Formal Verification of Software Countermeasures against Side-Channel Attacks

HASSAN ELDIB, CHAO WANG, and PATRICK SCHAUMONT, Virginia Polytechnic Institute and State University

A common strategy for designing countermeasures against power-analysis-based side-channel attacks is using *random masking* 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 assess how secure a countermeasure really is. We propose the first SMT-solver-based method for formally verifying the security of a masking countermeasure against such attacks. In addition to checking whether the sensitive data are *masked* by random variables, we also check whether they are *perfectly masked*, that is, whether the intermediate computation results in the implementation of a cryptographic algorithm are independent of the secret key. We encode this verification problem using 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 proposed method in a software verification tool based on the LLVM compiler frontend and the Yices SMT solver. Our experiments on a set of recently proposed masking countermeasures for cryptographic algorithms such as AES and MAC-Keccak show the method is both effective in detecting power side-channel leaks and scalable for practical use.

1. INTRODUCTION

Security analysis of the hardware and software systems implemented in embedded computing 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 that utilize secondary information resulting from the execution of sensitive algorithms on these devices. For example, the power consumption of an embedded device such as the SmartCard executing the instruction `tmp=text⊕key`

depends on the value of the secret key [Mangard et al. 2007]. This value can be reliably deduced using a statistical method known as *differential power analysis* (DPA) [Kocher et al. 1999]. In recent years, commercial systems in the embedded computing space have shown weaknesses against such power-analysis-based side-channel attacks [Paar et al. 2009; Moradi et al. 2011a; Balasch et al. 2012].

A common mitigation strategy against such attacks is *masking*, which is a randomization-based technique for removing the statistical dependency between the sensitive data and the side-channel emission. This can be done in multiple ways. For example, Boolean masking uses an XOR operation of a random number r with a sensitive variable a to obtain a masked (randomized) variable $a_m = a \oplus r$ [Chari et al. 1999; Balasch et al. 2012; Prouff and Rivain 2013]. Later, the sensitive variable can be restored by a second XOR operation with the same random number $a_m \oplus r = a$, thanks to the inherent property of the XOR operation. Other randomization-based countermeasures have used additive masking ($a_m = a + r \bmod n$), multiplicative masking ($a_m = a * r \bmod n$), and application-specific code transformations such as RSA blinding ($a_m = ar^e \bmod N$).

However, manually designing, implementing, and verifying such countermeasures are labor intensive and error prone and, currently, there is a lack of automated 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 code but rather side channels of the microprocessor hardware that executes the software. From the perspective of average software developers—who may not know all the physical and architectural details of the embedded computing 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 masking countermeasure.

In this article, we propose a new method for formally verifying the security of masking countermeasures. Our method uses an SMT solver to check whether any intermediate computation result of the software code statistically depends on the sensitive data. Since the security of the countermeasure against power analysis attacks is a statistical property, the problem cannot be solved by conventional techniques such as symbolic model checking based on Binary Decision Diagrams (BDDs) and satisfiability (SAT) solvers [Clarke et al. 1999; Wang et al. 2006, 2007; Yang et al. 2009]. Although there is a large body of work on language/type-based information flow analysis [Agat 2000; Sabelfeld and Myers 2003] in the literature, these methods are geared toward detecting information leakage in the standard computation flows as opposed to the *power* side channels, and therefore may lead to the classification of countermeasures as secure when they actually are not. In contrast, our new method always returns the precise result.

Bayrak et al. [2013] recently proposed a constraint-solver-based method for verifying masking countermeasures. However, their analysis is significantly less precise than ours, since they only check whether an intermediate computation result is *masked* by some random variables but do not check whether it is *perfectly masked*, that is, whether the result is statistically dependent on sensitive data. To the best of our knowledge, our method is the first SMT-solver-based fully automated verification method that can check for *perfect masking*. This is important because, when a software implementation is perfectly masked by some random variables, it is provably secure against any type of adversaries, regardless of their capabilities in carrying out power analysis attacks [Joye et al. 2005].

Figure 1 illustrates the difference between naive and perfect masking countermeasures. Here, we assume that 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. We assume

$o1 = k \land (r1 \land r2)$

$o2 = k \lor (r1 \land r2)$

$o3 = k \oplus (r1 \land r2)$

$o4 = k \oplus (r1 \oplus r2)$

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: the secret bit k is perfectly masked by random bits r1 and r2 at node o4, but not at nodes o1, o2, and o3.

all these variables are Boolean and we can construct the truth table in Figure 1 (right). The method by Bayrak et al. [2013] would have classified o1,o2,o3 as being securely masked because their values all logically depend on the random variables r1 and r2. However, they are still vulnerable to side-channel attacks. To see why, consider the case when o1 is logical 1. By observing the power side channel of the output of o1, we know for sure that k is logical 1, regardless of the values of the random variables. Similarly, when o2 is logical 0, we know for sure that k is logical 0. Although o3 does not *directly* leak 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 also logical 1 (or 0). Therefore, by launching a statistical-analysis-based attack such as DPA [Kocher et al. 1999; Taha and Schaumont 2013], an adversary can reliably deduce the value of k.

In contrast, we say that o4 is *perfectly masked* because the output value is statistically independent of the sensitive data k. When k is logical 1 (or 0), there is 50% chance that o4 is logical 1 (or 0) and vice versa. Therefore, the computation is provably secure against any *first-order* power analysis attack, where the adversary can observe the side channel of *at most one* intermediate computation result.

To sum up, the example in Figure 1 demonstrates a weakness of the existing method [Bayrak et al. 2013]. Since this existing method only checks whether an intermediate computation result is masked but not whether it is perfectly masked, 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. In addition to first-order attacks, our method can check for higher-order power analysis attacks, where the adverasary can observe the side-channel of *more than one* intermediate computation result. Finally, our method checks for power side-channel leaks at the bit level, where *leakage freedom* automatically implies the implementation is leakage free at coarser levels of granularity (e.g., at the word level).

We have implemented our new method in a verification tool based on the LLVM compiler frontend [Lattner and Adve 2004] and the Yices SMT solver [Dutertre and de Moura 2006]. We encode the verification problem using a series of quantifier-free first-order logic formulas, whose satisfiability can be decided by an off-the-shelf SMT solver such as Yices. Our SMT encoding scheme is significantly different from the ones used in standard verification methods such as bounded model checking [Biere et al. 1999; Li et al. 2005] because the *perfect masking* property checked by our tool is not a functional property but statistical in nature. Specifically, it involves the calculation of the probability for an intermediate computation result to be logical 1, based on the assumption that all random bits are uniformly distributed in the domain of $\{0, 1\}$. This is in contrast to the purely functional properties checked by standard verification methods.

For experimental comparison, we have also implemented the method of Bayrak et al. [2013] in our verification tool. We have conducted experiments on a set of masking countermeasures for cryptographic software, including the ones applied to AES

and the MAC-Keccak reference code submitted to Round 3 of NIST's SHA-3 competition [Bertoni et al. 2013]. Our experimental results show the new method is effective in detecting imperfectly masked implementations of countermeasures from the source code. Furthermore, our results show the new method is scalable enough to handle software code of practical size and complexity. In particular, it is able to formally verify one round of the MAC-Keccak implementation with 285K Boolean-level operations in a matter of minutes.

