SLEAK: A Side-channel Leakage Evaluator and Analysis Kit

Dan Walters, Andrew Hagen, and Eric Kedaigle
The MITRE Corporation
Bedford, Massachusetts 01730
Email: {dwalters,ahagen,ekern}@mitre.org

***Abstract*—Side-channel attacks (SCA) present a major threat to secure embedded systems. Effective software countermeasures against SCA are well known in theory, but in practice are difficult to implement properly due to issues such as unexpected compiler transformations and/or platform-specific leakage sources. Although several recent examples from industry and academia show that SCA is becoming increasingly simple and inexpensive to perform as an attacker, evaluating the security of a system against SCA can still be expensive and time-consuming. Furthermore, most evaluation techniques must be performed near the end of the development schedule which adds significant risk.**

In this paper, we present a new technique for testing software for SCA vulnerabilities in a fast, inexpensive, and automated manner. This testing could be applied to evaluate software-based SCA countermeasures even without access to source code, as may be the case with proprietary software libraries that are delivered pre-built, and without the use of side-channel collection equipment. Our implementation of the SLEAK tool demonstrates the efficacy of this technique by detecting vulnerabilities in an AES implementation that utilizes a masking countermeasure. We discuss the advantages and limitations of our technique and we conclude that it can be used to detect and understand the sources of many common SCA vulnerabilities early in the development schedule.

Keywords*—*Cryptography, side-channel attacks, masking, differential power analysis

I. Introduction

Over the last 15 years, it has been widely demonstrated that electronic devices leak information about their internal state as they perform computations. This leakage presents a major security threat to embedded systems [1]–[3], particularly those that perform cryptographic operations. Leaking information can be observed through so-called "side-channels" such as timing [4], instantaneous or static power consumption [5], [6], electromagnetic radiation [7], and perhaps others that have yet to be discovered. Through side-channel attacks (SCA), leakage is exploited to compromise implementations of otherwise secure algorithms.

While effective SCA countermeasures are well known, they are difficult to test and verify for effectiveness. This is typically done using penetration testing: attacking an actual system with the countermeasure to see if it makes the attack more difficult. There are several disadvantages to this approach. First, it requires side-channel collection equipment, specific tools for each target hardware platform, and expertise to collect and analyze the data. Second, the results are often inconsistent due to the myriad parameters that can drastically affect the test results. These include the skill and experience of the tester/attacker, the choice of side-channel, environmental conditions (e.g. temperature and background EM noise), data-collection equipment, and analysis techniques used. Even highly-experienced testers cannot guarantee that all potential vulnerabilities will be identified because new attacks and analysis techniques are constantly emerging. Finally, penetration testing cannot evaluate individual software or hardware countermeasures because it is difficult to isolate the effects of one from the other when studying the system as a whole.

In this paper, we present an automated evaluation method that addresses these challenges. That is, the proposed method does not require physical hardware or side-channel collection equipment and doesn't require the user to understand how to conduct a side-channel attack. Instead, we require only a target binary (source code optional), which will be analyzed as-is without making modifications or transformations that could affect the results. Note that our technique is applicable regardless of the countermeasures or algorithm implemented by the target binary, and because the source code is not used by our algorithm, we can evaluate any binary in the same manner regardless of the programming language used to write it. Because it also detects the worst-case information leakage, its results are applicable across varying hardware platforms and is capable of detecting vulnerabilities that originate from complex platform-specific behavior, without the need for a complex leakage model. We focus on evaluating the group of software-based countermeasures collectively known as "masking" (e.g. boolean and arithmetic masking [8]), though the proposed method could also be applied to other randomization-based countermeasures such as shuffling the execution order of independent operations [9], [10].

In Section II we summarize related works and the contributions that this paper makes to the state-of-the-art. Section III describes the technical details of our side-channel vulnerability evaluation technique and Section IV presents our proof-of-concept implementation, the SLEAK tool. Section V explains results from experimental verifications and describes vulnerabilities that SLEAK was able to identify in masked implementations of AES. Section VI concludes the paper with a summary of our contributions and an outline for future work.

II. Related Works

There are a number of related works that have helped to inform and shape our approach. In 2010, Bai *et al.* [11] proposed the use of hardware simulation tools to quickly

evaluate hardware designs for DPA vulnerabilities by generating simulated power traces rather than collecting from real hardware. In 2011, Bayrak *et al.* [12] outlined an analysis technique based on mutual information that could be used to identify vulnerable instructions by mapping samples from real power traces to their corresponding instructions. This work represented the first method that significantly deviated from penetration testing and could therefore be applied in a semi-automated manner. However, this approach still required real side-channel traces and was therefore hampered by all the challenges that arise from collecting side-channel data (as discussed earlier). Additionally, it relied on the ability to map each sample of side-channel data to the software instruction that was being executed during that time. This mapping can be challenging for many targets depending on their underlying hardware and software stack.

