QMS: Evaluating the Side-Channel Resistance of Masked Software from Source Code *

Hassan Eldib, Chao Wang
Department of ECE
Virginia Tech
Blacksburg, VA 24061, USA
{heldib,chaowang}@vt.edu

Mostafa Taha, Patrick Schaumont
Department of ECE
Virginia Tech
Blacksburg, VA 24061, USA
{mtaha,schaum}@vt.edu

ABSTRACT

Many commercial systems in the embedded space have shown weakness against power analysis based side-channel attacks in recent years. Designing countermeasures to defend against such attacks 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. The QMS can be automatically computed from the source code of a countermeasure implementation. Our experiments, based on side-channel measurement on real devices, show that the QMS accurately quantifies the side-channel resistance of the software implementation.

Categories and Subject Descriptors
K.6.5 [**Security and Protection**]: Physical security;
D.2.4 [**Software/Program Verification**]: Formal methods

General Terms: Security, verification

Keywords: Side channel attack, differential power analysis, countermeasure, quantitative masking strength, SMT solver

1. INTRODUCTION

In recent years, many commercial systems in the embedded space have shown weaknesses against power analysis based side-channel attacks [18, 16, 1], where an adversary can utilize secondary information such as heat and power dissipation and electromagnetic radiation resulting from the execution of sensitive algorithms on these devices. For example, the power consumption of an embedded device executing instruction `a=t`$\oplus$`k` may depend on the value of the secret `k` [14]. Masking, which is a randomization technique for removing the statistical dependency between 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: $a_m = a \oplus r$ [1, 19]. Later, the original variable can be restored by a second XOR operation: $a_m \oplus r = a$. Other similar countermeasures have used additive masking ($a_m = a + r \ mod \ n$), multiplicative masking ($a_m = a * r \ mod \ n$), as well as application-specific masking such as RSA blinding ($a_m = ar^e \ mod \ N$).

However, side-channel countermeasures are difficult to design and implement because the process is labor intensive and error prone. There is also a lack of formal analysis methods for quantifying how secure a countermeasure implementation really is. This is a problem because the source of the information leakage is not the cryptographic software but the hardware that executes the software. For average software developers who do not know all the architectural details of the device, it can be difficult to understand when side-channel information may be leaked.

In this paper, we introduce the notion of *quantitative masking strength (QMS)* to quantify 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 implementation, we record the number of traces required to successfully break the countermeasure. Our experimental results show that the number of 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 to compute the QMS of a given C program. The tool can also be used as a verification procedure to decide whether a program satisfies a given QMS requirement.

Our code analysis tool builds on the LLVM compiler and the Yices SMT solver [7]. We encode the problem into a series of quantifier-free first-order logic formulas, whose satisfiability can be decided by the SMT solver. Although in the literature there exists some work on checking the security of mask software code, e.g. using type-based information flow analysis [20], they are less accurate and may generate many false positives. Bayrak *et al.* [2] have used SAT solvers to check if a software is *masked*, but they cannot quantify the masking strength. To the best of our knowledge, our method is the first automated static analysis method for checking the strength of masking quantitatively.

We have conducted experiments on a set of cryptographic software implementations to evaluate the performance of our tool. The benchmarks include countermeasures proposed for AES as well as MAC-Keccak, a MAC based on the new SHA-3 standard. Our results show that the new method is effective in detecting side-channel leaks in the software code and is scalable enough to handle cryptographic software of practical size.

To sum up, this paper makes the following contributions:

- We propose the new notion of *quantitative masking strength (QMS)* as a way to quantify the side-channel resistance of a masked software implementation.
- We conduct DPA attack experiments on real devices to confirm that the QMS is indeed a good indicator of the side-channel resistance in practice.
- We propose a static code analysis method for computing the QMS of a software program without measurement. The tool can also be used to formally verify that a program satisfies a given QMS requirement.

*

DAC'14, June 01 - 05, 2014, San Francisco, California, USA.

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

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

Figure 1: Although `o1,o2,o3` are masked by random bits `r1` and `r2`, they may still leak secret information about `k`.

