Sleuth: Automated Verification of Software Power Analysis Countermeasures

Ali Galip Bayrak[1], Francesco Regazzoni[2,3], David Novo[1], and Paolo Ienne[1]

[1] School of Computer and Communication Sciences,
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{aligalip.bayrak,david.novobruna,paolo.ienne}@epfl.ch
[2] TU Delft, Netherlands
[3] ALaRI - University of Lugano, Switzerland
regazzoni@alari.ch

Abstract. Security analysis is a crucial concern in the design of hardware and software systems, yet there is a distinct lack of automated methodologies. In this paper, we remedy this situation for the verification of software countermeasure implementations. In this context, verifying the security of a protected implementation against side-channel attacks corresponds to assessing whether any particular leakage in any particular computational phase is statistically dependent on the secret data and statistically independent of any random information used to protect the implementation. We present a novel methodology to reduce this verification problem into a set of Boolean satisfiability problems, which can be efficiently solved by leveraging recent advances in SAT solving. To show the effectiveness of our methodology, we have implemented an automatic verification tool, named `Sleuth`, as an advanced analysis pass in the back-end of the LLVM compiler. Our results show that one can automatically detect several examples of classic pitfalls in the implementation of countermeasures with reasonable runtimes.

Keywords: Software verification, security, DPA.

1 Introduction

The average person was estimated to consume about 34 gigabytes of data per day in 2008 in the USA (including TV, gaming, movies, Internet, etc.) [9], and this number is growing. Considering the amount of personal data flowing through or processed by the everyday devices, ensuring the security of the information is becoming a crucial requirement within the design process. However, off-the-shelf compilers or *Electronic Design Automation* (EDA) tools still do not consider security as a design objective, and focus primarily on conventional design objectives, such as execution time, code size, area and energy. Recent works [4,7,10,11,25], however, indicate a nascent trend towards automating the application of hardware and software countermeasures to increase the security of the systems against certain side-channel attacks. Although this represents a promising direction, many challenges remain open. This paper targets

G. Bertoni and J.-S. Coron (Eds.): CHES 2013, LNCS 8086, pp. 293–310, 2013.
© International Association for Cryptologic Research 2013

# Unprotected 	
# Vulnerable 	
st = pt **xor** key
	# Masked
	# Safe
	st = key **xor** rnd
	st = st **xor** pt
	...
		# Masked
		# Vulnerable
		st = key **xor** rnd
		st = st **xor** pt
		st = st **xor** rnd

Fig. 1. Sample programs

one such challenge: the automatic verification of protected software implementations against power analysis attacks.

A standard verification process determines whether a given system satisfies certain properties described by the designer. Functional correctness is the most fundamental such property and has been extensively studied by the research community. In this work, we focus on a more specialized yet important property for the security-critical applications: *insensitivity* against power analysis attacks. We define an operation (or a group of operations) as *sensitive* if its associated leakage (e.g., power consumption) depends on secret data (e.g., key) but not on any random data. It is important to note that this definition does not necessarily cover all possible side-channel weaknesses; however, we can effectively use it to verify implementations of extensively studied countermeasures, such as Boolean and arithmetic masking [23] and random precharging [29], and to the best of our knowledge this is the first work in this direction.

As a simple motivating example, the operation in the first implementation in Fig. 1 is vulnerable to power analysis attacks when it is executed on most embedded devices [22]. This is because the device's power consumption during the execution of the operation will depend on the secret key (`key`), which can be recovered using a simple statistical analysis known as *Differential Power Analysis* (DPA) [21]. A well-accepted approach to avoid this vulnerability is to mask the secret variable with random masks so as to randomize the result of the intermediate calculations [23], as shown in the second implementation. The masks are propagated and then removed at the end of the whole implementation before outputting the ciphertext, which is not shown in the example. This method has been proven to be resistant against first-order DPA [8]. However, if the masks are removed too early, a secret intermediate value could be leaked, as shown in the third implementation. The value of `st` after the execution of the third operation is `key xor pt`, as the second masking with `rnd` removes the effect of the first one. Despite the triviality of the example, traditional type-based static information flow analysis would not detect this pitfall; these methods usually make their decisions based only on the types, but not on the associated variables. Hence, such methods will falsely conclude that the last operation has a random output since a variable that is random is involved in the operations. This behavior is an unacceptable over-simplification; therefore, the propagation rules should also consider the variables (e.g., random masks) in addition to types. Moss et al. [25]

used such a type system in their automatic masking method; however, their approach is limited only to certain operations (xor and table look-up) for certain (Boolean) masking schemes.

This simple example is just an illustration of many potential pitfalls in real implementations. Most such pitfalls are much harder to detect manually, e.g., when they appear in later operations of the program, when the program combines different Boolean and arithmetic functions, or when higher-order relations between operations are considered.

In this paper, we propose an approach for security verification that is fundamentally different from simple rule-based property propagation, which is used in many other instances of information flow analysis and other security problems. We convert the particular implementation under analysis into a set of satisfiability problems, which are then used to determine whether an intermediate computation leaks secret data in a deterministic way, making it vulnerable to certain attacks. In a sense, our methodology is agnostic to the protection schemes used; it is able detect pitfalls in the application of a countermeasure without making countermeasure specific considerations. Accordingly, it offers a broad application scope for verification of protected implementations.

2 Definition of the Power Analysis Sensitivity

In this section, we define the four main elements of our verification approach: *program*, *type system*, *leakage model* and *sensitivity*.

2.1 Program

A *straight-line program* is a sequence of branch-free operations. We use three-address form to represent the operations, and *Static Single Assignment* (SSA) form to represent the data dependencies [5].

