SMT-Based Verification of Software Countermeasures against Side-Channel Attacks

Hassan Eldib, Chao Wang, and Patrick Schaumont

Department of ECE, Virginia Tech, Blacksburg, VA 24061, USA
{heldib,chaowang,schaum} @vt.edu

Abstract. A common strategy for designing countermeasures against side channel attacks is using randomization techniques to remove the statistical dependency between sensitive data and side-channel emissions. However, this process is both labor intensive and error prone, and currently, there is a lack of automated tools to formally access how secure a countermeasure really is. We propose the first SMT solver based method for formally verifying the security of a countermeasures against such attacks. In addition to checking whether the sensitive data are *masked*, we also check whether they are *perfectly masked*, i.e., whether the joint distribution of any d intermediate computation results is independent of the secret key. We encode this verification problem into a series of quantifier-free first-order logic formulas, whose satisfiability can be decided by an off-the-shelf SMT solver. We have implemented the new method in a tool based on the LLVM compiler and the Yices SMT solver. Our experiments on recently proposed countermeasures show that the method is both effective and efficient for practical use.

1 Introduction

Security analysis of the hardware and software systems implemented in embedded devices is becoming increasingly important, since an adversary may have physical access to such devices and therefore can launch a whole new class of side-channel attacks, which utilize secondary information resulting from the execution of sensitive algorithms on these devices. For example, the power consumption of a typical embedded device executing the instruction `tmp=text⊕key` depends on the value of the secret `key` [12]. This value can be reliably deduced using a statistical method known as *differential power analysis* (DPA [10,19]). In recent years, many commercial systems in the embedded space have shown weaknesses against such attacks [16,14,1].

A common mitigation strategy against such attacks is using randomization techniques to remove the statistical dependency between the sensitive data and the side-channel information. This can be done in multiple ways. Boolean masking, for example, uses an XOR operation of a random number r with a sensitive variable a to obtain a masked (randomized) variable: $a_m = a \oplus r$ [1,17]. Later, the sensitive variable can be restored by a second XOR operation with the same random number: $a_m \oplus r = a$. Other randomization based countermeasures have used additive masking ($a_m = a + r \ mod \ n$), multiplicative masking ($a_m = a * r \ mod \ n$), and application-specific code transformations such as RSA blinding ($a_m = ar^e \ mod \ N$).

E. Ábrahám and K. Havelund (Eds.): TACAS 2014, LNCS 8413, pp. 62–77, 2014.
© Springer-Verlag Berlin Heidelberg 2014

However, designing and implementing such side-channel countermeasures are labor intensive and error prone, and currently, there is a lack of formal verification tools to evaluate how secure a countermeasure really is. Software countermeasures are particularly challenging to design, since the source of the information leakage is not the cryptographic software but the microprocessor hardware that executes the software. From the perspective of average software developers – who may not know all the architectural details of the device – it is difficult to predict the myriad possible ways in which side-channel information may be leaked. Furthermore, bugs in implementation can also break an otherwise secure countermeasure.

In this paper, we propose a new method for formally verifying the security of masking countermeasures. Our method uses an SMT solver to check if any intermediate computation result of a software statistically depends on the sensitive data. Since this is a statistical property, it cannot be directly checked by conventional formal verification methods [5,20,21,11]. Although in the literature, there exists some work on tackling the problem using type-based information flow analysis techniques [18], these methods are often overly conservative, leading to the classification of countermeasures as secure when they are not. In contrast, our method always returns the precise result. Although Bayrak *et al.* [2] also used a constraint solver in their method, the analysis is significantly less precise than ours. They check whether a variable is *masked* by some random variable, but not whether it is *perfectly masked*, i.e., whether the probability distribution is dependent on the sensitive data. To the best of our knowledge, our method is the first automated verification method that checks for *perfect masking*. This is important because with *order-d* perfect masking, an implementation is provably secure against any type of *order-d* (and lower-order) power analysis attack [9].

