Automatic Detection and Repair of Transition-Based Leakage in Software Binaries

Konstantinos Athanasiou(✉), Thomas Wahl, A. Adam Ding, and Yunsi Fei

Northeastern University, Boston, MA, USA
athanasiou.k@northeastern.edu

Abstract. The effectiveness of masking as a countermeasure against information leakage in cryptographic ciphers is contingent upon individual instructions leaking information independently. An example of dependent leakage is *transition-based leakage*: side-channels can leak data proportional to the exclusive-or of the old and the new value during a register write, despite proper first-order masking. In this paper we present a technique to detect and repair transition-based leakage. Our detection technique symbolically executes a binary to relate the old and new values during a register write. We then combine existing analyses to check for, and quantify, any leakage caused by the write. Our repair module closes a leak by flushing the affected register before writing it.

We also present a fully automated detection and repair tool called BATTL, which is the first to our knowledge to operate at the binary level and is thus sensitive to decisions at any compilation stage that may affect security. We evaluate BATTL against first-order secure implementations of the AES block cipher, and of a secure multiplicative inversion algorithm. BATTL identified a number of transition-based leakages, some with high leakage amounts. Our countermeasure removes all first-order leakages with only moderate runtime overhead.

Keywords: Side channels · Masking · Transition-based leakage

1 Introduction

The security of cryptographic algorithms, such as block ciphers, against crypto attacks, i.e. attacks that exploit how differences in the input correlate with the algorithm's output, is a well-studied problem [20]. Side-channel attacks use physical measurements of cryptographic implementations, such as timing, power consumption, and electromagnetic emanations [17,21,22], to recover the ciphers' secret, demonstrating that "correct" implementations of the cryptographic algorithms do not guarantee their security.

Countermeasures against such attacks are thus critical. The most widely used such measure, the *masking* of software, is a form of secret sharing that splits a secret in $d+1$ shares such that the joint distribution of any d shares is statistically independent of the secret. The implementation has to be refactored to compute its results over $d+1$ shares. If applied correctly, masking guarantees order-d security: any combination of d intermediate results of the implementation is statistically independent of the secret.

Work supported by the US National Science Foundation under award no. SaTC-1563697.

M. Christakis et al. (Eds.): NSV 2020/VSTTE 2020, LNCS 12549, pp. 50–67, 2020.
https://doi.org/10.1007/978-3-030-63618-0_4

These guarantees rest, however, on the *independent leakage assumption* [27]: the intermediate points in the computation should leak independently. Prior work has established that this assumption does not always hold [1,9]. In software implementations, physical effects that occur when a register's old value is overwritten with a new value give rise to leakage that depends on both values, a phenomenon known as *transition-based leakage* (TBL). Balash et al. experimentally demonstrated that a first-order ($d = 1$) secure implementation that splits its secret in ($d + 1 = 2$) shares, leaks the value of a secret to a first-order attacker (that observes $d = 1$ intermediate results) [1]. The leakage assessment required only a small number of power traces, which was collected and analyzed in a few minutes. Effectively, the paper showed that TBL incurs a reduction of the security order by a factor of 2: an attacker can now in effect observe (some) pairs of intermediate results. The same authors suggest that TBL can be averted simply by doubling the masking order. This approach, although easy to enforce, incurs a non-trivial performance overhead compared to the original cryptographic algorithm.

Prior work has proposed to instead address this problem using a modified compiler [31]: by determining pairs of program values that may cause TBL and taking this into account when assigning registers, it prevents TBL-inducing overwrites to begin with. This technique is light-weight and efficient to execute on a given input program. On the down-side, it comes with a heavy implementation cost: modifying a compiler's register allocation is non-trivial and compiler-specific, as it doesn't readily transfer to different compilation toolchains. Furthermore, this prior approach cannot account for leakages introduced during later compilation stages (e.g. linking).

In this paper, instead of analyzing the masked source code or an intermediate representation of it, we propose to analyze the compiled *binaries* of first-order masked programs. Our technique uses symbolic execution to locate register overwrites whose combined leakage involves all shares of the secret. We refer to these writes as *potential* TBLs. To check for statistical dependence of the potential leakage on the secret, we employ a secret dependence detection scheme that utilizes existing approaches [18,32] and quantifies the leakage intensity. If non-zero, we speak of *genuine* TBL.

Operating at the binary level, our technique detects leakages irrespective of whether they are present at the source code, or were introduced at some stage during the compilation. As an example, we show in this paper instances of leakage introduced during compiler optimizations, undetectable for source-code analyses. Our technique is also compiler-agnostic, allowing us to present experiments with different compilers and to compare the leakage characteristics of code generated by them. The technique is implemented in a publicly available tool called BATTL ("Binary Analysis Tool for Transition-based Leakage" [5]), which relies on the `angr` binary analysis platform [30].

If genuine TBL is detected in the binary, BATTL recompiles the masked program into assembly code, and applies to it a countermeasure based on register flushing. Given a register overwrite that causes genuine TBL, we assign a constant value to the register before the overwrite. At the end, the detection technique can be re-applied to the repaired binaries. In our experiments we confirmed that the flushing countermeasures indeed eliminated the detected leakages, and did not introduce new ones. This is not possible using the compiler-modification strategy proposed in [31].

2 Background and Problem Formalization

2.1 Abstract State Machines

We introduce a simple abstract state machine model, "machine" for short, to study the different leakage models of software implementations. Let V be a set of *values*, and $A \subseteq V$ a set of *addresses*. A machine *state* s is a mapping $s\colon A \to V$ from addresses to values; we denote by S the set of all such states. For $s \in S$ and $a \in A$, notation $s(a)$ therefore denotes the value stored at address a. Let $R \subseteq S \times S$ be a *transition relation*, formalizing the possible state changes of the machine. An *execution* of the machine is a sequence of states $s_0, s_1, s_2, \ldots$ such that, for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. The initial state s_0 contains the set of machine inputs I.

2.2 Leakage Modeling

The leakage behavior of a state or a transition is formalized using *leakage models*: functions assigning to each state, or each transition, a mapping from addresses to leakage measures. Let $\boxplus$ denote bit-wise XOR.

Definition 1. *The* ***value-based leakage function*** $V_L\colon S \to (A \to V)$ *and the* ***transition-based leakage function*** $T_L\colon R \to (A \to V)$ *are defined by*

$$V_L(s) = \{(a, s(a)) : a \in A\}\,, \qquad \textit{and} \tag{1}$$

$$T_L(s, s') = \{(a, s(a) \boxplus s'(a)) : a \in A\}\,. \tag{2}$$

For traceability, we denote XOR used to compute leakage (such as in T_L) by $\boxplus$, while XOR used in source code is denoted $\oplus$. Our definition of V_L above can simply be written as the identity function: $V_L(s) = s$; we write it in the form (1) to emphasize the contrast with (2). In practice, leakage functions often return more abstract measures of the value $s(a)$ stored at some address a, such as its Hamming weight [23]. Our definition is stronger, as it assumes the exact value $s(a)$ is leaked.