Summary of contributions. We make the following contributions in this article.

—We propose the first SMT-solver-based method for formally verifying the security of perfect masking countermeasures against power-analysis-based side-channel attacks.
—We implement the new method in a software tool built upon the LLVM compiler frontend and the Yices SMT solver for directly checking the C programs of cryptographic software implementations.
—We conduct experimental evaluation on a set of recently proposed masking countermeasures for cryptographic algorithms such as AES and MAC-Keccak to demonstrate the effectiveness of the proposed method.

Organization of the Article. The remainder of this work is organized as follows. We will establish notation in Section 2 before presenting our SMT-based verification algorithm in Section 3. Then, we will illustrate the verification process using an example in Section 4. We will present our incremental verification method in Section 6, which further improves the scalability of our SMT-solver-based verification method. We will present the experimental results in Section 7, review related work in Section 8, and finally give our conclusions in Section 9.

2. PRELIMINARIES

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

2.1. Side-Channel Attacks

Following the notation used by Blömer et al. [2004], we assume the program to be verified implements a function $c \leftarrow enc(x, k)$, where x is the plaintext, k the secret key, and c the ciphertext. We assume that x, k, and c are all finite-length bit-vectors. Furthermore, the implementation of the function $enc(x, k)$ consists of a sequence of computations. 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 $enc(x, k)$ function, where r is a random variable added to the implementation of $enc(x, k)$ to mask the secret key k. Here, r is an s-bit random number uniformly distributed in the domain $\{0, 1\}^s$. The purpose of using r to mask k is to make all intermediate computation results statistically independent of the secret key. We assume that, by default, the pseudorandom number generators available on chip are uniformly distributed, which is typically the case in practice.

When $enc(x, k)$ is a linear function of k in the Boolean domain, for example, masking and demasking are straightforward. Specifically, when x and k are Boolean variables, we can take advantage of the fact that $enc(x, k \oplus r) = enc(x, k) \oplus enc(x, r)$, by first masking k with r using the XOR operation and then demasking the result with the XOR of $enc(x, r)$ because, according to the property of XOR, $enc(x, k \oplus r) \oplus enc(x, r) = enc(x, k) \oplus enc(x, r) \oplus enc(x, r) = enc(x, k)$. In this case, the implementation of function $enc(x, k)$ does not need to be changed. However, when $enc(x, k)$ is a nonlinear function, masking and demasking become complicated, since they often require a complete redesign of the implementation of function $enc(x, k)$. Manually designing such a

masking countermeasure is labor intensive and error prone and, currently, there is a lack of automated tools to assess how secure the resulting countermeasure really is.

The verification method proposed in this article can accommodate power leakage models at different levels of granularity. The most generic and basic power model is the *Hamming Weight (HW) on variables* model [Mangard et al. 2007], which has been widely used in the context of differential power analysis. In this model, the power consumption of an embedded computing device executing an instruction is statistically dependent on the Hamming weight of the operands of this instruction. Here, the Hamming weight of a bit-vector is simply the number of bits that are set to the logical 1. For ease of presentation, we shall use the HW model during the illustration of our verification method. However, we can also plug in more accurate power models, such as the *Hamming Distance (HD)* model, if the exact register location for each program variable is known. The HD model relies on the number of bits flipped between the current and the previous values of the register overwritten by a program variable.

The attack model considered in this work is as follows. We assume 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, by observing the power side channels, the joint distribution of at most d intermediate computation results $I_1(x, k, r), \ldots, I_d(x, k, r)$. However, the adversary does not have access to the random variable r that is produced by a true random number generator. The goal of the adversary is to compute the secret key (k). In embedded computing devices such as the SmartCard, for instance, this is a realistic attack model. Kocher et al. demonstrated in their seminal work [1999] that, for $d = 1$, the sensitive data can be reliably deduced using a statistical method known as differential power analysis (DPA).

2.2. Perfect Masking

Given a pair (x, k) of plaintext and secret key for the function $enc(x, k)$, a random variable r, and d intermediate computation results $I_1(x, k, r), \ldots, I_d(x, k, r)$ of the function, 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 $R = \{0, 1\}^s$. Following Blömer et al. [2004], we consider the implementation vulnerable as long as there is any statistical dependence between $D_{x,k}(R)$ and the sensitive data k. In other words, we do not put restrictions on the technical capability of an adversary.

Definition 2.1. Given an implementation of function $enc(x, k)$ and a set of intermediate results $\{I_i(x, k, r)\}$ inside the function, we say the implementation is *order-d* perfectly masked if, for any d-tuple $\langle I_1, \ldots, I_d \rangle$, we have

$$D_{x,k}(R) = D_{x,k'}(R)$$

for any two pairs of plaintexts and keys, denoted (x, k) and (x, k').

The notion of *perfect masking* [Blömer et al. 2004] used in our work is more accurate than the notion of *sensitivity* used by Bayrak et al. [2013]. In their method, an intermediate computation result $I_i(x, k, r)$ is considered *sensitive* if its value is *logically* dependent on the secret data and *logically* independent of any random variable. A function f is logically dependent on a variable x if and only if the value of x can affect the value of f. We have demonstrated the difference between *logical* and *statistical* dependence using the example in Figure 1, where o1,o2,o3,o4 are all *insensitive* according to the method of Bayrak et al. while only o4 is *perfectly masked*. In general, if an intermediate computation result is perfectly masked, it is statistically independent of the secret input and therefore guaranteed insensitive. However, the reverse is not always true; an insensitive intermediate computation result may not be perfectly masked.

```
 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 : }
```
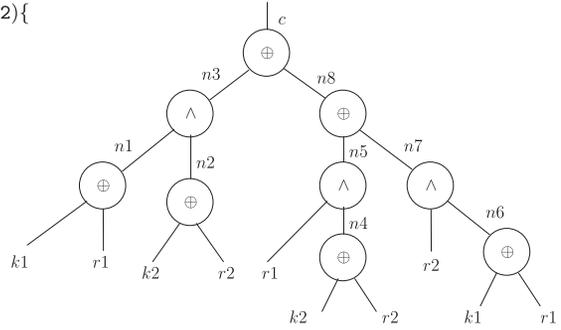
Fig. 2. Example: a Boolean program and its graphic representation (⊕ denotes XOR; ∧ denotes AND).

As the preceding definition of perfect masking has alluded, checking for violations of *perfect masking* requires us 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 $D_{x,k}(R)$ for a given pair of plaintext x and key k. We will present our SMT-solver-based solution to this problem in Section 3.

2.3. Cryptographic Software

In this work, we focus on verifying security-critical software programs, such as the C code, that implement cryptographic algorithms such as AES and MAC-Keccak, as opposed to arbitrary software applications. Indeed, our SMT-solver-based method would be too expensive for verifying general-purpose software applications. The class of C programs considered in this work, in general, does not have nonmanifest control (such as input-dependent loops), 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 targeted program can be transformed into a branch-free representation where the if-else branches are merged. Finally, since all variables in this type of programs are bounded integers, we can model them as finite-length bit-vectors or convert the entire program to a purely Boolean program through bit-blasting. Therefore, in the remainder of this article 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 Boolean 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 Figure 2. The program is a masked version of $c \leftarrow (k1 \wedge k2)$, where $k1$ and $k2$ are two secret-key bits, $r1$ and $r2$ are two random bits with independent and uniform distribution in the domain of $\{0, 1\}$, and c is the computation result. The objective of masking is to make the power side channel of the computing device independent of the values of the secret-key bits.

