Formally Proved Security of Assembly Code Against Power Analysis

A Case Study on Balanced Logic

Pablo Rauzy — Sylvain Guilley — Zakaria Najm
Institut Mines-Télécom ; Télécom ParisTech ; CNRS LTCI
{*firstname.lastname*}@telecom-paristech.fr

Abstract

In his keynote speech at CHES 2004, Kocher advocated that side-channel attacks were an illustration that formal cryptography was not as secure as it was believed because some assumptions (*e.g.*, no auxiliary information is available during the computation) were not modeled. This failure is caused by formal methods' focus on models rather than implementations. In this paper we present formal methods and tools for designing protected code and proving its security against power analysis. These formal methods avoid the discrepancy between the model and the implementation by working on the latter rather than on a high-level model. We then demonstrate our methods in a case study in which we generate a provably protected PRESENT implementation for an 8-bit AVR smartcard.

1 Introduction

The need to trust code is a clear and proved fact, but the code itself needs to be proved before it can be trusted. In applications such as cryptography or real-time systems, formal methods are used to prove functional properties on the critical parts of the code. Specifically in cryptography, some non-functional properties are also important, but are not typically subject to formal proof yet. Side-channel attacks are a real world threat to cryptosystems; they use auxiliary information gathered from implementations though physical channels such as power consumption, electromagnetic radiations, or time. The amount of leaked information depends on the implementation and as such appears difficult to model. As a matter of fact, physical leakages are usually not modeled when it comes to prove the security properties of a cryptographic algorithm. By applying formal methods directly on implementations we can avoid the discrepancy between the model and the implementation. Formally proving non-functional security properties then becomes a matter of modeling the leakage itself. In this paper we make a first step towards formally trustable cryptosystems, including for non-functional properties, by showing that modeling leakage and applying formal methods to implementations is feasible.

Many existing countermeasures against side-channel attacks are implemented at the hardware level, especially as far as smartcards are concerned. However, software level countermeasures are also very important, not only in embedded systems where the hardware cannot always be modified or updated, but also in the purely software world. For example, Zhang *et al.* [39] recently extracted private keys using side-channel attacks against a target virtual machine running on the same physical server as their virtual machine. Side-channel in software can also be found each

time there are some non-logic behaviors (in the sense that it does not appear in the equations / control-flow modeling the program) such as timing or power consumption (refer to [18]), but also some software-specific information such as packet size for instance (refer to [25]).

In many cases where the cryptographic code is executed on secure elements (smartcards, TPM, tokens, *etc.*) side-channel and fault analyses are the most natural attack paths. A combination of signal processing and statistical techniques on the data obtained by side-channel analysis allows to build key hypotheses distinguishers. The protection against those attacks is necessary to ensure that secrets do not leak, and most secure elements are thus evaluated against those attacks. Usual certifications are the common criteria (ISO/IEC 15408), the FIPS 140-2 (ISO/IEC 19790), or proprietary schemes (EMVCo, CAST, *etc.*).

Power analysis. It is a form of side-channel attack in which the attacker measures the power consumption of a cryptographic device. *Simple Power Analysis* (SPA) consists in directly interpreting the electrical activity of the cryptosystem. On unprotected implementations it can for instance reveal the path taken by the code at branches even when timing attacks [17] cannot. *Differential Power Analysis* [19] (DPA) is more advanced: the attacker can compute the intermediate values within cryptographic computations by statistically analyzing data collected from multiple cryptographic operations. It is powerful in the sense that it does not require a precise model of the leakage, and thus works blind, *i.e.*, even if the implementation is blackbox. As suggested in the original DPA paper by Kocher *et al.* [18], power consumption is often modeled by Hamming weight of values or Hamming distance of values' updates as those are very correlated with actual measures. Also, when the leakage is little noisy and the implementation is software, *Algebraic Side-Channel Attacks* (ASCAs [31]) are possible; they consist in modelling the leakage by a set of Boolean equations, where the key bits are the only unknown variables [5].

Thwarting side-channel analysis is complicated since an unprotected implementation leaks at every step. Simple and powerful attacks manage to exploit any bias. In practice, there are two ways to protect cryptosystems: "palliative" versus "curative" countermeasures. Palliative countermeasures attempt to make the attack more difficult, however without a theoretical foundation. They include variable clock, operations shuffling, and dummy encryptions among others (see also [15]). The lack of theoretical foundation make these countermeasures hard to formalize and thus not suitable for a safe certification process. Curative countermeasures aim at providing a leak-free implementation based on a security rationale. The two defense strategies are (*a*) make the leakage as decorrelated from the manipulated data as possible (masking [22, Chp. 9]), or (*b*) make the leakage constant, irrespective of the manipulated data (hiding or balancing [22, Chp. 7]).

Masking. Masking mixes the computation with random numbers, to make the leakage (at least in average) independent of the sensitive data. Advantages of masking are (*a priori*) the independence with respect to the leakage behavior of the hardware, and the existence of provably secure masking schemes [33]. There are two main drawbacks to masking. First of all, there is the possibility of high-order attacks (that examine the variance or the joint leakage); when the noise is low, ASCAs can be carried out on one single trace [32], despite the presence of the masks, that are just seen as more unknown variables, in addition to the key. Second, masking demands a greedy requirement for randomness (that is very costly to generate). Another concern with masking is the overhead it incurs in the computation time. For instance, a provable masking of AES-128 is reported in [33] to be 43 (resp. 90) times slower than the non-masked implementation with a 1st (resp. 2nd) order masking scheme. Further, recent studies have shown that masking cannot be analyzed independently from the execution platform: for example *glitches* are transient leakages that are likely to depend on

more than one sensitive data, hence being high-order [24]. Indeed, a glitch occurs when there is a race between two signals, *i.e.*, when it involves more than one sensitive variable. Additionally, the implementation must be carefully scrutinized to check for the absence of *demasking* caused by overwriting a masked sensitive variable with its mask.

Balancing. Balancing requires a close collaboration between the hardware and the software: two indistinguishable resources, from a side-channel point of view, shall exist and be used according to a dual-rail protocol. Dual-rail with precharge logic (DPL) consists in precharging both resources, so that they are in a common state, and then setting one of the resources. Which resource has been set is unknown to the attacker, because both leak in indistinguishable ways (by hypothesis). This property is used by the DPL protocol to ensure that computations can be carried out without exploitable leakage [38].

Contributions. The two main contributions of this paper are: (*a*) a design method for developing balanced assembly code by transforming it to make it obey the dual-rail with precharge logic (DPL) protocol, and a tool that implements this method; (*b*) a formal method (using symbolic execution) to statically prove the absence of power consumption leakage in assembly code (provided that the hardware it runs on satisfies a finite and limited set of requirements), and a tool which computes these proofs. We also provide a case study on a software implementation of the PRESENT [3] cipher running on an AVR micro-controller. DPL is a simple protocol that may look easy to implement correctly; however, in the current context of awareness about cyber-threats, it becomes evident that (independent) formal tools that are able to *generate* and *verify* a "trusted" implementation have a strong value.

Related work. The use of formal methods is not widespread in the domain of implementations security. When they exist, security proofs are done on mathematical models rather than implementations. An emblematic example is the Common Criteria [8], that bases its "formal" assurance evaluation levels on "Security Policy Model(s)" (class SPM) and not on implementation-level proofs. This means that it is the role of the implementers to ensure that their implementations fit the model, which is usually done by hand and is thus error-prone. For instance, some masking implementations have been proved; automatic tools for the insertion of masked code have even been prototyped [28]. However, masking relies on randomness, which is a rare resource and is hard to formally capture. Thus, many aspects of the security are actually displaced in the randomness requirement rather that soundly proved. Moreover, in the field of masking, most proofs are still literate (*i.e.*, verified manually, not by a computer program). This has led to a recent security breach in what was supposed to be a proved [33] masking implementation [6]. Previous similar examples exist, *e.g.*, the purported high-order masking scheme [34], defeated one year after in [9].