Definition 1. *A three-address form branch-free operation, or shortly an* operation*, $d \in \mathcal{D}$, is a 4-tuple (op, x, y, z), where op is the* operator*, and x, y, and z are the* operands*. An operand u represents values in $\{0,1\}^{w_u}$, where $w_u \in \mathbb{N}$ is the* bitwidth *of u. An* arithmetic/logic operation *is expressed as $x = y\ op\ z$, while an* array handling operation *is expressed as $x = y[z]$ or $y[z] = x$, where op is* load *or* store*, respectively. A* straight-line program*, or shortly a* program*, $p = (d_0, ..., d_{n-1}) \in \mathcal{P}$, is a sequence of n operations, where $n \in \mathbb{N}, d_i \in \mathcal{D}$ and $0 \leq i < n$.*

The left side of the assignment symbol (=) of an *operation* is known as the *l-value*; similarly, the right side is known as the *r-value*. An *operand* can be a *variable* or, in some cases, a *constant*. The variables of a program are classified as *input variables* and *intermediate variables*; this classification can be extracted unequivocally using standard compiler analysis [5]. For example, <`t = key xor pt; st = t xor rnd`> is a straight-line program which has two xor *operations*, three *input variables* (`key`, `pt` and `rnd`) and two *intermediate variables* (`t` and `st`).

In this work, we target programs that do not have any input-dependent control-flow. We automatically convert them into *straight-line programs* using standard static code transformations, i.e., loop unrolling and function inlining. We restrict our focus to this kind of programs for scalability reasons, since static program analysis complexity grows exponentially with the number of branches (e.g., [1]). Still, many provably-secure (against certain attacks) countermeasures, such as masking, can be implemented without input-dependent control-flows and can greatly benefit from our approach.

2.2 Type System

The use of type systems, a fundamental concept for programming languages and compilers, gives special meanings to sequences of bits. Traditional security analysis techniques (e.g., information flow analysis) use type systems to tag each variable with its level of secrecy; for example, it is a common practice to use two security types to represent each variable as either *public* or *secret*. In this work, we extend this notion and introduce another security type for *random* variables.

Definition 2. *Each input variable v of a program is tagged with a security* type, $t(v) \in T$, *where* $T = \{secret, public, random\}$. *A* secret *variable is one whose content should not be revealed (e.g., key), a* public *variable is one whose content is observable by third-parties (e.g., plaintext), and a* random *variable is one that takes uniformly distributed random values independently generated for each different fresh run of the program and is non-observable by third-parties (e.g., masks used in the application of masking countermeasure).*

We use the introduced type system to characterize the secrecy and randomness of the operations. The types of the input variables must be assigned explicitly by the user; types are automatically identified for the intermediate variables.

2.3 Leakage Model

Leakage is the information observable through the side channels (power consumption, EM radiation, etc.) during the execution of the program. A *leakage model* is a model of leakage imputable to one or more operations of the program. It can be defined to consider each operation independently (a univariate leakage) or a vector of operations together (bivariate, trivariate, and so on).

Definition 3. *A* leakage model, $l \in \mathcal{L}$, *is a function which models the side-channel leakage of a subset of operations* $d' = (d'_0, ..., d'_{m-1})$ *of a program* p *on a given device* $h \in \mathcal{H}$, *where* $m \in \mathbb{N}, d'_i \in \mathcal{D}$ *and* $0 \leq i < m$. *It returns a function* f *that, in turn, returns an estimated leakage value* $r \in \{0,1\}^s$ ($s \in \mathbb{N}$) *for an assignment of input variables of* d'; *hence, the domain of* f *is* $\{0,1\}^q$, *where* $q \in \mathbb{N}$ *represents the aggregate bitwidth of all input variables of* d'.

An example univariate leakage model, which is shown to be effective in practice for power analysis attacks, is *Hamming Weight* (HW) of the r-value of the operation (in this case, f is the HW function, which takes arbitrary length binary input and returns a non-negative integer represented in binary form). Similarly, a common bivariate leakage model is *Hamming Distance* (HD) of the r-values of the two operations. Needless to say, these models do not perfectly represent the leakage behaviors of the devices, but are the most common models used in the literature. Our methodology gives the flexibility to the user to define their leakage model; some sample models are presented in Sections 4 and 5. Note that, l can consider the device, the program and the operators in the formulation of f. Hence, one can define a device- or operator-specific leakage model.

2.4 Sensitivity

We describe a vector of operations, and its associated leakage, as *sensitive*, if the leakage of these operations satisfies two properties: (i) it statistically depends on at least one *secret* input variable and (ii) it is statistically independent of any *random* input variable. In other words, we check whether random inputs do not have any impact on the leakage and whether any secret information is leaked through a side-channel. Note that two variables are *statistically independent* if and only if their mutual information is zero.

Definition 4. *Given a program* $p = (d_0, ..., d_{n-1})$ *that has* k *input variables* $v = \{v_0, ..., v_{k-1}\}$, *the associated security types* $t = \{t_0, ..., t_{k-1}\}$ *of these variables, a device* h, *and a leakage model* l, *then the* sensitivity *of a subset* $d' = (d'_0, ..., d'_{m-1})$ *of operations of* p *is a Boolean value that represents whether the leakage* $l(d', p, h)$ *statistically depends on at least one input variable* v_i *such that* $t(v_i) = secret$, *but not on any input variable* v_j *such that* $t(v_j) = random$, *where* $i, j, k \in \mathbb{N}$ *and* $0 \leq i, j < k$.