Fig. 1 (left) illustrates the difference between naive and perfect masking. Here, `k` is the sensitive data, `r1` and `r2` are the random variables, and `o1, o2, o3`, and `o4` are the results of four different masking schemes. Assume that all variables are Boolean, we can construct the truth table in Fig. 1 (right). Although `o1,o2,o3` all seem to depend on the values of the random variables `r1` and `r2`, they are vulnerable to side-channel attacks. To see why, consider the case when `o1` is logical 1. In this case, we know for sure that `k` is 1, regardless of the values of the random variables. Similarly, when `o2` is logical 0, we know for sure that `k` is 0. Although `o3` does not *directly* leak the sensitive information about `k` as in `o1` and `o2`, the masking is still not perfect. When `o3` is logical 1 (or 0), there is a 75% chance that `k` is logical 1 (or 0). Therefore, an adversary may launch a power analysis attack to deduce the value of `k`.

```
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

Fig. 1. Masking examples: `o1,o2,o3` are not perfectly masked, but `o4` is perfectly masked

In contrast, `o4` is *perfectly masked* in that the output is statistically independent of the sensitive data. When `k` is logical 1 (or 0), there is 50% chance that `o4` is logical 1 (or 0). Therefore, it is provably secure against any first-order power analysis attack, where the adversary can observe one intermediate computation result. The example in Fig. 1 also demonstrates a weakness of the method in [2]: Since it only checks whether a variable is masked, but not whether its probability distribution depends on the key, it would (falsely) classify all of `o1,o2,o3,o4` as secure. In contrast, our new method can differentiate `o4` from the other three, since only `o4` is perfectly masked.

We have implemented our new method in a verification tool based on the LLVM compiler and the Yices SMT solver [6]. We encode the verification problem into a series of quantifier-free first-order logic formulas, whose satisfiability can be decided by Yices. Our SMT encoding scheme is significantly different from the ones used by standard verification methods, because the *perfect masking* property checked by our tool is statistical in nature. For comparison, we also implemented the method in [2] in our tool. We have conducted experiments on a large set of recently proposed countermeasures, including the ones applied to AES and the MAC-Keccak reference code submitted to Round 3 of NIST's SHA-3 competition. Our results show that the new method is effective in detecting flaws in the masking implementation. Furthermore, the method is scalable enough to handle programs of practical size and complexity.

The remainder of this paper is organized as follows. We establish notation in Section 2, before presenting our SMT based verification algorithm in Section 3. Then, we illustrate the entire verification process using an example in Section 4. We present our incremental verification method in Section 5, which further improves the scalability of our SMT-based method. We present our experimental results in Section 6, and finally give our conclusions in Section 7.

2 Preliminaries

In this section, we define the type of side-channel attacks considered in this paper and review the notion of *perfect masking*.

Side-Channel Attacks. Following the notation used by Blömer *et al.* [4], we assume that the program to be verified implements a function $c \leftarrow enc(x, k)$, where x is the plaintext, k is the secret key, and c is the ciphertext. Let $I_1(x, k, r)$, $I_2(x, k, r)$, ..., $I_t(x, k, r)$ be the sequence of intermediate computation results inside the function, where r is an s-bit random number in the domain $\{0, 1\}^s$. The purpose of using r is to make all intermediate results statistically independent of the secret key (k).

When $enc(x, k)$ is a linear function in the Boolean domain, masking and de-masking are straightforward. However, when $enc(x, k)$ is a non-linear function, masking and de-masking often require a complete redesign of the implementation. However, this manual design process is both labor intensive and error prone, and currently, there is a lack of automated tools to assess how secure a countermeasure really is.

We assume that an adversary knows the pair (x, c) of plaintext and ciphertext in $c \leftarrow enc(x, k)$. For each pair (x, c), the adversary also knows the joint distribution of at most d intermediate computation results $I_1(x, k, r), \ldots, I_d(x, k, r)$, through access to

some aggregated quantity such as the power dissipation. However, the adversary does not have access to r, which is produced by a true random number generator. The goal of the adversary is to compute the secret key (k). In embedded computing, for instance, these are realistic assumptions. In their seminal work, Kocher *et al.* [10] demonstrated that for $d = 1$ and 2, the sensitive data can be reliably deduced using a statistical method known as differential power analysis (DPA).

Perfect Masking. Given a pair (x, k) of plaintext and secret key for the function $enc(x, k)$, and d intermediate results $I_1(x, k, r), \ldots, I_d(x, k, r)$, we use $D_{x,k}(R)$ to denote the joint distribution of $I_1, \ldots, I_d$ – while assuming that the s-bit random number r is uniformly distributed in the domain $\{0, 1\}^s$. Following Blömer *et al.* [4], we do not put restrictions on the technical capability of an adversary. As long as there is information leak, we consider the implementation to be vulnerable.

Definition 1. *Given an implementation of function $enc(x, k)$ and a set of intermediate results $\{I_i(x, k, r)\}$, we say that the implementation is order-d perfectly masked if, for all d-tuples $\langle I_1, \ldots, I_d \rangle$, we have*

$$D_{x,k}(R) = D_{x',k'}(R) \quad \textit{for any two pairs } (x, k) \textit{ and } (x', k') \;.$$

The notion of *perfect masking* used here is more accurate than the *sensitivity* [2]. There, an intermediate result is considered to be *sensitive* if (1) it depends on at least one *secret* input and (2) it is independent of any *random* input. We have demonstrated the difference between them using the example in Fig. 1, where `o1,o2,o3,o4` are all *insensitive*, but only `o4` is *perfectly masked*. In general, if an intermediate result is perfectly masked, it is guaranteed to be insensitive. However, an insensitive intermediate result may not be perfectly masked.

To check for violations of *perfect masking*, we need to decide whether there exists a d-tuple $\langle I_1, \ldots, I_d \rangle$ such that $D_{x,k}(R) \neq D_{x',k'}(R)$ for some (x, k) and (x', k'). Here, the main challenge is to compute $D_{x,k}(R)$. We will present our solution in Section 3.

In this work, we focus on verifying security-critical programs, e.g. those that implement cryptographic algorithms, as opposed to arbitrary software programs. (Our method would be too expensive for verifying general-purpose software.) In general, the class of programs that we consider here do 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 variables are bounded integers, we can convert the program to a purely Boolean program through bit-blasting. Therefore, in this paper, we shall present our new verification method on the bit-level representation of a branch-free program. Our goal is to verify that all intermediate bits of the program are perfectly masked.

3 SMT Based Verification of Perfect Masking

We first illustrate the overall flow of our verification method using the program in Fig. 2. The program is a masked version of $c \leftarrow (k1 \wedge k2)$, where $k1$ and $k2$ are two secret

