Quantitative Masking Strength: Quantifying the Power Side-Channel Resistance of Software Code

Hassan Eldib, Chao Wang, *Member, IEEE*, Mostafa Taha, *Member, IEEE*, and Patrick Schaumont, *Senior Member, IEEE*

***Abstract*—Many commercial systems in the embedded space have shown weakness against power analysis-based side-channel attacks in recent years. Random masking is a commonly used technique for removing the statistical dependency between the sensitive data and the side-channel information. However, the process of designing masking countermeasures is both labor intensive and error prone. Furthermore, there is a lack of formal methods for quantifying the actual strength of a countermeasure implementation. Security design errors may therefore go undetected until the side-channel leakage is physically measured and evaluated. We show a better solution based on static analysis of C source code. We introduce the new notion of quantitative masking strength (QMS) to estimate the amount of information leakage from software through side channels. Once the user has identified the sensitive variables, the QMS can be automatically computed from the source code of a countermeasure implementation. Our experiments, based on measurement on real devices, show that the QMS accurately reflects the side-channel resistance of the software implementation.**

***Index Terms*—Countermeasure, differential power analysis (DPA), quantitative masking strength (QMS), satisfiability modulo theory (SMT) solver, security, verification.**

I. Introduction

In RECENT years, many commercial systems in the embedded space have shown weaknesses against side-channel attacks [2]–[4], where an adversary can utilize secondary information such as timing and power consumption resulting from the execution of sensitive algorithms on these devices. For example, the power consumption of an embedded device executing instruction `a=t⊕k` may depend on the value of the secret data `k` [5] and, as a result, `k` may be reliably deduced by an adversary using statistical methods such as the differential power analysis (DPA [6]).

Masking, which is a randomization technique for removing the statistical dependency between the sensitive data and the side-channel information, is a commonly used mitigation strategy. For example, Boolean masking uses an XOR operation of a random bit r with a variable a to obtain a masked variable: $a_m = a \oplus r$ [4], [7]. Later, the original Boolean variable can be restored by a second XOR operation: $a_m \oplus r = a$. Other similar countermeasures have used additive masking ($a_m = a + r \bmod n$), multiplicative masking ($a_m = a * r \bmod n$), as well as application-specific masking such as RSA blinding ($a_m = ar^e \bmod N$).

However, side-channel countermeasures based on masking are difficult to design and implement because the process is both labor intensive and error prone. There is also no formal method to quantify how secure a software implementation of the countermeasure really is. This is a problem in practice because the source of the information leakage is not the cryptographic algorithm itself, but the microprocessor hardware that executes the cryptographic software code. For software developers who do not know all the architectural details of the computing device, it can be difficult to understand where and when the side-channel information may be leaked.

In this paper, we solve the problem by first introducing the new notion of quantitative masking strength (QMS) to estimate the side-channel resistance of a software implementation. To demonstrate the effectiveness of QMS in quantifying the side-channel resistance, we conduct experiments on a set of cryptographic software on real devices while launching DPA attacks. For each software implementation, we record the number of power consumption measurement traces required to successfully break the countermeasure. Our experimental results show that the number of required measurement traces, which correlates to the difficulty in breaking the countermeasure, matches the QMS.

We also develop a design automation tool that leverages static code analysis techniques to compute the QMS of a given program, without executing it on the actual device. The tool can be used to evaluate the quality of the countermeasure implementation. It can also be used as a formal verification procedure to decide whether a given program satisfies a desired QMS requirement. In case that some intermediate computation

Manuscript received September 24, 2014; revised January 19, 2015; accepted March 4, 2015. Date of publication April 21, 2015; date of current version September 17, 2015. This work was supported in part by the NSF under Grant CNS-1128903 and Grant CNS-1115839, and in part by the ONR under Grant N00014-13-1-0527. Some of the preliminary results appeared in [1]. This paper was recommended by Associate Editor R. Karri.

H. Eldib, C. Wang, and P. Schaumont are with the Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University (Virginia Tech), Blacksburg, VA 24061 USA (e-mail: chaowang@vt.edu).

M. Taha is with the Department of Electrical and Computer Engineering, Worcester Polytechnic Institute, Worcester, MA 01609 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2015.2424951

results of the program do not satisfy the QMS requirement, our method can produce a side-channel attack scenario, consisting of a combination of the plaintext bits, the secret bits, and the relevant code region that leaks an excessive amount of information about the secret bits.

Our static code analysis tool builds upon the popular LLVM compiler [8] for C/C++ and the Yices satisfiability modulo theory (SMT) solver [9]. We encode the verification problem into a series of quantifier-free first-order logic (FOL) formulas, whose satisfiability can be decided by the SMT solver. Although in the literature, there is a large body of work on SAT/SMT solver-based verification techniques [10], [11] and some methods for analyzing side channels leaks [12]–[15], e.g., using type-based information flow analysis [16], [17], they do not focus on the power side-channel leakage. Bayrak *et al.* [18] proposed a method that uses a Boolean SAT solver to check if a piece of software code is masked by some random bits, but they cannot quantitatively check the masking strength. To the best of our knowledge, our method is the first fully automated static analysis method for quantitatively checking the strength of masking.

We have conducted experiments on a set of cryptographic software implementations to evaluate the performance of our static analysis tool. The benchmarks include several recent countermeasures for AES as well as message authentication code (MAC)-Keccak, an MAC based on the new secure hash algorithm (SHA-3) standard of NIST. Our experimental results show that the new method is effective in detecting vulnerabilities in cryptographic software code and is scalable enough to handle software code of practical size.

To sum up, this paper makes the following contributions.

1) We propose the new notion of QMS as a way to estimate the side-channel resistance of a software implementation in practice.
2) We conduct DPA attack experiments on real embedded computing devices to confirm that the QMS is indeed a good indicator of the side-channel resistance of the masked software.
3) We propose a static code analysis method for computing the QMS of a given program decorated with the sensitive variables and the masks. The method can also quantify the impact of bias in the random variables and formally verify that a program satisfies a QMS requirement.
4) When a program fails to satisfy the QMS requirement, our tool can produce an attack scenario, consisting of a plaintext and two sets of key bits, together with the related code region that have side-channel leaks.

The remainder of this paper is organized as follows. We establish notation in Section II and define the QMS in Section III. We present our static code analysis method in Section IV, and our method for modeling bias in random number generators in Section V. We describe our DPA attack experiments in Section VI. We present our experimental evaluation results for the static code analysis method in Section VII. We review related work in Section VIII, and finally, the conclusion is given in Section IX.

II. Preliminaries

In this section, we provide a brief introduction to power analysis-based side-channel attacks as well as randomization-based countermeasures.

Following the notation used by Blömer *et al.* [19], we assume that the program to be analyzed implements a function $c \leftarrow \text{enc}(x, k)$, where x is the plaintext, k is the secret key, and c is the ciphertext. Let $I_1, I_2, \ldots, I_t$ be the sequence of intermediate computation results inside the function, and each $I_i(x, k, r)$, where $1 \leq i \leq t$, be a function of x, k, and r. Here, r is a random number used to make the power leakage associated with the value of function I_i statistically independent of k.

When $f(x, k)$ is a linear function of k with respect to the XOR operator, masking and de-masking the sensitive computations inside $f(x, k)$ become trivial, because $f(x, k \oplus r) \oplus f(x, r) = f(x, k) \oplus f(x, r) \oplus f(x, r) = f(x, k)$. That is, we can mask the sensitive bit k by the XOR with r before the computation, and then de-masking the result by the XOR with $f(x, r)$. The main advantage of this approach is that the implementation of core computations inside $f(x, k)$ does not need to be modified. However, when $f(x, k)$ is a nonlinear function of k with respect to the XOR operator, masking and de-masking often require a complete redesign of the software. This process is both labor intensive and error prone. Indeed, designing a new masking scheme for a reputable cryptographic algorithm such as AES [20] would be considered as publishable work in top cryptography venues.