2. PRELIMINARIES

In this section, we provide a brief introduction to side-channel attacks and randomization based countermeasures. Following the notation used by Blömer *et al.* [4], we assume that the program to be analyzed implements a function $c \leftarrow F(x, k)$, where x is the plaintext, k is the secret key, and c is the ciphertext. Let I_1, I_2, ..., 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 I_i statistically independent of k.

When $F(x, k)$ is a linear function in the Boolean domain, masking and de-masking are trivial due to properties of the $\oplus$ operations. However, when $F(x, k)$ is a non-linear function, masking and de-masking often require a complete redesign of the software. This process is both labor intensive and error prone, and currently cannot be automated. Indeed, designing a new masking scheme for a reputable cryptographic algorithm such as AES or MAC-Keccak is considered publishable work in top cryptography venues.

In this paper, we assume that an adversary knows the pair (x, c) of plaintext and ciphertext in $c \leftarrow F(x, k)$. 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 a true random number. The goal of the adversary is to compute the secret key (k). Kocher *et al.* [12] demonstrated in their seminal work that it is possible to deduce k using a statistical method known as differential power analysis (DPA).

A necessary condition for side-channel resistance is for all the intermediate computation results of a function to be *insensitive*, as in Bayrak *et al.* [2]. An intermediate result I_i is *sensitive* if it depends on the secret/plaintext and, at the same time, it does not depend on any random variable. According to [2], this dependency analysis is equivalent to computing *don't cares (DCs)* in logic synthesis: If random bit r is a *don't care* of I_i, then I_i does not depend on r. Recall that r is a *don't care* 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 *depending on a random bit* does not mean that I_i is statistically independent from the secret.

Figure 1 shows an example, where `k` is the secret, `r1` and `r2` are the random variables, 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,r2` and therefore are *insensitive* [2], but three of them still leak secret information. 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). In contrast, `o4` is the only side-channel resistant output because it 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 side-channel analysis, a leakage model specifies the amount of side-channel information observable during program execution. In simple and differential power analysis based attacks, an effective and widely used leakage model, for a single instruction, is the *Hamming Weight (HW)* of the operand, and for two consecutive instructions, is the *Hamming Distance (HD)* of the two operands. It is also the model used in this paper.

3. QUANTITATIVE MASKING STRENGTH

Given a pair (x, k) of plaintext and secret key for the function $F(x, k)$, an s-bit random number r uniformly distributed in the domain $R = \{0, 1\}^s$, and d intermediate results $I_1, \ldots, I_d$, we use $D_{x,k}(R)$ to denote the joint distribution of $I_1, \ldots, I_d$. Here, d represents the maximum number of intermediate computation results whose power side-channel information can be observed by an adversary. If $D_{x,k}(R)$ is statistically independent of the secret k, we say that the function is *order-d* perfectly masked [4]. Otherwise, the function is vulnerable to side-channel attacks, and we would like to quantify the bias of $D_{x,k}(R)$, denoted Δ_{qms}, with respect to x and k.

Definition 1 *Given an implementation of function $F(x, k)$ and a set of intermediate computation results $\{I_i(x, k, r)\}$, we define the quantitative masking strength (QMS) as the minimal value of $(1 - \Delta_{qms})$ such that, for all d-tuple $\langle I_1, \ldots, I_d \rangle$,*

$$|D_{x,k}(R) - D_{x',k'}(R)| \leq \Delta_{qms} \quad \textit{for any } (x, k) \textit{ and } (x', k') \ .$$

In this sense, the *perfect masking* criterion introduced by Blömer *et al.* [4] is an extreme where $\Delta_{qms} = 0$. The *sensitivity* criterion introduced by Bayrak *et al.* [2] is another extreme where $\Delta_{qms} = 1$. 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. As an example, consider the four masking schemes in Figure 1. In the context of *order-1* side-channel attacks, we have

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

All four outputs are *insensitive* according to [2] because of their logical dependence on the random bits, but only `o4` is statistically independent of the secret k.