More recent works have eliminated the need for side-channel traces by performing the analysis solely on the data-flow graph of the target program. For example, the 2013 work of Bayrak *et al.* [13] presents a new tool, called "Sleuth," that converts the intermediate representation (IR) of the target program to a data-flow graph, and then uses that graph to generate satisfiability (SAT) queries. These queries can be solved with a SAT solver to detect unmasked intermediate values.

In 2014, Eldib *et al.* [15] showed that intermediate values could be partially masked and still leak information. Sleuth does not detect vulnerabilities related to partial or biased masking, so to address this limitation, Eldib *et al.* proposed a technique for computing "quantitative masking strength" (QMS) to give a numerical score to the side-channel resistance of a countermeasure implementation.

Sleuth and QMS represent significant progress in automated SCA evaluation, however, both are limited in the type of leakages that they can detect, and the type of programs that they can analyze. The first limitation is largely due to analyzing the target program at the IR-level. Since the IR represents the program at an intermediate stage of compilation, an analysis on the IR cannot detect any vulnerabilities that may be introduced during the final stages of compilation. While previous works have asserted that these final stages cannot generate vulnerable machine-code from protected IR [13]–[15], we show in Section V that this is not true. A second limitation, due to the conversion of the program to a data flow graph, is that the target program must be transformed into a *straight-line program* that is free from conditional branches. This process involves modifying the program by loop unrolling, function in-lining, and other transformations which could potentially introduce or eliminate vulnerabilities. If any of these modifications are made to the target binary, the analysis results may not reflect the true security of the fielded system.

The goals of our method, then, are to analyze the target binary without modification and to account for platform-specific leakage sources without requiring real side-channel data collection. We believe we've meet these goals, although at the expense of higher computational complexity.

III. Side-channel Vulnerability Assessment

We propose the use of a full system simulator to analyze platform-specific interactions that occur during program execution. More specifically, we use the simulator to determine which intermediate values depend on secret inputs and how these values are affected by the random inputs. Note that this is fundamentally different from using a simulator to generate power or EM traces because rather than attempt to model the real-world side-effects of computation, we model only the well-defined flow of data through system components. Since physical leakage can originate from any system component (registers, caches, memory, CPU flags, pipelines, and others) we compute a vulnerability metric for each component at each (simulated) clock tick. The vulnerability metric used is the mutual information (MI) between the secret values used by the target binary and intermediate values contained by system components. This metric indicates the extent to which each value in the system depends on a secret value. Note that this information-theoretic metric was originally proposed for this purpose by Standaert *et al.* [16].

A. Definitions

1) Target binary: The machine-code/executable program under evaluation. Note that we directly analyze the unmodified binary—we do not require that the program be branch-free. However, the analysis does require that the target binary provides an interface for setting all inputs.

2) Leakage model: A leakage model specifies the observable information that leaks via side-channels. We model this with a *leakage function* that takes the state of the machine as input and returns a value (or set of values) that represents the information that is observable from side-channels.

Our algorithm can be used with any arbitrary leakage function, including general functions that make very few assumptions about the specific way in which the hardware will leak information. For example, the *identity* leakage function takes the state of the machine as input and returns the entire state as output (currently a vector of the values held by the system registers). This function is both conservative and generic because it assumes that the exact value held by any individual component in the system will be revealed by a side-channel.

3) Intermediate Value: The value or state of some system component during the system's execution of the target binary. For example, the value held by Register-1 immediately after executing an instruction of the target binary.

4) Mutual Information: MI is a measure of the mutual dependence between two *random variables* (RV), measured in bits. In any computing system, all values under consideration are discrete, finite values and so are represented by *discrete* RVs. For two discrete RVs, X and Y, the mutual information is defined as

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log \left(\frac{p(x,y)}{p(x)p(y)} \right)$$

where $p(x,y)$ is the *joint probability distribution* of X and Y and $p(x)$ and $p(y)$ are the *marginal probability distributions* for each of X and Y.

Given this definition, it is easy to see that if the random variables are *independent* then

$$p(x, y) = p(x)p(y)$$

and the MI is exactly zero. Furthermore, it can be shown that the MI of two random variables is always non-negative. Intuitively, a zero MI indicates that no information about one variable can be obtained from observing the other. Otherwise, the greater the value of the MI, the more the two variables depend on each other and the more information one reveals about the other.

5) Leakage Location: A *leakage location* is an intermediate value that leaks secret information: $MI(V, S) > 0$ for an intermediate value, V, and secret value, S. A *location* is specified by the associated component name (e.g. Register-3) and point in execution time (e.g. clock tick 1 million).

6) Perfect Masking: Defined by Blömer *et al.* [17], an intermediate value is perfectly masked if it is statistically independent of the key. This definition is equivalent to stating that the MI between the intermediate value and the key is zero. The degree to which an intermediate value is not perfectly masked is given by the magnitude of the MI.