```
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 : }
```

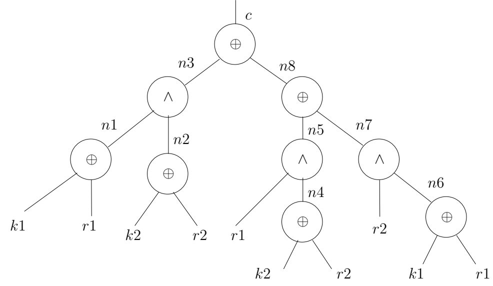


Fig. 2. Example: a program and its graphic representation ($\oplus$ denotes XOR; $\wedge$ denotes AND)

keys, $r1$ and $r2$ are random variables with independent and uniform distribution in $\{0, 1\}$, and c is the computation result. The objective of masking is to make the power consumption of the device executing this code independent from the values of the secret keys. This masking scheme originated from Blömer et al. [4]. The return value c is logically equivalent to $(k1 \wedge k2) \oplus (r1 \wedge r2)$. The corresponding demasking function, which is not shown in the figure, is $c \oplus (r1 \wedge r2)$. Therefore, demasking would produce a result that is logically equivalent to the desired value $(k1 \wedge k2)$.

Our method will determine if all the intermediate variables of the program are perfectly masked. We use the Clang/LLVM compiler to parse the input Boolean program and construct the data-flow graph, where the root represents the output and the leaf nodes represent the input bits. 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 result, we check whether it is perfectly masked. The order in which we check the internal nodes is as follows: $n1, n2, n3, n4, n5, n6, n7, n8$, and finally, c.

The Theory. 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 intermediate computation result $I(x, k, r)$ of the program, we check whether it is perfectly masked. Following Definition 1, we formulate this check as a satisfiability problem as follows:

$$\exists x . \exists k, k' \;.\; \left(\Sigma_{r \in \{0,1\}^s} I(x, k, r) \neq \Sigma_{r \in \{0,1\}^s} I(x, k', r) \right)$$

Here, x represents the plaintext bits, k and k' represent two different valuations of the key bits, and r is the random number uniformly distributed in the domain $\{0, 1\}^s$, where s is the number of random bits. For any fixed (x, k, k'),

- $\Sigma_{r \in \{0,1\}^s} I(x, k, r)$ is the number of satisfying assignments for $I(x, k, r)$, and
- $\Sigma_{r \in \{0,1\}^s} I(x, k', r)$ is the number of satisfying assignment for $I(x, k', r)$.

Assume that r is uniformly distributed in the domain $\{0, 1\}^s$, the above summations can be used to indicate the probabilities of I being logical 1 under two different key values k and k'.

If the above formula is satisfiable, there exists a plaintext x and two different keys (k, k') such that the distribution of $I(x, k, r)$ differs from the distribution of $I(x, k', r)$. In other words, some information of the secret key is leaked through I, and therefore we say that I is not perfectly masked. If the above formula is unsatisfiable, then such information leakage is not possible, and therefore we say that I is perfectly masked.

Another way to understand the above satisfiability problem is to look at the negation. Instead of checking the *satisfiability* of the formula above, we check the *validity* of the formula below:

$$\forall x.\forall k, k'.\ \left(\Sigma_{r\in\{0,1\}^s} I(x, k, r) = \Sigma_{r\in\{0,1\}^s} I(x, k', r)\right)$$

If this formula is valid – meaning that it holds for all valuations of x, k and k' – then we say that I is perfectly masked.

The Encoding. Let Φ denote the SMT formula to be created for checking 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 is not perfectly masked. 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 variable x.
- ***Boolean-to-int (Ψ_{b2i})***: It encodes the conversion of the Boolean valued output of $I(x, k, r)$ to an integer (true becomes 1 and false becomes 0), so that the integer values can be summed up later to compute $\Sigma_{r=1}^{2^s} I(x, k, r)$.
- ***Sum-up-the-1s (Ψ_{sum})***: It encodes the two summations of the logical 1s in the outputs of the 2^s program logic copies, one for $I(x, k, r)$ and the other for $I(x, k', r)$.
- ***Different sums (Ψ_{diff})***: It asserts that the two summations should have different results.

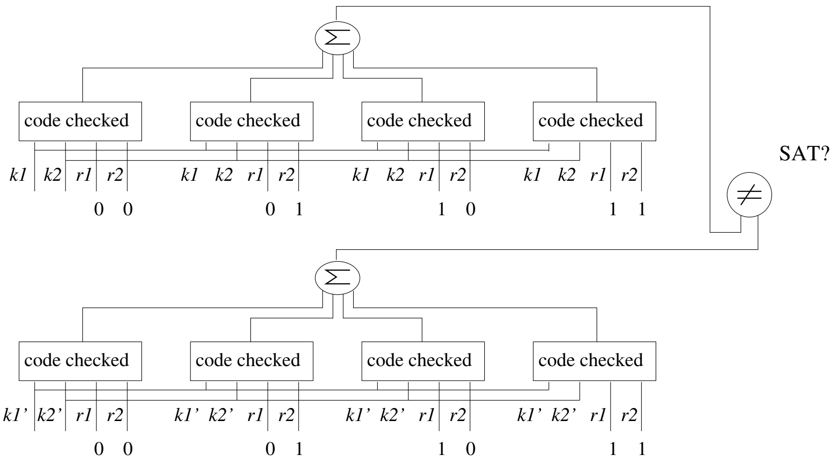


Fig. 3. SMT encoding for checking the statistical dependence of an output on secret data $(k1, k2)$

Fig. 3 is a pictorial illustration of our SMT encoding for an intermediate result $I(k1, k2, r1, r2)$, where $k1, k2$ are the secret key bits and $r1, r2$ are two random bits. Here, the first four boxes, encoding $\Psi_k^0, \ldots, \Psi_k^3$, are the four copies of the program logic for key bits ($k1k2$), with the random bits set to 00, 01, 10, and 11, respectively. The other four boxes, encoding $\Psi_{k'}^0, \ldots, \Psi_{k'}^3$, are the four copies of the program logic for key bits ($k1'k2'$), with the 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 are different.

The Running Example. Consider node $n8$ in Fig. 2 as the node under verification. The function is defined as `n8 = (r1 & (k2 xor r2)) xor (r2 & (k1 xor r1))`. The SMT formula that our method generates – by instantiating `r1r2` to 00, 01, 10, and 11 – is the conjunction of all of the formulas listed below:

```
n8_1 = (0 & (k2 xor 0)) xor (0 & (k1 xor 0))          // four copies of I(k, r)
n8_2 = (0 & (k2 xor 1)) xor (1 & (k1 xor 0))
n8_3 = (1 & (k2 xor 0)) xor (0 & (k1 xor 1))
n8_4 = (1 & (k2 xor 1)) xor (1 & (k1 xor 1))
n8_1' = (0 & (k2' xor 0)) xor (0 & (k1' xor 0))       // four copies of I(k',r)
n8_2' = (0 & (k2' xor 1)) xor (1 & (k1' xor 0))
n8_3' = (1 & (k2' xor 0)) xor (0 & (k1' xor 1))
n8_4' = (1 & (k2' xor 1)) xor (1 & (k1' xor 1))
(( num1 = 1 ) & n8_1 ) | ((num1=0) & not n8_1 )       // convert bool to integer
(( num2 = 1 ) & n8_2 ) | ((num2=0) & not n8_2 )
(( num3 = 1 ) & n8_3 ) | ((num3=0) & not n8_3 )
(( num4 = 1 ) & n8_4 ) | ((num4=0) & not n8_4 )
(( num1' = 1 ) & n8_1') | ((num1'=0) & not n8_1')     // convert bool to integer
(( num2' = 1 ) & n8_2') | ((num2'=0) & not n8_2')
(( num3' = 1 ) & n8_3') | ((num3'=0) & not n8_3')
(( num4' = 1 ) & n8_4') | ((num4'=0) & not n8_4')
(num1 + num2 + num3 + num4) != (num1' + num2' + num3' + num4')   // the check
```

We solve the conjunction of the above formulas using an off-the-shelf SMT solver called Yices [6]. In this particular example, the formula is satisfiable. For example, one of the satisfying assignments is `k1k2`=00 and `k1'k2'`=01. We shall show in the next section that, when the key bits are 00, the probability for $n8$ to be logical 1 is 0%; but when the key bits are 01, the probability is 50%. This makes it vulnerable to first-order DPA attacks. Therefore, $n8$ is not perfectly masked.

High-Order Attacks. For a masked code to be resistant to *first-order* differential power analysis (DPA) attacks, all the intermediate results must be perfectly masked. However, even if each intermediate result is perfectly masked, it is still not sufficient to resist *high-order* DPA attacks, where an adversary can simultaneously observe leakage from more than one intermediate computation result. For a masking scheme to be resistant to *order-d* DPA attacks, we need to ensure that the joint distribution of any d intermediate results (where $d = 2, 3, \ldots$) is independent of the secret key. That is, for any d intermediate results $I_1, \ldots, I_d$, we check the satisfiability of the following formula:

$$\exists x. \exists k, k' . \ \left(\Sigma_{r \in \{0,1\}^s} \Sigma_{i=1}^{d} I_i(x, k, r) \neq \Sigma_{r \in \{0,1\}^s} \Sigma_{i=1}^{d} I_i(x, k', r) \right)$$

Our encoding can be easily extended to implement this new check. In practice, most countermeasures assume that the adversary has access to the side-channel leakage of

either one or two intermediate results, which corresponds to first-order and second-order attacks. In our actual implementation, we handle both first-order and second-order attacks. In our experiments, we also evaluate our new method on verifying countermeasures against both first-order and second-order attacks (where $d = 1$ or 2).

4 The Working Example

Consider the automated verification of our running example in Fig. 2. For each internal node I, we first identify all the transitive fan-in nodes of I in the program to form a *code region* for the subsequent SMT solver based analysis. In the worst case, the extracted code region should start from the instruction (node) to be verified, and cover all the transitive fan-in nodes on which it depends. Then, the extracted code region is given to our SMT based verification procedure, whose goal is to prove (or disprove) that the node is statistically independent of the secret key.

Following a topological order, our method starts with node $n1$, which is defined in Line 3 of the program in Fig. 2. The extracted code region consists of $n1 = k1 \oplus r1$ itself. Since it involves only one key and one random variable in the XOR operation, a simple static analysis can prove that it is perfectly masked. Therefore, although we could have verified it using SMT, we skip it for efficiency reasons. Such simple static analysis is able to prove that $n2, n4$ and $n6$ are also perfectly masked.

Next, we check if $n3$ is perfectly masked. The truth table of $n3$ is shown in Fig. 4 (left). In all four valuations of $k1$ and $k2$, the probability of $n3$ being logical 1 is 25%. Therefore, $n3$ is perfectly masked. When we apply our SMT based method, the solver is not able to find any satisfying assignment for $k1$ and $k2$ under which the probability distributions of $n3$ are different. Note that our method does not check the probability of the output being logical 0, since having an equal probability distribution of logical 1 is equivalent to having an equal probability distribution for logical 0.

k1	k2	r1	r2	n3
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

k1	k2	r1	r2	n8
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

k1	k2	r1	r2	c
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Fig. 4. The truth-tables for internal nodes $n3$, $n8$, and c of the example program in Fig. 2

The verification steps for nodes $n5$ and $n7$ are similar to that of $n3$ – all of them are perfectly masked.

Next, we check if $n8$ is perfectly masked. The proof would fail because, as shown in the truth table in Fig. 4 (middle), the probability for $n8$ to be logical 1 is not the same under different valuations of the keys. For example, if the keys are 00, then $n8$ would be 0 regardless of the values of the random variables. Recall that we have shown the detailed SMT encoding for $n8$ in Section 3. Using our method, the solver can quickly find two configurations of the key bits (for example, 00 and 11) under which the probabilities of $n8$ being logical 1 are different. Therefore, $n8$ is not perfectly masked.

The remaining node is c, whose truth table is shown in Fig. 4 (right). Similar to $n8$, our SMT based method will be able to show that it is not perfectly masked.

It is worth pointing out that the result of applying the *Sleuth* method [2] would have been different. Although $n8$ and c are clearly vulnerable to first-order DPA attacks, the *Sleuth* method, based on the notion of *sensitivity*, would have classified them as "securely masked." This demonstrates a major advantage of our new method over *Sleuth*.

5 The Incremental Verification Algorithm

Note that the size of the formula created by our SMT encoding is linear in the size of the program and exponential in the number of random variables – for s random bits, we need to make 2^{s+1} copies of the program logic. This is the main bottleneck for applying our method to large programs. In this section, we propose an incremental verification algorithm, which applies SMT solver based analysis only to small code regions – one at a time – as opposed to the entire fan-in cone of the node under verification. This is crucial for scaling the method up to programs of practical size.

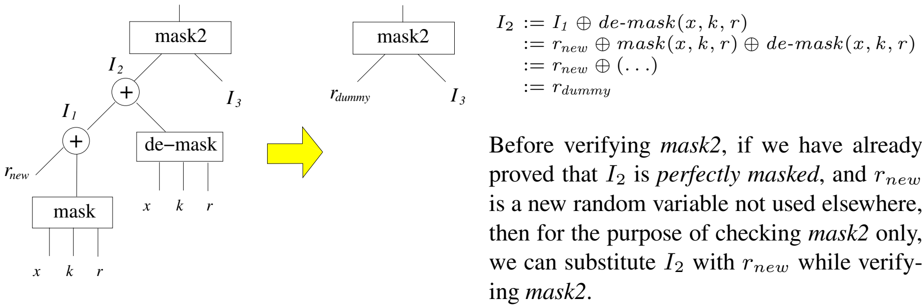


Fig. 5. Incremental verification: applying the SMT based analysis to a small fan-in region only

Extracting the Verification Region. In practice, a common strategy in 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. This can be illustrated by the example in Fig. 5, where the output of *mask(x,k,r)* is masked with the new random variable r_{new} before it is demasked from the old random variable r.

Due to *associativity* of the $\oplus$ operator, reordering the masking and demasking operations would not change the logical result. For example, in Fig. 5, the instruction being verified is in *mask2()*. Since the newly added 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} (or even r_{new} itself) while verifying *mask2()*. We shall see in the experimental results section that such opportunities are abundant in real-world applications. Therefore, in this subsection, we present a sound algorithm for extracting a small code region from the fan-in cone of the node under verification.

Our algorithm relies on some auxiliary data structures associated with the current node i under verification: *supportV[i]*, *uniqueM[i]* and *perfectM[i]*.

- *supportV[i]* is the set of inputs in the support of the function of node i.
- *uniqueM[i]* is the set of random inputs that each reaches i along only one path.
- *perfectM[i]* is a subset of *uniqueM[i]* where each random variable, by itself, guarantees that node i is perfectly masked.

These tables can be computed by a traversal of the program nodes as described in Algorithm 1. For example, for node I_1 in Fig. 5, *supportV[I_1]*$= \{x, k, r, r_{new}\}$, *uniqueM[I_1]* $= \{r, r_{new}\}$, and *perfectM[I_1]*$= \{r_{new}\}$, assuming r is not repeated in the mask block. For node I_2, we have *supportV[I_2]*$= \{x, k, r, r_{new}\}$, *uniqueM[I_2]*$= \{r_{new}\}$, since r reaches I_2 twice and so may have been de-masked, and *perfectM[I_2]*$= \{r_{new}\}$.

Algorithm 1. Computing the auxiliary tables for all internal nodes of the program.