To check if a function satisfies the given QMS requirement, we need to decide whether there exists a d-tuple $\langle I_1, \ldots, I_d \rangle$ such that $|D_{x,k}(R) - D_{x',k'}(R)| > \Delta_{qms}$ for some (x, k) and (x', k'). The function $F(x, k)$ satisfies the QMS requirement if and only if no such d-tuple exists for the given Δ_{qms} and the given d. Note that $d = 1, 2, \ldots, t$ specifies the order of the side-channel attack. In an order-d attack, we assume that an adversary can measure the leakage of d intermediate computation results simultaneously.

The main challenge for static code analysis – whether to compute the QMS of a given program or to verify that the program satisfies the given QMS requirement – is to compute $D_{x,k}(R)$. 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 $I(x, k, r)$, we check whether it satisfies the QMS requirement. Following Definition 1, we can formulate the *order-1* QMS check as a satisfiability problem as follows:

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

Here, x is the plaintext, k and k' are two different values of the secret key, and r is the s-bit random number in domain $R = \{0, 1\}^s$. For any fixed (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 $I(x, k', r)$. Assume that r is uniformly distributed in domain $R = \{0, 1\}^s$, the summations represent the probabilities of I being logical 1 under key values k and k', respectively.

If the above formula is satisfiable, there exist x and two keys (k, k') such that the distribution of $I(x, k, r)$ 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.

```
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 : }
```

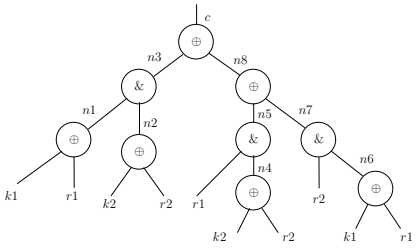

Figure 2: A program and the abstract syntax tree (AST) nodes.

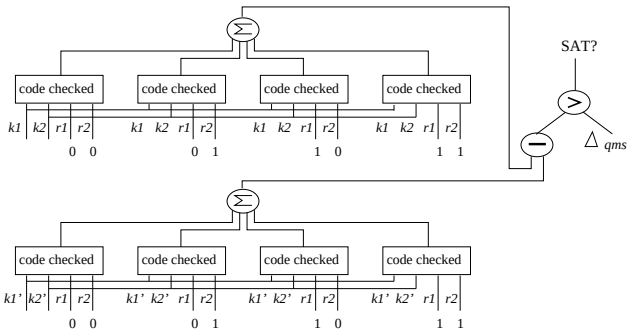

Figure 3: SMT encoding to verify the QMS w.r.t. $(k1, k2)$**.**

4. STATIC CODE ANALYSIS

In this section, we first present our verification procedure, which takes a program and a QMS as input and checks whether the program satisfies the QMS requirement. Then, we present our algorithm for estimating the QMS of a given program, which uses the aforementioned verification procedure as a subroutine.

4.1 Checking a Program against a QMS Requirement

Our method is based on translating the problem into a set of quantifier-free first-order logic (FOL) formulas and then deciding the formulas using an SMT solver. This is an extension of our previous work [9] on checking whether a cryptographic software program is *perfectly masked* [4]. For each intermediate computation result $I(x, k, r)$, we construct the formula Φ that is satisfiable if and only if there exist a plaintext x and two key values k and k' 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 satisfiability (SAT) based verification techniques have been widely used in EDA for checking functional correctness properties, our method is significantly different from them because QMS is a quantitative property and is statistical in nature. Since the property is statistical, it cannot be directly checked by functional verification techniques such as model checking [6, 22, 13, 25, 23].

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 Figure 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. 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 Figure 2 is a masked version of $c \leftarrow (k1 \& k2)$, where $k1$ and $k2$ are secret keys, $r1$ and $r2$ are random variables, and c is the computation result. The return value c is logically equivalent to $(k1 \& k2) \oplus (r1 \& r2)$. This masking scheme [4] is used to make the power consumption independent from the values of $k1$ and $k2$. The corresponding demasking function (not shown in the figure) is $c \oplus (r1 \& r2)$. Therefore, demasking would produce the desired value $(k1 \& k2)$.