B. Algorithm

Pseudo code for our SCA evaluation algorithm is shown in Algorithm 1. For each secret bit, we start by identifying the intermediate values that depend on that bit by calling a subroutine (*IdentifyDependencies*). Note that this first step is optional; it is an optimization to reduce the number of intermediate values that are tracked and used in later calculations. In SLEAK we implemented the *IdentifyDependencies* subroutine by running the target twice (once with the secret bit set to 0 and again with it set to 1) and contrasting the set of intermediate values from the two runs. [1]

Once a set of intermediate values have been identified, we calculate the MI between each intermediate value and each bit of the secret input. To perform this calculation, the target executable is run over all random input values[2] and a subset of the possible public ones. This is done both with the current secret bit set to zero, then again with it set to one. The set of intermediate values for each run is called a *trace.* Traces are used to calculate the mutual information and determine the amount of dependence between intermediate values and each secret bit. Those intermediate values with an MI greater than zero are leakage locations that present potential SCA vulnerabilities.

In order to run the target binary with the right set of parameter values, the user must provide an interface to set the inputs. Additionally, the user must indicate what the inputs are used for by assigning each of them to one of three classes: *secret*, *public*, or *random*. *Public* inputs are those that may be revealed to an attacker without consequence, such as the plaintext of a cryptographic algorithm. *Secret* inputs are meant to be kept hidden from the attacker, such as a secret key. *Random* inputs are those that are used as a source of entropy for masking or other side-channel countermeasures.

[1] Of course, there are some scenarios where this technique may not capture all possible dependencies in which case a more thorough dependency check could be performed, or the optimization could be eliminated completely.

[2] If the random input space is too large to run over all possible inputs, a random sampling can be used to calculate an approximation.

Algorithm 1 Calculate mutual information with secret for each intermediate value

Input: Target: the executable binary under analysis
Input: Public: the input space for the Public class
Input: Random: the input space for the Random class
Input: SecretBits: the set of input bits that are of type Secret
Output: Mutual information values for each leakage location

for $s \in SecretBits$ **do**
  $L \leftarrow$ IDENTIFYDEPENDENCIES(s)
  $i \leftarrow 0$
  Let P be a simple random sample of $Public$
  for $p \in P$ **do**
    for all $r \in Random$ **do**
      $trace0_i \leftarrow$ RUNTARGET$(p, r, s = 0, L)$
      $trace1_i \leftarrow$ RUNTARGET$(p, r, s = 1, L)$
      $i \leftarrow i + 1$
    end for
  end for
  for $c \in L$ **do** ▷ Calculate MI for each dependency
    $values0 \leftarrow (trace0[c]_0, trace0[c]_1, ..., trace0[c]_{i-1})$
    $values1 \leftarrow (trace1[c]_0, trace1[c]_1, ..., trace1[c]_{i-1})$
    Record CALCMI$(\vec{0}||\vec{1}, values0||values1)$
  end for
end for

C. Computational Complexity

This approach may have been previously overlooked as being computationally infeasible due to the difficulty of computing the full distributions of intermediate values. In this section we show that in some cases, for practical cryptographic algorithms, the exact distributions are computable, and in other cases, a sufficient approximation can be reached.

Ideally, the mutual information would be calculated over all possible public, secret, and random inputs. This calculation quickly becomes infeasible if there are many possible inputs and such is the case for cryptographic algorithms. However, as discussed in [12], performing this calculation over a random subset of inputs will yield results that closely approximate the true result, provided that the subset is sufficiently large. Our empirical studies, looking at AES implementations, have found that this mutual information estimate will rapidly converge to the true value, requiring only a small subset of the possible input values.

This fast convergence is partly due to properties of certain cryptographic algorithms, and in particular, substitution-permutation networks such as AES. To demonstrate this, consider a common set of operations in AES: a secret and public value XORed together and used as a lookup into an s-box table. In this example, let the public value be denoted p. The secret bits can be divided into the bit that is being tested, denoted k', and the rest of the bits, k, with the full secret value being the XOR of the two parts: $k' \oplus k$. The result of the s-box lookup can then be written as

$$sbox[(k' \oplus k) \oplus p] = sbox[k' \oplus (k \oplus p)]$$

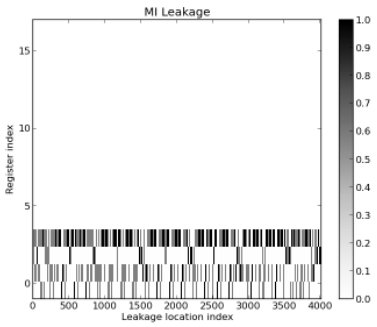

Fig. 1: Leakage timeline for each register

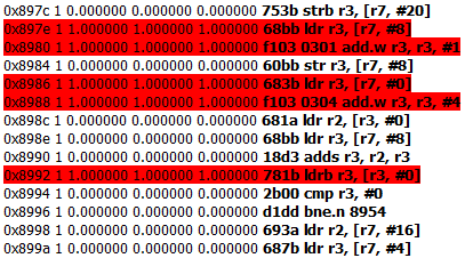

Fig. 2: Assembly view highlighting leaking instructions

Thus, as long as the subset of chosen inputs result in the values of $k \oplus p$ spanning a sufficiently large subset of the possible values, the mutual information of the s-box output will closely approximate that from a full sweep of the inputs.

