Synthesis of Masking Countermeasures against Side Channel Attacks

Hassan Eldib and Chao Wang

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

Abstract. We propose a new synthesis method for generating countermeasures for cryptographic software code to mitigate power analysis based side channel attacks. Side channel attacks may arise when computers and microchips leak sensitive information about the software code and data that they process, e.g., through power dissipation or electromagnetic radiation. Such information leaks have been exploited in commercial systems in the embedded space. Our new method takes an unprotected C program as input and returns a functionally equivalent but side channel leak free new program as output. The new program is guaranteed to be *perfectly masked* in that all intermediate computation results are made statistically independent from the secret data. We have implemented our new method in a tool based on the LLVM compiler and the Yices SMT solver. Our experiments on a set of cryptographic software benchmarks show that the new method is both effective and scalable for applications of realistic size.

1 Introduction

When cryptographic algorithms are proved to be secure against thousands of years of brute force cryptanalysis attacks, the assumption is that sensitive information can be manipulated in a closed computing environment. Unfortunately, real computers and microchips leak information about the software code and data that they process, e.g. through power dissipation or electromagnetic radiation. For example, the power consumption of a typical embedded device executing instruction $a = t \oplus k$ may depend on the value of the secret variable k [21]. Such information can be exploited by an adversary through statistical post-processing such as differential power analysis (DPA [19]), leading to successful attacks in linear time. In recent years, many commercial systems in the embedded space have shown weakness against such attacks [25,22,4].

In this paper, we propose a new synthesis method, which takes an unprotected software program as input and returns a functionally equivalent but side channel leak free new program as output. By leveraging a new verification procedure that we developed recently, called *SC Sniffer* [14,15], we can guarantee that the synthesized new program is secure by construction. That is, all intermediate computations of the program are *perfectly masked* [9] in that their computation results are statistically independent from the secret data. Masking is a popular and relatively low-cost mitigation strategy for removing the statistical dependency between sensitive data and side channel emissions. For example, Boolean masking [4,26] uses an XOR operation of a random bit r with variable a to obtain a masked variable: $a_m = a \oplus r$. The original value can be restored by a second XOR operation: $a_m \oplus r = a$. Since a_m no longer depends on the sensitive data a statistically, subsequent computations based on a_m will not leak information about the value of a .

When a computation $f(z)$ is in the linear domain in terms of $\oplus$ and with respect to the sensitive input z , masking can be implemented as $f(z \oplus r) \oplus f(r)$ since it is equivalent to $f(z) \oplus f(r) \oplus f(r) = f(z)$. That is, we mask z using an XOR with random bit r before the computation and de-mask using an XOR with $f(r)$ afterward. However, when $f(z)$ is a non-linear function, the computation $f(z)$ often needs to be completely redesigned, e.g., by splitting $f()$ into $f'()$ and $f''()$ such that $f'(z \oplus r) \oplus f''(r) = f(z)$. Finding the proper $f'()$ and $f''()$ is a highly creative process currently performed by cryptographic experts. Indeed, designing a new masking countermeasure for algorithms such as AES and SHA-3 would be publishable work in cryptographic venues.

Our new synthesis method relies on *inductive synthesis* and satisfiability modulo theory (SMT) solvers to search for masking countermeasures within a bounded design space. More specifically, given the software code to be masked, we use a set of quantifier-free first-order logic formulas to encode the two requirements of the synthesized new code – that it must be perfectly masked and that it must be functionally equivalent to the original code. The resulting formulas can be decided by an off-the-shelf SMT solver. Based on this formal analysis, we can guarantee that the synthesized program is provably secure against power analysis based side channel attacks even on devices with physical emissions.

In recent years, there is a growing interest in using compilers to automate the application of side-channel countermeasures [1,5,7,23]. However, these existing methods rely on matching known code patterns and applying predefined code transformations. They do not employ SMT solver based exhaustive search or the notion of *perfect masking* during the process. As a result, they cannot guarantee to find the leakage free new program even if such program exists, or formally prove that the generated code is leakage free. Our new method provides both guarantees. Although inductive synthesis has enjoyed remarkable success recently (e.g., [17,16,20,3,27]), this is the first time that it is applied to mitigating side channel attacks.

We have implemented our new method in a software tool called *SC Masker*, which builds upon the LLVM compiler [11] and the Yices SMT solver [12]. We have conducted experiments on a set of cryptographic software benchmarks, including both AES and MAC-Keccak. Our experiments show that the new method is both effective in eliminating side channel leaks and scalable for handling cryptographic software code of practical size.

To sum up, we have made the following contributions:

- We propose a new method for synthesizing *masking* countermeasures to protect cryptographic software code against power analysis attacks.
- We implement the method in a software tool, which takes an unprotected C program as input and returns a perfectly masked new program as output.
- We conduct experiments on a set of cryptographic software benchmarks to demonstrate the effectiveness and scalability of the new method.

The remainder of this paper is organized as follows. We will establish notation and define the synthesis problem in Section 2. We will illustrate the overall flow of our method using an example in Section 3. The detailed algorithms will be presented in Section 4, which include both inductively computing the candidate program and formally verifying the candidate program. We will present a partitioned synthesis procedure in Section 5 to further improve the runtime performance. Our experimental results will be presented in Section 6. Finally, we will give our conclusions in Section 7.

2 Preliminaries

Following the notation used by Blömer *et al.* [9], we assume that a sensitive computation $c \leftarrow f(x, k)$ takes a plaintext x and a secret key k as input and returns a ciphertext c as output. The implementation of function $f(x, k)$ consists of a sequence of intermediate operations. Each intermediate operation is referred to as a function $I_i(x, k)$, where i is the index of that operation.