Our method will determine if all intermediate variables of the program have a masking strengthen higher than Δ_{qms}. Let Φ denote the SMT formula to be created for checking the intermediate result $I(x, k, r)$. Let s be the number of random bits in r. Our encoding method ensures that Φ is satisfiable if and only if I violates the QMS requirement. Therefore, we define Φ as follows:

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

where the subformulas are defined as follows:

- ***Program logic*** **(**Ψ_k^r**)**: Each subformula Ψ_k^r encodes a copy of the functionality of $I(x, k, r)$, with the random variable r set to a concrete value in $\{0, \ldots, 2^s - 1\}$ and the key set to value k or k'. All copies share the same plaintext value x.
- ***Boolean-to-int*** **(**Ψ_{b2i}**)**: It encodes the conversion of the output of $I(x, k, r)$ from Boolean to integer (true becomes 1 and false becomes 0), so that the integer values can be summed up later to compute $\Sigma_{r \in R} I(x, k, r)$.
- ***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 $I(x, k, r)$ and the other for $I(x, k', r)$.
- ***Different sums*** **(**Ψ_{diff}**)**: It asserts that the difference between the two summations is bigger than the required Δ_{qms}.

Figure 3 is a pictorial illustration of the SMT encoding for output $I(k1, k2, r1, r2)$, where $k1, k2$ are the secret bits and $r1, r2$ 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 distributions of I differs from each other by more than Δ_{qms}.

4.2 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 algorithm, which applies the SMT based analysis only to small code regions of the program as opposed to the entire fan-in cone of each intermediate computation result. This is crucial for scaling our method to code of practical complexity.

Our incremental algorithm can be illustrated by Figure 4, where the output of *mask(x,k,r)* is masked again with the new random variable r_{new} before it is demasked from the old random variable r. 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 demasking operations would not change the logical result. For example, in Figure 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 the following observation. In practice, a common used strategy for implementing randomization based countermeasures is to have a chain of

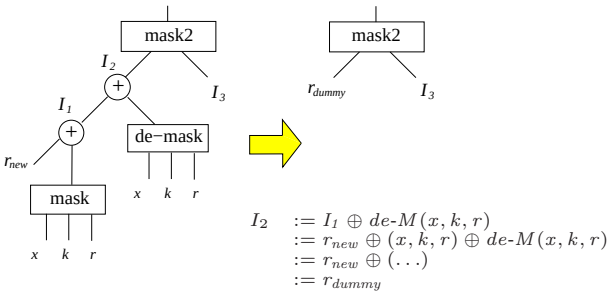

Figure 4: Incremental applying the SMT based analysis only to small fan-in region (assume r_{new} is not in the support of I_3).

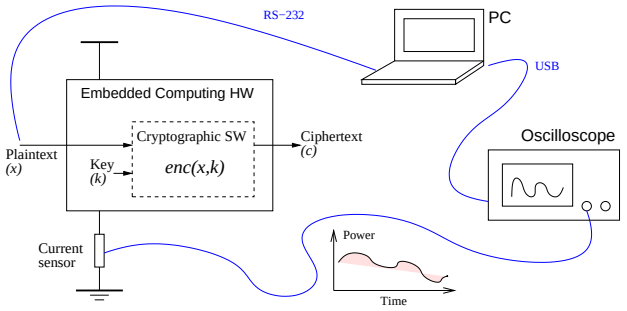

Figure 5: The side-channel attack measurement system setup.

modules, where the inputs of each module are masked before executing its logic, and are demasked afterward. To avoid having an unmasked intermediate value, the inputs to the successor module are masked with fresh random variables, before they are demasked from the random variables of the previous module. We shall see in the experimental results section that such optimization opportunities are abundant in real applications.

4.3 Estimating the QMS of a Given Program

Given a program, we can estimate the QMS of all the intermediate computation results by iteratively invoking our SMT based verification procedure as a subroutine. We start with $\Delta_{qms} = 1.0$, and check whether the program satisfies this QMS requirement. If the answer is no, then we decrease Δ_{qms} 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.