IV. The SLEAK Tool

Our proof-of-concept implementation of this algorithm is the Side-channel Leakage Evaluation and Analysis Kit (SLEAK). SLEAK uses Gem5 [18], an open-source, full-system simulator, to emulate the execution of the target binary. While we focused on this particular simulator in our tests, our approach could also be used with any other simulator — provided that it supports an interface for inspecting the state of the simulated machine during program execution. Gem5 provides this access to all components of the simulated machine's state, including the registers, memory, and caches. While all of these components can be analyzed as sources of side-channel vulnerabilities, our initial prototype only analyzes registers.

To analyze the large quantity of information it produces, SLEAK includes a number of visualization tools. Below are examples of the visualizations we used in our verification studies.

A. Leakage Timeline

Leakage timeline plots provide a high-level view of system leakage over time. The example plot in Figure 1 shows the leakage detected over the course of execution of a simple implementation of AES-256. Leakage was identified from each element of the platform's internal state—in this case, each register. The register index is given on the Y-axis and the CPU tick on the X-axis. No shading (white) represents no leakage (MI=0) while black represents full leakage (MI=1) and shades of gray indicate partial leakage. This summary view shows which specific components of the machine state are leaking information and at what point during execution.

B. Leakage Highlighting

In addition to reporting the MI of each leakage location, SLEAK stores the value of the program counter (PC) at each tick. This information is used to create a mapping from leakage locations to the instructions that caused them, which is then visually presented to the user as shown in Figure 2. This view shows a section of disassembled binary with instructions that leak secret information highlighted in red. Each instruction line begins with the program counter and the number of times that instruction was executed. Since the same instruction can be executed at multiple points, and its operands may be different each time, this view reports the minimum, maximum, and average MI values that yielded by that instruction across all ticks. Looking at the highlighted assembly, it is possible to identify the most vulnerable sections of the program and, if the binary was compiled with debugging enabled, the problems can be traced to the high-level source code.

V. Experimental Results

A. Verification against Hardware

To verify our approach we conducted several experimental studies to compare results from SLEAK to results based on side-channel measurements on physical hardware. These studies demonstrated that SLEAK accurately predicts side-channel leakage and can identify the source of a side-channel vulnerability for a given target binary.

The hardware platform used in our experimental studies was a BeagleBone Rev A6; a small Linux board with a 720MHz ARM Cortex-A8 processor running Ångström Linux. This is a common development board for hobbyist work and prototyping. The Cortex-A8 processor core is used in many system-on-chips (SoCs) and consumer devices (e.g. smart-phones), and more generally, the Cortex architecture is used in several mobile processors such as the Qualcomm Snapdragon and Samsung Exynos. The widespread use of this architecture makes it an interesting and relevant platform for embedded security research.

As explained in Section III, SLEAK can use a generic leakage model. We used this model to conduct most of our tests. When this generic model is used, SLEAK's results represent the worst-case leakage scenario. In an attempt to verify these results we chose to analyze the EM side-channel since it is widely believed to be more powerful than others [7], [19], [20] and because collection of EM data does not require any physical modifications to the target system.

The EM probe used in our collections was a Langer EMV-Technik near-field magnetic probe with a frequency range from 30MHz to 3GHz. The probe was positioned to maximize the EM energy observed during normal processing on the target system. We used a Detectus AB EMC-scanner to help identify this placement. The EM traces were collected with a 14-bit ADC running at 400 MS/s, and later filtered with

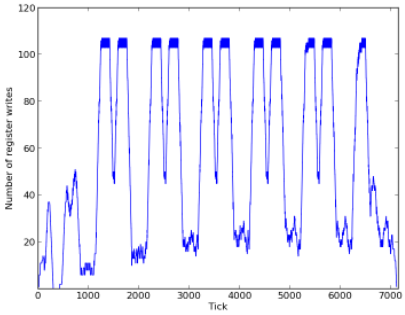

Fig. 3: Predicted EM Leakage

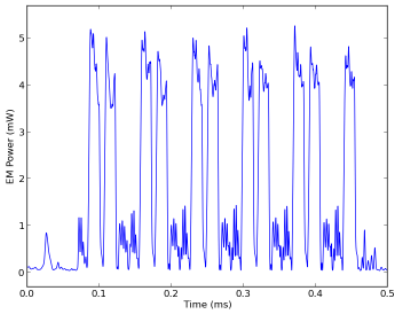

Fig. 4: Average Measured EM Trace

a 2MHz bandpass filter centered around 50MHz. This band was chosen based on the strength and structure of EM signals observed during cryptographic processing. We used a GPIO pin on the BeagleBone to trigger our collections and ensure that samples within each trace could be mapped to the proper CPU instructions. This precise triggering also greatly improved trace alignment, which would otherwise be a significant source of noise. To further reduce noise and environmental effects, we collected and averaged 10,000 traces for each test. The following sections describe the results of these tests.

