Formally Proved Security of Assembly Code Against Leakage

Pablo Rauzy
pablo.rauzy@telecom-paristech.fr

Sylvain Guilley
sylvain.guilley@telecom-paristech.fr

Zakaria Najm
zakaria.najm@telecom-paristech.fr

September 3, 2013

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. Of course, we can use formal methods to prove non-functional security properties such as the absence of side-channel leakages. But a common obstacle is that those properties are very low-level and appear incompatible with formalization. To avoid the discrepancy between the model and the implementation, we apply formal methods directly on the implementation. Doing so, we can formally prove that an assembly code is leak-free, provided that the hardware it runs on satisfies a finite (and limited) set of properties that we show are realistic. We apply this technique to prove that a PRESENT implementation in 8 bit AVR assembly code is leak-free.

keywords. side-channels, balancing countermeasure, formal methods, symbolic evaluation.

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 very important too, but are not typically subject to formal proof yet. Side-channel attacks are a real world threat to cryptography that use auxiliary information gathered on 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, including for non-functional properties, cryptosystems by showing that modeling leakage and applying formal methods to implementations is feasible.

Most existing countermeasures against side-channel attacks are implemented at the hardware level. 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 purely software world, as shown recently by Zhang *et al.* in [24], where they extract 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 and power consumption, but also some software-specific information such as packet size for instance.

In many cases where the cryptographic code executed on secure elements (smarcards, 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.*).

Thwarting side-channel analysis is complicated since an unprotected implementation leaks at every step, and powerful (and simple) attacks manage to exploit any bias. For instance, the differential power analysis [12] does not require a precise modelization of the leakage, and thus works blind, *i.e.*, even if the implementation is blackbox.

In practice, there are two options: "palliative" versus "curative" countermeasures. Palliative countermeasures attempt to make the attack more difficult, however without a theoretical foundation. They include variable clock, shuffling, dummy encryptions, among others (see also [9]). Since non-provable, they are 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:

1. make the leakage constant, irrespective of the manipulated data (hiding

or balancing strategy [13, Chp. 7]), or

2. make the leakage as decorrelated from the manipulated data (masking strategy [13, Chp. 9]) as possible.

Balancing. Balancing requires a close collaboration between the hardware and the software: two indistinguisable from a side-channel point of view resources shall exist and be used according to a DPL protocol. Dual-rail with precharge logic (DPL) consists in precharging the 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). As both resources leak identically, the attacker cannot determine which of them has been set and computations can be carried out without exploitable leakage [23].

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 absence of leakage behavior from the hardware, and the existance of provably secure masking schemes [18]. The drawbacks are the possibility of high-order attacks (that examine the variance or the joint leakage), and 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 the AES is reported in [18] to be 43 (resp. 90) times slower than the non-masked implémentation when a 1st (resp. 2nd) order masking scheme is used. Further, recent studies have shown that masking cannot be analyzed independently from the execution platform. For instance, the leakage models in Hamming distance mix two sensitive variables (and is thus already second-order). Also, glitches are transient leakages that are likely to depend on more than one sensitive data, hence being high-order. Indeed, a glitch occurs when there is a race between two signals, *i.e.*, when it involves more than one sensitive variable.

Lightweight cryptography, cost of countermeasures. Lightweight cryptography are cryptographic primitives that have been designed to meet as tightly as possible the security requirements in terms of resistance against cryptanalyses. They are thus dimensioned minimally, thereby making suitable for integration in resource-constrained devices, offering a mathematically provable security level at cost lower than non-lightweight primitives. Of course, lightweight primitives shall also be protected against side-channel attacks. Now, lightweight primitives shall remain lightweight even after their armoring with side-channel countermeasures. However, this is far from being obvious; let us consider the example of PRESENT [1], a 64 bit block cipher using a 80 bit key. PRESENT is extremely lightweight in hardware, as it is 2.17 times smaller than the AES for the same functionality (encryption / decryption) [1, Table 2]. But in software, the implementation is fairly inefficient [6], as it requires about twice as much time to execute as AES. Now, the 4×4 substitution box (S-Box) involved in