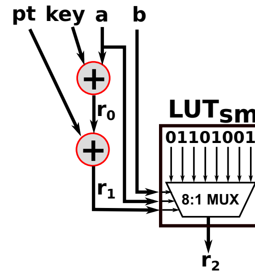
For example, given the program `<t = key xor pt; st = t xor rnd>`, the univariate leakage model "HW of the r-value of the operation", and the types (*secret,public,random*) of inputs (`key`,`pt`,`rnd`), the first operation is sensitive, since it has a leakage (HW(`key xor pt`)) that is statistically independent of `rnd`, and statistically dependent on `key`. The second operation, on the other hand, is *insensitive*.

3 Automatic Detection of the Sensitivity of Operations

In this section, we present how we determine *sensitive* operations (or vector of operations) in a given *program*, based on the definitions given in Section 2. The methodology is composed of two steps: first, we convert our *program* into a special *Data Flow Graph* (DFG) and, second, we analyze this graph to determine *sensitive* operations.

```
1 s[0] = 1;
2 s[1] = 0;
3 for (i=0 ; i<2 ; i++) {
4   t = s[i^a] ^ b;
5   sm[i] = t;
6 }
7 r0 = key ^ a;
8 r1 = pt ^ r0;
9 r2 = sm[r1];
```

(a) Sample implementation in C



(b) Corresponding DFG

Fig. 2. An example data-flow conversion from a given implementation. All variables are assumed to be a single bit for simplicity, without loss of generality. The inputs of the LUT_{sm} are statically generated using the formula LUT_{sm} `[sel] = s[i^a]^b`, where `sel` represents the selection line (`b` is the most significant bit followed by `a` and `i`).

3.1 Graph Representation

Our DFG nodes are either arithmetic/logic operations (`<x = y op z>`) or array handling operations (`<x = y[z]>` or `<y[z] = x>`). A node for an arithmetic/logic operation has two incoming edges for `y` and `z`, and one outgoing edge for `x`. Accurately covering data dependencies for the array handling operations requires special representation. This is because the accessed address, and hence the data it points to, might be determined by the input variables. For example, in the program `<a[0] = b; a[1] = c; t = a[d]>`, variable `t` is assigned either `b` or `c` depending on `d`. Hence, an array handling operation is treated according to whether the address it accesses is *statically-determinable* (i.e., the address is constant after propagating the constants statically along the program). For instance, the address of the second operation of the program `<i = 0; y = key[i]>` is statically-determinable, since `y` accesses `key[0]` independently of the inputs, while it is not statically-determinable for the program `<i = rnd; y = key[i]>` if `rnd` is an input. Accordingly, if all accesses to an array are statically-determinable, we refer to it as a *directly accessed array* and treat related accesses as ordinary assignment operations (e.g., `<y = key[0]>`). Otherwise, we call it an *indirectly accessed array* and create a *Look-Up-Table* (LUT) to mimic its behavior.

The LUT for an *indirectly accessed array* is represented as a $2^m : 1$ multiplexer, where m represents the total bitwidth of inputs that determines either the address or the data of the array. An example is given in Fig. 2(a); `sm` is an *indirectly accessed array* and there are $m = w_i + w_a + w_b$ bits (where w_x represents the bitwidth of x) that are involved in its calculation. The 2^m inputs of the LUT are set to fixed values that are calculated from *all* possible assignments of these m bits, e.g., the least significant input of the LUT is calculated by setting all of these m inputs to zero. Once the LUT is generated, the address of a load operation that accesses to that array is given as a select line together with the input bits used in the calculation. An example is shown in Fig. 2(b) for the single-bit case of the example in Fig. 2(a).

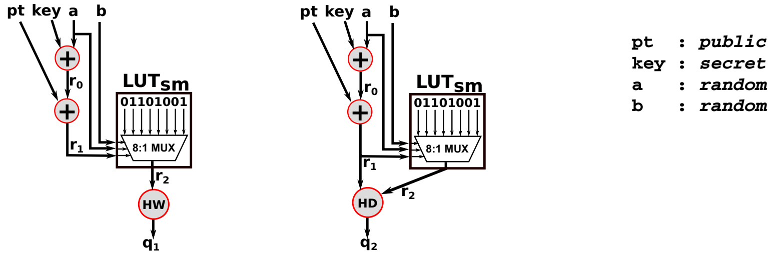


Fig. 3. The DFG corresponding to the code in Fig. 2 is redrawn to show leakage. Univariate leakage, q_1, for the last operation (`<r2 = sm[r1]>`) and bivariate leakage, q_2, caused by the consecutive execution of the last two operations (`<r1 = pt^r0; r2 = sm[r1]>`) are explicitly shown; HW and HD are selected as example leakage models.

This LUT-based approach is developed to cover *all* possible data-flows for perfect accuracy of the analysis, which is the primary concern of this work. It might be costly when we consider an application with a large number of pointer operations with huge points-to sets; however, such applications are rare, especially in power-analysis countermeasure implementations, since they are designed for memory-limited embedded systems and points-to sets are usually small. For example, for a masked AES implementation with different input and output masks, such a table has a size of $2^{24}\mathrm{B} = 16\mathrm{MB}$, which is not a huge table to be processed by a host computer (note that we run our verification tool on our own system, not the target system).

3.2 Sensitivity Detection

Once again, an operation (or a group of operations) is *sensitive* if two conditions hold simultaneously. First, the associated leakage must be statistically dependent on at least one secret input, and second, it must be statistically independent of any random input. Let us consider the left-hand side DFG of Fig. 3 as an example. The DFG computes the leakage of the operation on line 9 in Fig. 2(a) as the Hamming weight of the result of the look-up operation, which is $q1$. Then, dependence from the secret input key, first condition, can be expressed as $q_1 \sim key$, where $\sim$ denotes statistical dependence (i.e., $x \sim y$ means $\mathrm{I}(x; y) \neq 0$, where I stands for mutual information). Similarly, independence from the random inputs a and b, second condition, can be expressed as $\neg(q1 \sim a) \wedge \neg(q1 \sim b)$. A dual example for a bivariate leakage is shown at right-hand side DFG of Fig. 3. The latter models the leakage q_2 produced in the joint execution of the operations on lines 8 and 9. Accordingly, q_1 and q_2 will be $\mathrm{HW}(r_2)$ and $\mathrm{HD}(r_1, r_2)$, respectively.