The (leakage) *trace* of an execution $t = s_0, s_1, s_2, \ldots$ is obtained by lifting the application of the leakage function from states to traces for VBL and from pairs of states in R to traces of TBL, resulting in:

$$\begin{array}{lllll} t_{V_L} = & V_L(s_0), & V_L(s_1), & V_L(s_2), & \ldots \\ t_{T_L} = & & T_L(s_0, s_1), & T_L(s_1, s_2), & \ldots \end{array}$$

2.3 Masking and Threat Model

Masking [8] is a form of secret sharing [29] in which a secret k is split into *shares* that can be combined using a suitable function to recover the secret. We consider order-d Boolean masking, which uses $d+1$ share variables, called *(secret) shares*, computed by introducing d shares distributed uniformly at random, and defining the $(d+1)^{\text{st}}$ share as $k_d = k \oplus k_0 \oplus \ldots \oplus k_{d-1}$. The secret thus satisfies $k = k_0 \oplus \ldots \oplus k_d$. Masking is a countermeasure against *differential power analysis* on cryptographic algorithms [21], aimed at making every intermediate measure leaked by the machine appear random.

The plaintext, the secret, the random variables that give rise to the shares, and potentially additional random variables form the input set I of a machine implementing a cryptographic algorithm.

Definition 2. *The result of a leakage function f_L* ***statistically depends*** *on the secret if there exist secret values k, k' and a plaintext value p such that the distributions of values of f_L over all choices of the random inputs differ between inputs (k, p) and (k', p).*

We assume the standard Differential Power Analysis threat model [21]: the adversary has physical access to the machine, can provide plaintext inputs, can measure power consumption when executing over the secrets, can measure execution time and knows the implemented algorithm. She does not know the secret values in the implemented algorithm.

Definition 3. *An* ***execution*** *of a machine is* ***first-order secure*** *under leakage function f_L if each measure generated by f_L along its trace is statistically secret-independent.*

In this paper we focus on the problem of determining whether an execution of a given masked software is first-order secure under TBL. If not, we devise countermeasures that repair the leakage, while preserving the functional semantics of the program.

3 Motivating Example

3.1 Transition-Based Leakage

Compilers are known to compromise software security properties, due to the abstraction gap between source and binary code [12]. Physical resource allocation epitomizes this gap. The compiler maps program variables to physical locations, e.g. registers, which are not present in the source code. This has consequences: *transition-based leakage* [1,9], a known phenomenon in processors that occurs when a register's contents is updated, can accidentally be introduced by a compiler agnostic to the abstraction gap.

We use the *secure inversion* of a shared secret [28] to demonstrate this phenomenon in software binaries. Secure inversion, shown in Algorithm 1, is used in software implementations of the AES S-Box, parameterized by the order d of the masking applied to the source code. Secure multiplication, shown in Algorithm 2, is used inside secure inversion to multiply two shared secrets of size 1 byte each.

We consider the first call to SecMul in Algorithm 1 (Line 3) with symbolic arguments $\mathbf{z} = (\mathtt{x}_0^2 \oplus \mathtt{r}, \mathtt{x}_1^2 \oplus \mathtt{r})$ and $\mathbf{x} = (\mathtt{x}_0, \mathtt{x}_1)$, where $\mathtt{x} = \mathtt{x}_0 \oplus \mathtt{x}_1$ is the shared secret and $\mathtt{r}$ the random value introduced after the first call to RefreshMasks. We focus on the first time Line 4 of Algorithm 2 is executed. To illustrate TBL, we inspect the ARM assembly of the source code, which is shown in Listing 1.1 of Fig. 1: Line 1 sets up the first call to the `spamul` function, called in Line 2 to compute the product $a_0 \times b_1 := (\mathtt{x}_0^2 \oplus \mathtt{r}) \times \mathtt{x}_1$; Lines 3–5 compute the leftmost XOR prioritized by the parentheses, i.e. $r_{0,1} \oplus (a_0 \times b_1) := \mathtt{r}_{0,1} \oplus (\mathtt{x}_0^2 \oplus \mathtt{r}) \times \mathtt{x}_1$; Lines 6–14 set up the second call to `spamul` at Line 15, which computes $a_1 \times b_0 := (\mathtt{x}_1^2 \oplus \mathtt{r}) \times \mathtt{x}_0$.

Figure 2 depicts the register file contents after executing Lines 1 and 14 of Listing 1.1. Following the ARM calling convention, register `r1` holds the second argument

Algorithm 1. SecInv: secure inversion of shared secret x in GF(2^8).

Require: $\mathbf{x} = (x_0, \ldots, x_d)$ s.t. $\oplus_i x_i = x$
Ensure: $\mathbf{y} = (y_0, \ldots, y_d)$ s.t. $\oplus_i y_i = x^{-1}$
1: **for** i from 0 to d **do** $z_i := x_i^2$
2: RefreshMasks($\mathbf{z}$)
3: $\mathbf{y}$:=SecMul($\mathbf{z}$,$\mathbf{x}$)
4: **for** i from 0 to d **do** $w_i := y_i^4$
5: RefreshMasks($\mathbf{w}$)
6: $\mathbf{y}$:=SecMul($\mathbf{y}$,$\mathbf{w}$)
7: **for** i from 0 to d **do** $y_i := y_i^{16}$
8: $\mathbf{y}$:=SecMul($\mathbf{y}$,$\mathbf{w}$)
9: $\mathbf{y}$:=SecMul($\mathbf{y}$,$\mathbf{z}$)

Algorithm 2. SecMul: secure multiplication of shared secrets x and y.

Require: $(a_0, \ldots, a_d)$ and $(b_0, \ldots, b_d)$ s.t. $\oplus_i a_i = a$ and $\oplus_i b_i = b$
Ensure: $(c_0, \ldots, c_d)$ s.t. $\oplus_i c_i = a \times b$
1: **for** i from 0 to d **do**
2: **for** j from $i+1$ to d **do**
3: $r_{i,j} \in_R GF(2^n)$
4: $r_{j,i} := (r_{i,j} \oplus a_i \times b_j) \oplus a_j \times b_i$
5: **for** i from 0 to d **do**
6: $c_i := a_i \times b_i$
7: **for** j from 0 to $d, j \neq i$ **do**
8: $c_i := c_i \oplus r_{i,j}$

to `spamul`, i.e. $\mathtt{x}_1$ in Line 3 and $\mathtt{x}_0$ in Line 14. The overwrite of register `r1` caused by the `mov` instruction in Line 14 involves both shares of $\mathbf{x}$ in a single instruction. This fact alone does not yet indicate a secret leakage; we refer to it as a *potential* leak. To determine whether the leak is *genuine*, we use the leakage model function T_L (introduced in Sect. 2.2), i.e. the XOR between the old and new values of the register write [1,9]. The potential leak observed at Line 14 measures as $\mathtt{x}_1 \boxplus \mathtt{x}_0 = \mathtt{x}$ and constitutes a genuine leak of the value of `x` as the secret and the leakage expression are directly related.