PRESENT is a random (*i.e.*, unstructured) function which cannot be easily expressed as polynomial. Masking operations are embedded within a group, and thus traditionnal masking schemes, such as [18], cannot be applied; instead, generic masking methods (thus without optimizations) shall be employed [2]. We have estimated that a protected implémentation of PRESENT is roughly 32 times slower than an unprotected one, which is definitely a huge overhead.

As masking is inadequate for PRESENT, hiding must be considered.

The motivations are as follows:

- Given the simple Boolean equations of the S-Box, a bitslice implementation is not that long to compute, especially in regards of the extremely long time required to execute bit-level parts such as the permutation layer and the round keys computation.
- Such an implementation is standalone, not requiring any source of randomness (usually absent from low-cost embedded devices).
- DPL also protect against some fault injection attacks: the encoding being redundant, some error detection logic can be added; also it is likely that the fault alters the data in a way that makes it not compliant with the protocol, in which case the algorithm continues with data that have lost all relationships with the correct data, thereby whitening the sensitive values.

Contributions. The three contributions of this paper are: (1) a software bitsliced implementation of present that is faster than the current state-of-the-art implementation, (2) an automatic protection against side-channel leakages of assembly code by insertion of the dual-rail with precharge logic (DPL), and (3) a formal proof of the implementation's security with respect to side-channel analysis.

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 [5], 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 implementation fit the model, which is usually done by hand and is thus error-prone.

For instance, some masking implementation 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 [18] masking implementation [2].

In this light it is easy to conclude that the use formal methods to prove the security of implementations is a technological enabler: it would guarantee that

the instanciations of security principles are as strong as the security principles themselves.

In this paper we use formal methods (symbolic evaluation) to prove that an implementation (at the assembly code level) is leak-free, provided that the hardware it runs on satisfies a finite and limited set of requirements that we give and which can be tested in lab. We also provide a case study on a software implementation of the PRESENT algorithm running on an AVR micro-controller.

Organization of the paper. First of all, a state-of-the-art about constant leakage, irrespective of the sensitive variables, is drawn in Section 2: it explains the specificity of dual-rail with precharge logic (DPL) in software, and notably why the approach is realistic. The compilation flow to obtain a portable protected program (mappable to several CPU cores) is described in Section 3. The security proof of the programs generated by this compilation is given in Section 4. A case study is then presented in Section 5. Eventually, conclusions are in Section 6.

2 State-of-the-Art of DPL implementations

The hiding countermeasure has been employed against side-channel since the early 2004, with dual-rail with precharge logics. The countermeasure consist 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 [14], it can serve as a canary to detect a fault.

Various dual-rail with precharge logic 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 was that the indistinguishability hypothesis on the two resources holding y_{False} and y_{True} values was violated, which lead to the development of dedicated hardware. A first dissymetry comes from the gates driving y_{False} and y_{True}. In wave dynamic differential logic (WDDL [21]), 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 [22, 8], 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 [16]) style has been proposed: instead

