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 due to the fact that formal methods work with 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 not something that needs to be proved anymore. But the code itself needs to be proved in order to be trustable. 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.* [34] 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 [15]), but also some software-specific information such as packet size for instance (refer to [21]).

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 [14] cannot. *Differential Power Analysis* [16] (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.* [16], power consumption is often modeled by Hamming weight of values or Hamming distance of values' updates as it is very correlated with actual measures. Also, when the leakage is little noisy and the implementation is software, *Algebraic Side-Channel Attacks* (ASCAs [26]) are possible; they consist in modelling the leakage by a set of equations, where the key bits are the only unknown variables [4].

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 [12]). 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 [19, Chp. 9]), or (*b*) make the leakage constant, irrespective of the manipulated data (hiding or balancing [19, 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 [28]. 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 [27], despite the presence of the mask(s), that is(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 [28] 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 [20]. Indeed, a glitch occurs when there is a race between two signals, *i.e.*, when it involves more than one sensitive variable.

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 they leak the same (by hypothesis). This property is used by the DPL protocol to ensure that computations can be carried out without exploitable leakage [33].

Contributions. The two main contributions of this paper are: (*a*) a design method for developing balanced assembly code by transforming it to make it follow 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 [2] cipher running on an AVR micro-controller.

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 [7], 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. 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 recent security breach in what was supposed to be a proved [28] masking implementation [5].

Timing and cache attacks are an exception as they benefit from the work of Köpf *et al.* [17, 18]. Their tool, CacheAudit [9], implements formal methods that directly work on x86 binaries.

In this light it is easy to conclude that the use 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 the early 2004, with dual-rail with precharge logic. The DPL countermeasure consists in computing on a

redundant representation: each bit b 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 [23], 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 [31]), 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 [32, 11], 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 [25]) style has been proposed: instead of balancing exactly the lines carrying y_{False} and y_{True}, those were randomly swapped, according a random bit, as 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 gate is represented in a grey box. The synchronization logic are the gates outside of this box. In SecLib [10], the synchronization can be achieved by Muller C-elements (represented with a symbol C [30]), and act as a decoding of the

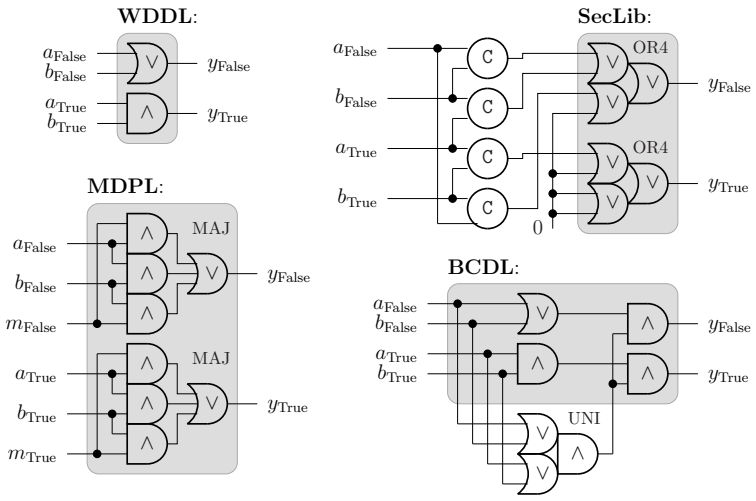

Figure 1: Four dual-rail with precharge logic styles.

inputs configuration. Another implementation, balanced cell-based differential logic (BCDL [24]), 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. However, as noted by Hoogvorst *et al.* [13], 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 to 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 they 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 they are 1 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 `/.+(001|100)/`, 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[1] 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    ::= ( Inst? [ ';' <comment> ] '\n' )*
Inst    ::= Label
          | Opcode0
          | Branch1 Addr
          | Opcode2 Val Val
          | Opcode3 Val Val Val
          | Branch3 Val Val Addr
Label   ::= <label-name> ':'
Opcode0 ::= 'nop'
Branch1 ::= 'jmp'
Opcode2 ::= 'not' | 'mov'
Opcode3 ::= 'and' | 'orr' | 'xor'
          | 'shl' | 'shr'
          | 'add' | 'mul'
Branch3 ::= 'beq' | 'bne'
Val     ::= 'r' <register-number>
          | '@' <memory-address>
          | '#' <immediate-value>
          | '!' Val ( ',' <offset> )?
Addr    ::= '#' <absolute-code-address>
          | '$' <relative-code-address>
          | <label-name>
```

Figure 3: Generic assembly syntax.

```
mov r1 r0
mov r1 a
and r1 r1 #3
shl r1 r1 #1
shl 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 semantic of the instructions is intuitive. For `Opcode2`, `Opcode3`, and `Branch2` the first operand is the destination and the others are the arguments. The `mov` instruction is used to

[1] the code will be available with non-anonymous version of the article.

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[2], 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 [1], many block ciphers have been implemented in bitslice for performance reasons, which mitigated 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 cleartext (as usually admitted in the "*only computation leaks*" paradigm; see for instance [28, §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`, `shl`, `shr`, 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 needs 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 is three operands in our generic assembly language, the destination is the same as one of the two others), which make 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 clean when adding the offset. 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`.