1) Leakage Patterns: Since the initial implementation of SLEAK is focused on leakage from registers, the first test was to check whether the EM radiation from the Cortex-A8 was affected by register writes. To perform this test we used an unprotected AES implementation [21] so that leakages would not be hidden by side-channel countermeasures. The compiled binary for this implementation was run on the BeagleBone platform while measuring its EM emanations. This same binary was then run in the Gem5 simulator while SLEAK tracked the number of register writes performed over a sliding time window. The results from these measurements are shown in Figure 3 and Figure 4. Note that the simulator-based predictions of leakage patterns, relative leakage strength, and timing are all strongly correlated with actual EM measurements. This correlation provides a strong indication that the simulator used by SLEAK is accurately modeling the execution of the binary and that register updates are a significant source of the observed EM leakage.

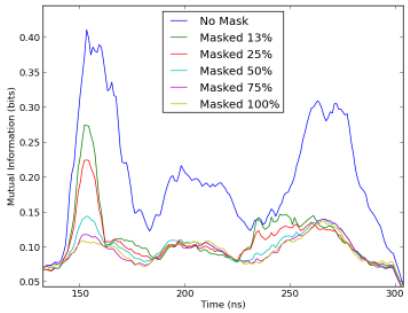

Fig. 5: EM Leakage of S-Box Lookups

2) Masking Strength: To verify SLEAK's measurement of various masking strengths, we developed a custom test program that performs an s-box table lookup on a secret input value. This table lookup operation was protected with a boolean masking countermeasure. We generated multiple versions of this program with different levels of masking strength by adjusting the distribution of the mask values. For example, if the mask is set to a constant value, the operation is effectively unmasked. At the other extreme, if the mask is drawn from a uniform random distribution of all possible 8-bit mask values, then the operation is perfectly masked. For effective masking strengths between these extremes, the mask is drawn from a non-uniform random distribution. In this way, we generated programs with six different degrees of masking strength. Each program binary was analyzed with SLEAK and with a traditional analysis of EM side-channel data, which was collected while running the target binary on the BeagleBone system. The mutual information between the EM data and the secret value used in the s-box lookup is shown in Figure 5. As expected, the effectively unmasked version yielded the most information leakage and mask distributions closer to uniform random produced progressively less leakage.

Figure 6 shows a comparison of SLEAK's MI results (X-axis) to the real EM leakage measured on hardware (Y-axis.) In this comparison, SLEAK used the generic leakage model described in Section III-A2, which is hardware agnostic and produces the worst-case scenario leakage predictions. Consequently, the predicted leakage from SLEAK is generally higher than the measured leakage and has an exponential relationship with it. This relationship highlights one of the advantages of using SLEAK for vulnerability testing: the difference between perfectly masked (MI=0) and a mask with significant bias is exploitable, yet it is challenging to detect from measured EM leakage.

Figure 7 shows the same plot, but with SLEAK using a hamming-weight leakage model instead of the generic model. In this case, the predicted leakage is highly correlated to the measured leakage ($\rho = 0.976$) and has a linear relationship. Note that while this leakage model produces very accurate predictions for this particular hardware platform, it may be less accurate for other platforms.

3) Key Extraction: To verify that the leakage detected by SLEAK is exploitable, we performed a partial key extraction on the unprotected implementation of AES-256 running on the

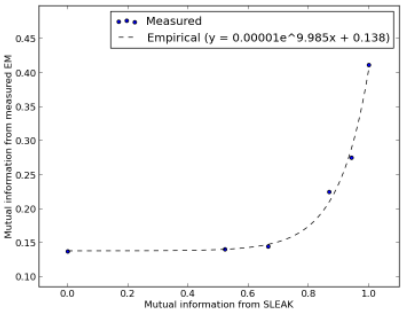

Fig. 6: Comparison between MI calculated with real EM measurements and MI from SLEAK using a generic leakage model

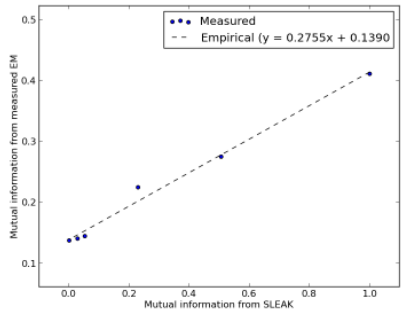

Fig. 7: Comparison between MI calculated with real EM measurements and MI from SLEAK using a hamming weight leakage model

BeagleBone. We first analyzed the implementation in SLEAK, which identified the leakage locations shown in Figure 1. Next, we mounted a side-channel attack at one of those leakage locations. We used a first-order correlation EM analysis (CEMA) attack, as described by Quisquater and Samyde [22]. The result is summarized in Figure 8. The incorrect key hypotheses are shown in gray and the correct key byte is shown in black. After roughly 2,000 traces, the key hypothesis represented by the black plot correlates significantly more than any other, revealing that it is the correct key. This successful attack demonstrates that the SCA vulnerabilities identified by SLEAK can be exploited through the EM side-channel, and used to extract bytes of the secret AES key.