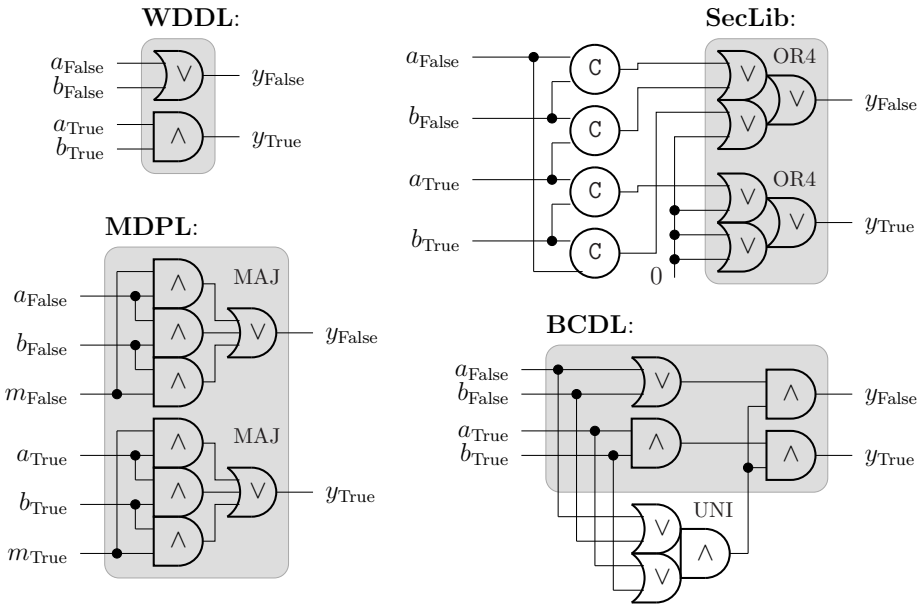

Figure 1: Four dual-rail with precharge logic styles.

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 – `MAJ`), but the routing is balanced thanks to the extra mask. However, it appeared that another dissymetry 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 cannot 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 with round corners. The synchronization logic are the gates outside of this box. In SecLib [7], the synchronization can be achieved by Muller C-elements (represented with a symbol `C` [20]), and act as a decoding of the inputs configuration. Another implementation, balanced cell-based differential logic (BCDL [15]), parallelize the synchronization with the computation.

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.

- There is no such problem as early evaluation, since the processor executes one operation after the other; therefore there are no unbalanced paths.

This has motived the design for DPL co-processors [17, 3, 4].

In this paper, we want to run dual-rail with precharge logic on an off-the-shelf processor. Therefore, we must:

1. identify two similar resources that can hold true and false values in an undiscernible way for a side-channel attacker;
2. play the DPL protocol by ourselves.

Basically, by an adequate characterization (called stochastic profiling [19]), it is easy to find two bits from the processor accumulator that have the same leakage property. Then, by a careful handwritten sequence in assembly language, the resources can be precharged and used in the DPL protocol. Our flow to generate a DPL program is discussed in the next section.

3 Generation of DPL code

This section develops on the generation of DPL code, Fig. 2 recaps the steps of this transformation. The point is to show that leak-free code actually exists and can be obtained methodically. Besides, it provides us with examples that serves as targets for our formal proofs.

Translation to abstractASM. We chose to implement the DPL protection at the assembly level for two main reasons. The optimization by any compiler would mess up the DPL scheme, which for instance requires logically useless precharge of values. Second, it is really on the actual implementation that we want to prove the absence of sensible information leakage.

However, it would be wasteful to develop formal methods and proven implementations for one particular physical architecture. This is why we chose to design our own assembly language, abstractASM. AbstractASM is a simple and generic assembly language which is easy to manipulate and, more importantly, whose instructions can be mapped one-to-one with those of actual assembly languages. This allows to develop tools that will be usable to formally prove the security of assembly code on multiple devices. To give the reader an idea of what it looks like, a BNF of abstractASM can be found in the appendix A.

Bitslice transformation. The DPL protocol has originally been designed to be implemented in hardware. Thus it operates at the bit level. This means that it is very generic and can be applied to virtually any algorithm, but also that

we cannot apply it to a software implementation "as is". The first code transformation is the translation of the algorithm into bitslice form: all operations must be bitwise and memory cells and registers must contain only one bit. In terms of speed, this transformation could seem prohibitive, but it can actually be benefical for some classes of algorithms. Indeed, bitslice implementations of some ciphers (precisely what interest us here) are parallelizable on as many blocks as the machine word length in bits with minimum efforts. In particular, ciphers whose internal data are seen as binary Galois field, *i.e.*, those that make extensive use of binary exclusive or, such as PRESENT, AES (block ciphers), Trivium, Grain, MICKEY (stream ciphers). On the other hand, algorithms whose data are seen as integers are less suitable for this transformation, for instance the round function of RC5 can be computed in two 32 bits instructions but would require hundreds of instructions in bitslice.