Side Channel Attacks. We assume that the plaintext x and the ciphertext c may be observed by an adversary, whereas the secret key k is hidden in the computing device. The goal of the adversary is to deduce k based on observing x , c , and the power leakage of the device. Based on the widely used Hamming Weight (HW) model, we assume that the power leakage of the device correlates to the values involved in the sensitive operations $I_1(x, k) \dots I_n(x, k)$. Here, $I_i(x, k)$ refers to the i -th instruction whose result is a function of both x and k . Given two different key values k and k' , for instance, the power consumption of $k \oplus x$ and $k' \oplus x$ may differ. The information leak may be exploited by techniques such as differential power analysis (DPA [19]).

To eliminate side channel leaks, a countermeasure called *masking* can be implemented to randomize the instantaneous power consumption to make it statistically independent from the secret data. For example, when the computation $f(z)$ is a linear function of variable z in the $\oplus$ domain, meaning that $f(z_1 \oplus z_2) = f(z_1) \oplus f(z_2)$, masking requires no modification of the original implementation of function $f(z)$.

$$f(z \oplus r) \oplus f(r) = f(z) \oplus f(r) \oplus f(r) = f(z).$$

Here, the random bit r is generated internally on the cryptographic device so the adversary cannot access its value. Due to commutativity of the XOR operation, we can mask z with r before the computation on the device and de-mask with $f(r)$ afterward.

However, when $f(z)$ is a non-linear function, the implementation of $f(z)$ often needs to be completely redesigned. Depending on the order of attacks to be mitigated, for instance, z may have to be divided into n chunks by using XOR operations with n random bits $r_1 \dots r_n$. Then, each chunk is fed to a newly designed cryptographic function $f'_i(z \oplus r_i, r_i)$, where $1 \leq i \leq n$. At the end, these results are combined to reconstruct $f(z)$ by using XOR operations with another function $f''_i(z \oplus r_i, r_i)$. Consider $n=1$ as an example. We require the new functions $f'()$ and $f''()$ to satisfy the following constraint:

$$f'(z \oplus r, r) \oplus f''(z \oplus r, r) = f(z).$$

However, the design of such cryptographic functions f' and f'' is a highly creative manual process currently undertaken by experts – it is labor intensive and error prone. Furthermore, even if the masking algorithm is provably secure, bugs introduced during the software coding process may still cause information leaks.

Iterative Inductive Synthesis. To overcome these problems, we propose using inductive synthesis to generate implementations of perfect masking countermeasures. We follow the iterative synthesis procedure shown in Fig. 1, which consists of three steps:

1. Given an unprotected program as input, we first compute a candidate new program that is masked and is functionally equivalent to the original program, at least for a small set of test inputs.

2. We try to prove that the candidate program is perfectly masked and is functionally equivalent to the original program under all possible test inputs.
3. If the verification succeeds, we are done. Otherwise, the candidate program is invalid. In the latter case, we block this solution, go back to Step 1, and try again.

The reason why we choose not to generate, in one step, a candidate program that is valid for all possible test inputs is because of performance concerns. A candidate program valid for all possible test inputs would be prohibitively more expensive for an SMT solver to compute. By separating the synthesis task into three subtasks, namely the inductive synthesis of candidate programs, the formal verification of candidate programs, and the iterative refinement step, we can make all three substeps practically feasible to complete.

Fig. 1. The iterative inductive synthesis procedure

In this work, the verification step will consist of two substeps. First, we prove that the candidate program is functionally equivalent to the original program under all possible inputs. Second, we prove that all intermediate computations in the candidate program are perfectly masked. Toward this end, we leverage a verification procedure that we developed recently, called *SC Sniffer* [14,15], which can check whether an intermediate computation result of the program is statistically dependent on the secret data.

Verifying Perfect Masking. Given a pair (x, k) of plaintext and key for the function $f(x, k)$ and an intermediate computation result $I_i(x, k, r)$ masked by the random variable r , we use $D_{x,k}(R)$ to denote the distribution of $I_i(x, k, r)$. Here, r is an s -bit random variable uniformly distributed in the domain $R = \{0, 1\}^s$; it is meant to be used to remove the information leakage of $I_i(x, k, r)$ while maintaining the input-output relation of function $f(x, k)$. If $D_{x,k}(R)$ is statistically independent from k , we say that the function is perfectly masked [9]. Otherwise, the function has side channel leaks.

Definition 1. Given an implementation of function $f(x, k)$ and a set of its intermediate results $\{I_i(x, k, r)\}$, we say that the function is perfectly masked if for each $I_i(x, k, r)$,

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

As an example, consider Fig. 2 where ciphertexts c_1, c_2, c_3, c_4 are results of four different masking schemes for plaintext bit x and key bit k using random bits r_1 and r_2 . According to the truth tables on the right-hand side, all of these four outputs are logically dependent on r_1, r_2 . However, this does not imply statistical independence from the secret k . Indeed, c_1, c_2, c_3 all leak sensitive information. Specifically, when x is logical 0, and when c_1 is 1, we know for sure that the secret k is also 1, regardless of the values of the random variables. Similarly, when c_2 is logical 0, we know for sure that k is also 0. When c_3 is logical 1 (or 0), there is a 75% chance that k is logical 1 (or 0). In contrast, c_4 is the only leak-free output because it is statistically independent of k – when k is logical 1 (or 0), there is 50% chance that c_4 is logical 1 (or 0).