In this paper, we assume that an adversary knows the pair (x, c) of plaintext and ciphertext in $c \leftarrow \text{enc}(x, k)$. For each pair (x, c), the adversary may measure the side-channel leakage of at most d intermediate computation results $I_1, \ldots, I_d$. However, the adversary does not have access to r, which is assumed to be the output of a true random number generator. The goal of the adversary is to compute the secret key (k). Kocher *et al.* [6] demonstrated in their seminal work that it is possible to deduce such information (k) from the power side channels using a statistical method known as the DPA.

A necessary condition for power side-channel resistance is for all the intermediate computation results of a function to be insensitive with respect to the secret bits, as defined by Bayrak *et al.* [18]. Here, we say that an intermediate result I_i is sensitive if it logically depends on the secret bits and, at the same time, it does not logically depend on any random variable. According to [18], this logical dependency analysis is equivalent to computing don't cares (DCs) in logic synthesis [21]. If random bit r is a DC of the Boolean function I_i, then the value of I_i does not depend on the value of r. Recall that, in logic synthesis, r is a DC if I_i remains unchanged whether r is set to logical 0 or 1. However, even an insensitive I_i may still leak secret information, because logically depending on random bits does not mean that I_i is statistically independent from the secret bits.

Fig. 1 shows an example, where `k` is the secret bit, `r1` and `r2` are the random bits, and `o1, o2, o3`, and `o4` are the results of four masking schemes. According to the truth table on the right-hand side, all four outputs depend on `r1` and `r2` and therefore are insensitive [18], but three of them still leak secret

o1 = x∧k∧(r1∧r2)
o2 = x∧k∨(r1∧r2)
o3 = x∧k⊕(r1∧r2)
o4 = x∧k⊕(r1⊕r2)

x	k	r1	r2	o1	o2	o3	o4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	1

Fig. 1. Four masking schemes with different side-channel leakages and the corresponding truth table when $x = 0$. Although o1, o2, and o3 are masked by random bits r1 and r2, they still leak secret information about k. In contrast, o4 does not have side-channel leakage.

information. For instance, when o1 is logical 1, we know for sure that the secret k is also 1, regardless of the values of the random variables. Similarly, when o2 is logical 0, we know for sure that k is also 0. When o3 is logical 1 (or 0), there is a 75% chance that k is logical 1 (or 0). Furthermore, in practice, techniques for mitigating the overhead of masking may lead to unbalanced masks, which can cause hidden leakages [22]. In contrast, o4 is the only side-channel resistant output because it is statistically independent of k. When k is logical 1 (or 0), there is 50% chance that o4 is logical 1 (or 0).

In the context of power side-channel analysis, a leakage model specifies the amount of side-channel information observable during program execution. In simple power analysis- and DPA-based attacks, an effective and widely used leakage model is the Hamming weight (HW) model. We will show in Section VI that, based on our measurements on real embedded computing devices, the HW model is accurate enough for our analysis.

Sometimes, however, a device does not leak the HW of the processed data, but the Hamming distance (HD) between these data and an initial state [23]. Extending our static code analysis method to handle this so-called HD model is left for future work.

III. Quantitative Masking Strength

In this section, we first introduce the new notion of QMS and then present methods for checking whether a program satisfies a given QMS.

A. Notion of Masking Strength

Given a pair (x, k) of plaintext and secret key for the function $\text{enc}(x, k)$, an s-bit random number r uniformly distributed in the domain $R = \{0, 1\}^s$, and d intermediate computation results $I_1(x, k, r), \ldots, I_d(x, k, r)$, if the joint distribution of $I_1, \ldots, I_d$ is statistically independent of the secret k, following the terminology of [19], we say that the function is order-d perfectly masked. Otherwise, we say that the function is vulnerable to side-channel attacks, and in this section, we would like to propose a new metric (Δ_{qms}) for quantifying the bias of each $I_i(x, k, r)$ with respect to $x = \chi$ and $k = \kappa$, where χ and κ are certain values for variables x and k, respectively.

Definition 1: Given an implementation of the function $\text{enc}(x, k)$ and a set of intermediate computation results $\{I_i(x, k, r)\}$ inside the function, we define the QMS as the minimal value of $(1 - \Delta_{\text{qms}})$ such that, for any $I_i(x, k, r)$

$$\left|E(I_i|k = \kappa \wedge x = \chi) - E\left(I_i|k = \kappa' \wedge x = \chi\right)\right| \leq \Delta_{\text{qms}}$$

holds for all values χ, κ, and κ' where $\kappa \neq \kappa'$.

In the context of DPA, we can regard the value κ in the above definition as the correct value of variable k, and κ' as any of the incorrect values. For verification purposes, we cannot fix κ to any particular value; instead, we keep it symbolic in order to prove that there is no side-channel information leakage for any κ and κ'.

As an example, consider the four masking schemes in Fig. 1. Let $\Delta_{\text{qms}}(I_i) = |E(I_i|k = \kappa \wedge x = \chi) - E(I_i|k = \kappa' \wedge x = \chi)|$, where $\chi = 0$, $\kappa = 0$, and $\kappa' = 1$. In the context of order-1 side-channel attacks, we have

$$\Delta_{\text{qms}}(o1) = 1/4 - 0/4 = 0.25 \quad \Delta_{\text{qms}}\left(\overline{o1}\right) = 4/4 - 3/4 = 0.25$$
$$\Delta_{\text{qms}}(o2) = 4/4 - 1/4 = 0.75 \quad \Delta_{\text{qms}}\left(\overline{o2}\right) = 3/4 - 0/4 = 0.75$$
$$\Delta_{\text{qms}}(o3) = 3/4 - 1/4 = 0.50 \quad \Delta_{\text{qms}}\left(\overline{o3}\right) = 3/4 - 1/4 = 0.50$$
$$\Delta_{\text{qms}}(o4) = 2/4 - 2/4 = 0.00 \quad \Delta_{\text{qms}}\left(\overline{o4}\right) = 2/4 - 2/4 = 0.00.$$

All four outputs satisfy the insensitivity requirement [18] because of their logical dependence on the random bits, but only o4 is statistically independent of the secret k. In other words, only for o4, we have $\Delta_{\text{qms}} = 0.0$, which is the same as $\text{QMS} = (1 - \Delta_{\text{qms}}) = 1.0$, indicating it is perfectly masked.

Our new notion of QMS in Definition 1 subsumes both the notion of perfect masking [19] and the notion of sensitivity [18] in the following sense. First, the perfect masking criterion introduced by Blömer *et al.* [19] can be viewed as an extreme where $\Delta_{\text{qms}} = 0$. Second, the insensitivity criterion introduced by Bayrak *et al.* [18] can be viewed as another extreme where $\Delta_{\text{qms}} = 1$. They represent two extreme cases of the spectrum, whereas QMS allows us to quantify the side-channel resistance of the vast number of design choices in between.

B. Checking the Masking Strength

To decide whether a function satisfies the given QMS requirement, we need to decide whether there exists any intermediate computation result $I_i(x, k, r)$ such that $|E(I_i|k = \kappa \wedge x = \chi) - E(I_i|k = \kappa' \wedge x = \chi)| > \Delta_{\text{qms}}$ for some (χ, κ) and (χ, κ'), where $\kappa \neq \kappa'$. The function $\text{enc}(x, k)$ satisfies the QMS requirement if and only if no such intermediate computation result exists for the given Δ_{qms}.

The main challenge for static code analysis methods—whether it is to compute the QMS of a given program or to verify that the program satisfies the QMS requirement—is to compute $E(I_i|k = \kappa \wedge x = \chi)$.

Our new method works as follows. As the starting point, we mark all the plaintext bits in x as public, the key bits in k as secret, and the mask bits in r as random. Then, for each node $I(x, k, r)$, which represents a Boolean function, we check whether it satisfies the QMS requirement. Following Definition 1, we formulate the order-1 QMS check as a satisfiability problem as follows:

$$\exists x, k, k' \ . \ \left(\Sigma_{r \in R} I(x, k, r) - \Sigma_{r \in R} I\left(x, k', r\right)\right) > \Delta_{\text{qms}}.$$

Here, the variable x represents the plaintext, variables k and k' represent two different secret keys, and r represents the s-bit random number in domain $R = \{0, 1\}^s$. For any fixed value of (x, k, k'), the summation $\Sigma_{r\in R} I(x, k, r)$ represents the number of satisfying assignments of $I(x, k, r)$ and the summation $\Sigma_{r\in R} I(x, k', r)$ represents the number of satisfying assignment of $I(x, k', r)$. In both cases, the assignment count is in terms of the random variable r. Assume that r is uniformly distributed in the domain $R = \{0, 1\}^s$, the summations represent the probabilities of I being logical 1 under certain values of (x, k) and (x, k'), respectively.