Timing and cache attacks are an exception as they benefit from the work of Köpf *et al.* [20, 21]. Their tool, CacheAudit [11], implements formal methods that directly work on x86 binaries.

In this light it is easy to conclude that the use of formal methods to prove the security of implementations against power analysis is a need, and a technological enabler: it would guarantee that the instantiations of security principles are as strong as the security principles themselves.

Organization of the paper. The DPL countermeasure is studied in Sec. 2. Sec. 3 details our method to balance assembly code and prove that the transformation is correct. Sec. 4 explains the formal methods used to compute a proof of the absence of power consumption leakage. Sec. 5 is a practical case study using the PRESENT algorithm on an AVR micro-controller. Conclusions and perspectives are drawn in Sec. 6.

2 Dual-Rail with Precharge Logic

Balancing (or hiding) countermeasures have been employed against side-channel since early 2004, with dual-rail with precharge logic. The DPL countermeasure consists in computing on a redundant representation: each bit y is implemented as a pair $(y_{\text{False}}, y_{\text{True}})$. The bit pair is then used in a protocol made up of two phases:

1. a *precharge* phase, during which all the bit pairs are zeroized $(y_{\text{False}}, y_{\text{True}}) = (0, 0)$, such that the computation starts from a known reference state;
2. an *evaluation* phase, during which the pair $(y_{\text{False}}, y_{\text{True}})$ is equal to $(1, 0)$ if it carries the logical value 0, or $(0, 1)$ if it carries the logical value 1.

The value $(y_{\text{False}}, y_{\text{True}}) = (1, 1)$ is unused. As suggested in [27], it can serve as a *canary* to detect a fault.

2.1 State of the Art

Various DPL styles for electronic circuits have been proposed. Some of them, implementing the same logical *and* functionality, are represented in Fig. 1; many more variants exist, but these four are enough to illustrate our point. The reason for the multiplicity of styles is that the indistinguishability hypothesis on the two resources holding y_{False} and y_{True} values happens to be violated for various reasons, which leads to the development of dedicated hardware. A first asymmetry comes from the gates driving y_{False} and y_{True}. In wave dynamic differential logic (WDDL [36]), these two gates are different: logical *or* versus logical *and*. Other logic styles are balanced with this respect. Then, the load of the gate shall also be similar. This can be achieved by careful place-and-route constraints [37, 14], that take care of having lines of the same length, and that furthermore do not interfere one with the other (phenomenon called "crosstalk"). As those are complex to implement exactly for all secure gates, the masked dual-rail with precharge logic (MDPL [30]) style has been proposed: instead of balancing exactly the lines carrying y_{False} and y_{True}, those are randomly swapped, according a random bit, represented as a pair $(m_{\text{False}}, m_{\text{True}})$ to avoid it from leaking. Therefore, in this case, not only the computing gates are the same (*viz.* a majority), but the routing is balanced thanks to the extra mask. However, it appeared that another asymmetry could be fatal to WDDL and MDPL: the gates pair could evaluates at different dates, depending on their input. It is important to mention that side-channel acquisitions are very accurate in timing (off-the-shelf oscilloscopes can sample at more than 1 Gsample/s, *i.e.*, at a higher rate than the clock period), but very inaccurate in space (*i.e.*, it is difficult to capture the leakage of an area smaller than about $1\,\text{mm}^2$ without also recording the leakage from the surrounding logic). Therefore, two bits can hardly be measured separately. To avoid this issue, every gate has to include some synchronization logic. In Fig. 1, the "computation part" of the gates is represented in a grey box. The rest is synchronization logic. In

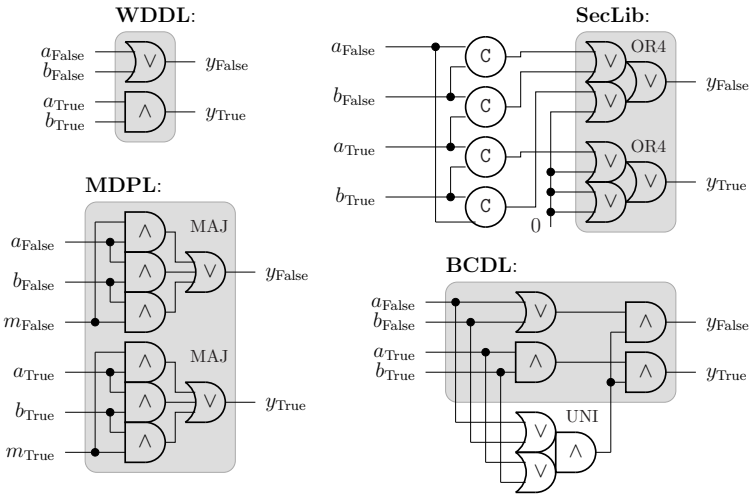


Figure 1: Four dual-rail with precharge logic styles.

SecLib [13], the synchronization can be achieved by Muller C-elements (represented with a symbol `C` [35]), and act as a decoding of the inputs configuration. Another implementation, balanced cell-based differential logic (BCDL [29]), parallelize the synchronization with the computation.

2.2 DPL in Software

In this paper, we want to run DPL on an off-the-shelf processor. Therefore, we must: (*a*) identify two similar resources that can hold true and false values in an indiscernible way for a side-channel attacker; (*b*) play the DPL protocol by ourselves, in software. We will deal with the former in Sec. 4.2. The rest of this section deals with the latter.

The difficulty of balancing the gates in hardware implementations is simplified in software. Indeed in software there are less resources than the thousands of gates that can be found in hardware (aimed at computing fast, with parallelism); thus the attention for the balancing can be focused. Also, there is no such problem as early evaluation, since the processor executes one operation after the other; therefore there are no unbalanced paths in timing. However, as noted by Hoogvorst *et al.* [16], standard micro-processors cannot be used *as is* for our purpose: instructions may clobber the destination operand without precharge; arithmetic and logic instructions generate numbers of 1 and 0 which depend on the data.

To reproduce the DPL protocol in software requires to (*a*) work at the bit level and (*b*) to duplicate (in positive and negative logic) the bit values. Every algorithm can be transformed so that all the manipulated values are bits (by the theorem of equivalence of universal Turing machines), so (*a*) is not a problem. Regarding (*b*), the idea is to use two bits in each register / memory cell to represent the logical value it holds. For instance using the two least significant bits, the logical value 1 could be encoded as 1 (`01`) and the logical value 0 as 2 (`10`). Then, any function on those bit values can be computed by a look-up table indexed by the concatenation of its operands. Each sensitive instruction can be replaced by a *DPL macro* which does the necessary precharge and fetch the result from the corresponding look-up table.

Fig. 2 shows a DPL macro for the computation of $d = a$ op b, using the two least significant bits for the DPL encoding. The register r_0 is an always-zero register, a and b hold one DPL encoded bit, and *op* is the address in memory of the look-up table for the op operation.

This DPL macro assumes that before it starts the state of the program is a valid DPL state (*i.e.*, that a and b are of the form `/.+(01|10)/`) and leaves it in a valid DPL state to make the macros chainable.

The precharge instructions (like $r_1 \leftarrow r_0$) erase the content of their destination register or memory cell before use. If the erased datum is sensitive it is DPL encoded, thus the number of bit flips (*i.e.*, the Hamming distance of the update) is independent of the sensitive value. If the erased value is not sensitive (for example the round counter of a block cipher) then the number of bit flips is irrelevant. In both cases the power consumption provides no sensitive information.

$$
\begin{array}{rcl}
r_1 & \leftarrow & r_0 \\
r_1 & \leftarrow & a \\
r_1 & \leftarrow & r_1 \wedge 3 \\
r_1 & \leftarrow & r_1 \ll 1 \\
r_1 & \leftarrow & r_1 \ll 1 \\
r_2 & \leftarrow & r_0 \\
r_2 & \leftarrow & b \\
r_2 & \leftarrow & r_2 \wedge 3 \\
r_1 & \leftarrow & r_1 \vee r_2 \\
r_3 & \leftarrow & r_0 \\
r_3 & \leftarrow & op[r_1] \\
d & \leftarrow & r_0 \\
d & \leftarrow & r_3
\end{array}
$$

Figure 2: DPL macro for $d = a$ op b.