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: }
```

It is worth pointing out that in this work, we focus on verifying 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 inlining 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.

5. MEASUREMENT ON REAL DEVICES

To check if QMS reflects the masking strength of a software, we conducted a set of side-channel attacks on implementations of countermeasures for MAC-Keccak, AES, and a few other cryptographic algorithms. We ran all software code on a 32-bit Microblaze processor [24] built on a Xilinx Spartan-3e FPGA (Figure 5). 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 differential power analysis (difference of means [12]). 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 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 compute 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 divide 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 compute the difference of means between those two bins, for each key guess. We select the key guess that result in the maximum difference.

We have conducted three sets of experiments. Table 1 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 ASE implementations [3, 17, 21, 5] 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 [2, 11, 15, 10, 4], with fixed QMS values, and measured the average number of traces needed to determine the secret key.

Figure 6 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 (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.

Figure 7 shows our results on the AES benchmark. Here, the measured data are similar to those in Figure 6. 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: $N_{trace} = \frac{1}{(1-\text{QMS})^c}$, where $c \approx 2.2$ for these two sets of experiments. In general, c is an empirical constant that ultimately will be decided by the actual hardware and measurement set-up. We shall leave the investigation of the theoretical nature of this constant to future work. What is important is that, overall, the

Table 1: The statistics of masked software benchmarks used in our measurement based DPA attack experiments on real devices. Here, *Key*, *Plain*, and *Rand* represent the number of bits in the secrete key, plaintext, and random variable, respectively.

Name	Description	Lines of Code	Intermediate Nodes	Key	Plain	Rand
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

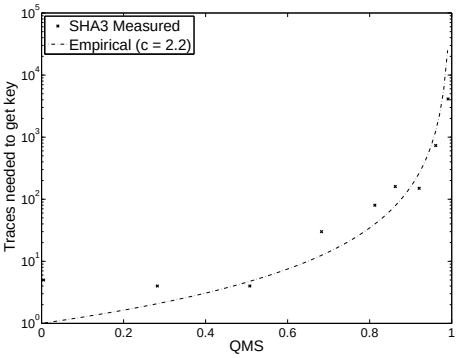

Figure 6: DPA attacks on SHA3: plotting the number of traces needed to determine the key with respect to the QMS value.

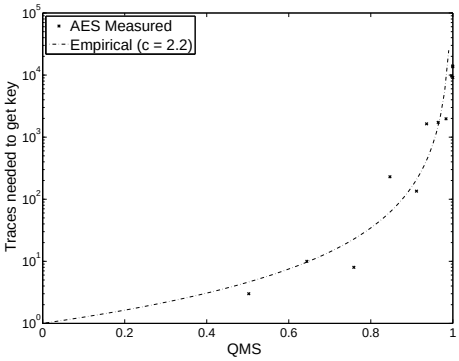

Figure 7: DPA attacks on AES: plotting the number of traces needed to determine the key with respect to the QMS value.

Table 2: 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.

side-channel resistance is exponentially dependent on QMS.

Table 2 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 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.

6. EXPERIMENTAL RESULTS

We have evaluated the efficiency of our new static code analysis method for QMS estimation and checking in the context of related work. Our experimental evaluation was designed to answer the following questions:

- Is it practical to compute the QMS of a C program through purely static code analysis?
- Does the new method offer significant advantages over existing methods such as *Slueth* [2]?

Our benchmarks included a set of recently published masking countermeasures [2, 5, 11, 15, 10, 4, 3, 17] whose statistics have been shown in Table 1. 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 3 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 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 [2] 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. Due to page limit, we omit the description of several 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 4 shows the results of applying our new method to check whether a program satisfies a given QMS requirement. For compar-

Table 3: 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

ison, we have re-implemented and evaluated the *Sleuth* algorithm of Bayrak *et al.* [2] 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. In addition to the P1-P10 examples, we have experimented on a set of full-sized MAC-Keccak implementations [3] (P11-P16) in order to compare the scalability of the two methods.