The masking scheme illustrated by Figure 2 came from Blömer et al. [2004]. After masking, the value c in the program is logically equivalent to $(k1 \wedge k2) \oplus (r1 \wedge r2)$. The corresponding demasking function (not shown in the figure) would be $c \oplus (r1 \wedge r2)$. Due to the property of the XOR operation, demasking would produce a result logically equivalent to the desired value $(k1 \wedge k2)$.

Our verification method will determine whether all the intermediate variables of the program in Figure 2 are not only logically dependent on some random bits (meaning they are masked) but also statistically independent of the secret bits (meaning they are perfectly masked). We use the Clang/LLVM compiler frontend to parse the input C

program and construct the dataflow graph, where the root node of the graph 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 Figure 2, our method starts by parsing the function `compute` on the left-hand side and creating the graph representation on the right-hand side. 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, our method checks whether the node is perfectly masked. The order in which the internal nodes are checked in the graph is as follows: $n1, n2, n3, n4, n5, n6, n7, n8$, and finally, c.

3.1. 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 bits in r as random. Then, for each intermediate computation result $I(x, k, r)$ of the function $enc(x, k)$ where $I(x, k, r)$ is a Boolean function in terms of x, k, and r, we check whether it is perfectly masked. Following Definition 2.1, we formulate this check as a satisfiability problem as follows:

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

With a little abuse of notion in the following paragraphs, we use x to represent the value of the plaintext bits, k and k' to represent two different valuations of the key bits, and r to represent the random value in the domain of $\{0, 1\}^s$, where s is the number of random bits: For any fixed value combination (x, k, k'):

—$\Sigma_{r \in \{0,1\}^s} I(x, k, r)$ is the number of assignments of r under which $I(x, k, r)$ evaluates to 1; and
—$\Sigma_{r \in \{0,1\}^s} I(x, k', r)$ is the number of assignments of r under which $I(x, k', r)$ evaluates to 1.

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

If the previous formula is satisfiable, then there exists a plaintext x and two different keys (k, k') such that the probability distribution of $I(x, k, r)$ differs from that of $I(x, k', r)$. In other words, some information of the secret key k is leaked through the power side channel. In this case, we say that I is not perfectly masked. If the preceding formula is unsatisfiable, it means that such an information leak is never possible. In this case, we say I is perfectly masked.

Another way to understand the aforesaid solution is to look at the negation of the satisfiability problem, which is a validity checking problem. That is, instead of checking the *satisfiability* of the preceding formula, we can check the *validity* of the following one.

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

If this formula is valid (meaning the equality holds for all valuations of x, k, and k') we say that I is perfectly masked.

3.2. The Encoding Method

We now explain the method for generating the satisfiability modulo theory (SMT) formula defined in the previous section. Let Φ denote the formula to be created for checking whether the intermediate computation result $I(x, k, r)$ has information leakage. Let s be the number of random bits in r. We want Φ to be satisfiable if and only if
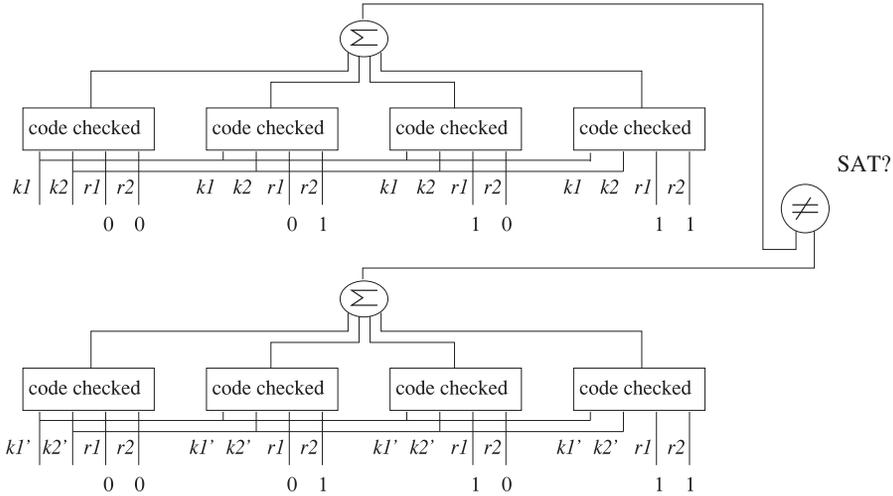
Fig. 3. SMT encoding for checking the statistical dependence of an output on secret data ($k1, k2$).

the current node $I(x, k, r)$ is not perfectly masked by the random variable r. We define Φ as

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

where the subformulas $\Psi_k^r, \Psi_{k'}^r, \Psi_{b2i}, \Psi_{sum}, \Psi_{diff}$ are defined as follows.

—*Program logic (Ψ_k^r)*. Each subformula Ψ_k^r encodes a copy of the input-output relation of function $I(x, k, r)$, with the random value r set to a concrete value in $\{0, \ldots, 2^s - 1\}$ and the key set to one of the two values k or k'. All copies of the program logic share the same plaintext value 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-1's (Ψ_{sum})*. It encodes the two summations of the logical 1's in the outputs of the 2^s program logic copies, one for the input-output relation of $I(x, k, r)$ and the other for the input-output relation of $I(x, k', r)$.

—*Different sums (Ψ_{diff})*. It asserts that the two summations should have different results.

Figure 3 is a pictorial illustration of our SMT encoding method for an example intermediate computation result $I(k1, k2, r1, r2)$, where $k1$ and $k2$ are the secret-key bits, and $r1$ and $r2$ are two random bits. Here, the first four boxes, which encode $\Psi_k^0, \ldots, \Psi_k^3$, are 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, which encode $\Psi_{k'}^0, \ldots, \Psi_{k'}^3$, are four copies of the program logic for key bits ($k1'k2'$) with the random bits set to 00, 01, 10, and 11, respectively. The comparison on the right-hand side checks for vulnerabilities against first-order DPA attacks as to whether there exist two sets of key values ($k1k2$ and $k1'k2'$) under which the probabilities of $I(x, k, r)$ being logical 1 are different.

Consider node $n8$ in Figure 2 as an example. The function of $n8$ in terms of $k1, k2, n1, n2$ is defined as n8 = (r1 ∧ (k2 ⊕ r2)) ⊕ (r2 ∧ (k1 ⊕ r1)). The SMT formula that our encoding method generates (by instantiating r1r2 to 00, 01, 10, and

```
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 satisfiability check
```

Fig. 4.   Example: the conjunction set of SMT formulas generated for checking the side-channel leaks of node n8 in Figure 2.

11) is the conjunction of all of the formulas shown in Figure 4. We solve the conjunction of these formulas using an off-the-shelf SMT solver called Yices [Dutertre and de Moura 2006]. In this particular example, the entire conjunctive formula is satisfiable. One satisfying assignment, for example, 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%, whereas when the key bits are 01, the probability for $n8$ to be logical 1 is 50%. This makes $n8$ vulnerable to first-order DPA attacks in that, by observing the power side channel corresponding to the value of $n8$, an adversary may deduce the values of the key bits. Therefore, we say node $n8$ is not perfectly masked.