The activity of the shift instructions (like $r_1 \leftarrow r_1 \ll 1$) is twice the number of DPL encoded bits in r_1 (and thus does not depend on the value when it is DPL encoded). The two most significant bit are shifted out and must be 0, *i.e.*, they cannot encode a DPL bit. The logical *or* instruction (as in $r_1 \leftarrow r_1 \vee r_2$) has a constant activity of one bit flip due to the alignment of its operands. The logical *and* instructions (like $r_1 \leftarrow r_1 \wedge 3$) flips as many bits as there are 1s after the two least significant bits (it's normally all zeros).

Accesses from/to the RAM (as in $r_3 \leftarrow op[r_1]$) cause as many bit flips as there are 1s in the transferred data, which is constant when DPL encoded. Of course, the position of the look-up table in the memory is also important. In order not to leak information during the addition of the offset ($op + r_1$ in our example), op must be a multiple of 16 so that its four least significant bits are 0 and the addition only flips the number of bits at 1 in r_1, which is constant since at this moment r_1 contains the concatenation of two DPL encoded bit values.

We could use other bits to store the DPL encoded value, for example the least and the third least significant bits. In this case a and b have to be of the form `/.+(0.1|1.0)/`, only one shift instruction would have been necessary, and the `and` instructions' mask would be 5 instead on 3.

3 Generation of DPL Protected Assembly Code

Here we present a generic method to protect assembly code against power analysis. To achieve that we implemented a tool (See App. A) which transforms assembly code to make it compliant with the DPL protocol described in Sec. 2.2. To be as universal as possible the tool works with a generic assembly language presented in Sec. 3.1. The details of the code transformation are given in Sec. 3.2. Finally, a proof of the correctness of this transformation is presented in Sec. 3.3.

3.1 Generic Assembly Language

Our assembly language is generic in that it uses a restricted set of instructions that can be mapped to and from virtually any actual assembly language. It has classical features of assembly languages: logical and arithmetical instructions, branching, labels, direct and indirect addressing. Fig. 3 gives the BNF of the language while Fig. 4 gives the equivalent code of Fig. 2 as an example of its usage.

```
Prog    ::= ( Label? Inst? ( ';' <comment> )? '\n' )*
Label   ::= <label-name> ':'
Inst    ::= Opcode0
          | Branch1 Addr
          | Opcode2 Lval Val
          | Opcode3 Lval Val Val
          | Branch3 Val Val Addr
Opcode0 ::= 'nop'
Branch1 ::= 'jmp'
Opcode2 ::= 'not' | 'mov'
Opcode3 ::= 'and' | 'orr' | 'xor' | 'lsl' | 'lsr' | 'add' | 'mul'
Branch3 ::= 'beq' | 'bne'
Val     ::= Lval | '#' <immediate-value>
Lval    ::= 'r' <register-number>
          | '@' <memory-address>
          | '!' Val ( ',' <offset> )?
Addr    ::= '#' <absolute-code-address>
          | <label-name>
```

Figure 3: Generic assembly syntax.

```
mov r1 r0
mov r1 a
and r1 r1 #3
lsl r1 r1 #1
lsl r1 r1 #1
mov r2 r0
mov r2 b
and r2 r2 #3
orr r1 r1 r2
mov r3 r0
mov r3 !r1,op
mov d r0
mov d r3
```

Figure 4: DPL macro of Fig. 2 in assembly.

The semantics of the instructions are intuitive. For `Opcode2` and `Opcode3` the first operand is the destination and the other are the arguments. The `mov` instruction is used to copy registers, load a value from memory, or store a value to memory depending on the form of its arguments. We remark that the instructions use the "`instr dest op1 op2`" format, which allows to map similar instructions from 32-bit processors directly, as well as instructions from 8-bit processors which only have two operands, by using the same register for `dest` and `op1` for instance.

3.2 Code Transformation

Bitsliced code. As seen in Sec. 2, DPL works at the bit level. Transforming code to make it DPL compliant thus requires this level of granularity. Bitslicing is possible on any algorithm[1], but we found that bitslicing an algorithm is hard to do automatically. In practice, every bitslice implementations we found were hand-crafted. However, since Biham presented his bitslice paper [2], many block ciphers have been implemented in bitslice for performance reasons, which mitigate this concern. So, for the sake of simplicity, we assume that the input code is already bitsliced.

DPL macros expansion. This is the main point of the transformation of the code.

Definition 1 (Sensitive value)**.** A value is said *sensitive* if it depends on sensitive data. A sensitive data depends on both the secret key and the plaintext (as usually admitted in the "*only computation leaks*" paradigm; see for instance [33, §4.1]).

Definition 2 (Sensitive instruction)**.** A *sensitive instruction* is an instruction which may modify the Hamming weight of a sensitive value.

All the sensitive instructions, must be expanded to a DPL macro. Thus, all the sensitive data must be transformed too. Each literal ("immediate" values in assembly terms), memory cells that contain initialized constant data (look-up tables, etc.), and registers values need to be DPL encoded. For instance, using the two least significant bits, the 1s stay 1s (`01`) and the 0s become 2s (`10`).

Since the implementation is bitsliced, only the logical (bit level) operators are used on sensitive instructions (`and`, `or`, `xor`, `lsl`, `lsr`, and `not`). To respect the DPL protocol, `not` instructions are replaced by `xor` which inverse the positive logic and the negative logic bits of DPL encoded values. For instance if using the two least significant bits for the DPL encoding, `not` a b is replaced by `xor` a b `#3`. Bitsliced code never needs to use shift instructions since all bits are directly accessible.

Thus, only `and`, `or`, and `xor` instructions need to be expanded to macros as the one shown in Fig. 4. This macro has the advantage that it actually uses two operands instructions only (when there are three operands in our generic assembly language, the destination is the same as one of the two others), which makes its instructions mappable one-to-one even with 8-bit assembly languages.

Look-up tables. As they appear in the DPL macro, the addresses of look-up tables are sensitive too. As seen in Sec. 2.2, the look-up tables must be located at an address which is a multiple of 16 so that the last four bits are available when adding the offset (in the case where we use the last four bits to place the two DPL encoded operands). Fig. 5 present the 16 values present in the look-up tables for `and`, `or`, and `xor`.

Address	0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111
`and`	00 , 00 , 00 , 00 , 00 , 01 , 10 , 00 , 00 , 10 , 10 , 00 , 00 , 00 , 00 , 00
`or`	00 , 00 , 00 , 00 , 00 , 01 , 01 , 00 , 00 , 01 , 10 , 00 , 00 , 00 , 00 , 00
`xor`	00 , 00 , 00 , 00 , 00 , 10 , 01 , 00 , 00 , 01 , 10 , 00 , 00 , 00 , 00 , 00

Figure 5: Look-up tables for `and`, `or`, and `xor`.

Values in the look-up tables which are not at DPL valid addresses, *i.e.*, addresses which are not a concatenation of `01` or `10` with `01` or `10`, are preferentially DPL invalid, *i.e.*, `00` or `11`. Like this if an error occurs during the execution (such as a fault injection for instance) it poisons the result and all the subsequent computations will be faulted too (infective computation).

[1]Intuitively, the proof invokes the Universal Turing Machines equivalence (those that work with only $\{0, 1\}$ as alphabet are as powerful as the others)

3.3 Correctness Proof of the Transformation

Formally proving the correctness of the transformation requires to define what we intend by "correct". Intuitively, it means that the transformed code does the "same thing" as the original one.

Definition 3 (Correct DPL transformation)**.** Let S be a valid state of the system (values in registers and memory). Let c be a sequence of instructions of the system. Let $\widehat{S}$ be the state of the system after the execution of c with state S, we denote that by $S \xrightarrow{c} \widehat{S}$. We write $dpl(S)$ for the DPL state (with DPL encoded values of the 1s and 0s in memory and registers) equivalent to the state S.
We say that c' is a *correct DPL transformation* of the code c if $S \xrightarrow{c} \widehat{S} \implies dpl(S) \xrightarrow{c'} dpl(\widehat{S})$.