To see that the presence of all shares of a secret in a leakage expression is not sufficient for genuine leakage, consider the first call to `spamul`: its return value $(\mathtt{x}_0^2 \oplus \mathtt{r}) \times \mathtt{x}_1$ is stored in register `r2` (not shown in Listing 1.1). The load at Line 7 overwrites `r2` with a constant value and forms the potential TBL $(\mathtt{x}_0^2 \oplus \mathtt{r}) \times \mathtt{x}_1 \boxplus \mathtt{0x7ffefd74}$. This leakage is not genuine since the random variable `r` eliminates any statistical dependence between the leakage expression and the secret $\mathbf{x}$. The goal of this paper is to detect genuine TBL, and to protect against it using countermeasures.

3.2 Complications Ahead: Value-Based Leakage

Ignoring security concerns during compilation can not only introduce TBL, as seen in Sect. 3.1, but can in fact sabotage the very leakage protection that source-code transformations like masking are supposed to provide. To see this, consider Line 4 of Algorithm 2 and its corresponding *optimized* ARM assembly, shown in Listing 1.2 of Fig. 1. The two calls to `spamul` are inlined and computed in parallel in Lines 1–9. After executing the load instruction at Line 10, registers `r6` and `r2` hold the two products computed in Line 4 of SecMul, namely $(\mathtt{x}_0^2 \oplus \mathtt{r}) \times \mathtt{x}_1$ and $(\mathtt{x}_1^2 \oplus \mathtt{r}) \times \mathtt{x}_0$ respectively, and register `r3` the fresh random value $\mathtt{r}_{0,1}$. In Lines 11 and 12, the compiler decides to exploit the associativity of XOR and change the order of operations, creating the expression $(\mathtt{x}_0^2 \oplus \mathtt{r}) \times \mathtt{x}_1 \oplus (\mathtt{x}_1^2 \oplus \mathtt{r}) \times \mathtt{x}_0$ for the symbolic value computed at Line 11. This expression is statistically dependent on the secret `x` (the random variable r cancels out and provides no protection). The resulting VBL demonstrates how the protection provided by the source-code level masking was inadvertently destroyed by an optimization that obliviously rewrote an expression.

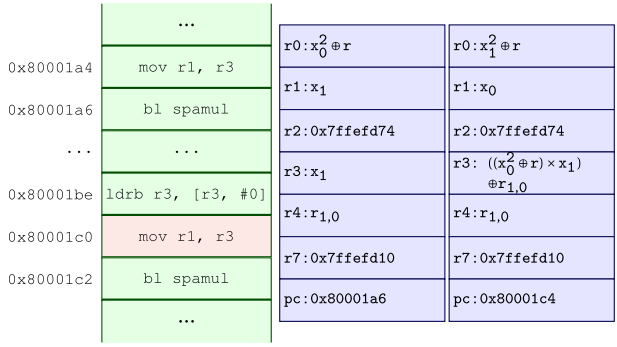
```
1  mov   r1, r3
2  bl    spamul
3  mov   r3, r0
4  eors  r3, r4
5  uxtb  r4, r3
6  ldr   r3, [r7, #24]
7  ldr   r2, [r7, #12]
8  add   r3, r2
9  ldrb  r0, [r3, #0]
10 ldr   r3, [r7, #28]
11 ldr   r2, [r7, #8]
12 add   r3, r2
13 ldrb  r3, [r3, #0]
14 mov   r1, r3
15 bl    spamul
```

Listing 1.1: Default asm (`-O0`)

```
1  ldrb  r2, [r2, #256]
2  ldrb  r3, [r3, #256]
3  ubfx  r6, r6, #7, #1
4  ubfx  r0, r0, #7, #1
5  orrs  r1, r5
6  and   r6, r6, r4, asr #7
7  and   r0, r0, r1, asr #7
8  mul   r6, r6, r2
9  mul   r2, r0, r3
10 ldrb  r3, [ip, #1]
11 eors  r2, r6
12 eors  r3, r2
13 strb  r3, [sp, #30]
14 ldrb  r2, [sp, #4]
```

Listing 1.2: Optimized asm (`-O2`)

Fig. 1. Assembly code for Line 4 of Algorithm 2 (SecMul) generated by default compiler flags (left) and optimization flags (right)



(a) Code segment (b) Register file contents

Fig. 2. (a) Code segment of Listing 1.1 causing transition-based leakage after executing the instruction highlighted red. (b) Register file contents after executing the first `mov` instruction (left), and after executing the highlighted `mov` instruction (right), which ultimately causes the leakage. (Color figure online)

While VBL due to compiler optimizations is not the main object of our study, it has important repercussions on the detection and repair of TBL: Line 14 of Listing 1.2 loads the constant value `0x7ffefd74` to register `r2` and produces the TBL $(\mathtt{x}_0^2 \oplus \mathtt{r}) \times \mathtt{x}_1 \oplus (\mathtt{x}_1^2 \oplus \mathtt{r}) \times \mathtt{x}_0 \boxplus \mathtt{0x7ffefd74}$, which is statistically dependent on `x`. While technically a TBL, the leakage is already present in the value stored in register `r2` and is unrelated to the confluence of all shares of a secret during the overwrite. We require for a potential TBL to be genuine, not only that the $\boxplus$ between the old and the new value of the register be statistically dependent on the secret, but also that neither value have genuine value-based leakage. Line 9 of Listing 1.2 is an example of a genuine TBL as it satisfies all these conditions. We formally define all these concepts in Sect. 4. This distinction also becomes relevant when it comes to eliminating the leakage as the TBL countermeasure we present in Sect. 5 is designed to disrupt said confluence of shares and will thus not repair TBLs that are in fact due to value-based leaks.

The latter type must be repaired using its own dedicated techniques, e.g. by preventing the above-mentioned unsafe expression rewriting.

4 Detection of Transition-Based Leakage

4.1 Potential and Genuine Leakage

We extend the definition of the abstract machine of Sect. 2.1 to a *symbolic* machine: let V_{sym} denote the set of *symbolic values* (expressions) over program inputs, A_{sym} the set of *symbolic addresses* (address variables), S_{sym} the set of *symbolic states* and R_{sym} the set of symbolic transitions. Let $Sh(k)$ be the set of shares into which the secret k is split, i.e. such that $\bigoplus_{k_i \in Sh(k)} k_i = k$. The shares are part of the symbolic input variables I_{sym}, i.e. $Sh(k) \subseteq I_{sym}$. Let $vars(v)$ be the set of symbolic variables in expression v.