```
 1. supportV[i] ← { v } for each input node i with variable v
 2. uniqueM[i] ← { v } for each input node i with random mask variable v
 3. perfectM[i] ← { v } for each input node i with random mask variable v
 4. for each (internal node i in a leaf-to-root topological order) {
 5.    L ← LEFTCHILD(i)
 6.    R ← RIGHTCHILD(i)
 7.    supportV[i] ← supportV[L] ∪ supportV[R]
 8.    uniqueM ← (uniqueM[L] ∪ uniqueM[R]) \ (supportV[L] ∩ supportV[R])
 9.    if (i is an XOR node)
10.       perfectM[i] ← uniqueM[i] ∩ (perfectM[L]∪perfectM[R])
11.    else
12.       perfectM[i] ← { }
13. }
```

Our idea of extracting a small code region for SMT based analysis is formalized in Algorithm 2. Given the node i under verification, and *uniqueM[i]* as the set of random variables that each reaches i along only one path, we call GETREGION(*i,uniqueM[i]*) to compute the region. Inside GETREGION, *uniqueM[i]* is renamed to *freshMasksATi*. More specifically, we start by checking each transitive fan-in node n of the current node i. If n is a leaf node (Line 2), then we add n and the input variable v to the region. If n is not a leaf node, we check if there is a random variable r $\in$*uniqueMATi* that, by itself, can perfectly mask node n (Line 4). In Fig. 5, for example, r_{new}, by itself, can uniformly mask node I_2. If such random variable r exists, then we add pair (n, r) to the region and return – skipping the entire fan-in cone of n. Otherwise, we recursively invoke GETREGION to traverse the two child nodes of n.

Algorithm 2. Extracting a code region for node i for the subsequent SMT based analysis.