Since our SMT encoding method closely follows the satisfiability formula defined in Section 3.2 that in turn closely follows Definition 2.1 of *perfect masking*, we have the following theorem regarding the correctness of our method.

THEOREM 3.1. *Let Φ be the formula created by the SMT encoding method for intermediate computation result $I(x, k, r)$. Formula Φ is satisfiable if and only if node $I(x, k, r)$ is perfectly masked.*

The size of the SMT formula resulting from our method is linear in the software code size and exponential in the number of random bits, due to the fact that we make 2^{s+1} copies of the program logic Ψ_k^r, one for each a distinct value of the s-bit random

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. 5. The truth tables for internal nodes $n3$, $n8$, and c of the example program in Figure 2.

variable. In practice, the formula size can become a performance bottleneck, which will be mitigated by our new incremental verification algorithm in Section 6.

4. THE WORKING EXAMPLE

Consider the verification of our running example in Figure 2. For each internal node I in the graph representation, we first identify all the transitive fanin 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 those transitive fanin nodes on which it depends logically. In the next section, we shall present new algorithms to systematically reduce the size of the code region, thereby leading to a faster and more scalable verification procedure. In this section, for ease of presentation we assume the region consists of the entire fanin cone.

Once the code region is extracted, it is given to our SMT-solver-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 the verification process with node $n1$ as defined in line 3 of the program in Figure 2. The extracted code region consists of $n1 = k1 \oplus r1$ itself. Since the code region involves only one secret key and one random variable in the XOR operation, a simple static analysis can prove it is perfectly masked. That is, a secret bit will be randomized if it is XOR-ed with a fresh random bit. By *fresh*, we mean that the random bit has not been used in other parts of the program. Therefore, although we could have verified $n1$ using the SMT-solver-based method, we avoid it for efficiency reasons. Such simple static analysis is able to prove that nodes $n2$, $n4$, and $n6$ are also perfectly masked.

Next, we check whether $n3$ is perfectly masked. The truth table of $n3$ is shown in Figure 5 (left). According to the truth table, in all four valuations of $k1$ and $k2$, the probability of $n3$ being logical 1 is 25%, therefore $n3$ is perfectly masked. This needs to be proved using our SMT-solver-based verification method. When we apply the SMT-solver-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 we do not need to check the probability of the output being logical 0, since having an equal probability distribution for logical 1 is equivalent to having an equal probability distribution for logical 0.

The verification steps for nodes $n5$ and $n7$ are similar to that of $n3$; all can be proved perfectly masked using the SMT-solver-based verification method.

Next, we check whether $n8$ is perfectly masked. The attempt to obtain a proof using the SMT solver would fail because, as shown in the truth table in Figure 5 (middle), the probability for $n8$ to be logical 1 is not the same for all valuations of the secret-key bits. For example, if the key bits are 00, then $n8$ would be logical 0 regardless of the values of the random variables. Recall that we have shown the detailed SMT encoding for $n8$ in Section 3.2. Using our new method, the SMT 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 Figure 5 (right). Similar to $n8$, our SMT-solver-based method would be able to show it is not perfectly masked.

It is worth pointing out that the result of applying the *Sleuth* method [Bayrak et al. 2013] to this example would have been different. The *Sleuth* method, based on the notion of *sensitivity*, would have (incorrectly) classified $n8$ and c as being "securely masked" despite the fact that our new method has shown $n8$ and c are still vulnerable to first-order DPA attacks. Therefore, the example in Figure 2 demonstrates a major advantage of our new method over *Sleuth*.

5. CHECKING FOR HIGH-ORDER ATTACKS

The encoding method presented in Section 3.2 considers only *first-order* attacks, where an adversary can have access to the side channel of at most one intermediate computation result. Under this assumption, for an implementation to resist power analysis attacks, each intermediate computation result must be perfectly masked. However, even if each intermediate computation result is perfectly masked, the implementation may still be vulnerable to *high-order* DPA attacks, where an adversary may simultaneously observe the side channels of multiple intermediate computation results. Here, the order of attack is defined as the number of intermediate computation results whose side channels are observable by the adversary.

In the general case, to defend against *order-d* side-channel attacks, the implementation must have the following property: the joint distribution of any d intermediate computation results (where $d = 1, 2, 3, \ldots$) must be statistically independent of the secret data. We formalize this requirement as a satisfiability problem as follows. There exist d intermediate computation results such that

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

Here, $\oplus_{i=1}^d$ computes the symmetric difference of the d intermediate computation results (their exclusive-or). When $d = 1$, the previous formula degenerates to the one we presented in Section 3.1. When $d > 1$, the SMT encoding method presented in Section 3.2 can be extended accordingly to implement this check.

Specifically, $I(x, k, r)$ and $I(x, k', r)$ in the encoding for checking first-order attacks are replaced by $\oplus_{i=1}^d I_i(x, k, r)$ and $\oplus_{i=1}^d I_i(x, k', r)$, respectively. Therefore, the resulting SMT formula, denoted Φ^d, contains $(2^r \times d)$ copies of the program logic; there are d intermediate computation results involved, each producing 2^r copies of its program logic. Since the method closely follows the definition of *perfect masking*, we have the following theorem regarding the correctness of our method.

THEOREM 5.1. *Let Φ^d be the formula created by the SMT encoding method for d intermediate computation results $I_1(x, k, r), \ldots, I_d(x, k, r)$. Formula Φ^d is satisfiable if and only if nodes $I_1(x, k, r), \ldots, I_d(x, k, r)$ are perfectly masked.*

In practice, most of the countermeasures proposed by cryptographic system engineers assume the adversary has access to the side-channel leakage of either one or two intermediate computation results, which corresponds to either first-order or second-order
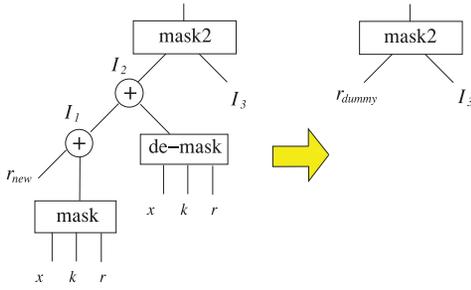
$$I_2 := I_1 \oplus de\text{-}mask(x, k, r)$$
$$:= r_{new} \oplus mask(x, k, r) \oplus de\text{-}mask(x, k, r)$$
$$:= r_{new} \oplus (\ldots)$$
$$:= r_{dummy}$$

Before verifying *mask2*, if we have already proved that I_2 is *perfectly masked*, and r_{new} is a fresh random bit not used elsewhere in the code, then for the purpose of checking *mask2*, we can replace I_2 with r_{new} or r_{dummy}.