Definition 4. *A **potential value-based leakage (PVBL)** is a pair $(s, a) \in S_{sym} \times A_{sym}$ such that $Sh(k) \subseteq vars(s(a))$. A **genuine value-based leakage (GVBL)** is a PVBL (s, a) such that $V_L(s)(a)$ statistically depends on k.*

Definition 4 captures the intuition that a VBL satisfies the necessary leakage condition of being *share-complete*: it must contain all secret shares. Our motivation to distinguish between *potential* and *genuine* is to have a sound and quick (although partial) way of checking statistical independence: it is implied by share-incompleteness.

We revisit some examples of leakages presented in Sect. 3. For a shared secret $\mathtt{x}$ such that $\mathtt{x}_0 \oplus \mathtt{x}_1 = \mathtt{x}$ and a random value $\mathtt{r}$, $(\mathtt{x}_0^2 \oplus \mathtt{r}) \times \mathtt{x}_1$ is a PVBL since it is share-complete, but it is not genuine since it is statistically independent of $\mathtt{x}$. In contrast, the PVBL $(\mathtt{x}_0^2 \oplus \mathtt{r}) \times \mathtt{x}_1 \oplus (\mathtt{x}_1^2 \oplus \mathtt{r}) \times \mathtt{x}_0$ is a GVBL: it does statistically depend on $\mathtt{x}$. We discuss in Sect. 6 how to check this dependence condition.

Definition 5. *A **potential transition-based leakage (PTBL)** is a triple $(s_{i-1}, s_i, a) \in R_{sym} \times A_{sym}$ such that*

(i) $Sh(k) \subseteq vars(s_{i-1}(a)) \cup vars(s_i(a))$, and
(ii) neither (s_{i-1}, a) nor (s_i, a) is a GVBL.

*A **genuine transition-based leakage (GTBL)** is a PTBL (s_{i-1}, s_i, a) such that the term $T_L(s_{i-1}, s_i)(a)$ statistically depends on k.*

Note how we design our notion of (P)TBL to depend on GVBL, in order to exclude leakages attributable to VBL (see Sect. 3.2). Under the above definitions, the leakage $(\mathtt{x}_0^2 \oplus \mathtt{r}) \times \mathtt{x}_1 \oplus (\mathtt{x}_1^2 \oplus \mathtt{r}) \times \mathtt{x}_0 \boxplus \mathtt{0x7ffefd74}$ from the overwrite of register $\mathtt{r2}$ discussed in Sect. 3.2 is not a GTBL since the lhs of $\boxplus$ (the old value of $\mathtt{r2}$) is (already) a GVBL.

4.2 Detection of Genuine Transition-Based Leakage

Following the definitions of Sect. 4.1, our TBL detection must be able to decide share-completeness and statistical dependence. We make use of forward symbolic execution to check for share-completeness by keeping track of the *symbolic transitions*, i.e. pairs containing the symbolic values of a location at two consecutive symbolic states.

```
1        0x22:mov r8, #0
2        0x24:mov r0, r9
3        0x26:ldrb r1, [r6, r8]
4        0x2a:mul r0, r0, r1
5        0x2e:mov r1, r10
6        0x30:mov r4, r0
7        0x32:ldrb r0, [r5, r8]
8        0x36:mul r0, r0, r1
9        0x3a:add r8, r8, #1
10       0x3c:cmp r8, #2
11       0x40:bne 0x24
```

Listing 1.3: ARM assembly program fragment. Symbolic inputs $a0, $a1, $b0, $b1 are stored in registers/locations r9, 0x7d, r10, 0x8d, resp. Registers r5, r6 hold the location addresses 0x7d, 0x8d. Symbols #0, #1, #2 denote constants.

Table 1. Symbolic transitions generated by executing Listing 1.3. ***Address*** is the address of an instruction in the listing, ***Location*** is the name of the instruction's target register. Symbols c_1, c_2, c_3 denote constants.

Address	*Location*	*Symbolic Transition* (old,new)
0x24	$r0	(c_1,$a0)
0x26	$r1	(c_2,$b1)
0x2a	$r0	($a0,$a0 * $b1)
0x2e	$r1	($b1,$b0)
0x30	$r4	(c_3,$a0 * $b1)
0x32	$r0	($a0 * $b1,$a1)
0x36	$r0	($a1,$a1 * $b0)

Consider Listing 1.3, the two secrets $a and $b and their corresponding sets of shares $Sh(\$a) = \{\$a0, \$a1\}$, $Sh(\$b) = \{\$b0, \$b1\}$. Table 1 lists the transitions generated by symbolically executing Listing 1.3. The instructions at Lines 2,3,6 don't generate interesting transitions because their target registers r0,r1,r4 contain constants before the instruction and thus no shares. The target registers of instructions at Lines 4 and 8 hold secret shares both before and after the transition; however, the before and the after value together do not involve *all* shares of secret $a, nor all shares of $b. Only instructions at Lines 5 and 7 involve all shares of one of the secrets (in red) and qualify as share-complete. Application of a leakage function to the contents of an address of a symbolic state generates a symbolic leakage expression. We need a leakage analysis method that can decide whether this expression statistically depends on the secret variable. Various techniques and metrics for qualitative and quantitative analysis of statistical dependence have been proposed in the literature [7, 16, 18, 19]. For now we present our GTBL detection method in a form parametric in the dependence analysis; we instantiate this parameter in Sect. 6.

Algorithm 3 shows our GTBL detection scheme implementing Definition 5. It uses the abstract predicate *SecretDep*(l) that decides whether a given symbolic leakage expression l statistically depends on the secret. The algorithm is *lazy* in the sense that it delays the (potentially expensive) secret dependence checks, in favor of the (syntactic) share-completeness checks: Line 1 checks condition (i) in Definition 5. The symmetric blocks starting in Lines 3 and 6 check condition (ii), by first determining share-completeness of the respective VBL expression and then, if necessary, check for statistical dependence on the secret, still for VBL. In Line 9 we know the TBL is potential; to determine its genuineness we must perform a secret dependence check.

Algorithm 3 is used at every step of the forward symbolic execution, to classify whether the location that is written to during that step constitutes a GTBL. The number of GTBL checks is thus linear in the number of states along the execution (instead of quadratic, as it would be for the alternative method of checking for second-order

Algorithm 3. GTBL checker

Require: Symbolic state s_i and its predecessor s_{i-1}, address a written at s_i.
Ensure: True if (s_{i-1}, s_i, a) is a GTBL; False otherwise.

```
if (s_{i-1}(a), s_i(a)) is share-incomplete then
    return False
if s_{i-1}(a) is share-complete then
    if SecretDep(V_L(s_{i-1})(a)) then
        return False
if s_i(a) is share-complete then
    if SecretDep(V_L(s_i)(a)) then
        return False
return SecretDep(T_L(s_{i-1}, s_i)(a))
```

```
10  ldr   r3, [r7, #28]
11  ldr   r2, [r7, #8]
12  add   r3, r2
13  ldrb  r3, [r3, #0]
14  mov   r1, #0
15  mov   r1, r3
16  bl    spamul
```

Listing 1.4: Flushing instruction eliminating the GTBL of L. 14 in List. 1.1.

```
7   and   r0, r0, r1, asr #7
8   mov   r7, #0
9   mov   r7, r6
10  mov   r6, #0
11  mul   r6, r7, r2
12  mov   r7, #0
13  mul   r2, r0, r3
```

Listing 1.5: Flushing gadget eliminating the GTBL of L. 8 in List. 1.2.

protection, by considering all possible pairs of states). Absence of GTBLs along the execution implies that the latter is first-order secure under TBL.

