and fewer bits when the maximum window size is larger), in our experiments the maximum window size was 6 and yet 55% of the exponent's bits are recovered using reconstruction of branch decision of only the large-change branches (those at lines 8 and 37 in the sliding-window program listing).

Furthermore, successful reconstruction of branch decisions that cause 4-instruction changes in program execution would recover *all* of the exponent's bits. Our experiments show that, when the received signal's bandwidth is 4% or more of the clock frequency, these 4-instruction-change branch decisions are recovered with >90% accuracy, and that accuracy exceeds 95% when using 16% bandwidth, and this alone allows recovery of 90% to 95% of the exponent's bits, far more than in prior work. Finally, note that information gained from reconstruction of large-change and 4-instruction-change decisions can be combined, leading to recovery of 84% of the exponent's bits even with 2% bandwidth, 95% with 4% bandwidth, and 98% with 16% bandwidth. For side-channel-based program monitoring, this implies that knowledge of how the branch decisions in the program are related to each other can be used to improve overall accuracy of tracking the program at basic-block granularity.

Finally, our results for 1-instruction-change branches in the modified fixed-window implementation further imply that *any* key-dependent branching should be avoided in cryptographic implementations, and that side-channel-based program monitoring with single-instruction granularity is *possible*, for at least some parts of the code.

4.3 Impact of Training on Branch Decision Reconstruction

Another important factor in both cryptanalysis and side-channel based execution monitoring is how much training is needed to achieve a desired level of accuracy, and how the accuracy improves as more or less training is available. To help answer this question, we focus on the branch that causes a 4-instruction change in the sliding-window exponentiation (the branch at line 28). The results of this study are shown in Figure 5, for signals received with bandwidth of 160 MHz (16% of the processor's clock frequency). The smallest amount of training we use consists of only one exponentiation, but since 2048-bit RSA decryption uses two 1024-bit exponents, even a single (1024-bit) exponent results in 130 to 140 signal "snippets" that correspond to window-forming loop (lines 25-33 in the sliding-window listing). Recall that for each such signal "snippet" the reconstruction decision has $2^5 = 32$ possible outcomes because each snippet contains 5 execution instances of the branch at line 28, so each 1024-bit exponentiation used in training on average provides slightly more than 4 signal snippets for each of these possibilities.

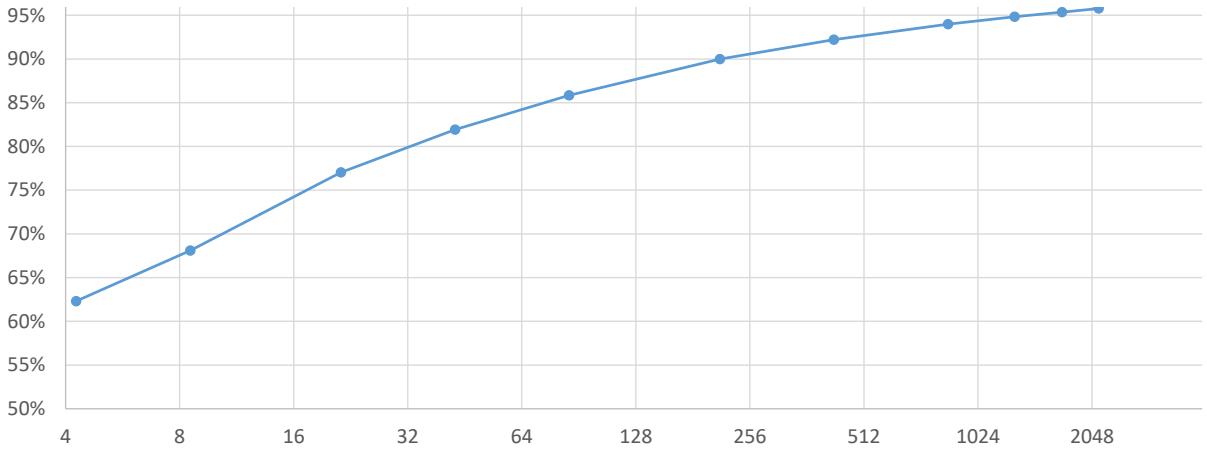

Figure 5. Accuracy of side-channel-based reconstruction of branch decisions (vertical axis) when changing the number of training examples for that point in the program code (horizontal axis, note the logarithmic scale).

We observe that training with only 4 examples for each distinct 5-branch reconstruction possibility results in 62% reconstruction accuracy, which is not very accurate but is noticeably more than the 50% accuracy that would result from purely random reconstruction. The accuracy dramatically improves with more training, reaching 86% when 85 training examples for each 5-branch reconstruction possibility, 95% when 1028 training examples per reconstruction possibility are available, and the improvement in accuracy continues, albeit more slowly, as even more training examples are added.