If the above formula is satisfiable, there exist values for x and (k, k') such that the distribution of $I(x, k, r)$, in terms of $r \in R$, differs from the distribution of $I(x, k', r)$ by more than Δ_{qms}. In other words, the secret values of k and k' are leaked, and the amount of information leakage is more than expected. On the other hand, if the above formula is unsatisfiable, then I satisfies the given QMS requirement.

IV. Static Software Code Analysis

In this section, we first present our verification procedure, which takes a Boolean program and a QMS as input and checks whether the program satisfies the QMS requirement. It is an extension of our verification method, called SC Sniffer [24], for verifying whether a program is perfectly masked. Then, we present our algorithm for estimating the QMS of a given program, which uses the aforementioned verification procedure as a subroutine.

A. Checking Program Against QMS Requirement

Our method is based on translating the verification problem into a set of quantifier-free FOL formulas and then deciding the formulas using an off-the-shelf SMT solver. For each intermediate computation result $I(x, k, r)$ of the given program, we construct a formula Φ such that it is satisfiable if and only if there exist a plaintext value $x = \chi$ and two different key values κ and κ' such that the probability for $I(x, k, r)$ to be logical 1 differs from the probability for $I(x, k', r)$ to be logical 1 by more than Δ_{qms}. Although SAT/SMT solver-based verification techniques have been widely used in electronic design automation for verifying hardware/software components, our method is different in that existing methods typically focus on functional correctness whereas the QMS checked by our method is nonfunctional—it is a quantitative property and is statistical in nature.

Given a Boolean program as input, we first construct a data-flow graph, where the root represents the return value and the leaf nodes represent the inputs. Each internal node represents the result of a Boolean operation of one of the following types: AND, OR, NOT, and XOR. For the example in Fig. 2, our method starts by parsing the program and creating a graph representation. This is followed by traversing the graph in a topological order, from the program inputs (leaf nodes) to the return value (root node). For each internal node, which represents an intermediate computation result, we check whether it satisfies the given QMS requirement.

```
1 : compute(bool k1, bool k2, bool r1, bool r2){
2 :     bool n1, n2, n3, n4, n5, n6, n7, n8, c;
3 :     n1 = k1 ⊕ r1;
4 :     n2 = k2 ⊕ r2;
5 :     n3 = n1 & n2;
6 :     n4 = k2 ⊕ r2;
7 :     n5 = r1 & n4;
8 :     n6 = k1 ⊕ r1;
9 :     n7 = r2 & n6;
10 :    n8 = n5 ⊕ n7;
11 :    c = n3 ⊕ n8;
12 :    return c;
13 : }
```

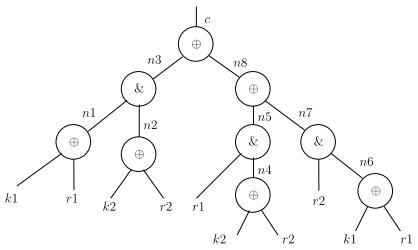


Fig. 2. Example for static analysis of the QMS. The input program and its abstract syntax tree (AST).

The order in which we check the internal nodes is as follows: $n1, n2, n3, n4, n5, n6, n7, n8$, and finally, c.

Notice that the program in Fig. 2 is a masked version of $c \leftarrow (k1 \wedge k2)$, where $k1$ and $k2$ are the secret keys, $r1$ and $r2$ are the random variables, and c is the computation result. The return value c is logically equivalent to $(k1 \wedge k2) \oplus (r1 \wedge r2)$. This masking scheme (from [19]) is used to make the power consumption of the computation of c independent of the values of $k1$ and $k2$. The corresponding de-masking function (not shown in the figure) is $c \oplus (r1 \wedge r2)$. Since $(k1 \wedge k2) \oplus (r1 \wedge r2) \oplus (r1 \wedge r2) = (k1 \wedge k2)$, de-masking would produce the desired value $(k1 \wedge k2)$.

Our method will determine if all intermediate variables of the program have a masking strength higher than Δ_{qms}. Toward this end, we formulate a satisfiability problem for each intermediate computation result $I(x, k, r)$. Let Φ denote the SMT formula to be created for checking the intermediate result $I(x, k, r)$. Recall that I is a Boolean function in terms of x, k and r. Let s be the number of random bits in r. Our encoding method ensures that Φ is satisfiable if and only if I does not satisfy the QMS requirement. Therefore, we define Φ as follows:

$$\Phi := \left(\bigwedge_{r=0}^{2^s-1} \Psi_k^r\right) \wedge \left(\bigwedge_{r=0}^{2^s-1} \Psi_{k'}^r\right) \wedge \Psi_{b2i} \wedge \Psi_{\text{sum}} \wedge \Psi_{\text{diff}}$$

where the subformulas are defined as follows.

1) *Program Logic (Ψ_k^r):* Each subformula Ψ_k^r encodes a copy of the function of $I(x, k, r)$ with the random variable r set to a concrete value in the domain $\{0, \ldots, 2^s - 1\}$ and the secret values set to k or k'. Notices that all copies of the program logics for Ψ_k^r and $\Psi_{k'}^r$ share the same plaintext value x.
2) *Boolean-to-Int (Ψ_{b2i}):* It encodes the conversion of the output of $I(x, k, r)$ from Boolean to integer (where true becomes 1 and false becomes 0), so that the integer values can be summed up later to compute $\Sigma_{r\in R} I(x, k, r)$.
3) *Sum-Up-the-1s (Ψ_{sum}):* It encodes the two summations of the logical 1s in the outputs of the 2^s copies of program logic, one for the program logic copies of the form $I(x, k, r)$ and the other for the program logic copies of the form $I(x, k', r)$.
4) *Different Sums (Ψ_{diff}):* It asserts that the difference between the two summations is bigger than the required Δ_{qms}—recall that the leakage and the QMS has the following relation: $\Delta_{\text{qms}} = 1 - \text{QMS}$.

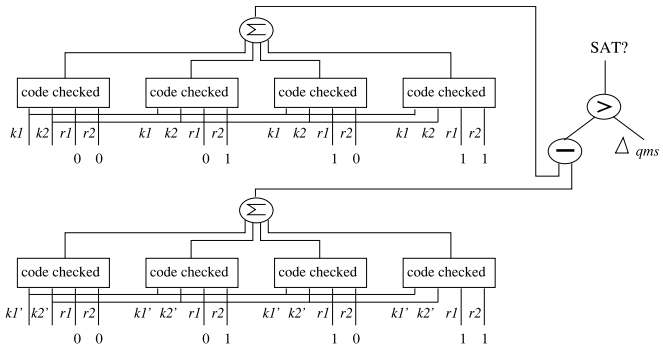


Fig. 3. Illustrating the SMT encoding for checking the QMS of the function $I(k1, k2, r1, r2)$.

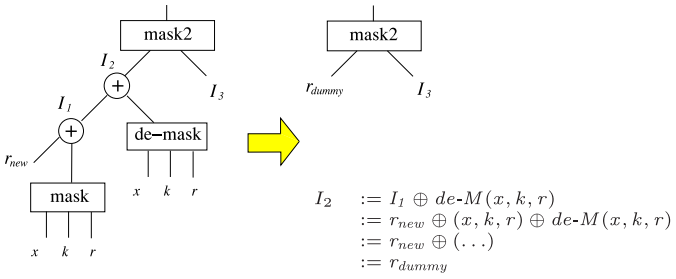


Fig. 4. Checking the QMS by incrementally applying the SMT-based analysis (see [24]). Here, we assume that r_{new} is fresh random bit that has not been used in the support of I_3.