Proposition 1 (Correctness of our code transformation)**.** *The expansion of the sensitive instructions into DPL macros such as presented in Sec. 2.2 is a correct DPL transformation.*

Proof. Let a and b be instructions. Let c be the code $a; b$ (instruction a followed by instruction b). Let X, Y, and Z be states of the program. If we have $X \xrightarrow{a} Y$ and $Y \xrightarrow{b} Z$, then we know that $X \xrightarrow{c} Z$ (by *transitivity*).

Let a' and b' be the DPL macro expansions of instructions a and b. Let c' be the DPL transformation of code c. Since the expansion into macros is done separately for each sensitive instruction, without any other dependencies, we know that c' is $a'; b'$. If we have $dpl(X) \xrightarrow{a'} dpl(Y)$ and $dpl(Y) \xrightarrow{b'} dpl(Z)$, then we know that $dpl(X) \xrightarrow{c'} dpl(Z)$.

This means that a chain of correct transformations is a correct transformation. Thus, we only have to show that the DPL macro expansion is a correct transformation.

Let us start with the `and` operation. Since the code is bitsliced, there are only four possibilities. Fig. 6 shows these possibilities for the `and d a b` instruction.

Fig. 8 shows the evolution of the values of a, b, and d during the execution of the macro which `and d a b` expands to. We assume the look-up table for `and` is located at address *and*. Fig. 7 sums up the Fig. 8 in the same format as Fig. 6.

	a, b, d			
Before	0, 0, ?	0, 1, ?	1, 0, ?	1, 1, ?
After	0, 0, 0	0, 1, 0	1, 0, 0	1, 1, 1

Figure 6: `and d a b`.

a, b, d			
10, 10, ?	10, 01, ?	01, 10, ?	01, 01, ?
10, 10, 10	10, 01, 10	01, 10, 10	01, 01, 01

Figure 7: DPL `and d a b` (see Fig. 8).

This proves that the DPL transformation of the `and` instructions are correct. The demonstration is similar for `or` and `xor` operations. □

The automatic DPL transformation of arbitrary assembly code has been implemented in our tool described in App. A.

4 Formally Proving the Absence of Leakage

Now that we know the DPL transformation is correct, we need to prove its efficiency security-wise. We prove the absence of leakage on the software, while obviously the leakage heavily depends on the hardware. Our proof thus makes an hypothesis on the hardware: we suppose that the bits we use for the positive and negative logic in the DPL protocol leak the same amount. This may seem like an unreasonable hypothesis, since it is not true in general. However, the protection can be implemented in a soft CPU core (LatticeMicro32, OpenRISC, LEON2, etc.), that would be

a, b	10, 10				10, 01				01, 10				01, 01			
	d	r1	r2	r3	d	r1	r2	r3	d	r1	r2	r3	d	r1	r2	r3
`mov r1 r0`	?	0	?	?	?	0	?	?	?	0	?	?	?	0	?	?
`mov r1` a	?	10	?	?	?	10	?	?	?	01	?	?	?	01	?	?
`and r1 r1 #3`	?	10	?	?	?	10	?	?	?	01	?	?	?	01	?	?
`shl r1 r1 #1`	?	100	?	?	?	100	?	?	?	010	?	?	?	010	?	?
`shl r1 r1 #1`	?	1000	?	?	?	1000	?	?	?	0100	?	?	?	0100	?	?
`mov r2 r0`	?	1000	0	?	?	1000	0	?	?	0100	0	?	?	0100	0	?
`mov r2` b	?	1000	10	?	?	1000	01	?	?	0100	10	?	?	0100	01	?
`and r2 r2 #3`	?	1000	10	?	?	1000	01	?	?	0100	10	?	?	0100	01	?
`orr r1 r1 r2`	?	1010	10	?	?	1001	01	?	?	0110	10	?	?	0101	01	?
`mov r3 r0`	?	1010	10	0	?	1001	01	0	?	0110	10	0	?	0101	01	0
`mov r3 !r1,`and[1]	?	1010	10	10	?	1001	01	10	?	0110	10	10	?	0101	01	01
`mov` d `r0`	0	1010	10	10	0	1001	01	10	0	0110	10	10	0	0101	01	01
`mov` d `r3`	10	1010	10	10	10	1001	01	10	10	0110	10	10	01	0101	01	01

[1] see Fig. 5

Figure 8: Execution of the DPL macro expanded from `and d a b`.

laid out in an FPGA or in an ASIC with special balancing constraints at place-and-route. The methodology follows the guidelines given by Chen *et al.* in [7]. Moreover, we will show in Sec. 4.2 how it is possible, using stochastic profiling, to find bits which leakages are similar enough for the DPL countermeasure to be sufficiently efficient even on non-specialized hardware. That said, it is important to note that the difference in leakage between two bits of the same register cannot be large enough for the attacker to break the DPL protection using SPA or ASCA.

Formally proving the balance of DPL code requires to properly define the notions we are using.

Definition 4 (Leakage model)**.** The attacker can measure the power consumption of parts of the cryptosystem. We model power consumption by the Hamming distance of values updates, *i.e.*, the number of bit flips. It is a commonly accepted model for power analysis, for instance with DPA [19] or CPA [4]. We write $H(a, b)$ the Hamming distance between the values a and b.

Definition 5 (Constant activity)**.** The activity of a cryptosystem is said to be constant if its power consumption does not depend on the sensitive data and is thus always the same.
Formally, let $P(s)$ be a program which has s as parameter (*e.g.*, the key and the plaintext). According to our leakage model, a program $P(s)$ is of *constant activity* if:

- for every values s_1 and s_2 of the parameter s, for each cycle i, for every sensitive value v, v is updated at cycle i in the run of $P(s_1)$ if and only if it is in the run of $P(s_2)$;
- whenever an instruction modifies some sensitive value from v to v', then the value of $H(v, v')$ does not depend on s.

4.1 Computed Proof of Constant Activity

To statically determine if the code is correctly balanced (*i.e.*, that the activity of a given program is constant according to Def. 5), our tool relies on symbolic execution. The idea is to run the code of the program independently of the sensitive data. This is achieved by computing on sets of all the possible values instead of values directly. The symbolic execution terminates in our case because we are using the DPL protection on block ciphers, and we avoid combinatorial explosion thanks to bitslicing, as a value can initially be only 1 or 0 (or rather their DPL encoded counterparts).

Our tool implements an interpreter for our generic assembly language which work with sets of values. The interpreter is equipped to measure all the possible Hamming distances of each value update, and all the possible Hamming weight of values. It watches updates in registers, in memory, and also in address buses (since the addresses may leak information when reading in look-up tables). If for one of these value updates there are different possible Hamming distances or Hamming weight,

then we consider that there is a leak of information: the power consumption activity is not constant according to Def. 5.

Example. Let a be a register which can initially be either 0 or 1. Let b be a register which can initially be only 1. The execution of the instruction `orr` a a b will set the value of a to be all the possible results of $a \vee b$. In this example, the new set of possible values of a will be the singleton $\{1\}$ (since $0 \vee 1$ is 1 and $1 \vee 1$ is 1 too). The execution of this instruction only modified one value, that of a. However, the Hamming distance between the previous value of a and its new value can be either 0 (in case a was originally 1) or 1 (in case a was originally 0). Thus, we consider that there is a leak.

By running our interpreter on assembly code, we can statically determine if there are leakages or if the code is perfectly balanced. For instance for a block cipher, we initially set the key and the plaintext (*i.e.*, the sensitive data) to have all their possible values: all the memory cells containing the bits of the key and of the plaintext have the value $\{0, 1\}$ (which denote the set of two elements: 0 and 1). Then the interpreter runs the code and outputs all possible leakage; if none are present, it means that the code is well balanced. Otherwise we know which instructions caused the leak, which is helpful when debugging but also to locate sensitive portions of the code.