For cryptanalysis, this amount of training is not difficult to obtain - even the rightmost data point in Figure 5 corresponds to using only 250 RSA decryptions (with known keys) for training, and this training can be accomplished in only a few minutes and the entire "dictionary" produced by training occupies only a few tens of megabytes of memory. For side-channel-based program monitoring, however, these results imply that training may possibly need to contain hundreds of signal examples for each point in the code, which is likely to create two kinds of problems. First, in any given application, some parts of the program are very rarely executed, so it may be hard to rapidly obtain enough signal examples for those parts of the code. Second, for very large software, hundreds of signal examples for each fine-grained part of the code, with possibly millions of such points in the code, would require an enormous memory/disk footprint to store all of the training data that is needed to monitor such an application. This means that significant further research may be needed on how to summarize and/or compress the training data without sacrificing reconstruction accuracy.

4.4 Impact of the Signal-to-Noise Ratio (SNR) on Branch Decision Reconstruction

Our results shown thus far use the signal collected in very close proximity (about 2 cm) of the monitored system, where the Signal-to-Noise Ratio (SNR) is about 30 dB, i.e. the power of the received signal is about 1,000 times the power of the received noise. To study how the SNR of the side channel signal affects reconstruction of branch decisions, we perform additional experiments where we reduce the SNR to 20 dB (signal has 100 times the power of the noise), 10 dB (signal has 10 times the power of the noise), and 5 dB (signal has only 3.2 times the power of the noise). We again use 160 MHz of bandwidth, and in Figure 6 show results for each of the three categories of branches – large-change and 4-instruction-change branches from the sliding-window exponentiation, and the 1-instruction-change branches from our modified fixed-window implementation.

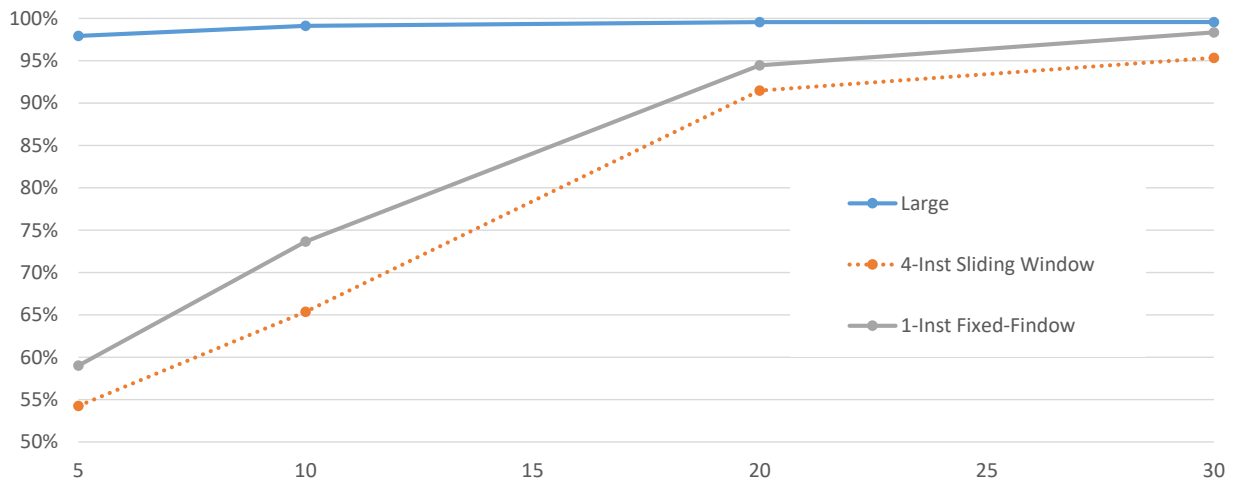

Figure 6. Accuracy of side-channel-based reconstruction of branch decisions (vertical axis) when changing the SNR (horizontal axis in decibels (dB), which implies a logarithmic scale).

From these results, we observe that a relatively low SNR (5 dB) still allows highly accurate (98%) reconstruction of branch decisions that cause large changes in program execution, while reconstruction of small-change branch decisions is only somewhat better (than pure random guessing (50%)). Increasing the SNR to 10 dB and then to 20 dB brings the large-change branch decisions very close to perfect reconstruction accuracy, and also dramatically improves reconstruction for small-change branch decisions, which at 20 dB SNR can be reconstructed with >90% accuracy. Finally, increasing the SNR to 30 dB has little impact on the (already close-to-perfect) reconstruction accuracy for large-change branch decisions, but does improve reconstruction accuracy for small-change branch decisions, which are not reconstructed with >95% accuracy.