B. Countermeasure Analysis

To verify the efficacy of SLEAK for real-world countermeasure analysis, we used it to analyze two different AES implementations with countermeasures: *1*) an implementation from the DPA Contest v4 [23]; and *2*) a custom boolean-masked version of AES. The DPA contest implementation uses a modern and sophisticated masking countermeasure called rotating s-box masking (RSM) [24]. Our custom version uses boolean masking and was intended to be completely secure against the register-based leakage detected by our current implementation of SLEAK. However, SLEAK identified unexpected vulnerabilities in both implementations. In the time

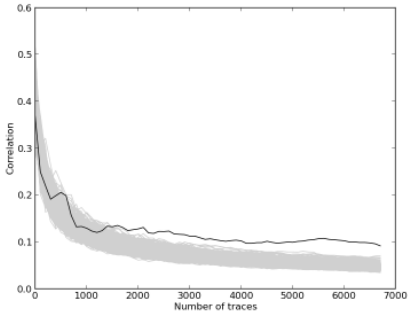

Fig. 8: The maximum correlation for each guess of an AES key-byte plotted against the number of traces. The correct guess is plotted in black and incorrect guesses in gray.

since the RSM countermeasure was released, a number of vulnerabilities have been exposed and documented [25]–[29]. Therefore, we focus on the vulnerabilities that we found exclusively through the use of SLEAK. The following sections explain the source of these vulnerabilities and how they were identified. [3]

1) Low-Entropy Mask: The RSM countermeasure is a low-entropy masking scheme (LEMS). That is, the number of discrete mask values are a subset of the possible intermediate values that it protects. In particular, RSM uses only 16 mask values, hence the mask entropy of this scheme is 4 bits, while it is used to protect 8-bit intermediate values. Although it is well-known that any LEMS is only partially masked and therefore potentially vulnerable, their reduced entropy allows for more efficient implementations and so presents a trade-off between security and performance. As a result of the low-entropy mask, SLEAK detected several leakage locations throughout the RSM implementation. Though these leakages are not surprising, it is worth pointing out that since SLEAK provides an approximation of the amount of leakage at each location, it is easy to ignore the locations with an acceptable level of leakage and investigate only those with higher-than-expected leakages. This technique allowed us to use SLEAK to search for more critical vulnerabilities in our RSM implementation.

2) Mask Cancellation From Register Updates: The second type of vulnerability that was identified in the RSM-protected version of AES manifests when leakage is a function of the two sequential values held in the same register. For example, if the initial register value is denoted v_i and the value being written is denoted v_f, then the value that is leaked is the delta between them: $v_i \oplus v_f$. SLEAK can detect these vulnerabilities by calculating the MI of all three values for each register (the initial, final, and delta). In this implementation of RSM, there are leakage locations where the initial and final register values themselves do not leak any information, but where the delta of the register does.

[3] For our testing, we ported the RSM code to run on the BeagleBone's ARM processor — the original code targeted an AVR processor. Consequently, the vulnerabilities identified may not apply to the traces provided by the DPA contest, since they were collected on an AVR platform.

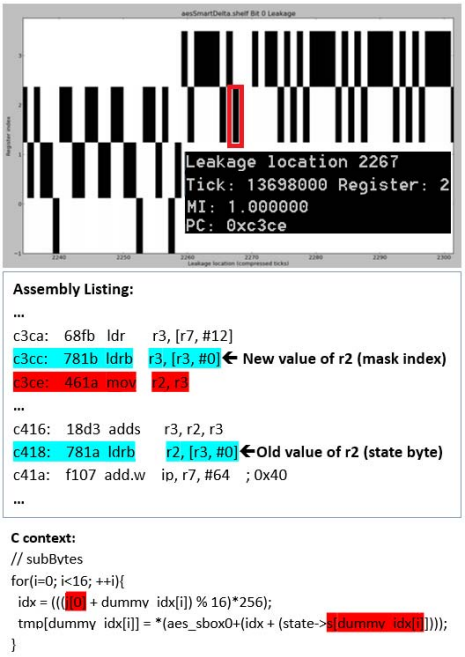

Fig. 9: Using SLEAK to identify the source of leakage

This vulnerability would likely go undetected by an analysis of the target at the IR-level because at that level, the compiler has not yet determined which registers will be used to store intermediate results. Therefore, it is impossible to determine at the IR-level which two values will be placed into the same register to create a vulnerable register delta.

Figure 9 shows SLEAK's output for one of these locations and the assembly and C contexts of the leaking instruction. At this leakage location, the register contained the starting mask index (the random value that provides the mask entropy) and was overwritten with a masked value. This caused the delta to be dependent on both the mask and the masking index, effectively allowing an attacker to see both values simultaneously. From an information theory perspective, the value of this leakage location is equivalent to $v \oplus m \oplus m$ for an intermediate value, v and mask m. Since $m \oplus m$ is the identity, the mask is canceled out and v is exposed.

Mapping the offending assembly instruction to the IR or to the original, C source code, as in Figure 9, there is no indication that this collision could occur. The fact that it did occur was wholly dependent on the register allocation chosen by the compiler backend. By analyzing the actual binary instead of higher-level representations, SLEAK is able to detect such problems.