Fig. 6.   Incremental verification: applying the SMT-solver-based analysis to small fanin regions only.

attacks. When $d = 2$, the symmetric differences become $(I_1(x, k, r) \oplus I_2(x, k, r))$ at the left-hand side and $(I_1(x, k', r) \oplus I_2(x, k', r))$ at the right-hand side. This is due to the relationship between the Hamming Distance (HD) model and the Hamming Weight (HW) model. In other words,

$$HD(I_1(x, k, r), I_2(x, k, r)) = HW(I_1(x, k, r) \oplus I_2(x, k, r)),$$
$$HD(I_1(x, k', r), I_2(x, k', r)) = HW(I_1(x, k', r) \oplus I_2(x, k', r)).$$

In the implementation of our verification tool, we have included the checks for both first- and second-order attacks. In our experiments, we have also evaluated our new method in detecting vulnerabilities against both first- and second-order power analysis attacks (the results will be presented in Section 7).

6. THE INCREMENTAL VERIFICATION ALGORITHM

Although our new method is extremely effective in detecting even minor side-channel leaks, the size of the SMT formula created by our encoding method can become large, since it is linear in the size of the program and exponential in the number of random variables. Recall that, for s random bits, we would need to make 2^{s+1} copies of the program logic Ψ_k^r, each for a distinct random value r. This is the main performance bottleneck for applying our method to large programs. In this section, we propose an incremental verification algorithm that applies the SMT-solver-based analysis only to small code regions, one at a time, as opposed to the entire fanin cone of the node under verification. This is crucial for scaling the new verification method to cryptographic software of practical size.

6.1. Extracting the Verification Region

Our incremental verification algorithm relies on the following observation. In practice, a common strategy used by cryptographic system engineers in implementing masking countermeasures is to create a chain of modules, where the inputs of each module are masked before executing its logic and demasked afterward. To avoid having unmasked intermediate values, the inputs to the successor module are masked with fresh random variables before they are demasked from the random variables used by the previous module. This can be illustrated by the example in Figure 6, 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 using the XOR with *de-mask(x,k,r)*.

Due to *associativity* of the $\oplus$ operator, reordering the masking and demasking operations would not change the logical result. For example, in Figure 6, we assume that the current instruction being verified is in *mask2()*. Since the newly added random variable r_{new} is not used inside *mask()* nor *de-mask()* nor in the support of I_3, we can safely replace the entire fanin cone of I_2 by r_{new} or a new random variable r_{dummy} while

verifying *mask2()*. We shall see in the experimental results section that such opportunities for performance optimization are abundant in real-world applications. In the remainder of this section, we present an algorithm that leverages this idea to soundly extract a small code region from the fanin cone of the node under verification.

Our algorithm relies on constructing some auxiliary data structures associated with node i, the current under verification. These data structures are defined as follows.

—*supportV[i]* is the set of inputs in the support of the function of node i.
—*uniqueM[i]* is the set of random inputs, each reaching node i along only one path in the graph representation.
—*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 topological traversal of the program nodes as described in Algorithm 1. The input of this procedure is the current node v and the output is the set of auxiliary data structures defined before.

For example, for node I_1 in Figure 6, we have $supportV[I_1] = \{x, k, r, r_{new}\}$, $uniqueM[I_1] = \{r, r_{new}\}$, 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 demasked, 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: }

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: }

Our idea of extracting a small code region from the fanin cone for the SMT-solver-based analysis is formalized in Algorithm 2. Given the node i under verification and

uniqueM[i] as the set of random variables, each reaching node i along only one path in the graph representation, we call GETREGION(*i,uniqueM[i]*) to compute the code region. Inside GETREGION, the table *uniqueM[i]* is renamed to *freshMasksATi*. We start by checking each transitive fanin 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 whether there is a random variable $r \in uniqueMATi$ that, by itself, can perfectly mask node n (line 4). In Figure 6, for example, r_{new}, by itself, can uniformly mask node I_2. If such a random variable r exists, then we add the pair (n, r) to the region and return, skipping the rest of the fanin cone of n. If such a random variable r does not exist, we recursively invoke GETREGION() to traverse the two child nodes of n.