For cryptographic implementations, these results imply that branch decisions that lead to large changes in what is subsequently executed would not be based on key material should, because that allows key material to be recovered through side channel signals even when they are received with low SNR, i.e. from larger distances or

in the presence of some noise-generating and/or shielding countermeasures. Accurate reconstruction of branch decisions that result in changing the program execution by only one or a few instructions, however, does require a reasonably good SNR (at least 20 dB), so countermeasures that dramatically reduce the SNR of the signal may be effective.

For side-channel-based monitoring of program execution, these results imply that significant changes in program execution, e.g. tracking which coarse-grained part of the program is execution or detecting long bursts of anomalous execution, can use signals received with relatively low SNR, but fine-grained (instruction-level) tracking of program execution and/or detection of small divergences from normal execution requires good-quality (at least 20 dB SNR).

5. CONCLUSIONS

In this paper, a widely used open-source cryptographic software is treated as a test subject to evaluate how the decisions of branches (control-flow decisions) in the program can be recovered (reconstructed) using side channel signals, and how the accuracy of this reconstruction depends on the signal's received bandwidth, the amount of training data, and the signal-to-noise ratio. Our results indicate that, with reasonable signal bandwidth (only 4% of the monitored system's clock frequency), enough training, and a good signal-to-noise ratio, even single-instruction-differences in program control flow can be recovered from the electromagnetic (EM) emanations. We also observe that branch decisions that control coarse-grained changes in program execution can be reconstructed even using signals that have relatively low bandwidth and SNR. For defending against side-channel attacks, our results imply that branch decisions that consider fine-grained parts (e.g. individual bits) of the cryptographic key should be avoided, especially when these branch decisions result in coarse-grained changes in subsequent execution. For side-channel-based monitoring of program execution, e.g. to obtain runtime performance information or detect anomalies, our results imply that even single-instruction-granularity monitoring is possible using signals collected with reasonable and compact equipment, e.g. sub-$1,000 receivers that provide >40 MHz of bandwidth and compact probes placed a few centimeters from the monitored system, but that monitoring of large applications will require advances in how to obtain enough training for rarely-executed parts of the application and in how to alleviate the memory/storage demands for storing the large amounts of training data that are needed to "cover" each of the many local part of the application with enough training examples.

REFERENCES

[1] Boneh, D. and Brumley, D., "Remote Timing Attacks are Practical," in [*Proceedings of the USENIX Security Symposium*], (2003).

[2] Coppens, B., Verbauwhede, I., Bosschere, K. D., and Sutter, B. D., "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors," in [*Proceedings of the 30th IEEE Symposium on Security and Privacy*], 45–60 (2009).

[3] Kocher, P., "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in [*Proceedings of CRYPTO'96, Springer, Lecture notes in computer science*], 104–113 (1996).

[4] Schindler, W., "A timing attack against RSA with Chinese remainder theorem," in [*Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2000*], 109–124 (2000).

[5] Bangerter, E., Gullasch, D., and Krenn, S., "Cache games - bringing access-based cache attacks on AES to practice," in [*Proceedings of IEEE Symposium on Security and Privacy*], (2011).

[6] Tsunoo, Y., Tsujihara, E., Minematsu, K., and Miyauchi, H., "Cryptanalysis of block ciphers implemented on computers with cache," in [*Proceedings of the International Symposium on Information Theory and its Applications*], 803–806 (2002).

[7] Wang, Z. and Lee, R. B., "New cache designs for thwarting software cache-based side channel attacks," in [*ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*], 494–505, ACM (2007).

[8] Aciiçmez, O., Koç, c. K., and Seifert, J.-P., "On the power of simple branch prediction analysis," in [*Proceedings of the 2nd ACM Symposium on Information, Computer and Communications security (ASIACCS)*], 312–320, ACM Press (Mar. 2007).

[9] Bayrak, A. G., Regazzoni, F., Brisk, P., Standaert, F.-X., and Ienne, P., "A first step towards automatic application of power analysis countermeasures," in [*Proceedings of the 48th Design Automation Conference (DAC)*], (2011).

[10] Chari, S., Jutla, C. S., Rao, J. R., and Rohatgi, P., "Towards sound countermeasures to counteract power-analysis attacks," in [*Proceedings of CRYPTO'99, Springer, Lecture Notes in computer science*], 398–412 (1999).

[11] Fahn, P. N. and Pearson, P. J., "A new class of power attacks," in [*Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999*], 173–186 (1999).

[12] Goubin, L. and Patarin, J., "DES and Differential power analysis (the "duplication" method)," in [*Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999*], 158–172 (1999).

[13] Kocher, P., Jaffe, J., and Jun, B., "Differential power analysis: leaking secrets," in [*Proceedings of CRYPTO'99, Springer, Lecture notes in computer science*], 388–397 (1999).