```
1.  GETREGION (n, uniqueMATi) {
2.    if (n is an input node with variable v)
3.      region.add ← (n, v)
4.    else if (∃ random variable r ∈ perfectM[n] ∩ uniqueMATi)
5.      region.add ← (n, r)
6.    else
7.      region.add ← (n, {})
8.      region.add ← GETREGION(n.Left,  uniqueMATi)
9.      region.add ← GETREGION(n.Right, uniqueMATi)
10.   return region
11. }
```

The Overall Algorithm. Algorithm 3 shows the overall flow of our incremental verification method. Given the program and the lists of secret, random and plaintext variables, our method systematically scans through all the internal nodes from the inputs to the return value. For each node i, our method first extracts a small code region (Line 4). Then, we invoke the SMT based analysis. If the node is not perfectly masked, we add it to the list of *bad* nodes.

Algorithm 3. Incremental verification of perfect masking.

```
1.  VERIFYPERFECTMASKING (Prog, keys, rands, plains) {
2.    badNodes ← { }
3.    for each (internal node i ∈ Prog in a topological order ) {
4.        region ← GETREGION(i, uniqueM[i])
5.        notPerfect ← CHECKMASKINGBYSMT (i, region, keys, rands, plains )
6.        if (notPerfect)
7.          badNodes.add( i )
8.    }
9.    return badNodes
10. }
```

To optimize the performance of Algorithm 3, we conduct a simple static analysis between Line 4 and Line 5 to quickly check whether it is fruitful to invoke the SMT solver. The first one checks if the region contains any secret keys, if not then the solver is not invoked and the instruction is perfectly masked. The second analysis checks some syntactic conditions – if all of these conditions are satisfied, the current node i is guaranteed to be perfectly masked. In such case, we also avoid invoking the SMT solver. The implemented syntactic conditions are listed as follows:

- The instruction has no secret input as its child. This guarantees that when a secret variable is introduced, its masking operation will be verified.
- None of the random variables appears in both operand's $supportV$ tables. This guarantees that no perfectly masking of a secret variable in any of the operands may be affected.

- Both operands are perfectly masked. This guarantees to find all the resultant imperfect masked instructions due to an initial imperfectly masked instruction.

To further optimize the performance of Algorithm 3, we implement a method for identifying random variables that are *don't cares* for the node i under verification, and use the information to reduce the cost of the SMT based analysis. Prior to the SMT encoding, for each random variable r ∈*supportV[i]*, we check if the value of r can ever affect the output of i. If the answer is no, then r is a *don't care*. During our SMT encoding, we will set r to logical 0 rather than treat r as a random variable, to to reduce the size of the SMT formula. This can lead to a significant performance improvement since the formula size is exponential in the number of relevant random variables.

We check whether $r \in$ *support[i]* is a *don't care* for node i by constructing a SAT formula and solving it using the SMT solver. The SAT formula is defined as follows:

$$\Psi^{r=0}_{region} \wedge \Psi^{r=1}_{region} \wedge \Psi_{diffO} \ ,$$

where $\Psi^{r=0}_{region}$ encodes the program logic of the region, with the random bit r set to 0, $\Psi^{r=1}_{region}$ encodes the program logic of the region, with the random bit r set to 1, and Φ_{diffO} asserts that the outputs of these two copies differ. If the above formula is unsatisfiable, then r is a *don't care* for node i.

6 Experiments

We have implemented our method in a verification tool called *SC Sniffer*, based on the LLVM compiler and the Yices SMT solver [6]. It runs in two modes: monolithic and incremental. The monolithic mode applies our SMT based encoding to the entire fan-in cone of each node in the program, whereas the incremental method tries to restrict the SMT encoding to a localized region. In addition, we implemented the *Sleuth* method [2] for experimental comparison. The main difference is that our method not only checks whether a node is masked (as in *Sleuth*), but also checks whether it is perfectly masked, i.e. it is statistically independent of the secret key.

We have evaluated our tool on some recently proposed countermeasures. Our experiments were designed to answer the following research questions:

- How effective is our new method? We know that in theory, the new method is more accurate than the *Sleuth* method. But does it have a significant advantage over the *Sleuth* method in practice?
- How scalable is our new method, especially in verifying applications of realistic code size and complexity? We have extended our SMT based method with incremental verification. Is it effective in practice?

Table 1 shows the statistics of the benchmarks. Column 1 shows the name of each benchmark example. Column 2 shows a short description of the implemented algorithm. Column 3 shows the number of lines of code – here, each instruction is a bit level operation. Column 4 shows the number of nodes that represent the intermediate computation results. Columns 5-7 show the number of input bits that are the secret key, the plaintext, and the random variable, respectively.

Table 1. The benchmark statistics: in addition to the program name and a short description, we show the total lines of code, the numbers of intermediate nodes and the various inputs