For now the bitslice transformation is done by hand. There is no tool to do that off-the-shelf yet, but our plan is to have this done by an automated compiler which will have been formally proved not to alter the semantic or behavior of the code.

DPLization of the state. The code and data are now in bitsliced form. Each literal (or "immediate" values in assembly terms), preset memory cells and registers values which correspond to data (by opposition to code or memory addresses) must be translated to their DPL equivalent in order to have a DPL state.

DPLization of the code. Our leakage model is the Hamming distance of updates. We want that, each time a register is updated, the Hamming distance between its old and new values be constant, independent of any sensitive value

Our approach is the one followed in [10]: each sensitive instruction is replaced by a sequence of instructions (a *macro*) which does the same computation but respect the constant Hamming distance constraint. The principle of the macros is to apply the DPL precharge before writing sensitive data, and to use lookup tables to compute the results of binary operations on DPLized bits. Macros assume that the current state is a valid DPL state and must leave a valid DPL state after their computation, so they can be chained.

Assuming that a and b are the registers containing the operands, that d is the destination, that op is the address of the lookup table for the binary operation that we "expand" with this macro, and that r_0 an always-0 register, Here is the macro:

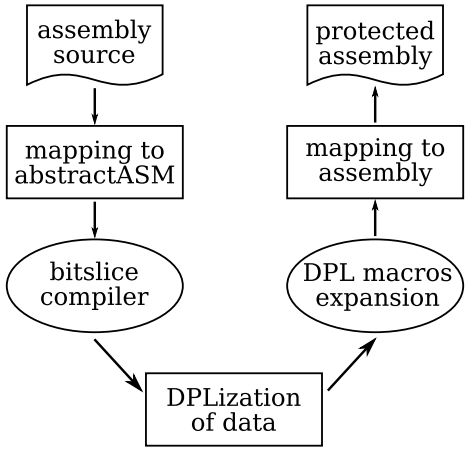

Figure 2: Generation of DPL code.

r_1	$\leftarrow$	r_0	`mov r1 r0`
r_1	$\leftarrow$	a	`mov r1` a
r_1	$\leftarrow$	$r_1 \wedge 3$	`and r1 r1 #3`
r_1	$\leftarrow$	$r_1 \ll 1$	`shl r1 r1 #1`
r_1	$\leftarrow$	$r_1 \ll 1$	`shl r1 r1 #1`
r_2	$\leftarrow$	r_0	`mov r2 r0`
r_2	$\leftarrow$	b	`mov r2` b
r_2	$\leftarrow$	$r_2 \wedge 3$	`and r2 r2 #3`
r_1	$\leftarrow$	$r_1 \vee r_2$	`orr r1 r1 r2`
r_3	$\leftarrow$	r_0	`mov r3 r0`
r_3	$\leftarrow$	$op[r_1]$	`mov r3 !r1,`op
d	$\leftarrow$	r_0	`mov` d `r0`
d	$\leftarrow$	r_3	`mov` d `r3`

The "expansion" of the sensible instructions is done automatically by a script. Provided that the lookup tables are built correctly (as explained in [10]), the semantic preservation of this transformation is provable by exhaustive case coverage. Since there's only four cases for each binary operation, it is sufficient to test them and see that $macro(DPL(a)\ op\ DPL(b)) = DPL(a\ op\ b)$, where $macro(inst)$ is the "expansion" of the instruction $inst$ and DPL is the encoding function for our DPL convention, *i.e.*, $DPL(0) = 0b10$ and $DPL(1) = 0b01$.