For an example in which the code is balanced, we can refer to the execution of the `and` DPL macro shown in Fig. 8. There we can see that the Hamming distance of the updates does not depend on the values of a and b. We also note that at the end of the execution (and actually, all along the execution) the Hamming weight of each value does not depend on a and b either. This allows to chain macros safely. Indeed, each value is precharged with 0 before being written to.

4.2 Hardware Characterization

The DPL countermeasure relies on the fact that the pair of bits used to store the DPL encoded values leak the same way, *i.e.*, that their power consumptions are the same. This property is generally not true in non-specialized hardware. However, using the two closest bits (in term of leakage) for the DPL protocol still helps reaching a better immunity to SCAs, especially ASCAs that operate on a limited number of traces.

The idea is to compute the leakage level of each of the bits during the execution of the algorithm, in order to choose the two closest ones as the pair to use for the DPL protocol and thus ensure an optimal balance of the leakage. This is facilitated by the fact that the algorithm is bitsliced. Indeed, it allows to run the whole computation using only a chosen bit while all the other stay zero. We will see in Sec. 5.1 how we characterized our smartcard in practice.

5 Case Study: PRESENT on an AVR Micro-Controller

5.1 Profiling the AVR Micro-Controller

We want to limit the size of the look-up tables used by the DPL macros. Thus, DPL macros need to be able store two DPL encoded bits in the four consecutive bits of a register. This let 13 possible DPL encoding layouts on 8-bit. Writing `X` for a bit that is used and `x` otherwise, we have: 1. `xxxxxxXX` 2. `xxxxxXXx` 3. `xxxxXXxx` 4. `xxxXXxxx` 5. `xxXXxxxx` 6. `xXXxxxxx` 7. `XXxxxxxx` 8. `xxxxxXxX` 9. `xxxxXxXx` 10. `xxxXxXxx` 11. `xxXxXxxx` 12. `xXxXxxxx` 13. `XxXxxxxx`. As explained in Sec. 4.2, we want to use the pair of bits that have the closest leakage properties, and also which is the closest from the least significant bit, in order to limit the size of the look-up tables.

To profile the AVR chip (we are working with an *Atmel ATmega163 AVR* smartcard, this is *notoriously leaky*), we ran eight versions of an unprotected bitsliced implementation of PRESENT,

each of them used only one of the 8 possible bits. We used the NICV (Normalized Inter-Class Variance [1], also called *coefficient of determination*) as a metric to evaluate the leakage level of the variables of each of the 8 versions. A key advantage of NICV is that it detects leakage using public information like input plaintexts or output ciphertexts only. We used a fixed key and a variable plaintext on which applying NICV gave us the leakage level of all the intermediate variables in bijective relation with the plaintext (which are all the sensible data as seen in Def. 1). As we can see on the measures plotted in Fig. 10 (which can be found in App. B), the least significant bit leaks very differently from the others, which are roughly equivalent in terms of leakage[2]. Thus, we chose to use the `xxxxxXXx` DPL pattern to avoid the least significant bit.

5.2 Generating Balanced AVR Assembly

We wrote an AVR bitsliced implementation of PRESENT that uses the S-Box in 14 logic gates from Courtois *et al.* [10]. This implementation was translated in our generic assembly language (see Sec. 3.1). The resulting code was balanced following the method discussed in Sec. 3, except that we used the DPL encoding layout adapted to our particular smartcard, as explained in Sec. 5.1. App. C present the code of the adapted DPL macro. The balance of the DPL code was then verified as in Sec. 4. Finally, the verified code was mapped back to AVR assembly. All the code transformations and the verification were done automatically using our tool.

5.3 Cost of the Countermeasure

The table in Fig. 9 compares the performances of the DPL protected implementation of PRESENT with the original bitsliced version from which the protected one has been derived. The DPL countermeasure multiplies by 1.88 the size of the compiled code, this low factor can be explained by the numerous instructions which it is not necessary to transform (the whole permutation layer of the PRESENT algorithm is left as is for instance). The protected version uses 64 more bytes of memory (sparsely, for the DPL macro look-up tables). It is also only 3 times slower[3].

	bitslice	DPL
code (B)	1620	3056
RAM (B)	288	352
#cycles	78,403	235,427

Figure 9: DPL cost.

5.4 Attacks

We attacked three implementations of the PRESENT algorithm: a bitsliced but unprotected one, a DPL one using the two less significant bits, and a DPL one using two bits that are more balanced in term of leakage (as explained in Sec. 5.1). On each of these, we computed the success rate of using monobit CPA of the output of the S-Box as a model. The monobit model is relevant because only one bit of sensitive data is manipulated at each cycle since the algorithm is bitsliced, and also because each register is precharged at 0 before a new intermediate value is written to it, as per the DPL protocol prescribe. Note that this means we consider the resistance against first-order attacks only. Actually, we are precisely in the context of [23], where the efficiency of correlation and Bayesian attacks gets close as soon as the number of queries required to perform a successful attack is large enough. This justifies our choice of the CPA for the attack evaluation.

The results are shown in Fig. 14 (which can be found in App. D.2). They demonstrate that the first DPL implementation is 10 times more resistant to first-order power analysis attacks (requiring

[2]These differences are due to the internal architecture of the chip, for which we don't have the specifications.

[3]Notice that PRESENT is inherently slow in software (optimized non-bitsliced assembly is reported to run in about 11,000 clock cycles on an *Atmel ATtiny 45* device [12]) because it is designed for hardware. Typically, the permutation layer is free in hardware, but requires many bit-level manipulations in software. Nonetheless, we precise that there are contexts where PRESENT must be supported, but no hardware accelerator is available.

almost 1500 traces) than the unprotected one. The second DPL implementation, which takes the chip characterization into account, is 34 *times more resistant* (requiring more than 4800 traces). Interpreting these results requires to bear in mind that, as detailed in App. D.2, the *attacks setting was largely to the advantage of the attacker*. As a comparison[4], an unprotected AES on the same smartcard breaks in 15 traces, and in 336 traces with a first order masking scheme using less powerful attack setting (see success rates of masking in App. D.1), hence a security gain of 22. Besides, we notice that our software DPL protection thwarts ASCAs. Indeed, ASCAs require a high signal to noise ratio on a single trace. This can happen both on unprotected and on masked implementation. However, our protection aims at theoretically cancelling the leakage, and practically manages to reduce it significantly, even when the chosen DPL bit pair is not optimal. Therefore, coupling software DPL with key-update [26] allows to both prevent against fast attacks on few traces (ASCAs) and against attacks that would require more traces (regular CPAs).

6 Conclusions and Perspectives

Contributions. We present a method to protect any bitsliced assembly code by transforming it to enforce the dual-rail with precharge logic (DPL) protocol, which is a balancing countermeasure against power analysis. We provide a tool which automates this transformation. We also formally prove that this transformation is correct, *i.e.*, that it preserves the semantic of the program.

Independently, we show how to formally prove that assembly code is well balanced. Our tool is also able to use this technique to statically determine whether some arbitrary assembly code's power consumption activity is constant, *i.e.*, that it does not depend on the sensitive data. In this paper we used the Hamming weight of values and the Hamming distance of values update as leakage models for power consumption, but our method is not tied to it and could work with any other leakage models that are computable. Characterizations on a real smartcard ensured that these models is relevant in practice.

The provably balanced DPL protected code can be at least 34 times more resistant to power analysis attacks than the unprotected version while being only 3 times slower. This shows that software balancing countermeasures are realistic. Indeed, our formally proved countermeasure is an order of magnitude less costly than the state of the art of formally proved masking [33].

Future work. Although the mapping from the internal assembly of our tool to the concrete assembly is straightforward, it would be better to have a formal correctness proof of the mapping.

We did not try to optimize our PRESENT implementation (neither for speed nor space). However, automated proofs enable optimization: indeed, the security properties can be checked again after any optimization attempt (using proofs computation as non-regression tests, either for changes in the DPL transformation method, or for handcrafted optimizations of the generated DPL code).