[14] Messerges, T. S., Dabbish, E. A., and Sloan, R. H., "Power analysis attacks of modular exponentiation in smart cards," in [*Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999*], 144–157 (1999).

[15] Agrawal, D., Archambeult, B., Chari, S., and Rao, J. R., "Advances in side-channel cryptanalysis electromagnetic analysis and template attacks," in [*RSA laboratories cryptobytes*], 20–32 (2003).

[16] Agrawal, D., Archambeult, B., Rao, J. R., and Rohatgi, P., "The EM side-channel(s)," in [*Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2002*], 29–45 (2002).

[17] Khun, M. G., "Compromising emanations: eavesdropping risks of computer displays," *The complete unofficial TEMPEST web page: http://www.eskimo.com/˜joelm/tempest.html* (2003).

[18] Backes, M., Durmuth, M., Gerling, S., Pinkal, M., and Sporleder, C., "Acoustic side-channel attacks on printers," in [*Proceedings of the USENIX Security Symposium*], (2010).

[19] Chari, S., Rao, J. R., and Rohatgi, P., "Template attacks," in [*Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2002*], 13–28 (2002).

[20] Shamir, A. and Tromer, E., "Acoustic cryptanalysis (On nosy people and noisy machines)." http://tau.ac.il/˜tromer/acoustic/.

[21] Liu, L., Yan, G., Zhang, X., and Chen, S., "Virusmeter: Preventing your cellphone from spies," in [*International Workshop on Recent Advances in Intrusion Detection*], 244–264, Springer (2009).

[22] Kim, H., Smith, J., and Shin, K. G., "Detecting energy-greedy anomalies and mobile malware variants," in [*Proceedings of the 6th international conference on Mobile systems, applications, and services*], 239–252, ACM (2008).

[23] Jacoby, G. A., Marchany, R., and Davis, N., "Battery-based intrusion detection a first line of defense," in [*Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*], 272–279, IEEE (2004).

[24] Buennemeyer, T. K., Nelson, T. M., Clagett, L. M., Dunning, J. P., Marchany, R. C., and Tront, J. G., "Mobile device profiling and intrusion detection using smart batteries," in [*Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*], 296–296, IEEE (2008).

[25] González, C. R. A. and Reed, J. H., "Power fingerprinting in sdr integrity assessment for security and regulatory compliance," *Analog Integrated Circuits and Signal Processing* **69**(2-3), 307 (2011).

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

[27] Nazari, A., Sehatbakhsh, N., Alam, M., Zajic, A., and Prvulovic, M., "Eddie: Em-based detection of deviations in program execution," in [*Proceedings of the 44th Annual International Symposium on Computer Architecture*], *ISCA '17*, 333–346, ACM, New York, NY, USA (2017).

[28] Callan, R., Behrang, F., Zajic, A., Prvulovic, M., and Orso, A., "Zero-overhead profiling via em emanations," in [*Proceedings of the 25th International Symposium on Software Testing and Analysis*], 401–412, ACM (2016).

[29] Sehatbakhsh, N., Nazari, A., Zajic, A., and Prvulovic, M., "Spectral profiling: Observer-effect-free profiling by monitoring em emanations," in [*Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*], 1–11, IEEE (2016).

[30] Percival, C., "Cache missing for fun and profit," in [*Proc. of BSDCan*], (2005).

[31] Bernstein, D. J., Breitner, J., Genkin, D., Bruinderink, L. G., Heninger, N., Lange, T., van Vredendaal, C., and Yarom, Y., "Sliding right into disaster: Left-to-right sliding windows leak." Conference on Cryptographic Hardware and Embedded Systems (CHES) 2017 (2017).

[32] Genkin, D., Pipman, I., and Tromer, E., "Get your hands off my laptop: physical side-channel key-extraction attacks on pcs," in [*Conference on Cryptographic Hardware and Embedded Systems (CHES)*], (2014).

[33] Genkin, D., Shamir, A., and Tromer, E., "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in [*International Cryptology Conference (CRYPTO)*], (2014).

[34] Olimex, "A13-olinuxino-micro user manual." https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware (accessed April 3, 2016).

[35] ARM, "Arm cortex a8 processor manual." https://www.arm.com/products/processors/cortex-a/cortex-a8.php (accessed April 3, 2016).

[36] Keysight, "N9020a mxa spectrum analyzer." https://www.keysight.com/en/pdx-x202266-pn-N9020A/mxa-signal-analyzer-10-hz-to-265-ghz?cc=US&lc=eng (accessed February 4, 2018).

[37] Ettus, "Usrp-b200mini." https://www.ettus.com/product/details/USRP-B200mini-i (accessed February 4, 2018).