Fig. 3 is a pictorial illustration of the SMT encoding for an intermediate computation result $I(k1, k2, r1, r2)$, where $k1$ and $k2$ are the secret bits and $r1$ and $r2$ are two random bits. The first four boxes, encoding $\Psi_k^0, \ldots, \Psi_k^3$, are copies of the program logic for key bits ($k1k2$) with random bits set to 00, 01, 10, and 11, respectively. The other four boxes, encoding $\Psi_{k'}^0, \ldots, \Psi_{k'}^3$, are copies of the program logic for key bits ($k1'k2'$) with random bits set to 00, 01, 10, and 11, respectively. The formula checks for security against first-order DPA attacks—whether there exist two sets of keys (*k1 k2* and *k1' k2'*) under which the probabilities of I (for being logical 1 and 0, respectively) differs from each other by more than Δ_{qms}.

B. Checking the Fan-In AST Nodes Incrementally

Since the SMT formula size is linear in the size of the program but exponential in the number of random variables, it may become a bottleneck if the program uses a large number s of random bits. To avoid the potential performance problem, we propose an incremental analysis algorithm similar to the one used in SC Sniffer [24], which applies the SMT-based analysis only to small code regions as opposed to the entire fan-in cone of each intermediate computation result. This is crucial for scaling our method to software code of practical size.

Our incremental algorithm can be illustrated by the example in Fig. 4, where the output of mask(x, k, r) is masked again with the new random variable r_{new} before it is de-masked from the old random variable r. In practice, a common used strategy for implementing randomization-based countermeasures is to have a chain of modules, where the inputs of each module are masked before executing its logic, and are de-masked afterward. To avoid having an unmasked intermediate value, the inputs to the successor module are masked with fresh random variables, before they are de-masked from the random variables of the previous module.

In Fig. 4, before verifying mask2, if we have already proved that I_2 is perfectly masked, and r_{new} is a new random variable not used elsewhere (not in computing I_3), then for the purpose of checking mask2, we can substitute I_2 with a new random variable r_{dummy} while verifying mask2.

Due to associativity of the $\oplus$ operator, reordering the masking and de-masking operations would not change the logical result. For example, in Fig. 4, the instruction being analyzed is in mask2(). Since random variable r_{new} is not used inside mask() or de-mask(), or in the support of I_3, we can replace the entire fan-in cone of I_2 by a new random variable r_{dummy} while verifying mask2().

The effectiveness of our incremental algorithm relies on this design pattern. We shall see in the experimental results section that such optimization opportunities are abundant in real applications.

Algorithm 1 Iteratively Computing the QMS of a Given Program

```
1: COMPUTEQMS (Prog) {
2:     Δ_low ← 0.00
3:     Δ_high ← 1.00
4:     while ( Δ_low ≤ Δ_high ) {
5:         Δ_mid ← (Δ_low + Δ_high)/2.0
6:         if ( CHECKQMS( Prog, Δ_mid) = SAT )
7:             Δ_low ← Δ_mid + 0.01;
8:         else
9:             Δ_high ← Δ_high − 0.01;
10:    }
11:    return Δ_low
12: }
```

C. Computing the QMS of Program

Given a C program, we can estimate the QMS of all the intermediate computation results of the program by iteratively invoking our SMT-based verification procedure as a subroutine. We start with $\Delta_{\text{qms}} = 1.0$, and check whether the program satisfies this QMS requirement. If the answer is no, then we decrease Δ_{qms} by a minimum step of 0.01 each time and check again. We stop as soon as the program satisfies the QMS requirement. At that moment, the value for Δ_{qms} is the estimated QMS of the given program.

Algorithm 1 shows the overall flow of our iterative procedure. To make it efficient, we have used the binary search. To further reduce the unnecessary runtime overhead of the SMT-based analysis, we use simple static analysis during preprocessing to quickly identify nodes where $\Delta_{\text{qms}} = 0.0$. For example, if the internal node n does not have any secret bit in its fan-in cone, then by definition there is no side-channel leakage associated with n. Sometimes, even if secret bits appear in the fan-in cone of a node n, they may be DCs. That is, the actual value of node n may be logically independent of these secret bits. In such cases, we can safely skip the SMT-based analysis. We will show in our experiments that

such simple static analysis allows us to skip the SMT-based analysis for a significantly large number of nodes.

In this paper, we focus on verifying security critical software code, such as implementations of cryptographic algorithms, as opposed to arbitrary software applications. The program under verification typically does not have input-dependent control flow, meaning that we can easily remove all the loops and function calls from the code using standard loop unrolling and function in-lining techniques. Furthermore, the program can be transformed into a branch-free representation, where the if-else branches are merged. Finally, since all program variables are bounded integers, we can convert the program to a purely Boolean program through bit-blasting. Therefore, in this paper, our static code analysis method is concerned with only the bit-level representation of a branch-free program.

V. Modeling Bias in Random Number Generators

In previous sections, we assume that random bits used by the masking countermeasures are unbiased in that they output a equal number of logical 1s and 0s over time. However, real-world random number generators may be biased, sometimes intentionally (e.g., a tradeoff between security and efficiency [25]) with either logical 1s or 0s predominating. Such bias may have a significant impact on the QMS of the countermeasure. However, in general, we are not aware of prior work on statically quantifying the impact of bias in masks on the effectiveness of masking countermeasures.

In this section, we first use an example to illustrate the problem. Then, we propose a nonintrusive method for modeling such bias during the static analysis of the software code, as well as the measurement of power leakage on real devices (Section VI).

Consider an example from the MAC-Keccak implementation, where the masking operation for an instruction is $n_{14} = (r_1 \oplus k_1) \oplus (\neg(r_2 \oplus k_2) \wedge (r_3 \oplus k_3)) \oplus ((r_2 \oplus k_2) \wedge r_3)$. Here, the secret bits are k_1, k_2, and k_3 and the random bits are r_1, r_2, and r_3. We can prove that the output n_{14} is perfectly masked if all the random bits are uniformly distributed, i.e., with an equal probability of having logical 1 and 0. However, when r_1, r_2, and r_3 are biased, this may no longer be true.

Take the two values 000 and 001 for the secret bits as an example. By setting k_1, k_2, and k_3 to 000 and 001, respectively, we can simplify the function of n_{14} to make it dependent only on r_1, r_2, and r_3. The Δ_{qms}, which is the difference in probability for n_{14} to be logical 1 and 0, is reduced to $|(P_{001} + P_{100}) - (P_{000} + P_{101})|$, where P_{001}, for instance, is the probability for $r_1 r_2 r_3$ to be 001. If we assume that the probabilities for r_1, r_2, and r_3 are all P_r, the Δ_{qms} is further simplified to $|4P_r^3 - 8P_r^2 + 5P_r - 1|$. In other words, we have $\text{QMS} = 1 - |4P_r^3 - 8P_r^2 + 5P_r - 1|$.

Fig. 5 shows a plot for the relation between the P_r and the QMS. The x-axis is the value of P_r, where 0.5 means there is no bias. The y-axis is the QMS, where 1.0 means there is no first-order side-channel leakage. When $P_r = 0.5$, we can see that QMS $= 1.0$, meaning that the output is perfectly masked. However, as P_r moves away from 0.5, the QMS starts to degrade, meaning that side-channel leakage starts to occur. Although the QMS reaches 1.0 again as P_r reaches 1.0, it does not mean that there is no side-channel leakage when the random bits become constant logical 1s. Instead, it only means that the two selected values (000 and 001) are indistinguishable. However, there are other pairs of values for the secret bits, for which the QMS is not 1.0 when $P_r = 1$. Indeed, among all the P_r values, the only one that leads to perfect masking of n_{14} is $P_r = 0.5$.

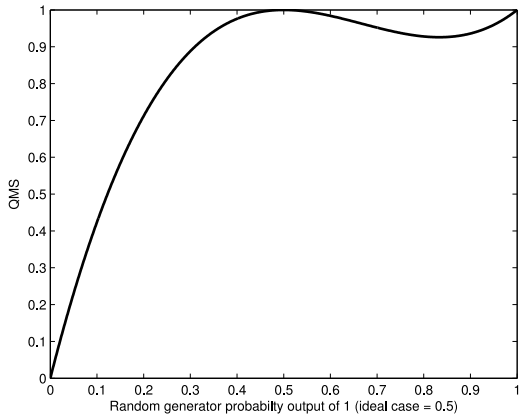


Fig. 5. Relationship between the QMS and the bias of the random number generator (0.5 on the x-axis means there is no bias).