Now that the conditions for sensitivity have been expressed, we need a practical method for checking statistical dependence. A naïve method would be to try all possible input values and compute the corresponding mutual information,

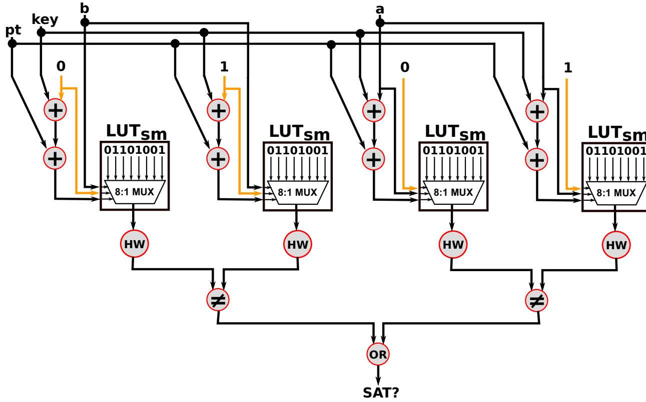


Fig. 4. The result of this SAT query determines whether the q_1 in Fig. 3 is randomized (when SAT?= 1) or not (when SAT?= 0). In this example, the answer is *true* and a satisfying assignment is $(a, b, key, pt) = (0, 0, 0, 0)$.

however, the sizes of the input bits of cryptographic algorithms are usually high (hundreds to thousands of bits), which make this method computationally infeasible.

Instead, inspired by the functional verification and logic synthesis communities, we propose to formulate our problem as a *satisfiability* (SAT) query and use SAT solvers to solve it. Thereby, we can leverage the latest advances in SAT technology, a hot area of research in the last decade (e.g., annual SAT competitions [3]), to speed up our analysis. Actually, the statistical dependence is analogous to the *don't-care* analysis of logic synthesis, since the input variables of the program are assumed to be, by definition, pairwise statistically-independent. An intermediate variable is statistically independent of an input variable if and only if the input variable is a *don't-care* for the intermediate variable. Such *don't-care* analysis have been successfully formulated as SAT queries in the context of logic synthesis [24].

Our solution initially checks the second condition (i.e., randomness) of the sensitivity as described in Algorithm 1. We check whether a bit is a *don't-care* for a particular leakage by using a SAT query that confirms whether setting the bit to 0 or to 1 can ever produce different results; if it can not, then the bit is a *don't care*. However, if we are able to find even a single bit that is not a *don't-care*, we conclude that the leakage depends on some random variables. Fig. 4 shows the query constructed based on Algorithm 1 to check the randomness for the q_1 of Fig 3, namely $\neg(q1 \sim a) \wedge \neg(q1 \sim b)$. The SAT solver reports *true*, confirming that q_1 is randomized and thus not sensitive. In the hypothetical case where the solver returns *false*, a second query is constructed with *secret* inputs instead of *random* ones, in order to determine whether there is any dependency to a *secret* variable, namely $q_1 \sim key$. If this second query returns *true*, the operation is *sensitive*; otherwise, it is *insensitive*.

Algorithm 1. Determine whether an output q depends on any random bit

Input: G : the DFG whose output is q
Input: I_R : the input bits of G which are random
Returns: a Boolean which determines whether G depends on any random bit
  for all $i \in I_R$ **do**
    $G_0 \leftarrow$ copy of G with i set to 0
    $G_1 \leftarrow$ copy of G with i set to 1
    if SAT($G_0 \neq G_1$) **then**
      return true
    end if
  end for
  return false

4 The Sleuth

In order to automate the methodology mentioned in Section 3, we developed `Sleuth`. We implemented `Sleuth` in the back-end of the `LLVM` [2] (*Low-Level Virtual Machine*) compiler; `LLVM` is an open-source compiler infrastructure that gained a significant popularity in the research community and industry in the last decade. `Sleuth` works on the input implementations given in `LLVM`-assembly language, which makes `Sleuth` compatible with many off-the-shelf tools. An implementation compiled for another embedded platform can easily be converted into `LLVM`-assembly; for instance, we implemented a straightforward conversion script from `AVR`- to `LLVM`-assembly. A high-level (C, C++, Python, etc.) implementation can be compiled into `LLVM`-assembly using off-the-shelf compilers, such as `llvm-gcc` or `clang`, if the user wants to analyze some high-level properties, such as verifying a given protection scheme.

4.1 User Interaction

`Sleuth` needs four inputs: the user's *original implementation* without any annotations, the *security type* annotations, the *leakage model* implementation, and the list of *operations* (or operation vectors) to be checked for sensitivity.

Original Implementation: The users provide their implementation in `LLVM`-assembly.

Security Type Annotations: The users provide a file that includes the *security types* of *all* the inputs. `Sleuth` reports an error to the users for the inputs that do not have any security type. This file consists of lines of the form $x_i : t_i$, where x_i is an *input* variable with *type* t_i (e.g., `key : secret`).