Table 4: Verifying a C program against the QMS requirement.

Program		*Sleuth* [2]			*New*		
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

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 non-zero). The results in Table 4 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 4.2.

7. CONCLUSIONS

We have proposed the notion of quantitative masking strength (QMS), which can, for the first time, represent the side-channel resistance of a masking countermeasure numerically. We have confirmed through experiments that the QMS is a good indicator of the actual masking strength of the software. We have developed a new static analysis tool to compute the QMS of a C program. The method can also be used as a procedure to formally verify a program against a QMS requirement. Our experimental results show that the new static analysis method is effective in detecting masking flaws and is scalable to handle cryptographic software code of practical size. For future work, we plan to extend this method to handle countermeasures that use other masking schemes such as additive masking, multiplicative masking, and RSA blinding. We also plan to leverage it in our incremental inductive synthesis framework [8] to generate countermeasures automatically.

REFERENCES

[1] J. Balasch, B. Gierlichs, R. Verdult, L. Batina, and I. Verbauwhede. Power analysis of Atmel CryptoMemory - recovering keys from secure EEPROMs. In *CT-RSA*, 2012.

[2] A. Bayrak, F. Regazzoni, D. Novo, and P. Ienne. Sleuth: Automated verification of software power analysis countermeasures. In *CHES*, 2013.

[3] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer. Keccak implementation overview. URL: http://keccak.neokeon.org/Keccak-implementation-3.2.pdf.

[4] J. Blömer, J. Guajardo, and V. Krummel. Provably secure masking of AES. In *Selected Areas in Cryptography*, 2004.

[5] J. Boyar and R. Peralta. A small depth-16 circuit for the AES S-Box. In *SEC*, pages 287–298, 2012.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[7] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.

[8] H. Eldib and C. Wang. An SMT based method for optimizing arithmetic computations in embedded software code. In *FMCAD*, 2013.

[9] H. Eldib, C. Wang, and P. Schaumont. SMT based verification of software countermeasures against side-channel attacks. In *TACAS*, 2014.

[10] L. Goubin. A sound method for switching between boolean and arithmetic masking. In *CHES*, pages 3–15, 2001.

[11] C. Herbst, E. Oswald, and S. Mangard. An AES smart card implementation resistant to power analysis attacks. In *ACNS*, pages 239–252, 2006.

[12] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[13] B. Li, C. Wang, and F. Somenzi. A satisfiability-based approach to abstraction refinement in model checking. *ENTCS*, 89(4), 2003.

[14] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Sards*. Springer, 2007.

[15] T. S. Messerges. Securing the AES finalists against power analysis attacks. In *Fast Software Encryption*, 2000.

[16] 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. *IACR Cryptology*, 2011.

[17] NIST. Keccak reference code submission to the SHA-3 competition. URL: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/Keccak_FinalRnd.zip.

[18] C. Paar, T. Eisenbarth, M. Kasper, T. Kasper, and A. Moradi. Keeloq and side-channel analysis-evolution of an attack. In *FDTC*, pages 65–69, 2009.

[19] E. Prouff and M. Rivain. Masking against side-channel attacks: A formal security proof. In *EUROCRYPT*. 2013.

[20] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[21] M. Taha and P. Schaumont. Differential power analysis of MAC-Keccak at any key-length. In *IWSEC*, 2013.

[22] C. Wang, G. D. Hachhtel, and F. Somenzi. *Abstraction Refinement for Large Scale Model Checking*. Springer, 2006.

[23] C. Wang, H. Jin, G. Hachtel, and F. Somenzi. Refining the SAT decision ordering for bounded model checking. In *DAC*, San Diego, CA, 2004.

[24] Xilinx. Microblaze soft processor core. URL: http://www.xilinx.com/tools/microblaze.htm.

[25] Z. Yang, C. Wang, F. Ivančić, and A. Gupta. Mixed symbolic representations for model checking software programs. In *MEMOCODE*, pages 17–24, July 2006.