3) Intrinsic Functions: Our custom implementation of AES uses boolean masking, which removes most of the leakage, but not all. When we analyzed this implementation with SLEAK, it still detected leakage locations that occurred within a memcpy call. In the LLVM IR of the binary, memcpy is a single *intrinsic function*, which is a built-in function that LLVM expands at a later compilation stage, shown below.

```
call void @llvm.memcpy.p0i8.p0i8.i32(
```

```
b138: ldrb r0, [r4, #9]
b13c: ldrb r2, [r1, #8]!
b140: orr r0, r2, r0, lsl #8
b144: ldrb r2, [r1, #2]
b148: ldrb r1, [r1, #3]
b14c: orr r1, r2, r1, lsl #8
b150: orr r0, r0, r1, lsl #16
b154: str r0, [sp, #12]
```

Fig. 10: Memcpy Assembly (partial listing)

```
i8* %2, i8* %3, i32 16, i32 1, i1 false)
```

Figure 10 shows a portion of the expansion for this single line of IR into several ARM assembly instructions. The instructions that leak are highlighted in gray. This code loads multiple bytes into one register and writes them all to memory at once. While this is a good performance optimization, it potentially nullifies the masking countermeasure. Before being written, the register contained four intermediate values, all masked with the same mask. Effectively, this code creates a 32-bit value that still uses a mask with only eight bits of entropy. Like the mask-cancellation vulnerability, there was no indication in either the C source or the IR that this vulnerability existed. It is important to notice that the implementation of an intrinsic function can be significantly different across different target platforms. In general, intrinsic functions cannot be evaluated for leakage at the IR-level because they are a blackbox; the fully-compiled binary must be analyzed to ensure that the lowering[4] of intrinsic functions does not introduce vulnerabilities.

VI. Conclusion

In this paper, we proposed and demonstrated, for the first time, the use of a full-system simulator to analyze software for side-channel vulnerabilities. Our algorithm demonstrates how to perform this analysis in an automated way such that it can be used by software developers without expertise in side-channel security. Furthermore, we have shown that our technique for approximating the mutual information is a practical approach that avoids calculating this metric exactly (which would be computationally infeasible).

Our proof-of-concept implementation, SLEAK, produces results that account for platform-specific leakage characteristics and compiler effects without the need for physical hardware or side-channel traces. Though the prototype focuses only on registers, SLEAK can be extended to evaluate other leakage sources, such as memory and caches. Our experimental verifications of SLEAK indicate that it correctly discerns between different levels of mask strength to identify intermediate values that can leak information. It also accurately identifies the system components that generate leakage and when it occurs. By tracing these leakage locations to their corresponding assembly and source code instructions, we show that there are real-world cases where high-level code and IR is secure, but has vulnerabilities introduced during the compiler's final stages.

[4]"Lowering" is the proper term for expanding intrinsic functions.

SLEAK and other automated software evaluation tools can help reduce the cost of verifying the security of software implementations. This testing can be performed early in the development cycle when it is easier to fix vulnerabilities. The development of automated tools, such as SLEAK, to assist with security testing is increasingly important in the face of the ever decreasing cost and difficulty of performing side-channel attacks.

Future work consists of evaluating other leakage sources, extending our technique to check for security against multivariate attacks, and to identify fault attack vulnerabilities.

REFERENCES

[1] A. Moradi, A. Barenghi, T. Kasper, and C. Paar, "On the vulnerability of fpga bitstream encryption against power analysis attacks extracting keys from xilinx virtex-ii fpgas," Cryptology ePrint Archive, Report 2011/390, 2011, http://eprint.iacr.org/.

[2] C. Paar, T. Eisenbarth, M. Kasper, T. Kasper, and A. Moradi, "Keeloq and side-channel analysis-evolution of an attack," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on*, Sept 2009, pp. 65–69.

[3] J. Balasch, B. Gierlichs, R. Verdult, L. Batina, and I. Verbauwhede, "Power analysis of atmel cryptomemory recovering keys from secure eeproms," in *Topics in Cryptology CT-RSA 2012*, ser. Lecture Notes in Computer Science, O. Dunkelman, Ed. Springer Berlin Heidelberg, 2012, vol. 7178, pp. 19–34. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27954-6_2

[4] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1109. Springer, 1996, pp. 104–113.

[5] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '99. London, UK, UK: Springer-Verlag, 1999, pp. 388–397. [Online]. Available: http://dl.acm.org/citation.cfm?id=646764.703989

[6] J. Giorgetti, G. Scotti, A. Simonetti, and A. Trifiletti, "Analysis of data dependence of leakage current in cmos cryptographic hardware," in *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '07. New York, NY, USA: ACM, 2007, pp. 78–83. [Online]. Available: http://doi.acm.org/10.1145/1228784.1228808

[7] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Cryptographic Hardware and Embedded Systems CHES 2001*, ser. Lecture Notes in Computer Science, e. Ko, D. Naccache, and C. Paar, Eds. Springer Berlin Heidelberg, 2001, vol. 2162, pp. 251–261.