Translation to concrete assembly. The obtained code is supposed to be protected. The security proof is done independently from the code generation and is exposed in Section 4. Once the proof is done, the abstractASM code can be mapped back to concrete assembly code for lab tests.

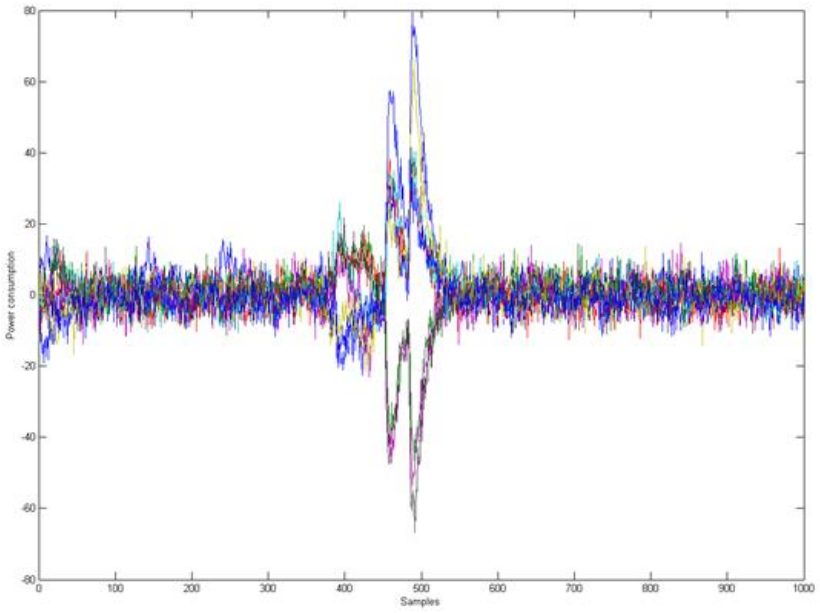

Figure 3: Stochastic characterization of every bit in a general purpose CPU.

4 Security Proof of our Code

The security proof is done independently from the DPL code generation. On the one hand there are other ways to obtain leak-free code and it should be possible to formally prove the absence of leakages on theses codes too. On the other hand the tool we developed to generate the proof, and which will be presented in Section 4.2, can be used as a non-regression test in case of attempts at optimization of the protected code.

4.1 Hardware Characterization

This is done by exhaustive execution of the routines and profiling. For instance, on an 8-bit 6502 core (see Fig. 3), the main register have bits that leak equally (and some others that leak exactly the opposite direction – those can be ignored). Such a characterization, based on real traces analyzed by a linear regression to infer the most adequate model (*i.e.*, the stochastic approach [19]) shows that most bits in a CPU register are indeed indistinguishable in practice.

4.2 Proof of Constant Activity

The aim of the DPL code protection against sensitive information leakage is to make the leakage activity constant. To prove that this goal is acheived the only

solution would be to record the leakage activity of the algorithm running for each input. This is obviously non-feasible using a naive brute-force approach, and this is were formal methods enter the game.

In [11], Kocher pointed out that side-channel attacks could exist because some assumptions were not modeled. We want to illustrate that formal methods, in return, can help thwart implementation-level attacks by properly capturing device-level leakage and deriving a formal assurance of resistance.

Formal methods are the key to certified security, and the key to formal methods is modelization. Being as low-level as assembly language could seem to be contradictory with modelization, since we lost almost all semantics of what the program does. However, we are not interested in what the program actually does. What interest us is what the implementation does, at the bit level. In this perspective, it is also an absolute benefit to work with bitsliced code.

A common model for power consumption and electromagnetic radiations side-channels is the Hamming distance of updates of values.