Our work would benefit from automated bitslicing, which would allow to automatically protect any assembly code with the DPL countermeasure. However, it is still a very challenging issue.

Finally, the DPL countermeasure itself could be improved: the pair of bits used for the DPL protocol could be chosen at random for each execution to better balance the leakage among multiple traces, and unused bits could be randomized instead of being zero to add noise on top of balancing.

We believe formal methods have a bright future concerning the certification of side-channel attacks countermeasures (including their implementation in assembly) for trustable cryptosystems.

[4]We insist that the comparison between two security gains is very platform-dependent. The figures we give are only valid on our specific setup. Of course, for different conditions, *e.g.*, lower signal-to-noise ratio, masking might become more secure than DPL.

References

[1] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. NICV: Normalized Inter-Class Variance for Detection of Side-Channel Leakage. In *International Symposium on Electromagnetic Compatibility" (EMC '14 / Tokyo)*. IEEE, May 12-16 2014. Session OS09: EM Information Leakage. Hitotsubashi Hall (National Center of Sciences), Chiyoda, Tokyo, Japan.

[2] Eli Biham. A Fast New DES Implementation in Software. In Eli Biham, editor, *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.

[3] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES*, volume 4727 of *LNCS*, pages 450–466. Springer, September 10-13 2007. Vienna, Austria.

[4] Éric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In *CHES*, volume 3156 of *LNCS*, pages 16–29. Springer, August 11–13 2004. Cambridge, MA, USA.

[5] Claude Carlet, Jean-Charles Faugère, Christopher Goyet, and Guénaël Renault. Analysis of the algebraic side channel attack. *J. Cryptographic Engineering*, 2(1):45–62, 2012.

[6] Claude Carlet, Louis Goubin, Emmanuel Prouff, Michaël Quisquater, and Matthieu Rivain. Higher-Order Masking Schemes for S-Boxes. In Anne Canteaut, editor, *FSE*, volume 7549 of *Lecture Notes in Computer Science*, pages 366–384. Springer, 2012.

[7] Zhimin Chen, Ambuj Sinha, and Patrick Schaumont. Using Virtual Secure Circuit to Protect Embedded Software from Side-Channel Attacks. *IEEE Trans. Computers*, 62(1):124–136, 2013.

[8] Common Criteria Consortium. Common Criteria (*aka* CC) for Information Technology Security Evaluation (ISO/IEC 15408), 2013.
Website: `http://www.commoncriteriaportal.org/`.

[9] Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. Side Channel Cryptanalysis of a Higher Order Masking Scheme. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *LNCS*, pages 28–44. Springer, 2007.

[10] Nicolas Courtois, Daniel Hulme, and Theodosis Mourouzis. Solving Circuit Optimisation Problems in Cryptography and Cryptanalysis. *IACR Cryptology ePrint Archive*, 2011:475, 2011. (Also presented in SHARCS 2012, Washington DC, 17-18 March 2012, on page 179).

[11] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *IACR Cryptology ePrint Archive*, 2013:253, 2013.

[12] Thomas Eisenbarth, Zheng Gong, Tim Güneysu, Stefan Heyse, Sebastiaan Indesteege, Stéphanie Kerckhof, François Koeune, Tomislav Nad, Thomas Plos, Francesco Regazzoni, François-Xavier Standaert, and Loïc van Oldeneel tot Oldenzeel. Compact Implementation and Performance Evaluation of Block Ciphers in ATtiny Devices. In Aikaterini Mitrokotsa and Serge Vaudenay, editors, *AFRICACRYPT*, volume 7374 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2012.

[13] Sylvain Guilley, Sumanta Chaudhuri, Laurent Sauvage, Philippe Hoogvorst, Renaud Pacalet, and Guido Marco Bertoni. Security Evaluation of WDDL and SecLib Countermeasures against Power Attacks. *IEEE Transactions on Computers*, 57(11):1482–1497, nov 2008.

[14] Sylvain Guilley, Philippe Hoogvorst, Yves Mathieu, and Renaud Pacalet. The "Backend Duplication" Method. In *CHES*, volume 3659 of *LNCS*, pages 383–397. Springer, 2005. August 29th – September 1st, Edinburgh, Scotland, UK.

[15] Tim Güneysu and Amir Moradi. Generic side-channel countermeasures for reconfigurable devices. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES*, volume 6917 of *LNCS*, pages 33–48. Springer, 2011.

[16] Philippe Hoogvorst, Jean-Luc Danger, and Guillaume Duc. Software Implementation of Dual-Rail Representation. In *COSADE*, February 24-25 2011. Darmstadt, Germany.

[17] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[18] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[19] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer-Verlag, 1999.

[20] Boris Köpf and David A. Basin. An information-theoretic model for adaptive side-channel attacks. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 286–296. ACM, 2007.

[21] Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *CSF*, pages 324–335. IEEE Computer Society, 2009.

[22] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, December 2006. ISBN 0-387-30857-1, `http://www.dpabook.org/`.

[23] Stefan Mangard, Elisabeth Oswald, and François-Xavier Standaert. One for All - All for One: Unifying Standard DPA Attacks. *Information Security, IET*, 5(2):100–111, 2011. ISSN: 1751-8709 ; Digital Object Identifier: 10.1049/iet-ifs.2010.0096.

[24] Stefan Mangard and Kai Schramm. Pinpointing the Side-Channel Leakage of Masked AES Hardware Implementations. In *CHES*, volume 4249 of *LNCS*, pages 76–90. Springer, October 10-13 2006. Yokohama, Japan.

[25] Luke Mather and Elisabeth Oswald. Pinpointing side-channel information leaks in web applications. *J. Cryptographic Engineering*, 2(3):161–177, 2012.

[26] Marcel Medwed, François-Xavier Standaert, Johann Großschädl, and Francesco Regazzoni. Fresh Re-Keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices. In *AFRICACRYPT*, volume 6055 of *LNCS*, pages 279–296. Springer, May 03-06 2010. Stellenbosch, South Africa. DOI: 10.1007/978-3-642-12678-9_17.

[27] Simon Moore, Ross Anderson, Robert Mullins, George Taylor, and Jacques J.A. Fournier. Balanced Self-Checking Asynchronous Logic for Smart Card Applications. *Journal of Microprocessors and Microsystems*, 27(9):421–430, October 2003.

[28] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler Assisted Masking. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES*, volume 7428 of *LNCS*, pages 58–75. Springer, 2012.

[29] Maxime Nassar, Shivam Bhasin, Jean-Luc Danger, Guillaume Duc, and Sylvain Guilley. BCDL: A high performance balanced DPL with global precharge and without early-evaluation. In *DATE'10*, pages 849–854. IEEE Computer Society, March 8-12 2010. Dresden, Germany.

[30] Thomas Popp and Stefan Mangard. Masked Dual-Rail Pre-charge Logic: DPA-Resistance Without Routing Constraints. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *LNCS*, pages 172–186. Springer, 2005.

[31] Mathieu Renauld and François-Xavier Standaert. Algebraic Side-Channel Attacks. In Feng Bao, Moti Yung, Dongdai Lin, and Jiwu Jing, editors, *Inscrypt*, volume 6151 of *Lecture Notes in Computer Science*, pages 393–410. Springer, 2009.

[32] Mathieu Renauld, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA. In *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 97–111. Springer, September 6-9 2009. Lausanne, Switzerland.

[33] Matthieu Rivain and Emmanuel Prouff. Provably Secure Higher-Order Masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *CHES*, volume 6225 of *LNCS*, pages 413–427. Springer, 2010.

[34] Kai Schramm and Christof Paar. Higher Order Masking of the AES. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *LNCS*, pages 208–225. Springer, 2006.

[35] Maitham Shams, Jo. C. Ebergen, and Mohamed I. Elmasry. Modeling and comparing CMOS implementations of the C-Element. *IEEE Transactions on VLSI Systems*, 6(4):563–567, December 1998.