In the presence of bias in the random number generators, we want to make sure that static analysis method still can compute the QMS correctly. Furthermore, we want to find a convenient way to measure the actual power leakage of the software with biased random variables. Toward this end, we propose a new method for constructing biased random bit streams from a set of unbiased random bit streams. We implement such construction in a piece of software code, which in turn can be merged with the original piece of software code, for both static analysis as well as hardware-based measurement.

The main idea of our construction can be illustrated by analyzing the impact of an AND gate on two random bit streams r_1 and r_2. Let $r = r_1 \wedge r_2$. Since the output of r is logical 1 only when both r_1 and r_2 are logical 1s, the probability for r to be logical 1 is $P_r = P_{r_1} P_{r_2}$. If $P_{r_1} = P_{r_2} = 0.5$, for example, meaning the two input bit streams are unbiased, we have $P_r = 0.25$, with logical 0s predominating in the output bit stream. Similarly, if we use an OR gate, i.e., $r = r_1 \vee r_2$, then we have $P_r = 0.75$, with logical 1s predominating in the output bit stream. If, on the other hand, we want to construct a random bit stream with the desired value $P_r = 0.125$, we need to use three unbiased random bit streams: $r = r_1 \wedge r_2 \wedge r_3$.

Using this approach, we can construct a random bit stream with the bias set to a rational value between 0.0 and 1.0. We generate a piece of software code that implements the above construction and merge it with the original software code that uses the biased random bit streams. The combined software code is then subjected to both static analysis of the QMS and hardware-based measure of the actual power leakage.

The advantage of this approach is that the bias modeling is made nonintrusive and transparent with respect to the

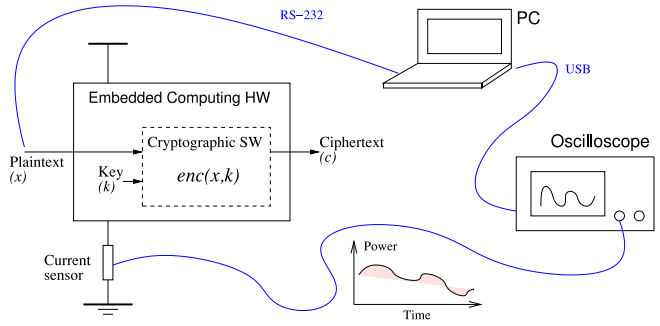


Fig. 6. Side-channel attack measurement system setup.

verification and measurement methods. More specifically, our SMT solver-based method for analyzing the QMS does not need to be changed. What is changed is the input, which includes not only the original code but also the additional code for generating biased random bit streams. Similarly, in our hardware-based measurement experiments (Section VI), we can easily control the bias of the random bit streams without using specialized hardware for biased random number generation.

VI. Measurement on Real Devices

To check whether the QMS actually reflects the masking strength of a piece of software, we conducted a set of power analysis-based side-channel attacks on implementations of MAC-Keccak, AES, and a few other cryptographic algorithms. We ran all software code on a 32-bit Microblaze processor [26] built on a Xilinx Spartan-3e FPGA (Fig. 6). To measure the power consumption of the processor core, we used a Tektronix DPO 3034 oscilloscope and a CT-2 current probe to sample the power consumption of the FPGA. The side-channel attack was conducted using the classic DPA (difference of means [6]). To limit the effect of measurement noise, we collected each trace after running the same software code 128 times and using the oscilloscope to calculate the average. Here, a trace refers to a set of samples taken during the execution of the software.

We used the DPA to determine whether a key guess was correct. Recall that DPA relies on the observation that power consumption variations correlate to the values of the sensitive bits being manipulated. Using the same input vector stream of plaintext as in the measured traces, we computed the value of the sensitive variable assuming that the secret key was one of the key guesses. For an n-bit key, there would be 2^n key guesses. For each key guess, we divided the set of measurement traces into two bins, one for all the sensitive values of logic 0, and one for all the sensitive values of logic 1. Then we computed the difference of means between those two bins, for each key guess. We selected the key guess that result in the maximum difference.

We conducted three sets of experiments. Table I shows the statistics of the benchmarks, including the name of the program, a short description, the lines of code, the number of computation nodes, as well as the numbers of key bits, plaintext bits, and random bits. The first two sets consist of various versions of the MAC-Keccak and AES implementations [27]–[30] with gradually degrading QMS values. We measured the average number of traces needed to determine the secret key. In the third set of experiments, we used a set of recently published software countermeasures [18], [19], [31]–[33], with fixed QMS values, and measured the average number of traces needed to determine the secret key.

Fig. 7 shows our results on the SHA3 benchmark. The x-axis is the QMS value, while the y-axis is the measured average number of traces needed to determine the secret key. Notice that the y-axis is in logarithmic scale. In addition to the measured data, we have plotted an empirical approximation rule generated by hit and trial (dotted curve) to estimate the measured data. We can see that when the QMS value approaches 1.0, the number of traces needed to determine the secret key will approach infinity. This is as expected because QMS $= 1.0$ means that the code is perfectly masked—since there is no information leakage, the implementation is provably secure. However, when the QMS value deviates from 1.0 slightly, the number of traces needed to determine the secret key drops drastically—QMS $= 0.90$ corresponds to around 100 DPA traces. Overall, the side-channel resistance, as measured by the number of traces needed to determine the secret key, is exponentially dependent on QMS.

Fig. 8 shows our results on the AES benchmark. Here, the measured data are similar to those in Fig. 7. Furthermore, we note that the approximate empirical formula computed to estimate the number of required DPA traces has the following relation with the QMS value:

$$N_{\text{trace}} = \frac{1}{(1 - \text{QMS})^c}$$

where $c \approx 2.0$ for these two sets of experiments. This is consistent with theoretical analysis results in the literature, which says that c should be 2.0 since $(1 - \text{QMS})$ represents the standard deviation of power analysis measurements under certain assumptions—see the rough approximation by Mangard [34] of the number of measurement traces needed for a successful DPA, which is a function of the standard deviation of the power analysis measurements. In the context of this paper, what is important is our discovery that the side-channel resistance is indeed closely dependent on QMS.

Table II shows our results on the third set of benchmarks. Here, columns 1 and 2 show the program name and the node to which we have applied the DPA attack. Column 3 shows the QMS value computed statically for the software code. Column 4 shows the number of traces needed to determine the secret key. T.O. means timed out after 100 000 traces are measured. It is worth pointing that we performed second order analysis on P3–P5. Overall, we have observed a similar exponential dependence between the number of measured traces and the QMS value. For example, when the QMS is 0.00—meaning that the node is not masked at all—we have found that the secret key can be determined with merely a handful of DPA traces. When the QMS is 1.00—meaning it is perfectly masked—the key cannot be determined within our time limit

TABLE I
STATISTICS OF MASKED SOFTWARE BENCHMARKS USED IN OUR MEASUREMENT-BASED DPA ATTACK EXPERIMENTS ON REAL DEVICES

Name	Description	Lines of Code	Intermediate Nodes	Key Bits	Plain Bits	Rand Bits
SHA3	A series of masked MAC-Keccak with varying levels of masking (biased random number generators from 0.01 to 0.5 to vary QMS from 0.0 to 1.0)	61	31	3	3	3
AES	A series of masked AES with varying levels of masking (biased random number generators from 0.01 to 0.5 to vary QMS from 0.5 to 1.0)	52	37	8	8	8
P1	CHES13 Masked Key Whitening	79	47	16	16	16
P2	CHES13 De-mask and then Mask	67	31	8	8	16
P3	CHES13 AES Shift Rows	21	21	2	2	2
P4	CHES13 Messerges Boolean to Arithmetic (bit0)	23	24	1	1	2
P5	CHES13 Goubin Boolean to Arithmetic (bit0)	27	60	1	1	2
P6	Logic Design for AES S-Box (1st implementation)	32	9	2	2	2
P7	Masked Chi function MAC-Keccak (1st implementation)	59	19	3	3	4
P8	Masked Chi function MAC-Keccak (2nd implementation)	60	19	3	3	4
P9	Syn. Masked Chi func MAC-Keccak (1st implementation)	66	22	3	3	4
P10	Syn. Masked Chi func MAC-Keccak (2nd implementation)	66	22	3	3	4