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

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`, must be DPL invalid, *i.e.*, `00` or `11`. Like this if an error occur during the execution (such as a fault injection for instance) it poisons the result and all the subsequent computations will be faulted too.

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 $X \xrightarrow{c} Z$.

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. □

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

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`.

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 laid out in an FPGA or in an ASIC with special balancing constraints at place-and-route. The methodology follows the guidelines given in [6]. 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 Simple Power Analysis or Algebraic Side-Channel Attack.

To formally prove the balancing of the DPL transformed code we first need to 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 [16] or CPA [3]. 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 cleartext). 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

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

We implemented an interpreter for our generic assembly language which work with sets of values. This interpreter is equipped to measure all the possible Hamming distances of each value update. 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, 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 cleartext (*i.e.*, the sensitive data) to have all their possible values: all the memory cells containing the bits of the key and of the cleartext 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.

Using stochastic profiling [29], or monobit CPA attack to measure the Pearson correlation coefficient between the actual power consumption of the targeted bit and the logical Hamming distance of its updates, it is possible to find a pair of bits that have close leakage properties and that are at suitable positions (*i.e.*, next to each other or separated by exactly one other bit) for the DPL protocol. We will see in Sec. 5.1 and 5.4 that in practice it is indeed possible to find a pair of bits that are sufficiently close in term of leakage.

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 to 16 bytes. Thus, DPL macros need to store two DPL encoded bits in the four least significant 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`. 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, to avoid shifting on unbalanced pairs as much as possible.

To profile the AVR chip (we are working with an *Atmel Atmega163 AVR* smartcard), we ran eight versions of an unprotected bitsliced implementation of PRESENT, each of them used only one of the 8 possible bits. We did a monobit CPA attack targeting the bit used by each version, using known keys and messages. As we can see on the measures plotted in App. A, the two closest bits in term of leakage correspond to the **TODO**th possible DPL encoding layout.

5.2 Generating Balanced AVR Assembly

We wrote an AVR bitsliced implementation[1] of PRESENT that uses the S-Box in 14 logic gates from Courtois *et al.* [8]. This implementation was translated in our generic assembly language (see Sec. 3.1), using a script[1] that does an instruction-to-instruction mapping. To balance the code we followed the method discussed in Sec. 3, except that we used the DPL encoding layout adapted to our particular smartcard. App. B present the code of the adapted DPL macro. We then used our tool[1] to verify that the code was well balanced as in Sec. 4. Finally, we mapped the code back to AVR assembly using another script[1] which does the reverse operation of the first one.

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 2.32 the size of the code, uses 48 more bytes of memory (for the DPL macro look-up tables), and is 3.27 times slower.

Implem.	#instr.	RAM (B)	#cycles
normal	599	288	76,862
DPL	1337	336	250,955

Figure 9: DPL cost.

5.4 Attack

We attacked both the unprotected and the DPL balanced implementations of PRESENT with a monobit CPA attack. The results are shown in Fig. **TODO** and demonstrates that the DPL balanced implementation is **TODO** times more resistant to the attack.

TODO ici le tableau et/ou la courbe qui résume l'attaque.

Besides, we notice that our software DPL protection thwarts ASCAs. Indeed, ASCAs require a high signal to noise 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. Therefore, coupling software DPL with key-update [22] allows to both prevent against fast attacks on few traces (ASCAs) and against attacks that would require more traces (regular CPA).

6 Conclusions and Perspectives

Contributions. We presented a method to protect any bitsliced assembly code by transforming it to follow the dual-rail with precharge logic (DPL) protocol, which is a balancing countermeasure against power analysis side-channel attacks. We provide a tool which do this transformation automatically. We also formally proved that this transformation is correct, *i.e.*, that it preserves the

semantic of the program.

Independently, we showed how to formally prove that assembly code is well balanced using symbolic execution of the code. We also provide a tool which use this technique to statically determine if some arbitrary assembly code is power consumption activity is constant, *i.e.*, that it does not depend on the sensitive data. In this paper we used the Hamming distance of values update as a leakage model (for power consumption), but our method is not tied to it and could work with any other leakage model that is computable.

The provably balanced DPL protected code is **TODO** times more resistant to power analysis attacks than the unprotected version while being only a bit more than 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 [28].

Future work. We did not try to optimize our implementation of PRESENT. However, formal proofs enable optimization, as they can be used as non-regression tests. Indeed, the security properties can be checked again after any optimization attempt. It would be interesting to see how much speed could be gained on our protected implementation of PRESENT.

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

The DPL countermeasure could be improved in several ways. The pair of bits used to store the DPL encoded value could be changed during the execution, or chosen at random at the beginning of the execution in order to better balance the leakage among multiple traces. Also, unused bits could be randomized instead of being zero, which would add a layer of masking on top of the balancing.

Our results are promising and we believe formal methods have a bright future concerning side-channel attacks countermeasures certification for trustable cryptosystems.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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).

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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/`.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] Werner Schindler, Kerstin Lemke, and Christof Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In LNCS, editor, *CHES*, volume 3659 of *LNCS*, pages 30–46. Springer, Sept 2005. Edinburgh, Scotland, UK.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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 Characterization of the AVR Micro-Controller

TODO

B DPL Macro for the AVR Micro-Controller

TODO