[36] Kris Tiri and Ingrid Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *DATE'04*, pages 246–251. IEEE Computer Society, February 2004. Paris, France. DOI: 10.1109/DATE.2004.1268856.

[37] Kris Tiri and Ingrid Verbauwhede. Place and Route for Secure Standard Cell Design. In Kluwer, editor, *Proceedings of WCC / CARDIS*, pages 143–158, Aug 2004. Toulouse, France.

[38] Kris Tiri and Ingrid Verbauwhede. A digital design flow for secure integrated circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(7):1197–1208, 2006.

[39] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 305–316. ACM, 2012.

A Our Tool

The name of the tool and an URL to its public code repository including usage instructions will be available with the final and non-anonymous version of the article.

We implemented our tool using the OCaml[5] programming language, which type safety helps to prevent many bugs. On our PRESENT case-study, it runs in negligible time ($\ll 1$ second), both for DPL transformation and simulation, including balance verification. The unprotected (resp. DPL) bitslice AVR assembly file consists of 641 (resp. 1456) lines of code. We use nibble-wise jumps in each PRESENT operation, and an external loop over all rounds.

Adapters. To easily adapt it to any assembly language, it has a system of plugins (which we call "adapters") that allows to easily write a parser and a pretty-printer for any language and to use them instead of the internal parser and pretty-printer (which are made for the internal language we use, see Sec. 3.1) without having to recompile the whole tool.

DPL transformation. If asked so, our tool is able to automatically apply the DPL transformation as explained in Sec. 3.2. It takes as arguments which bits to use for the DPL protocol, the offset at which to place the pattern for look-up tables (for example, we used an offset of 1 to avoid resorting to the least significant bit which leaks differently), and where in memory should the look-up tables start. Given these parameters, the tool verifies that they are valid and consistent according to the DPL protocol, and then it generates the DPL balanced code corresponding to the input code, including the code for look-up tables initialization. Optionally, the tool is able to compact the look-up tables (since they are sparse), still making sure that their addresses respect the DPL protocol (Sec. 2.2).

[5] `http://ocaml.org/`

Simulation. Our tool can simulate the execution of the code after its optional DPL transformation. The simulator is equipped to do the balance verification proof (see Sec. 4) but it is not mandatory to do the balance analysis when running it. It takes as parameters the size of the memory and the number of register to use, and initializes them to the set of two DPL encoded values of 1 and 0 corresponding to the given DPL parameters. The tool can optionally display the content of selected portions of the memory or of chosen registers after execution, which is useful for inspection and debugging purpose for example.

Balance verification. The formal verification of the balance of the code is an essential functionality of the tool. Indeed, bugs occur even when having a thorough and comprehensive specification, thus we believe that it is not sufficient to have a precise and formally proven method for generating protected code, but that the results should be independently verified (see Sec. 4).

B Characterization of the AVR Micro-Controller

Fig. 10 shows the leakage level computed using NICV [1] for each bit of the *Atmel ATmega163 AVR* smartcard that we used for our tests (see Sec. 5.1). We can see the first bit leaks very differently from the others. Thus it is not a good candidate to appear in the bit pair used for the DPL protocol.

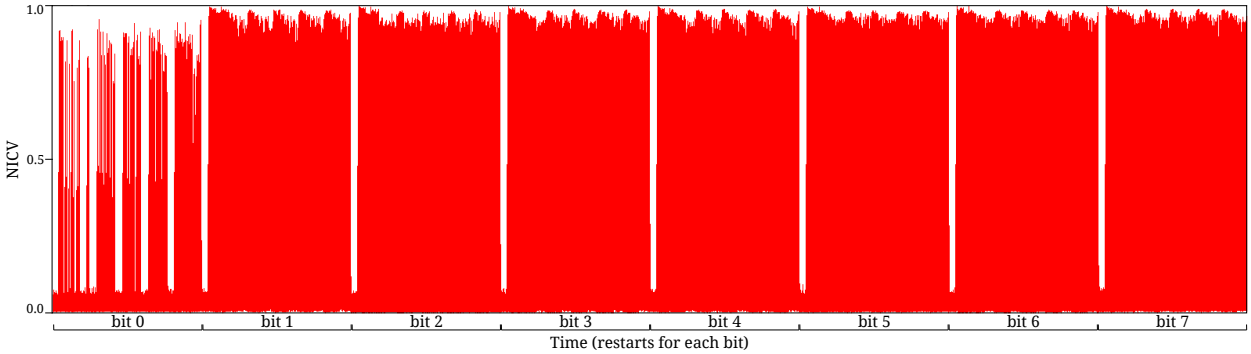


Figure 10: Leakage level during unprotected encryption for each bit of the *ATmega163*.

C DPL Macro for the AVR Micro-Controller

Once we profiled our smartcard as described in Sec. 5.1, we decided to use the bits 1 and 2 for the DPL protocol (`xxxxxXXx`), that is, the DPL value of 1 becomes 2 and the DPL value of 0 becomes 4. To avoid using the least significant bit (which leaks very differently from the others), we decided to align the two DPL bits for look-up table access starting on the bit 1 rather than 0 (`xxxXXXXx`). With these settings, the DPL macro automatically generated by our tool is presented in Fig. 11 (it follows the same conventions as Fig. 2). As we can see the only modification is the mask applied in the logical *and* instructions which is now 6 instead of 3 to reflect the new DPL pattern.

Note that the least significant bit is now unused by the DPL protocol and allowed our tool to compact the look-up tables used by the DPL macros. Indeed, their addresses need

$$
\begin{array}{rcl}
r_1 & \leftarrow & r_0 \\
r_1 & \leftarrow & a \\
r_1 & \leftarrow & r_1 \wedge 6 \\
r_1 & \leftarrow & r_1 \ll 1 \\
r_1 & \leftarrow & r_1 \ll 1 \\
r_2 & \leftarrow & r_0 \\
r_2 & \leftarrow & b \\
r_2 & \leftarrow & r_2 \wedge 6 \\
r_1 & \leftarrow & r_1 \vee r_2 \\
r_3 & \leftarrow & r_0 \\
r_3 & \leftarrow & op[r_1] \\
d & \leftarrow & r_0 \\
d & \leftarrow & r_3
\end{array}
$$

Figure 11: DPL macro for $d = a$ op b on the *ATmega163*.

to be of the form `/.+0000./` leaving the least significant bit free and thus allowing to put two look-up tables one on another without overlapping of their actually used cells (see Sec. 3.2).

D Attacks

D.1 Attack results on masking (AES)

For the sake of comparison, we provide attack results on the same smartcard tested with the same setup. Figure 12 shows the success rate for the attack on the first byte of an AES.

We estimate the number of traces for a successful attack as the abscissa where the success rate curve first intersects the 80% horizontal line.

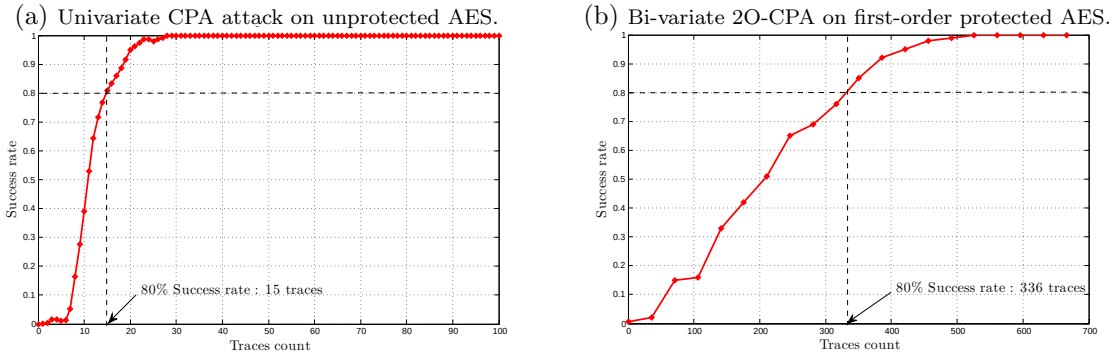


Figure 12: Attacking AES on the *ATmega163*: success rates.