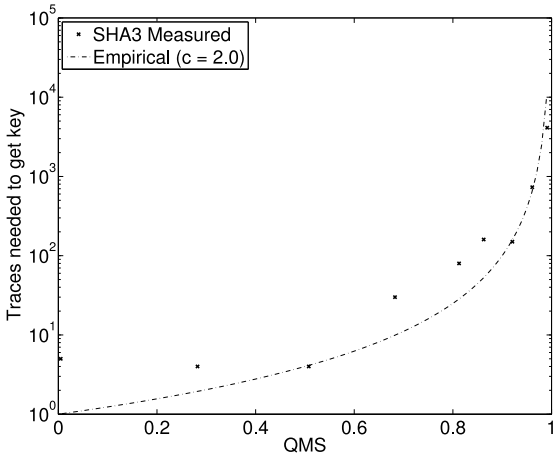


Fig. 7. DPA attacks on MAC-Keccak. Plotting the number of traces needed to determine the key with respect to the QMS value.

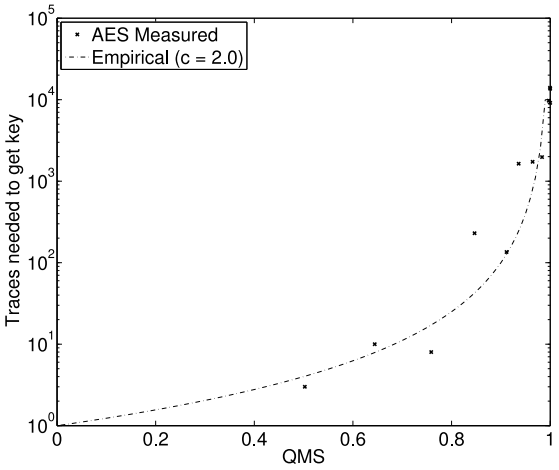


Fig. 8. DPA attacks on AES. Plotting the number of traces needed to determine the key with respect to the QMS value.

of 100 000 traces. When the QMS is between 0.00 and 1.00, the number of DPA traces closely follows the same empirical formula (exponential dependence on the QMS) that we have discovered earlier, but with a slightly different value for constant c.

TABLE II
DPA ATTACKS ON P1–P10: SHOWING THE RELATION BETWEEN QMS AND THE NUMBER OF TRACES NEEDED TO DETERMINE THE KEY

Name	Node	QMS	Trace	Name	Node	QMS	Trace
P1	n011	0.00	2	P1	n012	1.00	T.O.
P2	n21	0.00	3	P2	n 11	1.00	T.O.
P3	st10 $\oplus$ st2	0.00	2	P3	rx2 $\oplus$ st2	1.00	T.O.
P4	X $\oplus$ A3	0.00	2	P4	A1 $\oplus$ A3	1.00	T.O.
P5	X $\oplus$ R2	0.00	3	P5	T1 $\oplus$ R2	1.00	T.O.
P6	n09	0.50	936	P6	n07	1.00	T.O.
P7	n32	0.50	992	P7	n35	1.00	T.O.
P8	n02	0.50	587	P8	n23	1.00	T.O.
P9	n47	0.50	255	P9	n39	1.00	T.O.
P10	n47	0.50	426	P10	n48	1.00	T.O.

VII. EXPERIMENTS

We have implemented our new QMS checking algorithm in a static code analysis tool called SC Sniffer [24]. The new method, called Sniffer-QMS, can verify whether a piece of Boolean program satisfies an arbitrary QMS requirement (ranging from 0.0 to 1.0), whereas the original tool only checks whether it is perfectly masked. In addition, Sniffer-QMS can compute the QMS value for a given program. Our implementation uses the LLVM compiler [35] as the front end, and the Yices SMT solver [9] as the analysis engine at the back end.

During the experiments, we have evaluated the efficiency and effectiveness of our new static code analysis method for both QMS estimation and QMS checking. Our experimental evaluation was designed to answer the following questions.

1) Is it practical to compute the QMS of a program through purely static code analysis?
2) Does the new method offer significant advantages over existing methods such as Sleuth [18]?

Our benchmarks included a set of published implementations and/or masking schemes for cryptographic algorithms [18], [19], [27], [28], [30]–[33], many of which (e.g., P1–P6) were shown to be vulnerable to power side-channel attacks according to our method. The statistics of these benchmarks are presented in Table I. All our experiments were obtained on a desktop computer with a 3.4 GHz Intel i7-2600 CPU, 3.3 GB RAM, and a 32-bit Linux operating system.

Table III shows the results of applying our new method to compute the QMS of a given software. Column 1 shows the name of the software. Column 2 shows the number of internal nodes checked. Columns 3–6 show the QMS computed, including the minimal, maximal, local average, and

TABLE III
STATICALLY COMPUTING THE QMS OF THE C PROGRAMS

Program		QMS				Performance	
Name	Nodes	Min.	Max.	Local Avg	Global Avg	Iters	Time
P1	47	0.00	1.00	0.00	0.66	31	0.13s
P2	31	0.00	1.00	0.00	0.74	23	0.41s
P3	21	0.00	1.00	0.33	0.71	108	1.6s
P4	24	0.00	1.00	0.17	0.93	151	1.7s
P5	60	0.00	1.00	0.17	0.97	367	3.1s
P6	9	0.50	1.00	0.50	0.83	11	0.15s
P7	19	0.00	1.00	0.17	0.86	19	0.17s
P8	19	0.50	1.00	0.50	0.92	20	0.16s
P9	22	0.50	1.00	0.50	0.97	23	0.18s
P10	22	0.50	1.00	0.50	0.97	23	0.24s

TABLE IV
SMT RELATED STATISTICS DURING THE QMS COMPUTATION

Program	Nodes Checked			Iterations		Variables in SMT		
Name	Total	Static	SMT	Total	Per node	Avg.	Min.	Max.
P1	47	31	16	31	1.9	28	28	28
P2	31	23	8	23	2.9	26	26	26
P3	21	6	15	108	2.4	67	30	82
P4	24	15	9	151	6.0	106	30	143
P5	60	52	38	367	4.9	97	30	151
P6	9	7	2	11	5.5	196	174	228
P7	19	16	3	19	6.3	161	34	204
P8	19	16	3	20	6.7	161	34	204
P9	22	20	2	23	11.5	313	272	434
P10	22	19	3	23	7.7	312	275	434

global average. Columns 7 and 8 show the number of iterations and the total execution time. The number of iterations is for the combination of checks on all internal nodes. Also, for P3–P5, we have applied second-order DPA following [18] as opposed to first-order DPA, so each node has been checked against every other node of the program. The results show that our iterative method converged quickly in all cases. Our tool can report several addition pieces of useful information reported by our new method, e.g., which node in the program has the lowest QMS and therefore is the most vulnerable to side-channel attacks.

Table IV shows the statistics of SMT solver-based checks during the above experiments. Here, Column 1 shows the name of each benchmark. Columns 2–4 show the number of nodes in the program, the number of nodes skipped by the simple static analysis (Section IV-C), and the number of nodes checked by the SMT-based method in Algorithm 1, respectively. Columns 5 and 6 show the total number of iterations and the average number of iterations per node needed to compute the QMS in Algorithm 1. The last three columns show the average, minimum, and maximum number of variables in the formulas fed to the SMT solver.

Table V shows the results of applying our new method to check whether a program satisfies a given QMS requirement. For comparison, we have reimplemented and evaluated the Sleuth algorithm of Bayrak *et al.* [18] in our framework. Here, Columns 1 and 2 show the program name and the number of nodes checked. Columns 3–5 show the statistics of Sleuth, including whether it finds any unmasked node, the number of unmasked nodes, and the total execution time. Columns 6–8 show the statistics of our new method, including whether it finds any node that leaks side-channel information, the number of vulnerable nodes found, and the total execution time.