6.2. The Overall Algorithm

Algorithm 3 shows the overall flow of our incremental verification method. Given the program *Prog* and the lists of secret, random, and plaintext variables, our method systematically scans through all the internal nodes in topological order, from the inputs to the return value. For each node i, our method first extracts a small code region (line 4). Then, it checks whether the node is perfectly masked by invoking the SMT-solver-based verification method (line 6). 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:       if ( NEEDTOBECHECKED( i, region ) ) {
6:          notPerfect ← CHECKMASKINGBYSMT (i, region, keys, rands, plains )
7:          if (notPerfect)
8:             badNodes.add( i )
9:       }
10:    return badNodes
11: }
```

To reduce the runtime overhead of the SMT-solver-based verification method, we add a set of simple static checks between line 4 and line 6 to quickly decide whether the SMT-solver-based verification needs to be invoked. These simple static checks have been implemented inside the subroutine NEEDTOBECHECKED(*i,region*). There are three main computation steps inside this subroutine.

—*Don't-Care Random Variables*. We compute the set of random variables in the fanin cone of node i whose values do not affect the output of node i. The result of this computation will be used in the subsequent static checks.
—*No-Secret-Variable Check*. We check whether the code region associated with node i contains any secret-key bit. If the answer is no, then the region is guaranteed free of side-channel leaks. In this case, we can skip the check of masking by SMT.
—*Three Syntactic Conditions*. We check three static conditions associated with the current region and node i and, if all of these conditions are satisfied, the region is guaranteed perfectly masked. In such a case, we can skip the check of masking by SMT.

In the following paragraphs, we explain the first and third steps of NEEDTOBECHECKED in more detail.

We identify the set of random variables that are *don't-cares* for the current node i. These variables will be used later to reduce the computational cost of verifying masking by SMT. Specifically, for each random variable $r \in supportV[i]$, we check whether the value of r can ever affect the output of node i logically. If the answer is no, then r is a *don't-care*. If r has been proved a don't-care, we will leverage the information to speed up the subsequent SMT formulas. Specifically, during our SMT encoding, we will set r to logical 0 rather than treat r as a random variable, 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.

Whether a random variable $r \in support[i]$ is a *don't-care* for node i can be decided by using the SMT solver. Toward this end, we construct an SMT formula as

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

where $\Psi_{region}^{r=0}$ encodes the function of node i with the random bit r set to 0, $\Psi_{region}^{r=1}$ encodes the function of node i with the random bit r set to 1, and Φ_{diffO} asserts that their outputs differ. If the preceding formula is unsatisfiable, it means that the value of node i remains the same regardless of the value of r. In other words, r is a *don't-care* for node i.

We check the following three syntactic conditions in order to quickly decide whether the current code region is perfectly masked. All three conditions must be satisfied for us to conclude that the region is perfectly masked. If any of them is not satisfied, we need to continue with the check for masking by SMT.

—Node i has no secret input as its immediate child. The condition ensures that, whenever a secret variable is introduced to the region, its masking operation will be checked by SMT.
—None of the random variables appears in the *supportV* tables of both operands of node i. This condition ensures that no perfect masking of a secret variable in any of the operands may be affected.
—Both operands of node i are perfectly masked. This condition ensures our method will find all the resultant imperfectly masked nodes due to an initial imperfectly masked node.

The effectiveness of using these simple static checks to reduce the computational cost of the SMT-solver-based verification method will be evaluated empirically in Section 7.3.

7. EXPERIMENTS

We have implemented our method in a verification tool called *SC Sniffer*, based on the LLVM compiler frontend [Lattner and Adve 2004] and the Yices SMT solver [Dutertre and de Moura 2006]. Our verification tool runs in two modes: monolithic and incremental. In the monolithic mode, *SC Sniffer* applies the SMT-based encoding method (Section 3.2) to the entire fanin cone of each intermediate node in the program, whereas in the incremental mode, *SC Sniffer* restricts the SMT encoding to a localized code region (Section 6.2).

We have implemented the *Sleuth* method [Bayrak et al. 2013] for the purpose of experimental comparison. The main difference between these two methods is that our new method not only checks whether a node is masked (as in *Sleuth*), but also whether it is perfectly masked, that is, whether the node is statistically independent of the secret key.

We have evaluated our new method on a set of masking countermeasures for recently proposed cryptographic software implementations, including real-world algorithms such as AES and MAC-Keccak. Our new method is designed for verifying perfect

Table I. Benchmark Statistics

Name	Description	Code	Nodes	Keys	Plains	Rnds
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

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.

masking countermeasures for nonlinear operations. Therefore, all of the benchmarks used in our experiments have nonlinear operations/functions (e.g., the S-BOX in AES). During the construction of the SMT formula, we use both Boolean logic and bit-vector arithmetic, which can efficiently encode linear and nonlinear operations. 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 since it checks for statistical dependence in addition to logical dependence. But does it have a significant advantage over *Sleuth* in detecting side-channel leaks in practice?
—How scalable is our new method, especially when used for verifying cryptographic software of realistic size and complexity? We have extended our SMT-based method with incremental analysis. Is it effective in practice?

7.1. Benchmarks

Table I shows the statistics of the benchmarks. Column 1 shows the name of each benchmark example, while column 2 shows a short description of the implemented algorithm. Column 3 shows the number of lines of code—here, each instruction occupies a line and each instruction is a bit-level operation. Column 4 shows the number of nodes that represent the intermediate computation results and columns 5 through 7 show

the number of input bits that represent the secret key, the plaintext, and the random variable, respectively.

The benchmarks are classified into three groups. The first group of test cases (P1 to P5) are examples taken from the *Sleuth* benchmark [Bayrak et al. 2013], all of which contain intermediate variables that are not masked at all. More specifically, P1 is the masking key whitening code in Bayrak et al. [2013, page 12]. P2 is the AES8 example originated from Herbst et al. [2006]—it is a smartcard implementation of AES resistant to power analysis. P3 is the code in Bayrak et al. [2013, page 13] that also originated from Herbst et al. [2006], while P4 is the code in Bayrak et al. [2013, page 18] that originated from Messerges [2000]. Finally, P5 is the code in Bayrak et al. [2013, page 18] that originated from Goubin [2001]. The Boolean-to-arithmetic mask conversion algorithm has possible second-order attack points that can be successfully detected by our new method.

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. Specifically, P6 and P7 are the two examples used by Blömer et al. [2004, page 7], while P8 and P9 are the MAC-Keccak computation reordered examples that originated from Bertoni et al. [2013, page 46, Eq. (5.2)]. P10 and P11 are two experimental masking schemes for the Chi function in MAC-Keccak, neither perfectly masked.

The third group of test cases (P12 to P17) comes from the regeneration of the MAC-Keccak reference code submission to NIST in the SHA-3 competition [NIST 2013]. There are a total of 285K lines of Boolean operation code. The difference among these test cases is that they are protected by various masking countermeasures, some of which are perfectly masked (e.g., P12) whereas others not.

7.2. Results

Table II shows the experimental results obtained on a desktop machine with a 3.4 GHz Intel i7-2600 CPU, 4GB RAM, and a 32-bit Linux OS. We have compared the performance of three methods: *Sleuth*, *SC Sniffer* (monolithic), and *SC Sniffer* (incremental). Here, *Sleuth* is the method proposed by Bayrak et al. [2013], while the other two are our new methods. In this table, column 1 shows the name of each test program. Columns 2 through 5 show the results of running *Sleuth*, including whether the program passed the check, the number of nodes not masked, and the total number of nodes checked. Columns 6 through 9 show the results of running our new monolithic method. Here, MO (mem-out) means the method requires more than 4GB of RAM. Columns 10 through 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 that of nodes checked because many of them are already resolved by our static checks.