D.2 Attack results on DPL (PRESENT)

Fig. 14 shows the success rates and the correlation curves when attacking our three implementations of PRESENT. The sensitive variable we consider is in line with the choice of Kocher *et al.* in their CRYPTO'99 paper [18]: it is the least significant bit of the output of the substitution boxes (that are 4×4 in PRESENT).

In Fig. 13, we give, for the unprotected bitslice implementation, the correspondence between the operations of PRESENT and the NICV trace. The zones of largest NICV correspond to operations that access (read or write) sensitive data in RAM. To make the attacks more powerful, they are not done on the maximal correlation point over the full first round of PRESENT[6] ($500,000$ samples), but rather on a smaller interval (of only 140 samples, *i.e.*, one clock period of the device) of high potential leakage revealed by the NICV computations, namely sBoxLayer. This makes the attack much more powerful and has to be taken into account when interpreting its results.

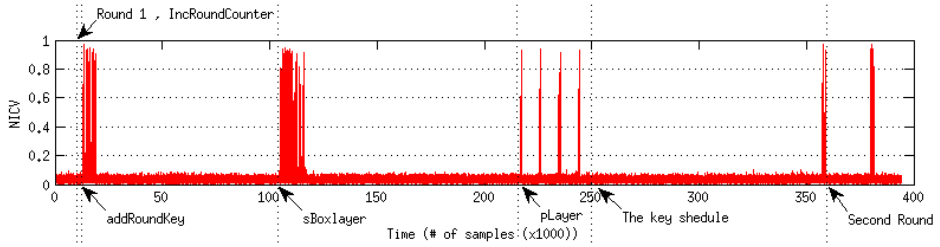
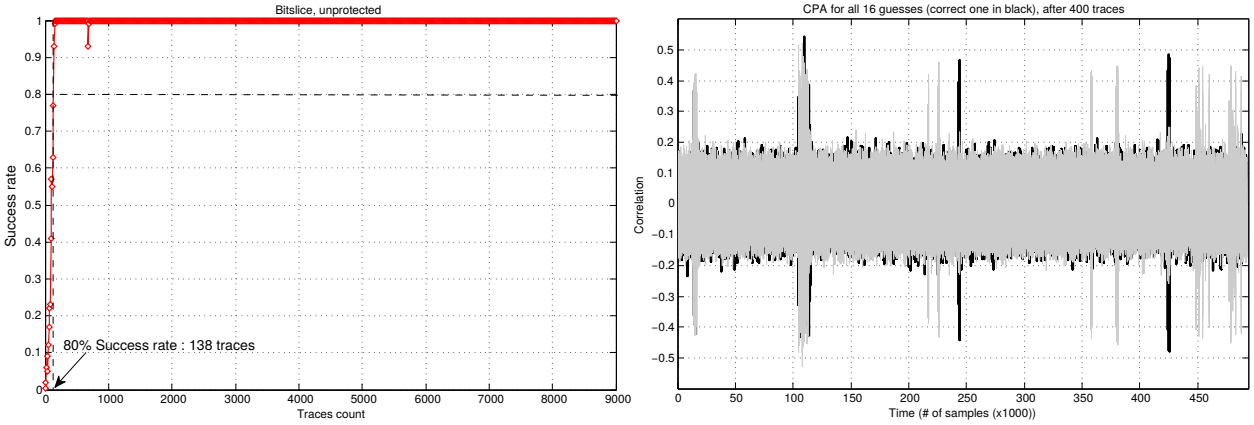


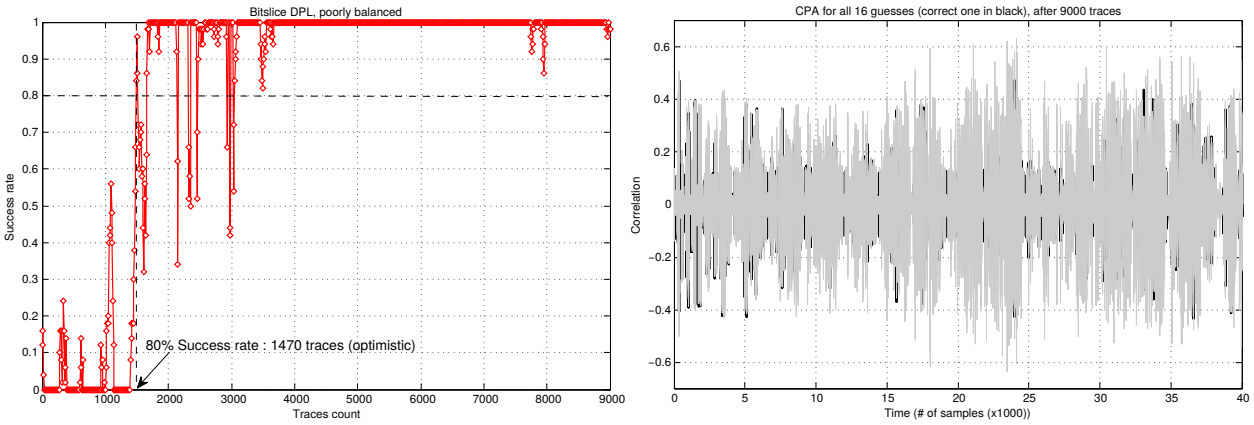
Figure 13: Correspondence between NICV and the instructions of PRESENT.

[6]Note that using the maximum correlation point to attack the DPL implementations resulted in the success rate remaining always at $\approx 1/16$ (there are 2^4 key guesses in PRESENT when targeting the first round, because the substitution boxes are 4×4) in average (at least on the number of traces we had ($> 10,000$)) on both on them.

(a) Monobit CPA attack on unprotected bitslice implementation.



(b) Monobit CPA attack on poorly balanced DPL implementation (bits 0 and 1).



(c) Monobit CPA attack on better balanced DPL implementation (bits 1 and 2).

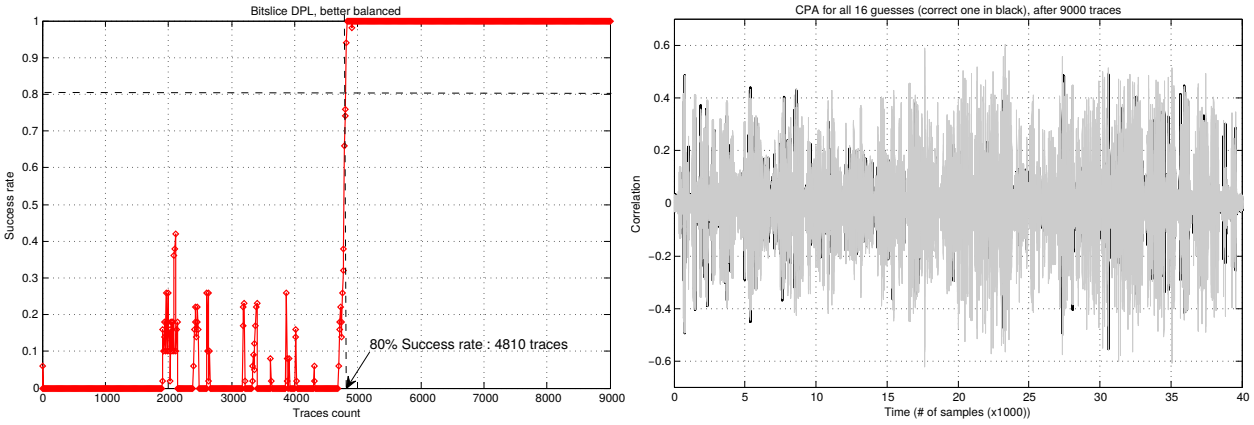


Figure 14: Attacks on our three implementations of PRESENT;
Left: success rates (estimated with 100 attacks/step), and
Right: CPA curves (whole first round in (a), and only sBoxLayer for (b) and (c)).