5 Repair of Transition-Based Leakage

TBL violates the independent leakage assumption via a register overwrite that combines the value computed at the current step and the value last stored in the register. To eliminate the TBL, it therefore suffices to disconnect the confluence of the two values by *flushing* the contents of the overwritten register. The flushing countermeasure has to be applied before the register write (i.e. before the *definition*, in compiler terminology) that manifests the leakage, yet after the previous definition of the same register. We refer to the TBL-inducing register as TBL register. Additionally, to guarantee that the original program's I/O semantics is maintained (correctness requirement), flushing has to be applied *after* the last read (i.e. *use*) of the TBL register, to ensure that uses see the original contents instead of the flushed ones.

Simply inserting a *flushing instruction*, such as one that sets the contents of the TBL register to 0 right before the leaky instruction, guarantees in most cases that the above conditions are met, and results in a straightforward and efficient countermeasure. Listing 1.4 shows how a flushing instruction eliminates the GTBL described in Sect. 3.1.

The flushing instruction approach assumes that the last use of the TBL register occurs at a different, earlier instruction in the program. This assumption is violated in the case of instructions that both use and define the TBL register, which we refer to as *update instructions*. Listing 1.5 shows a *flushing gadget* for repairing the TBL of the

Algorithm 4. Flushing Countermeasure

Require: Instruction `I` whose target register is to be flushed.
Ensure: Sequence of instructions that is I/O equivalent to `I` and GTBL-free.

```
1: r_d := Def(I)
2: if r_d ∉ Use(I) then
3:     return Flush(r_d); I
4: else
5:     r_t      := Dead()
6:     I_0;I_1 := Split(I, r_t)
7:     return Flush(r_t); I_0; Flush(r_d); I_1; Flush(r_t)
```

update instruction **`mul`** `r6, r6, r2` at Line 8 of Listing 1.2. The gadget splits the instruction and eliminates the use of the TBL register `r6`, as follows. First, it transfers the old contents of `r6` to an unused register, namely `r7`, in Line 9 of Listing 1.5.[1] Second, it performs the flush of the TBL register (Line 10). Third, it modifies the original leaky instruction, by replacing `r6` on the right-hand side (the use) by `r7` (Line 11), so that the modified instruction is equivalent to the original update. Finally, and critically, the gadget must rule out that `r7` accidentally introduce leakage through its previous or future contents. Therefore, `r7` is flushed at Lines 8 and 12.

We formalize our flushing countermeasure in Algorithm 4, which takes as input an instruction `I` whose target register is to be flushed, and makes use of the following subroutines:

Def(`I`):	the register instruction `I` defines;
Use(`I`):	the set of registers instruction `I` uses;
Flush(`r`):	given register r, return the instruction **`mov`** `r, #0`;
Split(`I`,r_t):	given an update instruction `I` (**`opc`** `ra,ra,rb`) and a temporary register r_t, return an I/O-equivalent sequence of two instructions (**`mov`** `rt,ra`; **`opc`** `ra,rt,rb`) that eliminate the use of the target register of `I`, by using the temporary register instead;
Dead():	an available register, i.e. one that is not used along any path between the instruction we are flushing and its next definition.

Theorem 6. *Application of the flushing countermeasure (Algorithm 4) to each GTBL-induc-ing instruction of an assembly program P results in an I/O-equivalent and GTBL-free assembly program P_f.*

Proof (Sketch): I/O-equivalence follows easily from the subroutine specifications given above. Flushing eliminates each GTBL (s_{i-1}, s_i, a) by splitting it into its corresponding non-genuine VBLs (s_{i-1}, a) and (s_i, a), and the zero leakage VBL (s_f, a), where s_f is the state introduced by Flush. □

[1] If an unused register is not available, we free a used register, by spilling its contents to memory.

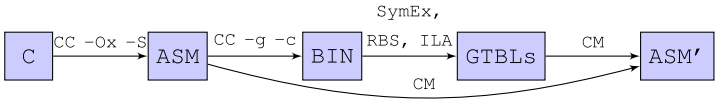


Fig. 3. BATTL's GTBL detection and repair pipeline. Arrows indicate the invocation of the components mentioned above them; boxes indicate the intermediate results produced by BATTL's flow.

6 BATTL: Binary Analysis for Transition-Based Leakage

6.1 Implementation

We have implemented the detection and repair of transition-based leakage in BATTL, our BINARY ANALYSIS TOOL FOR TRANSITION-BASED LEAKAGE [5], written in Python and C. Figure 3 shows the toolflow of BATTL. It expects a masked C program, compiles it to binary, reports the detected GTBLs, and outputs an I/O-equivalent and TBL-free masked assembly (asm) program. BATTL is parametric to the compiler used for generating the binary and asm programs, and requires no compiler modifications.

Detection. BATTL's symbolic execution component (SymEx) is built on top of the `angr` binary analysis platform [30] and uses the latter's symbolic execution engine on the compiled binaries to generate symbolic transitions. BATTL is imported as a library in user provided *drivers*, which are python programs specifying the inputs and secrets of the binary. We use two complementary techniques, in order of increasing hardness and precision, for the implementation of *SecretDep*. First, we adapt the *rule-based* approach of Gao et al. [18] to operate on bitvector expressions. This rule-based system (RBS) statically checks the symbolic leakage expressions for semantic properties that imply the distribution class they fall into, namely one of three classes: (i) `RUD`, denoting a random-uniform distribution; (ii) `SID`, denoting a secret-independent distribution, or (iii) `UKD`, denoting an unknown distribution. Class `RUD` is a subset of `SID`; both permit the conclusion of leakage-freedom. A `UKD` expression, however, may or may not leak; in this sense, RBS is incomplete.

To resolve this incompleteness, our implementation of *SecretDep* resorts to a second technique, the *Information Leakage Amount* (ILA) metric function [32]. ILA quantifies how much one secret value is distinguishable from the remaining secret values, by computing the averaged square of the L_2-distances of the Hamming weights of the f_L-measures that each different value of the secret gives rise to. ILA maps a symbolic expression to a real number in the range $[0,4]$ that signifies its leakage intensity. Zero ILA implies that the expression doesn't statistically depend on the secret; in this case *SecretDep* returns False. Positive ILA implies dependence on the secret; higher values indicate more leakage and thus easier exploitability. The upper bound of 4 derives from the number of bits of the secret variable divided by 2: we have 1-byte secrets, stored in 32-bit machine integers.