Our method in *SC Sniffer* [14,15] relies on translating the verification problem into a set of satisfiability (SAT) problems, each of which is encoded as a logical formula.

	x	k	$r1$	$r2$	$c1$	$c2$	$c3$	$c4$		x	k	$r1$	$r2$	$c1$	$c2$	$c3$	$c4$
$c1 = x \oplus k \wedge (r1 \wedge r2)$	0	0	0	0	0	0	0	0		1	0	0	0	0	1	1	1
	0	0	0	1	0	0	0	1		1	0	0	1	0	1	1	0
$c2 = x \oplus k \vee (r1 \wedge r2)$	0	0	1	0	0	0	0	1		1	0	1	0	0	1	1	0
	0	0	1	1	0	1	1	0		1	0	1	1	1	1	0	1
$c3 = x \oplus k \oplus (r1 \wedge r2)$	0	1	0	0	0	1	1	1		1	1	0	0	0	0	0	0
	0	1	0	1	0	1	1	0		1	1	0	1	0	0	0	1
$c4 = x \oplus k \oplus (r1 \oplus r2)$	0	1	1	0	0	1	1	0		1	1	1	0	0	0	0	1
	0	1	1	1	1	1	0	1		1	1	1	1	0	1	1	0

Fig. 2. The values of $c1, c2, c3$ are statistically dependent on the key bit k although they are masked by random bits $r1$ and $r2$ – knowing the value of these ciphertexts and plaintext x , an adversary can deduce the value of k with high probability. In contrast, $c4$ is perfectly masked.

These formulas can be decided using an off-the-shelf SMT solver. More specifically, we start by marking all the plaintext bits in x as public, the key bits in k as secret, and the mask bits in r as random. Then, we traverse the entire program and for each intermediate computation $I(x, k, r)$, check the satisfiability of the following formula:

$$\exists x, k, k' . \left(\sum_{r \in R} I(x, k, r) \neq \sum_{r \in R} I(x, k', r) \right)$$

Here, k and k' are two different values of the secret key and R is the set of values of random variable r . The summation $\sum_{r \in R} I(x, k, r)$ represents the number of values of r that can make $I(x, k, r)$ evaluate to logical 1, and the summation $\sum_{r \in R} I(x, k', r)$ represents the number of values of r that can make $I(x, k', r)$ evaluate to logical 1. Assume that random variable r is uniformly distributed in the domain R , the above two summations represent the probabilities of I being logical 1 under key values k and k' , respectively. If the above formula is satisfiable, then we have found a plaintext x and two values (k, k') such that the distributions of $I(x, k, r)$ and $I(x, k', r)$ differs – it means that the value of the secret key bit is leaked. In contrast, if the formula is unsatisfiable, it is a formal proof that $I(x, k, r)$ is perfectly masked by r . We will present the detailed SMT encoding in Section 4.2.

3 Motivating Example

In this section, we illustrate the overall flow of our synthesis method using an example. Our example is part of the implementation of MAC-Keccak, the newly standardized SHA-3 cryptographic hashing algorithm [24], after three rounds of competitions by cryptographic experts worldwide. The MAC-Keccak code [8] consists of five main functions that are repeated for 24 rounds on the input bits (plaintext and key) in order to compute the output (ciphertext). The computation in a single round can be represented by $out = \iota.\chi.\pi.\rho.\theta(in)$, where $\iota()$, $\pi()$, $\rho()$ and $\theta()$ are linear functions in the domain of $\oplus$, consisting of operations such as XOR, SHIFT and ROTATE, whereas $\chi()$ is a nonlinear function, containing nonlinear operations such as AND.

Our synthesis procedure takes the MAC-Keccak code as input and returns a perfectly masked version of the code as output. It starts by transforming the original program into an intermediate representation (IR) using the LLVM compiler front-end. Since we focus on cryptographic software, not general purpose software, we can assume that all

```

1: Chi(bool i1, bool i2, bool i3) {
2:   bool n1, n2, n3;
3:   n3 = ¬i2;
4:   n2 = n3 ∧ i3;
5:   n1 = n2 ⊕ i1;
6:   return n1;
7: }

```

i1	i2	i3	n3	n2	n1
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	0	1
1	1	0	0	0	1
1	1	1	0	0	1

```

1: mChi(bool i1, bool i2, bool i3) {
2:   bool r1, r2, r3; //random bits added
3:   bool b1, b2, b3, n1, n2, n3, n4, n5, n6, n7, n8, n9;
4:   b1 = i1 ⊕ r1;
5:   b2 = i2 ⊕ r2;
6:   b3 = i3 ⊕ r3;
7:   n9 = b3 ∧ r2;
8:   n8 = r3 ∧ r2;
9:   n7 = r3 ∨ b2;
10:  n6 = r1 ⊕ n9;
11:  n5 = n7 ⊕ n8;
12:  n4 = b2 ∨ b3;
13:  n3 = n5 ⊕ n6;
14:  n2 = n4 ⊕ b1;
15:  n1 = n2 ⊕ n3;
16:  return n1;
17: }

```

Fig. 3. The original χ function in MAC-Keccak, its truth table, and the synthesized χ function. Here $\neg$ denotes NOT, $\wedge$ denotes AND, $\vee$ denotes OR, and $\oplus$ denotes XOR.

program variables are bounded integers and there is no input-dependent control flow. (Cryptographic software typically do not have input-dependent control flow because it is vulnerable to timing attacks.) Therefore, it is relatively straightforward to transform the input program into a Boolean program, e.g., by merging if-else conditions, unwinding loops, inlining functions, and bit-blasting the integer operations. Thus, from now on, we are only concerned with an IR where all instructions operate on bits. Focusing on the bit-level analysis allows us to detect leaks at the finest granularity possible.

The next step is traversing the abstract syntax tree of the Boolean program in a topological order, starting at the input nodes and ending at the output node. For each internal node, we first check whether its function is linear in the domain of $\oplus$. As we have shown earlier, for a linear function $f(z)$, we can mask the input z with an XOR of a random bit r before the computation and demask with an XOR of $f(r)$ afterward. Furthermore, to make sure that all intermediate nodes stay masked, we need to chain the mask-demask segments together, by masking the output of a function with a new random variable before demasking it with the previous random variable.