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 runtime is small. However, on large programs such as the MAC-Keccak reference code, the monolithic method could not finish since it quickly ran out of the 4GB RAM, whereas the incremental method finished in a reasonable amount of time. Moreover, although the *Sleuth* method implements a significantly simpler (and hence weaker) check, it is also based

Table II. Experimental Results: Comparing our *SC Sniffer* Method with the *Sleuth* Method [Bayrak et al. 2013]

Name	*Sleuth* [Bayrak et al. 2013]				*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	91m	-	0	34	MO	Yes	0	128K	0	10m
P13	No	2560	128k	92m	No	1	46	MO	No	2560	128K	2560	14m
P14	Yes	0	128k	97m	-	0	31	MO	No	1024	128K	1024	18m
P15	Yes	0	152k	132m	-	0	32	MO	No	512	152K	1024	37m
P16	No	512	128k	113m	-	0	40	MO	No	1536	128K	1536	17m
P17	No	4096	131k	103m	-	0	34	MO	No	4096	131K	4096	17m
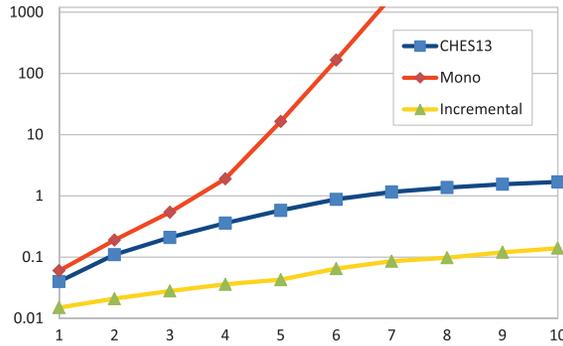
Fig. 7. Result: comparing the scalability of the three methods. The *x*-axis is the size of the program under verification, and the *y*-axis is the runtime in seconds.

on a monolithic approach. Our results in Table II show that, on large examples, our incremental method is significantly faster than *Sleuth*.

As a measurement of the scalability of the three algorithms, we have conducted more 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 Figure 7. In this figure, the *x*-axis shows the program size and the *y*-axis the total verification time in seconds. The results show that, among the three methods, our incremental method is the most scalable.

7.3. Statistics

In this section, we show the statistics of running our incremental verification method on the same set of benchmarks.

Table III shows how many of the code regions are verified using the SMT-solver-based verification method as opposed to the simple static checks as described in Section 6.2. Recall that, before applying the SMT-based verification, we first check whether the code region contains any secret variable and then check three syntactic conditions. If the static checks can guarantee the region is free of side-channel leaks, we will skip the SMT-based verification. In Table III, columns 1 and 2 show the program name and the number of nodes checked. Columns 3 through 5 show the number of nodes discharged by the two static checks and the SMT-based method, respectively. The results show that many of the verification subproblems were solved by the two simple static checks.

Table IV shows the number of calls to the SMT solver during the computation of don't-care random variables and during the verification of perfect masking. In this table, columns 2 and 3 show the number of calls to the SMT solver and the time spent on computing the don't-care random variables. Similarly, columns 4 and 5 show the number of calls to the SMT solver and the time spent on checking for perfect masking by SMT. For example, in P12, the execution time was spent entirely on computing the don't-care random variables, which invoked the SMT solver 38,400 times and spent 8 minutes and 15 seconds. Since all nodes are discharged by the simple static checks, the number of calls to the SMT solver during the check for perfect masking is 0.

Table V shows the average, minimum, and maximum number of variables in the SMT formulas generated during the computation of don't-care random variables and during the verification of perfect masking, respectively. Here, column 1 shows the name of the test program. Columns 2 through 4 show the statistics for computing don't-care

Table III. Statistics: Number of Nodes Discharged by the Two Simple Static Checks and the SMT-Based Verification Method, respectively

Benchmark		Nodes Discharged in Different Verification Steps		
name	nodes checked	no-secret-variable check	three syntactic conditions	check masking by SMT
P1	47	0	31	16
P2	31	8	15	8
P3	21	0	3	18
P4	24	0	16	8
P5	60	0	40	20
P6	9	4	3	2
P7	6	3	2	1
P8	19	6	10	3
P9	19	7	10	2
P10	22	8	12	2
P11	22	7	12	3
P12	128k	11,648	116,251	0
P13	128k	9,088	116,351	2,560
P14	128k	11,136	115,839	1,024
P15	152k	13,152	138,399	1,024
P16	128k	10,624	115,839	1,536
P17	131k	11,136	115,327	4,096

Table IV. Statistics: Number of Calls to SMT Solver during the Computation of Don't-Care Random Variables and during the Incremental Check for Perfect Masking by SMT, respectively

	Compute Don't Care Random Variables		Check for Perfect Masking by SMT	
Name	calls to SMT solver	total SMT solving time	calls to SMT solver	total SMT solving time
P1	0	0.00s	16	0.08s
P2	8	0.04s	8	0.04s
P3	54	0.32s	18	0.12s
P4	77	0.38s	8	0.15s
P5	161	0.96s	20	0.07s
P6	6	0.03s	2	0.03s
P7	3	0.02s	1	0.01s
P8	9	0.06s	3	0.04s
P9	9	0.06s	2	0.03s
P10	12	0.10s	2	0.05s
P11	12	0.08s	3	0.08s
P12	38,400	8m 15s	0	0.00s
P13	38,400	8m 43s	2,560	2m 15s
P14	38,400	8m 04s	1,024	7m 42s
P15	47,552	14m 1s	1,024	14m 57s
P16	37,888	8m 28s	1,536	5m 30s
P17	38,400	8m 32s	4,096	5m 35s

random variables and columns 5 through 7 show the statistics for the verification of perfect masking by SMT. The results show most of the hard-to-solve SMT formulas are generated during the verification of perfect masking.

Together, the experimental results presented in this section confirm our conjecture that the set of simple static checks added before the more heavyweight SMT-based verification method are effective in reducing the overall computational cost.

Table V. Experimental Results: Average, Minimum, and Maximum Number of Variables in the SMT Formulas Generated during the Verification Process

Name	Compute Don't Care Random Variables			Check for Perfect Masking by SMT		
	avg. variables	min. variables	max. variables	avg. variables	min. variables	max. variables
P1	-	-	-	28	28	28
P2	12	12	12	26	26	26
P3	14	12	19	30	30	30
P4	17	12	19	49	30	67
P5	16	12	19	60	30	67
P6	28	25	31	194	175	212
P7	27	25	28	91	91	91
P8	30	25	37	131	34	183
P9	32	25	41	179	175	183
P10	40	28	52	353	272	434
P11	39	33	46	336	281	434
P12	92	59	232	-	-	-
P13	88	54	232	79	75	85
P14	83	59	218	708	688	750
P15	94	59	233	1,310	996	1,710
P16	80	59	218	291	149	355
P17	84	59	228	112	83	221

7.4. Discussion

It is worth pointing out that the side-channel leaks detected by our new method are real security threats. We have confirmed these vulnerabilities in a follow-up study using real embedded computing hardware, where we conducted a set of power-analysis-based side-channel attacks on implementations of MAC-Keccak, nonlinear components of AES (such as the S-BOX), and a few other cryptographic algorithms. Specifically, we ran all software code on a 32-bit Microblaze processor [Xilinx 2014] built on a Xilinx Spartan-3e FPGA. We used a Tektronix DPO 3034 oscilloscope and a CT-2 current probe to sample the power consumption of the microprocessor core. The side-channel attack was conducted using differential power analysis (difference of means [Kocher et al. 1999]). Overall, the power side-channel attack resistance, as measured by the number of power measurement traces needed to determine the secret key, is dependent on the side-channel leaks detected by our method. For more information, please refer to the paper Eldib et al. [2014b].

8. RELATED WORK

Throughout the article, we have reviewed most of the related work on detecting power side-channel leaks in the source code of cryptographic software, such as *Sleuth* [Bayrak et al. 2013]. In this section, we will review the related work in general on the masking methods, compiler-assisted masking, and the detection/mitigation of other types of side-channel leaks.

8.1. Masking

There is a large body of work on designing and implementing masking countermeasures for cryptographic algorithms [Messerges 2000; Goubin 2001; Oswald et al. 2005; Herbst et al. 2006; Canright and Batina 2008; Moradi et al. 2011b]. However, in all these prior works, the countermeasures were manually designed and implemented and there was a lack of verification tools to ensure the masking countermeasures are indeed secure.

The notion of *perfect masking* was introduced by Blömer et al. [2004], following Shannon's notion of *perfect secrecy*. Blömer et al. also proposed a provably secure masking

scheme for AES. When implemented properly, perfect masking countermeasures are provably secure against power analysis attacks regardless of the technological capability of the adversary, even on computing hardware that has power emissions. To the best of our knowledge, the SMT-solver-based method proposed in this article is the first for formally verifying *perfect masking* countermeasures.

8.2. Compiler-Assisted Masking

Beyond verification, an interesting line of research is the automated construction of countermeasures against side-channel attacks. There has been some recent work along this line, including the software tool developed by Bayrak et al. [2011], the compiler-assisted masking method proposed by Moss et al. [2012], the code morphing method proposed by Agosta et al. [2012], and our method for generating perfect masking countermeasures using inductive synthesis [Eldib and Wang 2014]. Unlike previous work that is based on compiler transformation, which typically relies on matching known code patterns and then applying some predetermined transformation rules, our inductive synthesis-based method for constructing perfect masking countermeasures [Eldib and Wang 2014] is application agnostic and able handle both known and unknown vulnerabilities.