Leakage Model: The definition of the leakage model is used by the *sensitivity* detection algorithm mentioned in Section 3.2. The user has the flexibility to define a univariate (e.g., HW) or a multivariate (e.g., [15,17]) leakage model. The function takes an *operation vector* as parameter, which is used for defining

operator-dependent (e.g., the leakage of a multiplication operation could be defined differently than of a Boolean operation) or operand-dependent (e.g., the leakage might consider *only* some operands) leakage models; see Appendix A for samples. Nothing in the fundamental idea of this paper (Sections 2 and 3) prevents the implementation of leakage models that consider device-specific features or physical concerns (pipelining, caches, etc.); however, the challenge is in implementing such a leakage model and a software interface that conveys all the required information on the program and yet allows [experienced] users to capture the physical phenomenon in a practical, realistic, and somehow intuitive way. Thus, in this first version of the `Sleuth` we have a restricted interface to implement simple yet most popular (e.g., HW, HD) and effective leakage models; future versions might support implementing advanced physical features.

Queried Operations: The user can specify to `Sleuth` which leakages they want to analyze. A query file consists of any number of lines having the form `<num s_begin:s_end {all,consecutive}>`, where `num` represents the order of relations to analyzed (i.e., univariate, bivariate, etc.), `s_begin:s_end` represents the scope of the analysis (e.g., between lines 10 and 20), and `all/ consecutive` represents whether the analysis should be performed for all operations or only between pairwise consecutive ones (i.e., only valid in case of multivariate leakages). The users can specify as many such queries as they want ensuring the feasibility of the number of queries. Some results on the execution time of the queries in typical implementations are provided in Section 5.

At the end of its run, `Sleuth` reports the result of each query; a result is either `successful`, meaning that there is no *sensitive* leakage for the given query, or `failed` along with the operations causing the *sensitive* leakages. The user also has access to the DFGs generated for the sensitivity queries, which could be exploited to analyze the underlying problem.

4.2 Implementation Details

We implemented `Sleuth` in the back-end of the `LLVM` compiler, using standard `LLVM` libraries. `Sleuth` works in the following three fully-automated main steps:

- It first parses the *original implementation* in LLVM-assembly and converts it into a *straight-line program*, by unrolling loops and inlining function calls, without applying any optimizations. The user must handle any desired optimization before giving it as input to `Sleuth`, because we also analyze the low-level relations, such as operation ordering, which could be destroyed by the optimization.
- The *straight-line program* is converted to a DFG as explained in Section 3.1.
- The user specified *operations* are *queried* for sensitivity as explained in Section 3.2, using the user specified *security type annotations* and *leakage model*.

We used `KLEE` [1] as a base system, especially for parsing and its interface with SAT-solver. `KLEE` is a verification tool (more specifically, automatic test

generation tool) that also works in the back-end of the `LLVM`. We then implemented the mentioned features for generating the DFG and the SAT queries from the user-provided inputs.

5 Experimental Studies

In this section, we present some examples on how `Sleuth` can automatically detect *sensitive* leakages. We first give examples on different implementations of state-of-the-art countermeasures and present some potential programming pitfalls, which can be easily detected by `Sleuth`. Afterwards, we show that `Sleuth` can also be used to analyze the high-level algorithmic behavior of the countermeasures by showing the vulnerabilities of well-known Boolean to arithmetic mask conversion algorithms.

5.1 High-Level Code to Assembly Compilation Problems

As we motivate in the introduction and show in Fig. 1, randomly masking the key before operating with the plaintext is a common method for protection. However, if we implement our masking scheme in a high-level language and compile it for our target device using an off-the-shelf compiler, we might not always get the desired behavior in the low level assembly code. Figure 5 shows an example. Although we force the priority of the xor operation between `key` and `mask` in the C code using parentheses (line 9), our `AVR` port of the `gcc` compiler does the opposite (see lines 9-13 of assembly code): the xor (`eor` in AVR-assembly) between `key` and `pt` is performed first. This is not a bug from the functionality perspective since the xor operation is associative and the compiler made an arbitrary choice during one of the optimization phases. However, it is a serious problem from the security perspective: we are not doing a proper masking in the given assembly code, because the operation on line 11 leaks sensitive information (plaintext xor key). As shown by Mangard et al. [22], such an operation might be vulnerable to standard DPA attacks. Note that, although this is a small example, it is not a simple or local problem. Key whitening (plaintext xor key) is the first operation in many cryptographic algorithms and these key whitening operations (i.e., assembly operations on line 11, 17 and fourteen more lines for each iteration of the loop) might render the whole implementation vulnerable against DPA attacks even if the rest of the implementation is perfectly masked.

`Sleuth` is able to detect this problem easily. We converted the mentioned AVR assembly into LLVM assembly using a simple conversion script we implemented. The ordering of the operations was preserved during the conversion. We created the necessary files for `Sleuth` to define the types of the variables and the scope of the queries. Finally, we used the Hamming weight univariate leakage model. `Sleuth` was able to spot all of the sixteen *sensitive* xor operations (for each iteration of the loop) in 0.02 seconds on a standard desktop computer.