For nonlinear functions, such as $\chi()$, there are no easy ways of generating the countermeasures. In this work, we rely on the iterative inductive synthesis and SMT solvers to search for a valid countermeasure in a bounded design space. Given the $\chi()$ function in Fig. 3 (left), our method will produce the new code in Fig. 3 (right). Our method ensures that these two versions have the same input-output relation, and at the same time, all the intermediate computation results in the new program are perfectly masked with random bits $r1$, $r2$ and $r3$. Our method has two main advantages over the state of the art. First, it is more economical and sustainable than the manual mitigation approach, especially when considering the rapid increase in the application size and platform variety. Second, it eliminates both the design errors and the implementation errors while guaranteeing that the synthesized program is secure by construction. That is, by assuming that each of $r1$, $r2$ and $r3$ in Fig. 3 (right) is randomly distributed, our method guarantees that the probability of each intermediate result being logical 1 (or 0) is statistically independent from the values of $i1$, $i2$ and $i3$.

4 Synthesis of Masking Countermeasures

In this section, we present our basic algorithm for iteratively synthesizing a masked version of the input Boolean program. We leave performance optimizations to the next section. The pseudocode is shown in Algorithm 1, where P is the original program, $inputs$ is the set of inputs, and $output$ is the output. The input variables also have annotations indicating whether they are plaintext bits, key bits, or random bits. The synthesis procedure returns a new program $newP$ whose input-output relation is equivalent to that of P . At the same time, all internal nodes of $newP$ are perfectly masked. New random bits may be added by the synthesis procedure gradually, on a *need-to* basis.

Algorithm 1. Inductive synthesis of a masked version of the input program P .

```

1. SYNTHESIZEMASKING ( $P, inputs, output$ ) {
2.   blocked  $\leftarrow \{ \}$ ;
3.   testSet  $\leftarrow \{ \}$ ;
4.   size  $\leftarrow 1$ ;
5.   while ( $size < MAX\_CODE\_SIZE$ ) {
6.     newP  $\leftarrow$  COMPUTECANDIDATE( $P, inputs, output, size, blocked, testSet$ );
7.     if ( $newP$  does not exist)
8.       size  $\leftarrow size + 1$ ;
9.     else {
10.      test1  $\leftarrow$  CHECKEQUIVALENCE( $newP, P$ );
11.      test2  $\leftarrow$  CHECKINFOLEAKAGE( $newP$ );
12.      if ( $\{test1, test2\} == \{ \}$ )
13.        return newP;
14.      blocked  $\leftarrow$  blocked  $\cup \{newP\}$ ;
15.      testSet  $\leftarrow$  testSet  $\cup \{test1, test2\}$ ;
16.    }
17.  }
18.  return no_solution;
19. }
```

The synthesis procedure iterates through three elementary steps: (1) compute a candidate program $newP$ which is functionally equivalent to the original program P , at least for a selected set of test inputs; (2) verify that $newP$ is functionally equivalent to P for all possible inputs and is perfectly masked; (3) if any of the two verification substeps fails, we block this solution, add the failure triggering inputs to $testSet$, and repeat. The synthesis procedure iteratively searches for a new candidate program with increasing code size, until the size reaches MAX_CODE_SIZE . We record the bad solutions in the set $blocked$ to avoid repeating them in the future. We record in $testSet$ all the test cases that led to failures at some previous verification steps.

In the remainder of this section, we present the detailed algorithms for two elementary steps: computing the candidate program and verifying the candidate program.

4.1 Computing the Candidate Program

The first step in computing $newP$ from P is to create a parameterized AST that captures all possible masked Boolean programs up to a bounded size. Following the notation

used in [13], we call this parameterized AST as a *skeleton*. An example is shown in Fig. 4, which has 11 nodes. Each node is either an *Op* node or a *V* node. The internal node *Op* can be instantiated to any bit-level operation such as $\oplus$, $\&$, $|$, or $!$. The *V* node can be instantiated to any variable in the original program, or fresh random bit added by the synthesis procedure, or constant (logical 0 or 1). The instantiation of *Op* nodes and *V* nodes is controlled by a set of auxiliary variables, whose values will be assigned by the SMT solver.

Fig. 4. A candidate program skeleton consisting of 11 parameterized AST nodes

Fig. 5. The synthesized candidate program with instantiated Boolean masking

As an example, consider node n_8 in Fig. 4. The corresponding logical constraint may be $((N8 == V1) \&\&bV1) \mid \mid (N8 == V2) \&\&bV2$, where $N8$ denotes the output of n_8 and $V1$ and $V2$ are two variables in the input program. Auxiliary variables $bV1$ and $bV2$ are added to decide which of the node types are chosen – we would add another constraint saying that one and only one of $bV1$ and $bV2$ must be true. Based on which variable is set to true by the SMT solver, the output of node n_8 is determined. For node n_1 , the constraint may be $((N1 == (N2 \& N3)) \&\&bAND1) \mid \mid ((N1 == (N2 \mid N3)) \&\&bOR1) \mid \mid ((N1 == (N2 \oplus N3)) \&\&bXOR1) \mid \mid ((N1 == (\oplus N2)) \&\&bNOT1)$ where $N1$, $N2$ and $N3$ denote the output of node n_1 , n_2 , and n_3 , respectively. Auxiliary variables $bAND1$, $bOR1$, $bXOR1$, and $bNOT1$ are constrained such that one and only one of them must be true. Fig. 5 shows a masked candidate program synthesized by the SMT solver, which represents $n_1 = i_1 \oplus i_2$.

The next step is to build an SMT formula Φ that imposes two additional requirements: (1) the input-output relation of the candidate program *skeleton* is equivalent to the original program P , and (2) the internal nodes of the candidate program are all masked by some random variables. More formally, the formula Φ is defined as follows:

$$\Phi = \Phi_P \wedge \Phi_{skel} \wedge \Phi_{iEqv} \wedge \Phi_{oEqv} \wedge \Phi_{masked} \wedge \Phi_{testSet} \wedge \Phi_{blocked},$$

where the subformulas are defined as follows:

- Φ_P encodes the program logic of P .
- Φ_{skel} encodes the program logic of the *skeleton*.
- Φ_{iEqv} asserts that the input variables of P and *skeleton* have the same values.
- Φ_{oEqv} asserts that the outputs of P and *skeleton* have the same value.
- Φ_{masked} asserts that all internal nodes are masked by some random bits – some random bit must appear in the support of the function of each node.