TABLE V
VERIFYING A C PROGRAM AGAINST THE QMS REQUIREMENT

Program		*Sleuth* [18]			*Sniffer-QMS*		
name	nodes	masked	nodes failed	time	masked qms=1.0	nodes failed	time
P1	47	No	16	0.16s	No	16	0.09s
P2	31	No	8	0.21s	No	8	0.14s
P3	21	No	9	1.17s	No	9	1.14s
P4	24	No	2	0.58s	No	2	1.25s
P5	60	No	2	1.19s	No	2	2.53s
P6	9	Yes	0	0.06s	No	2	0.08s
P7	19	No	1	0.15s	No	3	0.12s
P8	19	Yes	0	0.13s	No	2	0.10s
P9	22	Yes	0	0.18s	No	1	0.16s
P10	22	Yes	0	0.20s	No	1	0.18s
P11	128k	Yes	0	91m53s	Yes	0	11m20s
P12	128k	No	2560	92m59s	No	2560	14m45s
P13	128k	Yes	0	97m38s	No	1024	19m26s
P14	152k	Yes	0	132m10s	No	512	37m17s
P15	128k	No	512	113m12s	No	1536	17m44s
P16	131k	No	4096	103m56s	No	4096	18m29s

In addition to the P1–P10 examples, we have experimented on a set of full-sized MAC-Keccak implementations [27] (P11–P16) in order to compare the scalability of the two methods.

From the results, we have observed several advantages of our new method over Sleuth. First, our new method can check for the quantitative masking strength—for any QMS value ranging from 0.00 to 1.00—whereas Sleuth can only check whether a node is masked (whether the QMS is zero or nonzero). The results in Table V clearly show that there are many cases (e.g., in P6 and P8) where the nodes are masked by some random bits, but the masking is not perfect, and therefore the nodes can still leak sensitive information. Second, our new method is more scalable than Sleuth. Although the two methods have comparable run time on small programs, our new method is significantly faster than Sleuth on large programs, despite the fact that it is checking a more sophisticated quantitative property. This is due to the fact that we are using incremental SMT analysis as described in Section IV-B.

VIII. RELATED WORK

This paper is an extension of [1] that proposed the notion of QMS for the first time. In this extended version, we have provided a more detailed description of the static analysis algorithm, explained our modeling of bias in random number generators, and presented more experimental results.

The notion of perfect masking was first introduced by Blömer *et al.* [19], following Shannon's notion of perfect secrecy. They also proposed a provably secure masking scheme for AES. When implemented properly, perfect masking countermeasures are provably secure against power analysis attacks regardless of the technological capability of the adversary. However, they did not consider quantifying the strength of masking countermeasures. To the best of our knowledge, we are the first to propose the new notion of QMS as a metric for evaluating masking countermeasures.

We also propose the first SMT solver-based method for checking the QMS of the source code of cryptographic software. The Sleuth [18] method only checks whether sensitive data are masked by some random bits, but does not check

whether it is perfectly masked. The SC Sniffer method proposed in [24] can check whether sensitive data are perfectly masked, but cannot check it against an arbitrary QMS requirement. Furthermore, neither of these previous methods can statically estimate the QMS of a piece of software code.

There is a large body of work on designing and implementing masking countermeasures for cryptographic algorithms [31]–[33], [36]–[38]. However, in all these prior works, the countermeasures were manually designed and implemented, and in general, there was a lack of automated verification tools to ensure that the masking countermeasures were indeed secure. Our new method would be particularly useful in this situation.

There have been some recent works on automated construction of side-channel countermeasures [39]–[41]. However, they are typically based on compiler transformations that match known leakage patterns and apply predetermined transformation rules. In contrast, we recently proposed a new method for generating masking countermeasures [42] based on inductive synthesis, which is application agnostic and therefore can handle unknown leakage patterns.

There are many other types of side channels through which sensitive information of the software code may be leaked. They include the execution time [43], [44], faults [45], and cache side channels [46], etc. Techniques for detecting and mitigating such side-channel attacks have also been proposed. For example, Köpf *et al.* [12] and Backes *et al.* [13] proposed methods for conducting quantitative information flow analysis. Doychev *et al.* [14] developed a static analysis tool for detecting information leaks through cache side channels. Barthe *et al.* [15] proposed a mitigation method designed for defending against concurrent cache attacks. Since these methods focus on other types of side channels, they are orthogonal to this paper.

IX. Conclusion

We have proposed the new notion of QMS, which can, for the first time, represent the resistance of a masking countermeasure numerically. We have confirmed through evaluations on real devices that the QMS is a good indicator of the actual masking strength of the software. We have developed a static analysis tool to compute the QMS of a C program; the method can also be used to verify a program against a QMS requirement. Our experiments show that the new static analysis method is effective in detecting masking flaws and scalable for handling software of practical size. We have analyzed the source code of a full SHA-3 implementation. For future work, we plan to analyze a full AES. We are also interested in extending our method to handle countermeasures that use other masking schemes such as additive masking, multiplicative masking, and RSA blinding.

Acknowledgment

Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] H. Eldib, C. Wang, M. Taha, and P. Schaumont, "QMS: Evaluating the side-channel resistance of masked software from source code," in *Proc. ACM/IEEE Design Autom. Conf.*, San Francisco, CA, USA, 2014, pp. 1–6.

[2] C. Paar, T. Eisenbarth, M. Kasper, T. Kasper, and A. Moradi, "Keeloq and side-channel analysis-evolution of an attack," in *Proc. Int. Workshop Fault Diagn. Toler. Cryptography*, Lausanne, Switzerland, 2009, pp. 65–69.

[3] 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," in *Proc. 18th ACM Conf. Comput. Commun. Security*, Chicago, IL, USA, 2011, pp. 111–124.

[4] J. Balasch, B. Gierlichs, R. Verdult, L. Batina, and I. Verbauwhede, "Power analysis of Atmel CryptoMemory—Recovering keys from secure EEPROMs," in *Proc. Cryptograph. Track RSA Conf. (CT-RSA)*, San Francisco, CA, USA, 2012, pp. 19–34.

[5] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. New York, NY, USA: Springer, 2007.

[6] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proc. 19th Annu. Int. Cryptol. Conf. Adv. Cryptol. (CRYPTO)*, Santa Barbara, CA, USA, 1999, pp. 388–397.

[7] E. Prouff and M. Rivain, "Masking against side-channel attacks: A formal security proof," in *Advances in Cryptology—EUROCRYPT*. Berlin, Germany: Springer, 2013, pp. 142–159.

[8] C. Lattner and V. Adve, "The LLVM instruction set and compilation strategy," Dept. Comput. Sci., Univ. Illinois Urbana-Champaign, Champaign, IL, USA, Tech. Rep. UIUCDCS-R-2002-2292, Aug. 2002.

[9] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Proc. 18th Int. Conf. Comput. Aided Verif.*, Seattle, WA, USA, 2006, pp. 81–94.

[10] C. Wang, G. D. Hachhtel, and F. Somenzi, *Abstraction Refinement for Large Scale Model Checking*. New York, NY, USA: Springer, 2006.

[11] Z. Yang, C. Wang, F. Ivančić, and A. Gupta, "Mixed symbolic representations for model checking software programs," in *Proc. IEEE Int. Conf. Formal Methods Models Codesign*, Napa, CA, USA, Jul. 2006, pp. 17–24.

[12] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *Proc. Int. Conf. Comput. Aided Verif.*, Berkeley, CA, USA, 2012, pp. 564–580.

[13] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *Proc. IEEE Symp. Security Privacy*, Berkeley, CA, USA, 2009, pp. 141–153.

[14] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A tool for the static analysis of cache side channels," in *Proc. 22nd USENIX Conf. Security*, Auckland, New Zealand, 2013, pp. 431–446.

[15] G. Barthe, B. Köpf, L. Mauborgne, and M. Ochoa, "Leakage resilience against concurrent cache attacks," in *Proc. 3rd Int. Conf. Principles Security Trust*, Grenoble, France, 2014, pp. 140–158.