The calculation of ILA involves exhaustive enumeration of the possible values of the secret and random variables of the leakage expression and can therefore be expensive. Our implementation of *SecretDep*, which returns the ILA value (rather than just a

Boolean value, as suggested by Algorithm 3), uses a sampling-based approach to compute ILA for expressions with 4 or more variables. The ILA computation component is implemented as a C embedding in the original source code of the analyzed program.

Repair. The flushing countermeasure of Sect. 5 is implemented as a component of BATTL named CM. In BATTL's pipeline we instruct the compiler to generate the (possibly optimized) assembly code to which we add debug information and then compile to an executable binary. BATTL detects the binary's GTBLs and using debug information it reports the asm source lines and instruction that cause the GTBLs. The instructions and the assembly code are the input of CM, which applies the flushing countermeasure. To accommodate the Dead subroutine required by the flushing gadgets without the need of spilling, we perform a register liveness analysis during the SymEx phase.

6.2 Evaluation

We evaluate BATTL by attempting to validate the following hypotheses.

- **H1:** Genuine TBLs naturally occur in first-order secure implementations;
- **H2:** GTBL is not restricted to specific compilation toolchains or options;
- **H3:** GVBLs appear in binaries obtained from masked source code due to "careless" optimizations; they influence the number of reported GTBLs;
- **H4:** Flushing countermeasures are effective and incur only small overhead on the original implementation.

We present a thorough analysis of the above hypotheses using BATTL. We are not aware of other tools that are able to identify TBL at the binary level. The prior work on compiler-based elimination of TBL [31], which operates on `llvm` IR, does not offer a publicly available implementation.

Benchmarks. We test these hypotheses against benchmarks from two algorithms: SecInv [28] (shown in Sect. 3) and a masked version of AES [10,11,33]. Both algorithms are parametric in the protection order d, which we set to $d = 1$ since we focus on the detection of first-order TBL. While SecInv is nominally part of the AES SBox, the implementation of AES used in the evaluation of BATTL makes use of the *common share* [10] and *randomness reduction* [33] approaches and performs improved mask refreshing [11].[2] The SecInv benchmark is therefore not part of the AES benchmark.

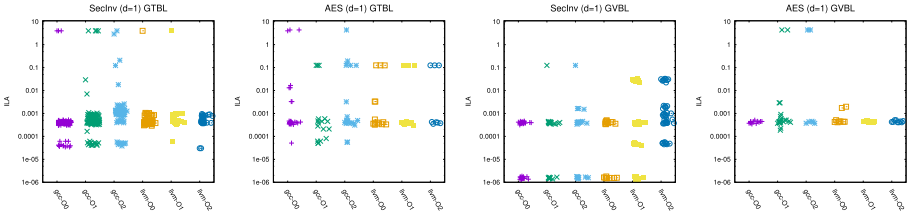
Our benchmarks consist of binaries generated from `C` implementations of SecInv (≈100 LOC) and AES (≈500 LOC) that target the ARM Cortex-M3 family of microprocessors and vary along two orthogonal axes: the optimization level (`O0`–`O2`) and the compilation toolchain (`gcc`, `llvm`). SecInv operates on 1 secret key byte, and AES on 16 secret key bytes. We instruct BATTL to symbolically execute the full SecInv binaries and identify leaks of its secret key. In the case of the AES binaries we stop after executing the first AES round (the algorithm consists of 10 rounds in total) and check for leaks stemming from the first key byte.[3]

[2] https://github.com/knarfrank/Higher-Order-Masked-AES-128.

[3] BATTL can be configured by the user to check for any number of bytes of the secret key. We choose one, the first, for presentation purposes.

Table 2. Rows 1–4: Potential and genuine TBL summary for SecInv and AES. #PTBL is the total number of PTBLs. The fraction of PTBLs shown to be secret-independent using the RBS is given in row #RUD/SID, that of PTBLs shown secret-independent using the ILA metric in row #ILA=0, and that of PTBLs shown to leak genuinely on account of a positive ILA in row #GTBL. Rows #RUD/SID, #ILA=0, #GTBL add up to row #PTBL. Rows 5–7: Wall-clock running time of BATTL's components. Rows RBS and ILA report the total runtime spent on the analysis of all PTBLs of each benchmark.

	SecInv (d=1)						AES (d=1)					
	gcc			llvm			gcc			llvm		
	-O0	-O1	-O2	-O0	-O1	-O2	-O0	-O1	-O2	-O0	-O1	-O2
1: #PTBL	442	262	194	309	200	121	156	92	113	108	90	92
2: #RUD/SID	263	143	106	222	132	71	121	67	85	87	69	75
3: #ILA=0	43	20	12	17	23	19	4	8	6	6	9	9
4: #GTBL	136	99	76	70	45	31	31	17	22	15	12	8
5: SymEx	1m57s	21s	20s	53s	27s	19s	6m08s	2m5s	2m27s	2m47s	1m55s	1m38s
6: RBS	3s	4s	3s	4s	2s	1s	1s	1s	1s	1s	1s	1s
7: ILA	4h26m	1h45m	1h4m	2h3m	1h56m	1h4m	1h53m	57m32s	46m29s	32m19s	36m22s	23m48s



(a) (b) (c) (d)

Fig. 4. Scatter plots of GTBLs and GVBLs for SecInv and AES benchmarks; each point represents a genuine leakage with the ILA value reported on the vertical axis.

Experimental Results. Table 2 summarizes the numbers of TBL leaks reported by BATTL across all benchmarks. It confirms **H1** and **H2** and shows that first-order source-code protection does not avert first-order TBL, as BATTL identified GTBLs across *all* configurations of optimization options and compiler toolchains. Figure 4 displays the distribution of different leakage amounts across the different experiments and different leakage points (code locations). We see that in most benchmarks the manifested leakage can be large. All of the SecInv benchmarks except for `llvm-O2` have at least one GTBL case near the maximum ILA value of 4, which is easily exploitable by an adversary [32]. For AES, 2/6 benchmarks show ILA values of 4, with the rest having a large count of GTBLs with ILA ≈ 0.1. Our experiments also suggest that `llvm`-generated binaries show fewer TBL points, with smaller ILA values, than those generated by `gcc`, for both algorithms across all flags, essentially providing a more narrow attack surface. As a possible explanation, we observed on our benchmarks that `llvm` aggressively unrolled loops and generated larger basic blocks and, therefore, altogether larger

Table 3. Performance analysis of repaired benchmarks. Columns FI, FG show the number of flushing instructions and flushing gadgets per benchmark, respectively. Columns CC show the number of clock cycles of the repaired benchmarks. Columns (%) show the percentage increase in cycles between the original and the repaired benchmarks.