- $\Phi_{testSet}$ asserts that the input variables should take values only from $testSet$.
- $\Phi_{blocked}$ asserts that the previously failed solutions should not be selected.

If formula Φ is satisfiable, a candidate solution is found, and it will be verified for equivalence and perfect masking in the following step. Otherwise, the skeleton *size* will be incremented and the SMT solver will be invoked again on the new formula.

4.2 Verifying the Candidate Program

Given a candidate program $newP$, which is computed by the SMT solver for a set of selected test inputs, we verify that it is a valid solution for all possible inputs. We formulate the verification problem into two satisfiability subproblems, where we look for counterexamples, or test inputs, under which either $newP$ is not equivalent to P , or some nodes in $newP$ are not perfectly masked.

Checking Functional Equivalence. We construct formula Ψ_1 such that it is satisfiable if and only if there exists a test input under which $newP$ and P have different outputs. The formula is defined as follows:

$$\Psi_1 = \Phi_P \wedge \Phi_{newP} \wedge \Phi_{iEqv} \wedge \Phi_{oDiff},$$

where Φ_P and Φ_{newP} encode the input-output relations of the two programs, Φ_{iEqv} asserts that they have the same input values, and Φ_{oDiff} asserts that they have different outputs. If Ψ_1 is satisfiable, we find a test case showing that $newP$ is a bad solution. If Ψ_1 is unsatisfiable, then $newP$ and P are functional equivalent.

Checking for Information Leakage. We construct formula Ψ_2 such that it is satisfiable if and only if there exists an intermediate node in $newP$ that leaks sensitive information. Toward this end, we leverage our recently developed verification procedure [14,15] to check, for each intermediate node $I(x, k, r)$, whether there exist a plaintext x and two key values k, k' such that $\sum_{r \in R} I(x, k, r) \neq \sum_{r \in R} I(x, k', r)$. As we have explained in Section 2, this inequality means that the probabilistic distributions of $I(x, k, r)$ and $I(x, k', r)$ differ for the two key values k and k' . The formula Ψ_2 is defined as follows:

$$\Psi_2 := \left(\bigwedge_{r \in R} \Phi_{I(x, k, r)} \right) \wedge \left(\bigwedge_{r \in R} \Phi_{I(x, k', r)} \right) \wedge \Phi_{b2i} \wedge \Phi_{sum} \wedge \Phi_{sumDiff},$$

where the subformulas are defined as follows:

- *Program logic* ($\Phi_{I(x, k, r)}$): Each subformula $\Phi_{I(x, k, r)}$ encodes the input-output relation of $I(x, k, r)$ with a fixed value $r \in R$ and variable k . Each subformula $\Phi_{I(x, k', r)}$ encodes the input-output relation of $I(x, k', r)$ with a fixed value $r \in R$ and variable k' . All subformulas share the same plaintext variable x .
- *Boolean-to-int* (Φ_{b2i}): This subformula encodes the conversion of the bit output of each $I(x, k, r)$ to an integer (true becomes 1 and false becomes 0), which will be summed up later to compute $\sum_{r \in R} I(x, k, r)$ and $\sum_{r \in R} I(x, k', r)$.
- *Sum-up-the-1s* (Φ_{sum}): This subformula encodes the two summations of the logical 1's in the outputs of the $|R|$ copies of $I(x, k, r)$ and the $|R|$ copies of $I(x, k', r)$.
- *Different sums* ($\Phi_{sumDiff}$): It asserts that the two summations have different results.

Fig. 6. SMT encoding for checking the statistical dependence of an output on secret data (k_1, k_2)

If Ψ_2 is unsatisfiable, then the intermediate result I is perfectly masked. If Ψ_2 is satisfiable, then I has information leakage.

Fig. 6 provides a pictorial illustration of our SMT encoding for an intermediate result $I(k_1, k_2, r_1, r_2)$, where k_1, k_2 are the key bits and r_1, r_2 are the random bits. The first four boxes encode the program logic of $\Phi_{I(x, k, 0)} \dots \Phi_{I(x, k, 3)}$ for key bits $(k_1 k_2)$, with the random bits set to 00, 01, 10, and 11, respectively. The other four boxes encode the program logic of $\Phi_{I(x, k', 0)} \dots \Phi_{I(x, k', 3)}$ for key bits $(k'_1 k'_2)$, with the random bits set to 00, 01, 10, and 11, respectively. The entire formula checks whether there exist two sets of key values $(k_1 k_2$ and $k'_1 k'_2)$ under which the probabilities of I being logical 1 are different.

As a more concrete example, consider the computation $c_2 = x \oplus k \vee (r_1 \wedge r_2)$ in Fig. 2. The SMT solver may return the solution $x=0, k=0$ and $k'=1$ because, according to the truth table in Fig. 2, $\sum_{r \in R} c_2(0, 0, r) = 1$ whereas $\sum_{r \in R} c_2(0, 1, r) = 4$. Consider $c_4 = x \oplus k \oplus (r_1 \oplus r_2)$ in Fig. 2 as another example. The SMT solver will not be able to find any solution because it is perfectly masked. For instance, when $x=0, k=0$ and $k'=1$, we have $\sum_{r \in R} c_4(0, 0, r) = 2$ and $\sum_{r \in R} c_4(0, 1, r) = 2$.