[16] J. Agat, "Transforming out timing leaks," in *Proc. ACM SIGACT-SIGPLAN Symp. Principles Program. Lang.*, Boston, MA, USA, 2000, pp. 40–53.

[17] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[18] A. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated verification of software power analysis countermeasures," in *Cryptographic Hardware and Embedded Systems—CHES*. Berlin, Germany: Springer, 2013.

[19] J. Blömer, J. Guajardo, and V. Krummel, "Provably secure masking of AES," in *Selected Areas in Cryptography*. Berlin, Germany: Springer, 2004, pp. 69–83.

[20] M. Rivain and E. Prouff, "Provably secure higher-order masking of AES," in *Cryptographic Hardware and Embedded Systems—CHES*. Berlin, Germany: Springer, 2010, pp. 413–427.

[21] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. New York, NY, USA: Springer, 2006.

[22] A. Moradi, S. Guilley, and A. Heuser, "Detecting hidden leakages," in *Proc. Int. Conf. Appl. Cryptography Netw. Security*, Lausanne, Switzerland, 2014, pp. 324–342.

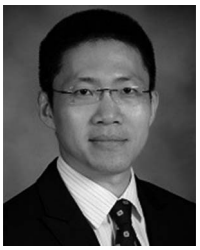
[23] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems—CHES*. Cham, Switzerland, 2004, pp. 16–29.

[24] H. Eldib, C. Wang, and P. Schaumont, "SMT based verification of software countermeasures against side-channel attacks," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2014, pp. 62–77.
[25] 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 *Proc. Design Autom. Test Europe Conf. Exhibit.*, Dresden, Germany, Mar. 2012, pp. 1173–1178.
[26] Xilinx. (2014). *Microblaze Soft Processor Core*. [Online]. Available: http://www.xilinx.com/tools/microblaze.htm
[27] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer. (2013). *Keccak Implementation Overview*. [Online]. Available: http://keccak.neokeon.org/Keccak-implementation-3.2.pdf
[28] NIST. (2013). *Keccak Reference Code Submission to NIST's SHA-3 Competition (Round 3)*. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/Keccak_FinalRnd.zip
[29] M. Taha and P. Schaumont, "Differential power analysis of MAC-Keccak at any key-length," in *Proc. 8th Int. Conf. Adv. Inf. Comput. Security*, Okinawa, Japan, 2013, pp. 68–82.
[30] J. Boyar and R. Peralta, "A small depth-16 circuit for the AES S-Box," in *Proc. Inf. Security Privacy Conf. (SEC)*, Crete, Greece, 2012, pp. 287–298.
[31] C. Herbst, E. Oswald, and S. Mangard, "An AES smart card implementation resistant to power analysis attacks," in *Proc. 4th Int. Conf. Appl. Cryptography Netw. Security*, Singapore, 2006, pp. 239–252.
[32] T. S. Messerges, "Securing the AES finalists against power analysis attacks," in *Fast Software Encryption*. Berlin, Germany: Springer, 2000, pp. 150–164.
[33] L. Goubin, "A sound method for switching between Boolean and arithmetic masking," in *Cryptographic Hardware and Embedded Systems—CHES*. Berlin, Germany: Springer, 2001, pp. 3–15.
[34] S. Mangard, "Hardware countermeasures against DPA—A statistical analysis of their effectiveness," in *Proc. Topics Cryptol. Cryptograph. Track RSA Conf. (CT-RSA)*, San Francisco, CA, USA, Feb. 2004, pp. 222–235.
[35] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, San Jose, CA, USA, 2004, pp. 75–88.
[36] E. Oswald, S. Mangard, N. Pramstaller, and V. Rijmen, "A side-channel analysis resistant description of the AES S-Box," in *Proc. Int. Workshop Fast Softw. Encrypt.*, Paris, France, 2005, pp. 413–423.
[37] D. Canright and L. Batina, "A very compact 'perfectly masked' S-Box for AES," in *Proc. 6th Int. Conf. Appl. Cryptography Netw. Security*, New York, NY, USA, 2008, pp. 446–459.
[38] A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang, "Pushing the limits: A very compact and a threshold implementation of AES," in *Advances in Cryptology—EUROCRYPT*. Berlin, Germany: Springer, 2011, pp. 69–88.
[39] A. G. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne, "A first step towards automatic application of power analysis countermeasures," in *Proc. ACM/IEEE Design Autom. Conf.*, New York, NY, USA, 2011, pp. 230–235.
[40] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *Cryptographic Hardware and Embedded Systems—CHES*. Leuven, Belgium, 2012, pp. 58–75.
[41] G. Agosta, A. Barenghi, and G. Pelosi, "A code morphing methodology to automate power analysis countermeasures," in *Proc. ACM/IEEE Design Autom. Conf.*, San Francisco, CA, USA, 2012, pp. 77–82.
[42] H. Eldib and C. Wang, "Synthesis of masking countermeasures against side channel attacks," in *Proc. Int. Conf. Comput. Aided Verif.*, Vienna, Austria, 2014, pp. 114–130.
[43] P. C. Kocher, "Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems," in *Proc. 16th Annu. Int. Cryptol. Conf. (CRYPTO)*, Santa Barbara, CA, USA, 1996, pp. 104–113.
[44] B. Köpf and M. Dürmuth, "A provably secure and efficient countermeasure against timing attacks," in *Proc. 22nd IEEE Symp. Comput. Security Found.*, Port Jefferson, NY, USA, 2009, pp. 324–335.
[45] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Proc. 17th Annu. Int. Cryptol. Conf. (CRYPTO)*, Santa Barbara, CA, USA, 1997, pp. 513–525.
[46] P. Grabher, J. Großschädl, and D. Page, "Cryptographic side-channels from low-power cache memory," in *Proc. 11th Int. Conf. Cryptography Cod.*, Cirencester, U.K., 2007, pp. 170–184.

Hassan Eldib received the B.S. and M.S. degrees from the Arab Academy for Science and Technology and Maritime Transport, Alexandria, Egypt, in 2006 and 2009, respectively. He is currently pursuing the Ph.D. degree with the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA.

His current research interests include developing automated synthesis and verification methods for embedded control software and cryptographic software.

Mr. Eldib was a recipient of the 2013 FMCAD Best Paper Award.



Chao Wang (M'02) received the Ph.D. degree from the University of Colorado Boulder, Boulder, CO, USA, in 2004.

He is currently an Assistant Professor with the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA. He has published a book and over 50 papers in top venues in the areas of software engineering and formal methods.

Dr. Wang was a recipient of the ACM SIGDA Outstanding Ph.D. Dissertation Award in 2004, the ACM TODAES Best Journal Paper of the Year Award in 2008, the ACM SIGSOFT Distinguished Paper Award in 2010, the NSF Faculty CAREER Award in 2012, and the FMCAD Best Paper Award and the ONR Young Investigator Award in 2013.

Mostafa Taha (S'12–M'14) received the B.E. and M.S. degrees in electrical engineering from Assiut University, Asyut, Egypt, in 2004 and 2008, respectively, and the Ph.D. degree in computer engineering from Virginia Tech, Blacksburg, VA, USA, in 2014.

He is currently an Assistant Professor with Assiut University, Asyut, Egypt. His current research interests include hardware security and implementation attacks.

Mr. Taha served as an Academic Reviewer for several conferences, including CHES, COSADE, CARDIS, HOST, and several journals, including the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, and IACR JCEN. He is a member of IACR.

Patrick Schaumont (SM'06) received the Ph.D. degree in electrical engineering from the University of California at Los Angeles, Los Angeles, CA, USA, in 2004.

He is currently an Associate Professor of Computer Engineering with Virginia Polytechnic Institute and State University, Blacksburg, VA, USA. His current research interests include cryptographic engineering, and the conception, design, and implementation of next generation embedded systems.

Prof. Schaumont is an Associate Editor with several journals, including the IEEE TRANSACTIONS ON COMPUTERS, IACR JCEN, ACM TODAES, and ACM TECS. He has served on the Program Committee of international conferences such as CHES, DATE, DAC, and IEEE HOST. He is a member of the IACR.