	SecInv (d=1)								AES (d=1)							
	gcc				llvm				gcc				llvm			
	FI	FG	CC	(%)	FI	FG	CC	(%)	FI	FG	CC	(%)	FI	FG	CC	(%)
-O0	16	4	6640	7.58	8	3	4978	7.65	8	5	36847	5.47	3	4	28757	3.73
-O1	12	5	1856	19.88	8	2	2688	9.38	7	5	11599	9.24	1	3	13422	1.38
-O2	9	13	1645	15.74	11	2	1086	6.08	7	4	10781	7.96	4	2	9666	0.33

binaries. We believe this code layout approach leads to fewer TBLs as it (accidentally) promotes the use of more registers in a basic block.

Our experiments were performed on first-order source-code secure implementations and might therefore be expected by the unaware programmer to be free of value-based leakage. Our findings contradict this expectation and confirm **H3**: for SecInv the two points with ILA ≈ 0.12 in Fig. 4c for `gcc-O1/O2` correspond to the VBL due to expression reordering (explained in Sect. 3.2). `llvm-O1/O2` also shows measurable leakage with ILA ≈ 0.02. For AES, Fig. 4d `gcc-O1/O2` shows full leakage of the secret due to VBL with ILA ≈ 4. The remaining points in Fig. 4d have very small ILA values obtained via sampling, indicating that they might in fact be leak-free. The `llvm` compiler fares strictly better than `gcc`, regarding re-introduction of VBL. We attribute this to differently implemented expression rewriting (`-ftree-reassoc`, `-reassociate` flags) in the two toolchains.

Analysis Performance. Table 2 summarizes BATTL's performance. For each benchmark, we execute BATTL on a single core of a desktop Intel-i7-4770@3.40 GHz with 16 GB RAM. Block cipher code consists of loops with a constant number of iterations and lacks input-dependent control-flow variations. As a result, SymEx doesn't suffer from path explosion; its runtime variations across benchmarks result from the different program sizes. `angr`'s engine scaled well with increasing expression size: no benchmark required more than 6 GB of RAM for its analysis.

BATTL spends a negligible amount of time on the RBS module; the ILA module dominates the time required for deciding *SecretDep* for PTBLs. For each benchmark, the number of ILA checks is equal to the sum of i) #PTBL – #RUD/SID and ii) the number of ILA checks performed when determining the set of PTBLs caused by GVBL. The runtime of each ILA check depends on the number of variables present in the leakage expression (see Sect. 6.1).

Flushing Countermeasure and Overhead. We have used CM to flush all the GTBLs of our benchmarks in the compiler-generated assembly. We use the internal clock cycle

counter of ARM Cortex-M3 [24] to measure the number of clock cycles of one invocation of SecInv and one round of AES, shown in Table 3. CM eliminates GTBLs without an increase in the masking order and incurs small performance overhead, confirming **H4**. For comparison, we have measured second-order (d = 2) implementations to incur more than a 100% overhead. The difference in clock cycles between SecInv and AES is due to the larger size of the latter, which naturally results in longer execution times. SecInv is subject to larger overheads (%) compared to AES. Per Table 2, SecInv has a larger amount of GTBLs and consequently requires more flushing instructions and flushing gadgets, to eliminate the leakage. Spilling was required in 14/24 of the flushing gadgets applied to the AES benchmarks (it was never required in SecInv).

The slowdown (%) incurred by our countermeasure is comparable to that reported by the earlier compiler-based approach [31]. In contrast to that work, we can apply BATTL to the final repaired binary to confirm the effectiveness of our countermeasure. (Note that Theorem 6 applies only to the repaired assembly (ASM' in Fig. 3) and does not account for possible interference by later compilation stages, such as the linker.)

7 Related Work

The majority of related work targets the VBL model. They use formal techniques to verify the correctness of masking countermeasures [3,7,13,15,18] and correct-by-construction approaches that automatically mask the source code, either at compile time [4,6,25], or synthesized independently [14]. Some of the verification techniques can in principle be extended to the TBL model by verifying second-order masking (since a second-order secure implementation is free of first-order TBL, as shown in [1]), although their experimental evaluations show that the analysis does not scale [7,18].

One can think of our technique, dedicated to TBL, as a *focused* second-order analysis: instead of considering all pairs of intermediate variables, only those that constitute consecutive assignments to the same register are investigated. This vastly reduces the complexity of the analysis from quadratic to linear in the length of the execution path. Based on forward symbolic execution, our technique is a form of (path-)bounded analysis: it reports all leakages up to the point of the symbolic execution. It is suitable for code with modest control-flow variations, or code with loops of a constant number of iterations. Block ciphers, the main target of power attacks, enjoy such characteristics.

Only recent work has considered the issue of TBL explicitly [2,26,31]. The work closest to ours is the correct-by-construction approach of Wang et al. [31]. They present a sound static analysis (based on the same rule system [18] used in BATTL) that is applied on the `llvm` IR. To avoid considering all pairs of instructions in the IR (which lacks register information) they use additional static analysis to overapproximate pairs of instructions that share registers, and apply the rule system to these pairs. Any pair classified as `UKD` is considered sensitive. To eliminate leaks that could arise from sensitive pairs, they add constraints to the register allocation and DAG combination passes of `llvm` to disallow register overwrites and to eliminate update instructions respectively.

The notion of *sensitive pair* [31] is an overapproximation of our notion of PTBL (the latter are precise in terms of register overwrites) and can thus not prove the existence of leakages, only their absence. Being a purely static analysis, their approach favors

efficiency over precision. Their technique incurs small, if any, performance overhead. However, it is (i) susceptible to interference from the compilation framework after the countermeasure passes are applied, and (ii) bound to their modified `llvm` compiler. As our analysis is performed on binaries, BATTL doesn't suffer from the above issues: it can be run on the repaired binaries to confirm the absence of compiler interference, and it is compiler agnostic. The refined definition of a GTBL that accounts for leakages due to GVBLs and is unique to our methodology, crucially influences the security guarantees of countermeasures specialized for TBL. BATTL reports such cases to the user and doesn't apply countermeasures to them, since they would anyway be ineffective. The user is responsible for eliminating the GVBL that induces the TBL, before starting a fresh round of detection and repair. Wang et al. do not distinguish cases of TBL due to VBL that cannot be eliminated by TBL-specific countermeasures and as a result the binaries they produce are subject to leakage.

MaskVerif [2], a tool based on a relational verification approach [3], is aimed at the analysis of (very) high-order software implementations under VBL and TBL. It requires transformation of the masking algorithm to some intermediate representation, contrary to our approach that operates directly on binary programs. It is geared towards circuit implementations that operate on Boolean variables and therefore doesn't handle software implementations. These shortcomings are common among previously proposed methods: they operate on intermediate representations such as `llvm` IR [7,18,31] or other internal languages [2] and target Boolean programs [2,18]. Work that, like ours, operates on ARM assembly does not handle TBL or high-order implementations [13].