5 Partitioned Synthesis to Improve Performance

SMT solver based inductive synthesis has the advantage of being exhaustive during the search of countermeasures within a bounded design space. With the help of the verification subprocedure, our method also guarantees that the resulting program is secure by construction. However, its main disadvantage is the limited scalability, since the SMT solver slows down quickly as the program size increases. Although we expect SMT solvers to continue improving in the coming years, it is unlikely that a monolithic SMT based synthesis procedure will scale up to large programs (this is consistent with what others in the field have observed [3,2]). In this section, we propose a *partitioned* synthesis procedure to combine static code analysis with judicious use of inductive synthesis so that the combined method can handle cryptographic software code of realistic size.

The partitioned synthesis procedure (Fig. 7) starts by traversing the AST nodes of the program in a topological order, from the inputs to the output. Depending on whether the

Fig. 7. The partitioned synthesis procedure for applying masking countermeasures locally

AST node n is linear or nonlinear as shown in Algorithm 2, it invokes either MASKLINEAR or SYNTHESIZEMASKING (presented in the previous section). When n is a linear function, we mask its input variables and demask the output with random variables, without modifying the linear function itself, as explained in Section 2. When n is a nonlinear function, instead of invoking SYNTHESIZEMASKING for the entire fan-in cone of n , we partition it into small code regions, and synthesize a masked version for each region. Then, we substitute the original code region reg in program P with the new code region new_reg . The entire synthesis procedure terminates when all small code regions of all nonlinear AST nodes in program P are perfectly masked.

Algorithm 2. Partitioned synthesis algorithm for masking the program P .

```

1. PARTITIONEDSYNTHESIS ( $P$ ,  $inputs$ ,  $output$ ) {
2.   for each (AST node  $n \in P$ ) {
3.     if ( $n$  represents a linear function)
4.        $new\_n \leftarrow \text{MASKLINEAR}(P, inputs, n)$ ;
5.       replace  $n$  in program  $P$  with  $new\_n$ ;
6.     else {
7.       while ( $\exists$  unprotected code region  $reg \in FanIn(n)$ ) {
8.         Let  $(reg\_ins, reg\_out)$  be the inputs and output of  $reg$ ;
9.          $new\_reg \leftarrow \text{SYNTHESIZEMASKING}(P, reg\_ins, reg\_out)$ ;
10.        replace  $reg$  in program  $P$  with  $new\_reg$ ;
11.      }
12.    }
13.  }
14.  return  $P$ ;
15. }
```

Selecting a Code Region. While selecting a code region in $FanIn(n)$ of a nonlinear node n , we first start from an AST node $m \in FanIn(n)$ that is not yet perfectly masked, and then include a number of its connected unprotected nodes. The exact number of fan-in nodes to be included in the code region of node m is controlled by a user specified bound. Choosing the right bound, and hence the size of the code region, is a tradeoff between the compactness of the synthesized program and the computational overhead.

On the one hand, if we set the bound to positive infinity, the *partitioned* synthesis procedure would degenerate to the *monolithic* approach. On the other hand, if we set the bound to a small number, the synthesized solution may be suboptimal in that some of the masking operations are unnecessary.

For illustration purposes only, we consider an extreme case where the region size is set to 1, meaning that each nonlinear AST node is masked separately. Under this assumption, in Fig. 3, we illustrate the process of masking the $\chi()$ function from Fig. 8. The first code region involves the NOT operation at Line 3, whose masked version is shown in the middle column. The second code region involves the AND operation at Line 4, whose masked version is shown in the middle column. The third code region involves the XOR of $n2$ and $i1$ at Line 5, whose masked version is shown in the middle column. It is worth pointing out that, in this extreme case, the resulting program will be suboptimal. However, the actual implementation of our partitioned synthesis procedure was able to obtain a perfectly masked countermeasure whose size is more compact.

L3: $n3 = \neg i2;$	→	$b1 = i2 \oplus r1;$ $t1 = \neg b1;$ $n3 = t1 \oplus r1;$	→	$b1 = i2 \oplus r1;$ $t1 = \neg b1;$ $n3 = t1 \oplus r2; //\text{swap with } r1$
L4: $n2 = n3 \wedge i3;$	→	$b3 = i3 \oplus r3;$ $b2 = n3 \oplus r2;$ $t10 = \neg b2;$ $t9 = b3 \wedge r2;$ $t8 = \neg r3;$ $t7 = t10 \wedge r3;$ $t6 = b2 \wedge b3;$ $t5 = \neg t9;$ $t4 = t8 \vee r2;$ $t3 = t6 \vee t7;$ $t2 = t4 \oplus t5;$ $n2 = t2 \oplus t3;$	→	$b3 = i3 \oplus r3;$ $b2 = n3 \oplus r1; //\text{swap with } r2$ $t10 = \neg b2;$ $t9 = b3 \wedge r2;$ $t8 = \neg r3;$ $t7 = t10 \wedge r3;$ $t6 = b2 \wedge b3;$ $t5 = \neg t9;$ $t4 = t8 \vee r2;$ $t3 = t6 \vee t7;$ $t2 = t4 \oplus t5;$ $n2 = t2 \oplus r4; //\text{swap with } t3$
L5: $n1 = n2 \oplus i1;$	→	$b4 = n2 \oplus r4;$ $b5 = i1 \oplus r1;$ $t12 = b4 \oplus b5;$ $t11 = r1 \oplus r4;$ $n1 = t11 \oplus t12;$	→	$b4 = n2 \oplus t3; //\text{swap with } r4$ $b5 = i1 \oplus r1;$ $t12 = b4 \oplus b5;$ $t11 = r1 \oplus r4;$ $n1 = t11 \oplus t12;$

Fig. 8. Example: the process of masking individual code regions and composing them together

Replacing the Code Region. Continue with the above *extreme case* exercise, we now explain how to use the newly synthesized code region (*new_reg*) to replace the original code region (*reg*) in program *P*. The replacement process is mostly straightforward, due to the fact that our partitioned synthesis procedure traverses regions in a bottom-up topological order. However, there is one caveat – before demasking the output of the new region *new_reg*, we need to mask it with another random variable; otherwise, the output of *new_reg* would become unmasked.