Name	Description	Code Size	Nodes	Keys	Plains	Rands
P1	CHES13 Masked Key Whitening	79	47	16	16	16
P2	CHES13 De-mask and then Mask	67	31	8	0	16
P3	CHES13 AES Shift Rows [2nd-order]	21	21	2	0	2
P4	CHES13 Messerges Boolean to Arithmetic (bit0) [2-order]	23	24	1	0	2
P5	CHES13 Goubin Boolean to Arithmetic (bit0) [2-order]	27	60	1	0	2
P6	Logic Design for AES S-Box (1st implementation)	32	9	2	0	2
P7	Logic Design for AES S-Box (2nd implementation)	40	6	2	0	3
P8	Masked Chi function MAC-Keccak (1st implementation)	59	19	3	0	4
P9	Masked Chi function MAC-Keccak (2nd implementation)	60	19	3	0	4
P10	Syn. Masked Chi func MAC-Keccak (1st implementation)	66	22	3	0	4
P11	Syn. Masked Chi func MAC-Keccak (2nd implementation)	66	22	3	0	4
P12	MAC-Keccak 512b Perfect masked	285k	128k	288	288	805
P13	MAC-Keccak 512b De-mask and then mask – compiler error	285k	128k	288	288	805
P14	MAC-Keccak 512b Not-perfect Masking of Chi function (v1)	285k	128k	288	288	805
P15	MAC-Keccak 512b Not-perfect Masking of Chi function (v2)	285k	152k	288	288	805
P16	MAC-Keccak 512b Not-perfect Masking of Chi function (v3)	285k	128k	288	288	805
P17	MAC-Keccak 512b Unmasking of Pi function	285k	131k	288	288	805

Table 2. Experimental results: comparing our *SC Sniffer* method with the *Sleuth* method [2]

Name	*Sleuth* [2]				*SC Sniffer (monolithic)*				*SC Sniffer (incremental)*				
	masked	nodes failed	nodes checked	time	masked perfect	nodes failed	nodes checked	time	masked perfect	nodes failed	nodes checked	SMT mask	time
P1	No	16	47	0.16s	No	16	47	0.22s	No	16	47	16	0.09s
P2	No	8	31	0.21s	No	8	31	0.20s	No	8	31	8	0.09s
P3	No	9	21	1.17s	No	9	21	1.27s	No	9	21	18	0.46s
P4	No	2	24	0.58s	No	2	24	0.65s	No	2	24	8	0.57s
P5	No	2	60	1.19s	No	2	60	1.40s	No	2	60	20	1.12s
P6	Yes	0	9	0.06s	No	2	9	0.10s	No	2	9	2	0.08s
P7	Yes	0	6	0.04s	No	1	6	0.07s	No	1	6	1	0.03s
P8	No	1	19	0.15s	No	3	19	0.26s	No	3	19	3	0.11s
P9	Yes	0	19	0.13s	No	2	19	0.27s	No	2	19	2	0.10s
P10	Yes	0	22	0.18s	No	1	22	0.32s	No	1	22	2	0.14s
P11	Yes	0	22	0.20s	No	1	22	0.37s	No	1	22	3	0.18s
P12	Yes	0	128k	91m53s	-	0	34	mem-out	Yes	0	128K	0	10m48s
P13	No	2560	128k	92m59s	No	1	46	mem-out	No	2560	128K	2560	14m10s
P14	Yes	0	128k	97m38s	-	0	31	mem-out	No	1024	128K	1024	18m20s
P15	Yes	0	152k	132m10s	-	0	32	mem-out	No	512	152K	1024	37m37s
P16	No	512	128k	113m12s	-	0	40	mem-out	No	1536	128K	1536	17m24s
P17	No	4096	131k	103m56s	-	0	34	mem-out	No	4096	131K	4096	17m35s

The benchmarks are classified into three groups. The first group of test cases (P1 to P5) are taken from the *Sleuth* benchmark [2], all of which contain intermediate variables that are not masked at all. More specifically, P1 is the masking key whitening code on Page 12 of the *Sleuth* paper. P2 is the AES8 example, a smart card implementation of AES resistant to power analysis, originated from Herbst *et al.* [8]. P3 is the code on Page 13 of the *Sleuth* paper, also originated from Herbst *et al.* [8]. P4 is the code on Page 18 of the *Sleuth* paper, originated from Messerges [13]. P5 is the code on Page 18 of the *Sleuth* paper, originated from Goubin [7].

The second group of test cases (P6 to P11) are examples where most of the intermediate variables are masked, but none of the masking schemes is perfect. P6 and P7 are

the two examples used by Blömer *et al.* [4] (on Page 7). P8 and P9 are the SHA3 MAC-Keccak computation reordered examples, originated from Bertoni *et al.* [3] (Eq. 5.2 on Page 46). P10 and P11 are two experimental masking schemes for the Chi function in SHA3, none of which is perfectly masked.

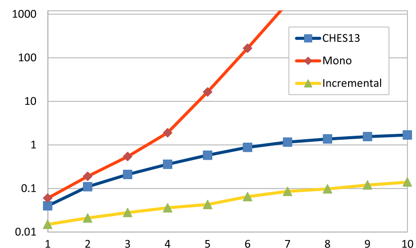
The third group of test cases (P12 to P17) comes from the regeneration of MAC-Keccak reference code submission to NIST in the SHA-3 competition [15]. There are a total of 285k lines of Boolean operation code. The difference among these test cases is that they are protected by various countermeasures, some of which are perfectly masked (e.g. P12) whereas others are not.

Table 2 shows the experimental results run on a machine with a 3.4 GHz Intel i7-2600 CPU, 4 GB RAM, and a 32-bit Linux OS. We have compared the performance of three methods: *Sleuth*, New (monolithic), and New (incremental). Here, *Sleuth* is the method proposed by Bayrak *et al.* [2], while the other two are our own method. In this table, Column 1 shows the name of each test program. Columns 2-5 show the results of running *Sleuth*, including whether the program passed the check, the number of nodes failed the check, and the total number of nodes checked. Columns 6-9 show the results of running our new monolithic method. Here, `mem-out` means that the method requires more than 4 GB of RAM. Columns 10-14 show the results of running our new incremental method. Here, we also show the number of SMT based masking checks made, which is often much smaller than the number of nodes checked, because many of them are resolved by our static analysis.

First, the results show that our new algorithm is more accurate than *Sleuth* in deciding whether a node is securely masked. Every node that failed the security check of *Sleuth* would also fail the security check of our new method. However, there are many nodes that passed the check of *Sleuth*, but failed the check of our new method. These are the nodes that are masked, but their probability distributions are still dependent on the sensitive inputs – in other words, they are not perfectly masked.