ASCOLD [26] is a tool for detecting transition-based leakages in AVR assembly. It keeps track of register overwrites and conservatively reports all cases that are share-complete. This leads to a vast amount of false positive leakage reports and unnecessary overhead when attempting to apply countermeasures. ASCOLD is restricted in the programs it can handle. It has been used on hand-written assembly code and supports but a fragment of the AVR assembly instruction set and it cannot be applied to arbitrary, compiler-generated binaries. Both MaskVerif and ASCOLD rely on manual insertion of countermeasures to TBL.

8 Conclusion and Future Work

We have presented a technique for the automated detection of transition-based leakage in software binaries, and a countermeasure for their repair. Our analysis showed that such leakage is prevalent in block ciphers and of high intensity among different compiler configurations. Our countermeasure was able to eliminate the reported leakages with a moderate performance overhead. By operating on binaries, our detection approach is sensitive to any leaks introduced by a leakage-oblivious compilation chain.

We leave the detection of transition effects through *memory writes*, in addition to register writes, as an immediate step for future work. In terms of repair, we believe that by using information provided by the forward symbolic execution we can devise even smaller gadgets that don't require flushing of registers used to temporarily hold values.

References

1. Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., Standaert, F.-X.: On the cost of lazy engineering for masked software implementations. In: Joye, M., Moradi, A. (eds.) CARDIS 2014. LNCS, vol. 8968, pp. 64–81. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16763-3_5
2. Barthe, G., Belaïd, S., Cassiers, G., Fouque, P.-A., Grégoire, B., Standaert, F.-X.: maskVerif: automated verification of higher-order masking in presence of physical defaults. In: Sako, K., Schneider, S., Ryan, P.Y.A. (eds.) ESORICS 2019. LNCS, vol. 11735, pp. 300–318. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29959-0_15
3. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.-A., Grégoire, B., Strub, P.-Y.: Verified proofs of higher-order masking. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 457–485. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_18
4. Barthe, G., et al.: Strong non-interference and type-directed higher-order masking. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 116–129. ACM (2016)
5. BATTL: Binary Analysis Tool for Transition-Based Leakage. https://gitlab.com/athanasiou.k/BATTL
6. Bayrak, A.G., Regazzoni, F., Brisk, P., Standaert, F.X., Ienne, P.: A first step towards automatic application of power analysis countermeasures. In: Proceedings of the 48th Design Automation Conference, pp. 230–235. ACM (2011)
7. 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). https://doi.org/10.1007/978-3-642-40349-1_17
8. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_26
9. Coron, J.-S., Giraud, C., Prouff, E., Renner, S., Rivain, M., Vadnala, P.K.: Conversion of security proofs from one leakage model to another: a new issue. In: Schindler, W., Huss, S.A. (eds.) COSADE 2012. LNCS, vol. 7275, pp. 69–81. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29912-4_6
10. Coron, J.-S., Greuet, A., Prouff, E., Zeitoun, R.: Faster evaluation of SBoxes via common shares. In: Gierlichs, B., Poschmann, A.Y. (eds.) CHES 2016. LNCS, vol. 9813, pp. 498–514. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53140-2_24
11. Coron, J.-S., Prouff, E., Rivain, M., Roche, T.: Higher-order side channel security and mask refreshing. In: Moriai, S. (ed.) FSE 2013. LNCS, vol. 8424, pp. 410–424. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43933-3_21
12. D'Silva, V., Payer, M., Song, D.: The correctness-security gap in compiler optimization. In: 2015 IEEE Security and Privacy Workshops (SPW), pp. 73–87. IEEE (2015)
13. El Ouahma, I.B., Meunier, Q.L., Heydemann, K., Encrenaz, E.: Symbolic approach for side-channel resistance analysis of masked assembly codes. In: Security Proofs for Embedded Systems (2017)
14. Eldib, H., Wang, C.: Synthesis of masking countermeasures against side channel attacks. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 114–130. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_8
15. Eldib, H., Wang, C., Schaumont, P.: Formal verification of software countermeasures against side-channel attacks. ACM Trans. Softw. Eng. Methodol. (TOSEM) **24**(2), 11 (2014)

16. Eldib, H., Wang, C., Taha, M., Schaumont, P.: QMS: evaluating the side-channel resistance of masked software from source code. In: Proceedings of the 51st Annual Design Automation Conference, pp. 1–6. ACM (2014)
17. 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). https://doi.org/10.1007/3-540-44709-1_21
18. Gao, P., Zhang, J., Song, F., Wang, C.: Verifying and quantifying side-channel resistance of masked software implementations. ACM Trans. Softw. Eng. Methodol. (TOSEM) **28**(3), 16 (2019)
19. Gierlichs, B., Batina, L., Tuyls, P., Preneel, B.: Mutual information analysis. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 426–442. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85053-3_27
20. Knudsen, L.R., Robshaw, M.: The Block Cipher Companion. Springer Science & Business Media, Heidelberg (2011). https://doi.org/10.1007/978-3-642-17342-4
21. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_25
22. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
23. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards, vol. 31. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-38162-6
24. Cortex-M4 Technical Reference Manual: Data watchpoint and trace unit. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337h/BIIFBHIF.html
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). https://doi.org/10.1007/978-3-642-33027-8_4
26. Papagiannopoulos, K., Veshchikov, N.: Mind the gap: towards secure 1st-order masking in software. In: Guilley, S. (ed.) COSADE 2017. LNCS, vol. 10348, pp. 282–297. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64647-3_17
27. Renauld, M., Standaert, F.-X., Veyrat-Charvillon, N., Kamel, D., Flandre, D.: A formal study of power variability issues and side-channel attacks for nanoscale devices. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 109–128. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20465-4_8
28. Rivain, M., Prouff, E.: Provably Secure Higher-Order Masking of AES. Cryptology ePrint Archive, Report 2010/441 (2010), https://eprint.iacr.org/2010/441
29. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979). https://doi.org/10.1145/359168.359176
30. Shoshitaishvili, Y., et al.: SOK: (state of) the art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy (2016)
31. Wang, J., Sung, C., Wang, C.: Mitigating Power Side Channels during Compilation. arXiv preprint arXiv:1902.09099 (2019)
32. Zhang, L., Ding, A.A., Fei, Y., Luo, P.: A unified metric for quantifying information leakage of cryptographic devices under power analysis attacks. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9453, pp. 338–360. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48800-3_14
33. Zhang, R., Qiu, S., Zhou, Y.: Further improving efficiency of higher order masking schemes by decreasing randomness complexity. IEEE Trans. Inf. Forensics Secur. **12**(11), 2590–2598 (2017)