We solve this problem by asserting, while computing the candidate program in procedure SYNTHESIZEMASKING, that the output and all inputs must be an XOR operation with some random variables. Due to the associativity of XOR operations, and the fact that now two adjacent code regions are connected through two XOR operations, we can switch the order of the two XOR operations during region replacement, without modifying the functionality of the final output.

In Fig. 8, the right-hand-side column shows an example for chaining the three new code regions of the χ function obtained in the middle column, by swapping their adjacent XOR operations with random bits.

Reusing Random Variables. To further reduce the size of the synthesized program, we reuse random variables as much as possible while masking the non-adjacent code regions. Specifically, while building the candidate program *skeleton* for a code region *reg* (see Section 4.1), we first need to create a list of random variables to be used in the *V* nodes. The number of random variables is at most as large as the number of input variables in *reg*. However, we do not have to create fresh random variables every time they are needed. Instead, we can reuse existing random variables in the program, as long as they are not used in the code regions adjacent to *reg*. This optimization can significantly reduce the number of random bits required in the masked new program, while at the same time soundly maintaining the statistical independence of the masked nodes.

6 Experiments

We have implemented our new synthesis method in a software tool called *SC Masker*, which builds upon the LLVM compiler front-end and the Yices SMT solver. Our tool runs in two modes: the monolithic mode and the partitioned mode, to facilitate experimental comparison of the two approaches. We have evaluated our method on a set of cryptographic software benchmarks. Our experimental evaluation was designed to answer the following questions:

- How effective is the new synthesis method in eliminating side channel leaks? Is the synthesized program as compact as the countermeasures handcrafted by experts?
- How scalable is the tool in handling code of realistic size? Our partitioned synthesis procedure is designed to address the scalability problem. Is it effective in practice?

Our benchmarks fall into three categories. The first set, from P1 to P8, are medium sized cryptographic functions that are partially masked. Specifically, P1 and P2 are taken from Bayrak *et al.* [6], which are incorrectly masked computations due to code motion in compiler optimization. P3 and P4 are from Herbst *et al.* [18], which are gate-level implementations of partially masked AES. P5 and P6 are masked versions of the χ function from Bertoni *et al.* [8], after integer to Boolean compilation with optimizations. P7 and P8 are two modified versions of the MAC-Keccak nonlinear χ functions. The second set, from P9 to P12, are small to medium sized cryptographic functions that are completely unmasked. Specifically, P9 is the original MAC-Keccak χ function taken from the reference implementation [8] (Equation 5.2 on Page 46). P10 and P11 are two nonlinear functions, *mul4* and *inv4*, from an implementation of AES in [10]. P12 is a single-round complete implementation of AES found in [10]. The third set, from P13 to P18, are partially masked large programs with a significant number of instructions not yet masked. These programs are generated by us from the MAC-Keccak reference code [24] after converting it from an integer program to a Boolean program. In each case, the whole program has been transformed into a single function to test the scalability of our new methods.

Table 1 shows the experimental results obtained on a machine with a 3.4 GHz Intel i7-2600 CPU, 4 GB RAM, and a 32-bit Linux OS. Columns 1-6 show the statistics

Table 1. Comparing performance of the monolithic and partitioned methods in *SC Masker*

Program						Monolithic			Partitioned		
Name	LoC	Keys	Plains	Rands	Nodes	Rands	Nodes	Time	Rands	Nodes	Time
P1	79	16	16	16	47	16	85	2.9s	16	85	2.9s
P2	67	8	8	16	31	16	55	1.5s	16	55	1.5s
P3	32	2	2	2	9	4	15	8.3s	4	15	8.1s
P4	32	2	2	2	6	6	9	0.2s	6	9	0.2s
P5	59	3	3	4	18	8	24	19m17s	8	27	8.3s
P6	60	3	3	4	18	8	24	0.5s	8	24	0.5s
P7	66	3	3	4	22	8	25	0.3s	8	25	0.3s
P8	66	3	3	4	22	8	25	0.3s	8	25	0.3s
P9	9	3	0	0	3	-	-	TO	4	14	3.1s
P10	57	8	0	0	37	-	-	TO	8	264	4m36s
P11	82	8	0	0	48	-	-	TO	4	485	13m10s
P12	365	8	0	0	182	-	-	TO	8	1072	22m10s
P13	56k	58	161	58	19k	-	-	TO	58	20k	24m7s
P14	56k	58	161	58	19k	-	-	TO	58	21k	41m37s
P15	56k	58	161	58	19k	-	-	TO	58	21k	36m21s
P16	56k	58	161	58	19k	-	-	TO	58	21k	35m42s
P17	56k	58	161	58	19k	-	-	TO	58	21k	48m15s
P18	56k	58	161	58	19k	-	-	TO	58	20k	23m41s

of each benchmark, including the name, the lines of code, the number of key bits, the number of plaintext bits, the number of random bits, and the number of operations (Nodes). Columns 7-9 show the results of the monolithic synthesis algorithm, including the number of random bits and the number of operations (Nodes) in the synthesized program, as well as the run time. Columns 10-12 show the results of the partitioned synthesis algorithm, including the number of random bits and the number of operations (Nodes) in the synthesized program, as well as the run time. Here, TO means that the *SC Masker* tool ran out of the time limit of 4 hours.

The experimental results show that our new synthesis method, especially when it runs in the partitioned mode, is scalable in handling cryptographic software of realistic size. On the first set of test cases, where the programs are small, both monolithic and partitioned procedures can complete quickly, and the differences in run time and compactness of the new program are small. However, on large programs such as AES and MAC Keccak, the monolithic method can not finish within four hours, whereas the partitioned method can finish in a reasonably small amount of time. Furthermore, we can see that most of the existing random bits in the original programs were reused.