[8] T. Messerges, "Securing the aes finalists against power analysis attacks," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, and B. Schneier, Eds. Springer Berlin Heidelberg, 2001, vol. 1978, pp. 150–164. [Online]. Available: http://dx.doi.org/10.1007/3-540-44706-7_11

[9] C. Herbst, E. Oswald, and S. Mangard, "An aes smart card implementation resistant to power analysis attacks," in *Applied Cryptography and Network Security*, ser. Lecture Notes in Computer Science, J. Zhou, M. Yung, and F. Bao, Eds. Springer Berlin Heidelberg, 2006, vol. 3989, pp. 239–252. [Online]. Available: http://dx.doi.org/10.1007/11767480_16

[10] M. Rivain, E. Prouff, and J. Doget, "Higher-order masking and shuffling for software implementations of block ciphers," in *Cryptographic Hardware and Embedded Systems - CHES 2009*, ser. Lecture Notes in Computer Science, C. Clavier and K. Gaj, Eds. Springer Berlin Heidelberg, 2009, vol. 5747, pp. 171–188. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04138-9_13

[11] X. Bai, Y. Wang, Y. Wang, and X. Hu, "A power analysis attack software simulation platform design and its applications," in *2010 2nd International Conference on Computer Engineering and Technology*, vol. 6, 2010.

[12] A. G. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne, "A first step towards automatic application of power analysis countermeasures," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 230–235. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024778

[13] A. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated verification of software power analysis countermeasures," in *Cryptographic Hardware and Embedded Systems - CHES 2013*, ser. Lecture Notes in Computer Science, G. Bertoni and J.-S. Coron, Eds. Springer Berlin Heidelberg, 2013, vol. 8086, pp. 293–310. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40349-1_17

[14] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi, "Compiler-based side channel vulnerability analysis and optimized countermeasures application," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 81:1–81:6. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488833

[15] H. Eldib, C. Wang, M. Taha, and P. Schaumont, "Qms: Evaluating the side-channel resistance of masked software from source code," in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 209:1–209:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593193

[16] F.-X. Standaert, T. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *Advances in Cryptology - EUROCRYPT 2009*, ser. Lecture Notes in Computer Science, A. Joux, Ed. Springer Berlin Heidelberg, 2009, vol. 5479, pp. 443–461. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01001-9_26

[17] J. Blmer, J. Guajardo, and V. Krummel, "Provably secure masking of aes," in *Selected Areas in Cryptography*, ser. Lecture Notes in Computer Science, H. Handschuh and M. Hasan, Eds. Springer Berlin Heidelberg, 2005, vol. 3357, pp. 69–83. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30564-4_5

[18] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[19] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The em side-channel(s)," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '02. London, UK, UK: Springer-Verlag, 2003, pp. 29–45. [Online]. Available: http://dl.acm.org/citation.cfm?id=648255.752713

[20] J. Heyszl, S. Mangard, B. Heinz, F. Stumpf, and G. Sigl, "Localized electromagnetic analysis of cryptographic implementations," in *Topics in Cryptology CT-RSA 2012*, ser. Lecture Notes in Computer Science, O. Dunkelman, Ed. Springer Berlin Heidelberg, 2012, vol. 7178, pp. 231–244. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27954-6_15

[21] (2013, Nov.) Aes and combined encryption/authentication modes. [Online]. Available: http://brgladman.org/oldsite/AES/index.php

[22] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (ema): Measures and counter-measures for smart cards," in *Smart Card Programming and Security*, ser. Lecture Notes in Computer Science, I. Attali and T. Jensen, Eds. Springer Berlin Heidelberg, 2001, vol. 2140, pp. 200–210. [Online]. Available: http://dx.doi.org/10.1007/3-540-45418-7_17

[23] (2014, Sept) Dpacontest v4. [Online]. Available: http://www.dpacontest.org/home/

[24] M. Nassar, Y. Souissi, S. Guilley, and J.-L. Danger, "Rsm: A small and fast countermeasure for aes, secure against 1st and 2nd-order zero-offset scas," in *DATE*, 2012, pp. 1173–1178.

[25] (2014, Sept) Dpacontest v4 - hall of fame. [Online]. Available: http://www.dpacontest.org/v4/hall_of_fame.php

[26] S. Kutzner and A. Poschmann, "On the security of rsm - presenting 5 first- and second-order attacks," in *Constructive Side-Channel Analysis and Secure Design*, ser. Lecture Notes in Computer Science, E. Prouff, Ed. Springer International Publishing, 2014, pp. 299–312. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10175-0_20

[27] X. Ye and T. Eisenbarth, "On the vulnerability of low entropy masking

schemes," in *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, 2013, pp. 44–60. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08302-5_4

[28] A. Moradi, S. Guilley, and A. Heuser, "Detecting hidden leakages," Cryptology ePrint Archive, Report 2013/842, 2013, http://eprint.iacr.org/.

[29] L. Lerman, S. Medeiros, G. Bontempi, and O. Markowitch, "A machine learning approach against a masked aes," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, A. Francillon and P. Rohatgi, Eds. Springer International Publishing, 2014, pp. 61–75. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08302-5_5