Similar problems can arise in later operations of an implementation; these are harder to detect because of the size of the problem and variety of the types of

```c
1  unsigned char st[16];
2  unsigned char key[16];
3  unsigned char pt[16];
4  unsigned char mask[16];
5  void ARK() {
6    unsigned char i;
7    for (i=0 ; i<16 ; i++) {
8      st[i] = pt[i] ^
9        (key[i] ^ mask[i]);
10   }
11 }
```

```
1    .text
2  .global ARK
3    .type
   ARK, @function
4  ARK:
5  /* prologue: function */
6  /* frame size = 0 */
7  /* stack size = 0 */
8  .L__stack_usage = 0
9    lds r24,key
10   lds r25,pt
11   eor r24,r25
12   lds r25,mask
13   eor r24,r25
14   sts st,r24
15   lds r24,key+1
16   lds r25,pt+1
17   eor r24,r25
18   ...
```

Fig. 5. A masked key whitening operation in C and the corresponding AVR-assembly code compiled with `avr-gcc -O3 -S` with `avr-gcc-4.5.3`. The ordering of the `xor` operations are not preserved because of its associative property.

operations. To show the effectiveness of `Sleuth` in mixture of arithmetic/logic and array handling operations, we implemented a protected round of AES which uses the Boolean masking algorithm of Herbst et al. [19], and tested it. There are a couple of points the designer must be extremely careful about when implementing this algorithm. For example, an improper implementation of the mask changing routine between the SubBytes and MixColumns can cause a sensitive leakage. This routine, briefly, changes the mask of row i of state matrix of AES from M' to M_i. In order to do this, we can simply xor all state variables in row i with $M' \oplus M_i$. However, the ordering of these xor operations is important; if we perform an xor operation between a state variable and M' before xor'ing with M_i, we will cancel the mask and reveal the original (unmasked) value of variable. This is highly likely when converting from a high-level implementation to the assembly as we show above or when programming manually. We intentionally inserted such bugs and ran `Sleuth` to see its performance.

`Sleuth` was able to detect all of such bugs in 430 seconds while creating a table of size 16MB for the S-Box. Considering the value of the feedback we get, this amount of time and memory appears quite reasonable. Note that we do not give any implementation specific parameter to `Sleuth`; it automatically analyzes the code, builds the graphs and LUTs and uses them to detect all *sensitive* operations for the given leakage model without even knowing that the masking countermeasure is applied.

5.2 Consecutive Execution Related Problems

In the previous examples, it is enough to use a univariate leakage model to detect the DPA-vulnerability of the individual operations. Sometimes, although each of the individual operations are *insensitive*, the consecutive execution of two (or more) of them might cause a *sensitive* leakage. Let us give an example.

```
1 // swap st[2] with st[10]
2 tmp = st[2];
3 st[2] = st[10];
4 st[10] = tmp;
```

This is a standard way of implementing the ShiftRows phase of AES for the third row of its state matrix. If we implement the Boolean masking algorithm of Herbst et al. [19], the input masks of `st[2]` and `st[10]` will be identical; i.e., original unmasked values are `st_orig[2]` $=$ `st[2]` $\oplus$ `m` and `st_orig[10]` $=$ `st[10]` $\oplus$ `m`, respectively. Since the ShiftRows phase propagates the masks without changing them, we can argue that if there is no information leaked before these operations, no information will be leaked here. However, this highly depends on the leakage (e.g., power consumption) characteristic of the target device and how we implement these operations at a low level. The corresponding optimized AVR-assembly implementation generated by avr-gcc is as follows:

```
1 lds r24,st+2
2 sts tmp,r24
3 lds r25,st+10
4 sts st+2,r25
5 sts st+10,r24
```

This code might be problematic when it is executed on a device whose power consumption characteristic is dominated by bit switching activity, as is the case with most of the state-of-the-art embedded devices. The power consumption of such a device is correlated with the Hamming distance of the number of bit switches between two consecutively executed operations [22]. Hence, consecutively executing the operations on lines 2 and 3 will have a power consumption value that is correlated with HD(`st[2]`,`st[10]`) = HD(`st_orig[2]`$\oplus$`m`, `st_orig[10]`$\oplus$`m`) = HW(`st_orig[2]`$\oplus$`st_orig[10]`). Note that this value does not depend on the mask m and only depends on the original, key dependent and non-randomized, values; hence, it is *sensitive*.

As a result, it is not only the individual operations themselves that can cause a sensitive leakage—the actual ordering of the operations is important, too. However, sensitive orderings could not be avoided in a standard compilation process (e.g., using `gcc`) or using automated protection tools (e.g., the tool of Moss et al. [25]) that does not consider the ordering problem specifically. Hence, we can use `Sleuth` to detect whether such a problem exists in a given implementation. We ran `Sleuth` to detect the sensitive bivariate leakages caused by *consecutive* operations and used Hamming distance as the leakage model. `Sleuth` was able to find all occurrences of the undesired consecutive operation orderings in 477 seconds for this example. In addition, for the above assembly code, it discovers that the consecutive ordering is also a problem for the operation pair on lines 4 and 5.

It is important to note that the *random precharging* countermeasure [29] is based on the idea of using random operations before and after sensitive ones to randomize the power consumption of consecutive executions. Using the same parameters we described here, we can easily detect the problems in an implementation of it.

5.3 Countermeasure Related Problems

In the previous examples, we focused on implementation-related issues. `Sleuth` can also be used to analyze the protection schemes as well. As an example, we analyze Boolean to arithmetic masking conversion algorithms by Messerges [23] and Goubin [18] (shown in Fig. 6 in Appendix B for quick reference). They both convert a Boolean masked variable into arithmetic masked version, i.e., if x' is the Boolean masked value of x and the mask is r_x, they find the value $\mathtt{A}$ such that $x' \oplus r_x = \mathtt{A} + r_x$.

The algorithm of Messerges [23] reveals either the original unmasked values (e.g., x) or the bitwise-negation of them (e.g., $\overline{x}$) during the intermediate calculations, based on the value of a random bit. The assumption is that the attacker would not know the value of the random bit; hence, any bit of an intermediate value (e.g., x or $\overline{x}$) is equally likely to be 0 or 1. The argument is valid when we consider each bit of x independently; however, as pointed by Coron and Goubin [12] later, the probability distribution of xor of last two bits of x or $\overline{x}$ is independent of the random value, i.e., $x_{[1]} \oplus x_{[0]} = \overline{x}_{[1]} \oplus \overline{x}_{[0]}$ always holds (where $x_{[i]}$ represents the ith bit of x). Hence, an attacker can use this power model (xor of two last bits of A) to successfully attack an implementation of this method. We implemented this conversion algorithm in C. `Sleuth` was able to detect the problem in about 0.02 seconds using different leakage models (e.g., xor of last two bits of the result, parity of the result, etc.).