As far as the compactness of the new program is concerned, we know of only one benchmark (P9) that has a previously published masking countermeasure. The countermeasure [8] handcrafted by cryptographic engineering experts has 14 operations. The countermeasure synthesized by our own tool (using the partitioned approach) also has 14 operations. Therefore, at least for this example, it is as compact as the handcrafted countermeasure. However, recall that our method has the additional advantages of being fully automated and at the same time guaranteeing that the synthesized new program is provably secure. Furthermore, when given more CPU time – for example, by setting the time limit to 10 hours and using a larger region size – our synthesis procedure in *SC Masker* was able to produce a countermeasure with only 12 operations, which is more compact than the countermeasure handcrafted by experts. We can also show that this is the smallest possible solution because reducing the skeleton size further causes the SMT solver to report unsatisfiability.

As another measurement of the scalability of our new methods, we conducted experiments on a parameterized version of test program P1 by expanding it from 1 encryption round up to 10 rounds. In each program, the input for one round is the output from the previous round. We ran the *SC Masker* tool twice, once with the monolithic approach and once with the partitioned approach. The results are plotted in Fig. 9, where the x -axis shows the program size in terms of the number of encryption rounds and the y -axis shows the run time in seconds. Also note that the y -axis is in logarithmic scale. Whereas the monolithic approach quickly ran out of time for programs with ≥ 5 rounds, the execution time increase of the partitioned approach remains modest – this demonstrates the capability of our *partitioned* method in handling large programs.

Fig. 9. Comparing the run time of the two synthesis methods in *SC Masker* (the time is in seconds)

7 Conclusions

We have presented a new synthesis method for automatically generating perfect masking countermeasures for cryptographic software to defend against power analysis based side channel attacks. Our method guarantees that the resulting software code is secure by construction. We have implemented our method in the *SC Masker* tool and evaluated it on a set of cryptographic software benchmarks. Our experiments show that the new method is effective in eliminating side channel leaks and at the same time is scalable for handling programs of practical size. For future work, we plan to continue optimizing our SMT based encoding and at the same time extending it to handle other masking schemes, including additive masking, multiplicative masking, as well as application specific masking such as RSA blinding.

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

References

1. Agosta, G., Barengi, A., Pelosi, G.: A code morphing methodology to automate power analysis countermeasures. In: ACM/IEEE Design Automation Conference, pp. 77–82 (2012)
2. Akiba, T., Imajo, K., Iwami, H., Iwata, Y., Kataoka, T., Takahashi, N., Moskal, M., Swamy, N.: Calibrating research in program synthesis using 72,000 hours of programmer time. Technical report, Microsoft Research (2013)
3. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: International Conference on Formal Methods in Computer-Aided Design, pp. 1–17 (2013)

4. Balasch, J., Gierlichs, B., Verdult, R., Batina, L., Verbauwhede, I.: Power analysis of Atmel CryptoMemory – recovering keys from secure EEPROMs. In: RSA Conference Cryptographers' Track, pp. 19–34 (2012)
5. Bayrak, A., Regazzoni, F., Brisk, P., Standaert, F.-X., Ienne, P.: A first step towards automatic application of power analysis countermeasures. In: ACM/IEEE Design Automation Conference, pp. 230–235 (2011)
6. 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)
7. Bayrak, A., Velickovic, N., Ienne, P., Burleson, W.: An architecture-independent instruction shuffler to protect against side-channel attacks. *ACM Transactions on Architecture and Code Optimization* 8(4), 20 (2012)
8. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V., Keer, R.V.: Keccak implementation overview, <http://keccak.neokeon.org/Keccak-implementation-3.2.pdf>
9. 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)
10. Boyar, J., Peralta, R.: A small depth-16 circuit for the AES S-Box. In: International Workshop on Security, pp. 287–298 (2012)
11. Lattner, C., Adve, V.: The LLVM Instruction Set and Compilation Strategy. Tech. report, CS Dept., Univ. of Illinois at Urbana-Champaign (August 2002)
12. 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)
13. Eldib, H., Wang, C.: An SMT based method for optimizing arithmetic computations in embedded software code. In: International Conference on Formal Methods in Computer-Aided Design (2013)
14. Eldib, H., Wang, C., Schaumont, P.: SMT-based verification of software countermeasures against side-channel attacks. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 62–77. Springer, Heidelberg (2014)
15. Eldib, H., Wang, C., Taha, M., Schaumont, P.: QMS: Evaluating the side-channel resistance of masked software from source code. In: ACM/IEEE Design Automation Conference (2014)
16. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 62–73 (2011)
17. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 281–292 (2008)
18. 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)
19. 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)
20. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Software synthesis procedures. *Commun. ACM* 55(2), 103–111 (2012)
21. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks – Revealing the Secrets of Smart Cards, pp. 1–337. Springer (2007)
22. Moradi, A., Barengi, A., Kasper, T., Paar, C.: On the vulnerability of FPGA bitstream encryption against power analysis attacks – extracting keys from Xilinx Virtex-II FPGAs. *IACR Cryptology* (2011)
23. Moss, A., Oswald, E., Page, D., Tunstall, M.: Compiler assisted masking. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 58–75. Springer, Heidelberg (2012)
24. NIST. Keccak reference implementation code submission to the SHA-3 competition, http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/Keccak_FinalRnd.zip

25. Paar, C., Eisenbarth, T., Kasper, M., Kasper, T., Moradi, A.: Keeloq and side-channel analysis – evolution of an attack. In: International Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 65–69 (2009)
26. 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)
27. Solar-Lezama, A.: Program sketching. International Journal on Software Tools for Technology Transfer 15(5-6), 475–495 (2013)