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 ($a_m = a + r \mod n$), as well as application-specific masking such as RSA blinding ($a_m = a^r \mod N$).

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 masked 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC'14, June 01 - 05, 2014, San Francisco, California, USA.
Copyright 2014 ACM 978-1-4503-2730-5/14/06 ...$15.00.
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 \( I_1, I_2, \ldots, I_t \) 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 \( I_i \) 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 \( \text{don't care (DCs)} \) 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 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, \( r_1 \) and \( r_2 \) are the random variables, and \( o_1, o_2, o_3 \), and \( o_4 \) are the results of four masking schemes. According to the truth table on the right-hand side, all four outputs depend on \( r_1, r_2 \) and therefore are insensitive [2], but three of them still leak secret information. When \( o_1 \) is logical 1, we know for sure that the secret \( k \) is also 1, regardless of the values of the random variables. Similarly, when \( o_2 \) is logical 0, we know for sure that \( k \) is also 0. When \( o_3 \) is logical 1 (or 0), there is a 75% chance that \( k \) is logical 1 (or 0). In contrast, \( o_4 \) 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 \( o_4 \) 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 \( \Delta_{qms} \), with respect to \( x \) and \( k \).

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

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

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

\[
\Delta_{qms}(o_1) = 1/4 - 0/4 = 0.25 \quad \Delta_{qms}(o_2) = 4/4 - 3/4 - 0.25 \\
\Delta_{qms}(o_3) = 3/4 - 1/4 - 0.50 \quad \Delta_{qms}(o_4) = 2/4 - 2/4 - 0.00
\]

All four outputs are insensitive according to [2] because of their logical dependence on the random bits, but only \( o_4 \) 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 \( d \)-tupple \( (I_1, \ldots, I_d) \) such that

\[
|D_{x,k}(R) - D_{x', k'}(R)| \leq \Delta_{qms}
\]

for some \( (x, k) \) and \( (x', k') \). The function \( F(x, k) \) satisfies the QMS requirement if and only if no such \( d \)-tupple exists for the given \( \Delta_{qms} \) 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:

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

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 \( \Sigma_{x \in R} I(x, k, r) \) represents the number of satisfying assignments of \( I(x, k, r) \), and the summation \( \Sigma_{x \in R} I(x, k', r) \) represents the number of satisfying assignment of \( I(x, k', r) \). Assume that \( r \) is uniformly distributed in domain \( R = \{0, 1\}^t \), 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 \( \Delta_{qms} \). 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.
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 \( \Phi \) 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 \( \Delta_{qms} \). 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 with a Boolean operation of one of the following types: AND, OR, XOR. Each internal node represents the result of the operation between two key values. The corresponding demasking function is as follows:

\[
\Phi := \bigwedge_{r=0}^{2^n-1} \Psi_{r}^{k} \land \bigwedge_{r=0}^{2^n-1} \Psi_{r}^{k'} \land \Psi_{sum} \land \Psi_{diff},
\]

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

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 of random variables. 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 commonly 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 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 $\Delta_{qms}$ and check again. We stop as soon as the program satisfies the QMS requirement. After that moment, the value for $\Delta_{qms}$ 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: COMPUTE QMS (Prog) {
2: $\Delta_{low} \leftarrow 0.00$
3: $\Delta_{high} \leftarrow 1.00$
4: while ($\Delta_{low} \leq \Delta_{high}$) {
5: $\Delta_{mid} \leftarrow (\Delta_{low} + \Delta_{high})/2.0$
6: if (CHECK QMS (Prog, $\Delta_{mid}$) = SAT) {
7: $\Delta_{low} \leftarrow \Delta_{mid} + 0.01$
8: } else {
9: $\Delta_{high} \leftarrow \Delta_{high} - 0.01$
10: }
11: return $\Delta_{low}$
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 AES 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: $N_{\text{traces}} = \epsilon \cdot (\text{QMS})^c$, where $\epsilon \approx 2.2$ 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 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 secret key, plaintext, and random variable, respectively.

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

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.

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.

<table>
<thead>
<tr>
<th>Program</th>
<th>QMS</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Min.</td>
<td>Max.</td>
</tr>
<tr>
<td>P1</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>P2</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>P3</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>P4</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>P5</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>P6</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>P7</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>P8</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>P9</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>P10</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

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