8.3. Other Types of Side Channels

Beside power side channels, sensitive information may be leaked through many other side channels, such as the execution-time [Kocher 1996; Köpf and Dürmuth 2009], faults [Biham and Shamir 1997], and cache side channels [Grabher et al. 2007]. Various leak detection and mitigation techniques have also been proposed for these types of side channels. For example, Köpf et al. [2012] proposed methods for conducting quantitative information flow analysis and Backes et al. [2009]. Doychev et al. [2013] developed a static analysis tool for detecting information leaks through cache side channels. Barthe et al. [2014] proposed a mitigation method designed for defending against concurrent cache attacks. Since these methods focus on other types of side channels, they are orthogonal to the new verification method proposed in this work.

9. CONCLUSIONS

We have presented a fully automated verification method for deciding whether a cryptographic software implementation is *perfectly masked* by random variables and therefore provably secure against power-analysis-based attacks. Our new method relies on translating the verification problem into a sequence of satisfiability problems that in turn can be decided by an off-the-shelf SMT solver such as Yices. We have also proposed an incremental analysis procedure to drastically improve the scalability of the SMT-based method. Our experiments on a set of recently proposed masking countermeasures for cryptographic algorithms such as AES and MAC-Keccak show the new method is not only more precise than existing methods in detecting power side-channel leaks, but also scalable for practical use.

REFERENCES

Johan Agat. 2000. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'00)*. 40–53.

Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. 2012. A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the 49th Annual ACM/IEEE Design Automation Conference (DAC'12)*. 77–82.

Michael Backes, Boris Kopf, and Andrey Rybalchenko. 2009. Automatic discovery and quantification of information leaks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (SP'09)*. 141–153.

Josep Balasch, Benedikt Gierlichs, Roel Verdult, Lejla Batina, and Ingrid Verbauwhede. 2012. Power analysis of atmel cryptomemory - Recovering keys from secure eeproms. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA'12)*. 19–34.

Gilles Barthe, Boris Kopf, Laurent Mauborgne, and Martin Ochoa. 2014. Leakage resilience against concurrent cache attacks. In *Proceedings of the 3rd International Conference on Principles of Security and Trust (POST'14)*. 140–158.

Ali Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. 2013. Sleuth: Automated verification of software power analysis countermeasures. In *Proceedings of the 15th International Workshop Cryptographic Hardware and Embedded Systems (CHES'13)*. 293–310.

Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, Francois-Xavier Standaert, and Paolo Ienne. 2011. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC'11)*. 230–235.

Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. 2013. Keccak implementation overview. http://keccak.neokeon.org/Keccak-implementation-3.2.pdf.

Armin Biere, Alessandro Cimatti, Edmund Clarke, Masahiro Fujita, and Yunshan Zhu. 1999. Symbolic model checking using sat procedures instead of BDDS. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference (DAC'99)*. 317–326.

Eli Biham and Adi Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'97)*. 513–525.

Johannes Blomer, Jorge Guajardo, and Volker Krummel. 2004. Provably secure masking of AES. In *Proceedings of the 11th International Conference on Selected Areas in Cryptography (SAC'04)*. 69–83.

David Canright and Lejla Batina. 2008. A very compact "perfectly masked" s-box for AES. In *Proceedings of the 6th International Conference on Applied Cryptography and Network Security (ACNS'08)*. 446–459.

Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. 1999. Towards sound approaches to counteract power-analysis attacks. In *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO'99)*. 398–412.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA.

Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*. 431–446.

Bruno Dutertre and Leonardo de Moura. 2006. A fast linear-arithmetic solver for DPLL(t). In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. Springer, 81–94.

Hassan Eldib and Chao Wang. 2014. Synthesis of masking countermeasures against side channel attacks. In *Proceedings of the International Conference on Computer Aided Verification (CAV'14)*. 114–130.

Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014a. SMT based verification of software countermeasures against side-channel attacks. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'14)*. 62–77.

Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. 2014b. QMS: Evaluating the side-channel resistance of masked software from source code. In *Proceedings of the 51st Annual Design Automation Conference (DAC'14)*. 1–6.

Louis Goubin. 2001. A sound method for switching between boolean and arithmetic masking. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*. 3–15.

Philipp Grabher, Johann Großschadl, and Dan Page. 2007. Cryptographic side-channels from low-power cache memory. In *Proceedings of the 11th IMA International Conference on Cryptography and Coding*. 170–184.

Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. 2006. An AES smart card implementation resistant to power analysis attacks. In *Proceedings of the 4th International Conference on Applied Cryptography and Network Security (ACNS'06)*. 239–252.

Marc Joye, Pascal Paillier, and Berry Schoenmakers. 2005. On second-order differential power analysis. In *Proceedings of the 7th International Conference on Cryptographic Hardware and Embedded Systems (CHES'05)*. 293–308.

Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the International Cryptology Conference (CRYPTO'96)*. 104–113.

Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Proceedings of the International Cryptology Conference (CRYPTO'99)*. 388–397.

Boris Kopf and Markus Durmuth. 2009. A provably secure and efficient countermeasure against timing attacks. In *Proceedings of the 22nd IEEE Symposium on Computer Security Foundations (CSF'09)*. 324–335.

Boris Kopf, Laurent Mauborgne, and Martin Ochoa. 2012. Automatic quantification of cache side-channels. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*. 564–580.

Chris Lattner and Vikram S. Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'04)*. 75–88.

Bing Li, Chao Wang, and Fabio Somenzi. 2005. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. *Int. J. Softw. Tools Technol. Transfer* 7, 2, 143–155.

Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer.

Thomas S. Messerges. 2000. Securing the AES finalists against power analysis attacks. In *Proceedings of the 7th International Workshop on Fast Software Encryption (FSE'00)*. 150–164.

Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. 2011a. On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-ii FPGAs. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. 111–124.

Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. 2011b. Pushing the limits: A very compact and a threshold implementation of AES. In *Proceedings of the 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'11)*. 69–88.

Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2012. Compiler assisted masking. In *Proceedings of the 14th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'12)*. 58–75.

NIST. 2013. Keccak reference code submission to NIST's SHA-3 competition (round 3). http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/KeccakFinalRnd.zip.

Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. 2005. A side-channel analysis resistant description of the AES s-box. In *Proceedings of the International Workshop on Fast Software Encryption (FSE'05)*. 413–423.

Christof Paar, Thomas Eisenbarth, Markus Kasper, Timo Kasper, and Amir Moradi. 2009. KeeLoq and side-channel analysis-Evolution of an attack. In *Proceedings of the International Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'09)*. 65–69.

Emmanuel Prouff and Matthieu Rivain. 2013. Masking against side-channel attacks: A formal security proof. In *Proceedings of the 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'13)*. Springer, 142–159.

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Select. Areas Comm.* 21, 1, 5–19.

Mostafa Taha and Patrick Schaumont. 2013. Differential power analysis of MAC-Keccak at any key-length. In *Proceedings of the International Conference on Advances in Information and Computer Security (IWSEC'13)*. 68–82.

Chao Wang, Gary D. Hachhtel, and Fabio Somenzi. 2006. *Abstraction Refinement for Large Scale Model Checking*. Springer.

Chao Wang, Zijiang Yang, Franjo Ivancic, and Aarti Gupta. 2007. Disjunctive image computation for software verification. *ACM Trans. Des. Autom. Electronic Syst.* 12, 2.

XILINX. 2014. MicroBlaze soft processor core. http://www.xilinx.com/tools/microblaze.htm.

Zijiang Yang, Chao Wang, Aarti Gupta, and Franjo Ivancic. 2009. Model checking sequential software programs via mixed symbolic analysis. *ACM Trans. Des. Autom. Electronic Syst.* 14, 1.