Similarly, we can use `Sleuth` to analyze the Boolean to arithmetic mask conversion algorithm of Goubin [18]. In this experiment, we ran our analysis on all pairs of operations to find out the possible second-order attack points against this algorithm. `Sleuth` reported us that a second-order attack might be possible when we use the Hamming distance of the results of the operations (i.e., on lines 3 and 6, $\mathrm{HD}(x' \oplus R, R \oplus r_x) = \mathrm{HW}(x' \oplus R \oplus R \oplus r_x) = \mathrm{HW}(x' \oplus r_x) = \mathrm{HW}(x)$ only depends on the secret value x). `Sleuth` was again able to find this bivariate sensitive leakage in 0.02 seconds.

Note that, using the correct leakage model is important to get the desired result; however, we support an extensible library of different leakage models, thus, less experienced users could conservatively try them all without knowledge of the exact model to be used.

6 State of the Art

Automatic generation and verification of secure systems is becoming increasingly popular in the security and design automation communities. The first attempts concerning the automatic generation of robust implementations began soon after the introduction of side-channel attacks [16,21,26] and the countermeasures against them. Initially, the research community focused mainly on the automation for hardware countermeasures, in the synthesis [27,30] or placement and routing phases [6,31]. More recently, automation has begun to be adopted in software countermeasures. Firstly, Bayrak et al. [7] presented a general framework which identifies sensitive instructions against power analysis attacks and

protects them by applying state-of-the-art countermeasures; in their study, random precharging was used. Soon after, Agosta et al. [4] proposed an automatic code morphing technique and Moss et al. [25] introduced a type system for automatic application of Boolean masking. These works focused mainly on power analysis attacks, whereas Cleemput et al. [10] proposed compiler mitigations for timing attacks. Our verification framework can be used to verify the output of some of these automatically-generated protected implementations. For example, Moss et. al [25] assumed that the information leakage is independent for each executed instruction. However, as shown in Section 5.2, consecutive execution of operations that are independently insensitive might cause vulnerabilities even against first-order attacks, and `Sleuth` can detect such problems.

State-of-the-art automatic verification methods have mainly focused on information flow analysis and timing vulnerability detection. Information-flow analysis techniques aim to detect whether there is any transfer of information that is undesired (such as a flow from a high-security variable to a low-security one) [28]. Such techniques have been applied to several areas, such as network security [20], operating system security [13] and gate-level hardware security [32]. These techniques usually do not cover side-channels. Ford et al. [14] proposed an information flow technique for detecting timing side-channels. Similarly, Vieira proposed a formal verification methodology for timing channels [33]. In this work, instead, we focused more on power side-channels and showed that we can successfully detect potential programming pitfalls in power analysis software countermeasure implementations.

7 Conclusions

In this work, we address, for the first time, the automated verification of power analysis countermeasures. We propose a fully-automated methodology based on converting a given implementation into a DFG and solving a set of satisfiability problems on this graph. We showed that this methodology is capable of detecting first-order or higher-order power analysis vulnerabilities in both the countermeasure itself and an improper implementation of it. We implemented the proposed methodology in a tool, `Sleuth`, which we used to verify several real world software routines aimed to be resistant against power analysis attacks. `Sleuth` demonstrates the effectiveness of our approach by automatically detecting, in a reasonable amount of time, not only the implementation related security bugs (such as improper ordering of instructions and undesired mask cancellation in an intermediate calculation), but also the hidden problems in the countermeasure itself, such as the ones in the Boolean to arithmetic masking conversion algorithm proposed by Messerges [23].

Although the presented SAT-based methodology is generic and powerful, the success of an analysis depends on the reliability of the used leakage model; this is a common problem in *any* power analysis based work that uses a leakage model. The limitation of the *tool* (not the proposed SAT-based methodology) is that the software interface has limited capabilities to define device-specific features; we are planning to improve the interface in future versions.

In terms of scalability, the usefulness of the tool is limited by the efficiency of the SAT-solver. However, almost any other verification tool in any domain (hardware, software) suffers from the same limitation, and yet verification tools are invaluable tools to designers. The initial results are promising: currently we detect all problems in an AES round within a few minutes, whereas a brute-force approach is intractable. There is vast literature on how to improve efficiency, potentially also trading off some accuracy; future research direction might explore such potential improvements.