Second, the results show that our incremental method is significantly more scalable than the monolithic method. On the first two groups of test cases, where the programs are small, both methods can complete, and the difference in run time is small. However, on large programs such as the Keccak reference code, the monolithic method could not finish since it quickly ran out of the 4 GB RAM, whereas the incremental method can finish in a reasonable amount of time. Moreover, although the *Sleuth* method implements a significantly simpler (and hence weaker) check, it is also based on a monolithic verification approach. Our results in Table 2 show that, on large examples, our incremental method is significantly faster than *Sleuth*.

As a measurement of the scalability of the algorithms, we have conducted experiments on a 1-bit version of test program P1 for 1 to 10 encryption rounds. In each parameterized version, the input for each round is the output from the previous round. We ran the experiment twice, once with an unmasked instruction in each round, and once with all instructions perfectly masked. The results of



the two experiments are almost identical, and therefore, we only plot the result for the perfectly masked version. In the right figure, the x-axis shows the program size, and the y-axis shows the verification time in seconds. Among the three methods, our incremental method is the most scalable.

7 Conclusions

We have presented the first fully automated method for formally verifying whether a software implementation is *perfectly masked* by uniformly random inputs, and therefore is secure against power analysis based side-channel attacks. Our new method relies on translating the verification problem into a set of constraint solving problems, which can be decided by off-the-shelf solvers such as Yices. We have also presented an incremental checking procedure to drastically improve the scalability of the SMT based algorithm. We have conducted experiments on a large set of recently proposed countermeasures. Our results show that the new method is not only more precise than existing methods, but also scalable for practical use.

Acknowledgments. This work is supported in part by the NSF grant CNS-1128903 and the ONR grant N00014-13-1-0527.

References

1. Balasch, J., Gierlichs, B., Verdult, R., Batina, L., Verbauwhede, I.: Power analysis of Atmel CryptoMemory – recovering keys from secure EEPROMs. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 19–34. Springer, Heidelberg (2012)
2. Bayrak, A.G., Regazzoni, F., Novo, D., Ienne, P.: Sleuth: Automated verification of software power analysis countermeasures. In: Bertoni, G., Coron, J.-S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 293–310. Springer, Heidelberg (2013)
3. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V., Keer, R.V.: Keccak implementation overview, http://keccak.neokeon.org/Keccak-implementation-3.2.pdf
4. Blömer, J., Guajardo, J., Krummel, V.: Provably secure masking of AES. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 69–83. Springer, Heidelberg (2004)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
6. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
7. Goubin, L.: A sound method for switching between boolean and arithmetic masking. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 3–15. Springer, Heidelberg (2001)
8. Herbst, C., Oswald, E., Mangard, S.: An AES smart card implementation resistant to power analysis attacks. In: Zhou, J., Yung, M., Bao, F. (eds.) ACNS 2006. LNCS, vol. 3989, pp. 239–252. Springer, Heidelberg (2006)
9. Joye, M., Paillier, P., Schoenmakers, B.: On second-order differential power analysis. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 293–308. Springer, Heidelberg (2005)
10. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)

11. Li, B., Wang, C., Somenzi, F.: A satisfiability-based approach to abstraction refinement in model checking. Electronic Notes in Theoretical Computer Science 89(4) (2003)
12. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks - Revealing the Secrets of Smart Cards. Springer (2007)
13. Messerges, T.S.: Securing the AES finalists against power analysis attacks. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 150–164. Springer, Heidelberg (2001)
14. Moradi, A., Barenghi, A., Kasper, T., Paar, C.: On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs. In: ACM Conference on Computer and Communications Security, pp. 111–124 (2011)
15. NIST. Keccak reference code submission to NIST's SHA-3 competition (Round 3), `http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/Keccak_FinalRnd.zip`
16. Paar, C., Eisenbarth, T., Kasper, M., Kasper, T., Moradi, A.: Keeloq and side-channel analysis-evolution of an attack. In: FDTC, pp. 65–69 (2009)
17. Prouff, E., Rivain, M.: Masking against side-channel attacks: A formal security proof. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 142–159. Springer, Heidelberg (2013)
18. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
19. Taha, M., Schaumont, P.: Differential power analysis of MAC-Keccak at any key-length. In: Sakiyama, K., Terada, M. (eds.) IWSEC 2013. LNCS, vol. 8231, pp. 68–82. Springer, Heidelberg (2013)
20. Wang, C., Hachhtel, G.D., Somenzi, F.: Abstraction Refinement for Large Scale Model Checking. Springer (2006)
21. Yang, Z., Wang, C., Ivančić, F., Gupta, A.: Mixed symbolic representations for model checking software programs. In: Formal Methods and Models for Codesign, pp. 17–24 (July 2006)