References

1. The KLEE symbolic virtual machine, http://klee.llvm.org
2. The LLVM compiler infrastructure, http://llvm.org
3. SAT competitions, http://www.satcompetition.org
4. Agosta, G., Barenghi, A., Pelosi, G.: A code morphing methodology to automate power analysis countermeasures. In: Design Automation Conference, DAC 2012, pp. 77–82 (2012)
5. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques, & Tools. Pearson (2006)
6. Badel, S., Guleyupoglu, E., Inac, O., Martinez, A.P., Vietti, P., Grkaynak, F.K., Leblebici, Y.: A generic standard cell design methodology for differential circuit styles. In: Design, Automation and Test in Europe, DATE 2008, pp. 843–848 (2008)
7. Bayrak, A.G., Regazzoni, F., Brisk, P., Standaert, F.X., Ienne, P.: A first step towards automatic application of power analysis countermeasures. In: Design Automation Conference, DAC 2011, pp. 230–235 (June 2011)
8. 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)
9. Bohn, R.E., Short, J.E.: How much information? 2009 Report on American consumers (December 2009)
10. Cleemput, J.V., Coppens, B., de Sutter, B.: Compiler mitigations for time attacks on modern x86 processors. ACM Transactions on Architecture and Code Optimization 8(4), 23:1–23:20 (2012)
11. Computer Aided Cryptography Engineering (CACE European Project), http://www.cace-project.eu
12. Coron, J.-S., Goubin, L.: On Boolean and arithmetic masking against differential power analysis. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, pp. 231–237. Springer, Heidelberg (2000)
13. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: USENIX Conference on Operating Systems Design and Implementation, pp. 1–6 (2010)
14. Ford, B.: Plugging side-channel leaks with timing information flow control. arXiv preprint arXiv:1203.3428 (2012)
15. Fumaroli, G., Martinelli, A., Prouff, E., Rivain, M.: Affine masking against higher-order side channel analysis. Cryptology ePrint Archive, Report 2010/523 (2010), http://eprint.iacr.org/
16. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: Concrete results. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 251–261. Springer, Heidelberg (2001)

17. Gierlichs, B., Batina, L., Preneel, B., Verbauwhede, I.: Revisiting higher-order DPA attacks. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 221–234. Springer, Heidelberg (2010)
18. Goubin, L.: A sound method for switching between Boolean and arithmetic masking. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 3–15. Springer, Heidelberg (2001)
19. 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)
20. Gray III, J.W.: Toward a mathematical foundation for information flow security. Journal of Computer Security 1(3), 255–294 (1992)
21. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999)
22. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks - Revealing the Secrets of Smart Cards. Springer (2007)
23. Messerges, T.S.: Securing the AES finalists against power analysis attacks. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 150–164. Springer, Heidelberg (2001)
24. Mishchenko, A., Brayton, R.K.: SAT-based complete don't-care computation for network optimization. In: Design, Automation and Test in Europe, DATE 2005, pp. 412–417 (2005)
25. 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)
26. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
27. Popp, T., Kirschbaum, M., Zefferer, T., Mangard, S.: Evaluation of the masked logic style MDPL on a prototype chip. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 81–94. Springer, Heidelberg (2007)
28. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
29. Tillich, S., Großschädl, J.: Power analysis resistant AES implementation with instruction set extensions. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 303–319. Springer, Heidelberg (2007)
30. Tiri, K., Verbauwhede, I.: A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In: Design, Automation and Test in Europe, DATE 2004, pp. 246–251 (2004)
31. Tiri, K., Verbauwhede, I.: A digital design flow for secure integrated circuits. IEEE Transactions on CAD of Integrated Circuits and Systems 25(7), 1197–1208 (2006)
32. Tiwari, M., Wassel, H.M.G., Mazloom, B., Mysore, S., Chong, F.T., Sherwood, T.: Complete information flow tracking from the gates up. ACM Sigplan Notices 44(3), 109–120 (2009)
33. Vieira, B.: Formal Verification of Cryptographic Software Implementations. Ph.D. thesis, Universidade do Minho, Portugal (2012)

A Sample Leakage Definition in Sleuth

In this section, we provide a sample leakage model implementation for `Sleuth`. The user can overwrite this function, and define other multivariate leakage models that consider any number of operations. The declaration of the function (in C) is as follows:

```
uint32_t Sleuth_leakage(Operation* ops, uint32_t sz);
```

where `ops` and `sz` represent the *operation vector* and its *size*, respectively. An *operation* is a structure

```
struct Operation{
  char* opname; uint32_t x; uint32_t y; uint32_t z;};
```

where `opname` is the operation name used as in LLVM-assembly (e.g., `"xor"`) and `x`, `y` and `z` are the operands (see Section 2.1 for details).

This function returns the value of the leakage in terms of the elements of the operations. We used 32-bits leakage values; realistic measurements (e.g, using an oscilloscope) usually have even less precision. The default univariate and bivariate leakage model definition is as follows:

```
// HW and HD functions return Hamming weight and distance
uint32_t Sleuth_leakage(Operation* ops, uint32_t sz) {
        if (sz == 1) { // univariate leakage
                return HW(ops[0].x);
        } else if (sz == 2) { // bivariate leakage
                return HD(ops[0].x,ops[1].x);
        } // one can define a multivariate leakage similarly
        return 0;
}
```

B Boolean to Arithmetic Masking Conversion Algorithms

The Boolean to arithmetic masking conversion algorithms of Messerges [23] and Goubin [18] are shown in Fig. 6.

```
1  BooleanToArithmetic_Messerges(x',r_x) {
2    // randomly select: C = 0 or C = -1
3    B = C ⊕ r_x; /* B=r_x or B=~r_x */
4    A = B ⊕ x'; /* A=x or A=~x */
5    A = A - B;  /* A=x - r_x or A=~x - ~r_x */
6    A = A + C;  /* A=x - r_x or A=~x - ~r_x */
7    A = A ⊕ C;  /* A=x - r_x */
8  }
9
10
```

```
1  BToA_Goubin(x',r_x){
2    // random R
3    T = x' ⊕ R;
4    T = T - R;
5    T = T ⊕ x';
6    R = R ⊕ r_x;
7    A = x' ⊕ R;
8    A = A - R;
9    A = A + T;
10 }
```

Fig. 6. Two different algorithms for switching from Boolean to arithmetic masking, proposed by Messerges [23] and Goubin [18], respectively. They both find the value of A such that $x = x' \oplus r_x = \mathrm{A} + r_x$, where x is the unmasked original value and r_x is the random mask.