We model the execution of the code by running it on a software simulated processor which we call abstractCPU. AbstractCPU is equipped to compute the leakages as we modeled them. But this does not solve the problem of computing the leakages for all the possible inputs. We address this problem by using symbolic evaluation: rather than computing using values, it computes using sets of values. We don't have the risk of combinatorial explosion given the few possible values that each set can contain due to the bitslicing of the code.

The use of sets of possible values allow to compute the Hamming distance between all the previous values to all the new values for each update. Thus compute all the possible leakages for each instruction.

The invariant that we want to obtain is that for each instruction there's at most one possible leak value. While this invariant holds, the code is provably leak-free.

For example, let's say that at some point in the computation r_1 is $\{1, 2\}$ (it could either be 1 or 2) and r_2 is $\{3, 4\}$ (it could either be 3 or 4). If at this point the next instruction is $r_1 \leftarrow r_1 \oplus r_2$ then after its evaluation, r_1 becomes $\{1, 2, 5, 6\}$. In that case, the value of the Hamming distance of r_1's update can be either 0 (for $1 \rightarrow 1$, $1 \rightarrow 2$, $2 \rightarrow 1$, $2 \rightarrow 2$) or 1 (for $1 \rightarrow 5$, $1 \rightarrow 6$, $2 \rightarrow 5$, $2 \rightarrow 6$). This means that the leakage depends on the possible values in the registers, which is exactly what we want to avoid.

For our purpose, the abstractCPU checks the possible leakages in six "places": the register write bus, the register read bus, the register address bus, the memory write bus, the memory read bus, and the memory address bus. We are not checking the code read and address buses (although we could easily have done so, using the exact same methods as for the six others buses). Mainly because the target device of our tests (an Atmel AVR) has an Harvard architecture, so the memory and code are actually separated. Also, the code is supposed public and there's no conditional jump that depends on sensitive data in our examples (as we will see in Section 5). Fig. 4 presents what abstractCPU looks like in this configuration.

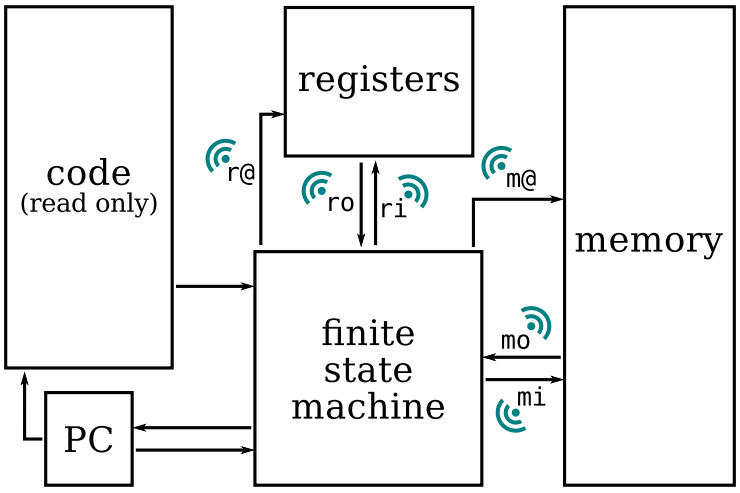

Figure 4: abstractCPU.

4.3 Support for Optimizations

The code generation methodology is rather generic, hence using protected code for all possible operations. Now, due to:

1. The existence of non-sensitive signals (*e.g.*, the selection of key size); or loop counters;
2. The limited data range of some variables, that makes some parts of the code use constant variables;
3. The possibility to go from one macro to the other through register, thereby saving time from the memory transfers;
4. The possibility to merge instructions given certain patterns;
5. The use of architecture-specific intructions not included in our abstractASM.

There is certainly room for optimization of the obtained DPL code. Thanks to the fact that our proof method is independant from DPL code generation, it can still be applied to optimized code to verify if the optimizations are valid with respect to security. Only the last of the five points would require an adaptation of our tool, in order to implement the behavior of the architecture-specific instructions.

5 Case study on PRESENT

We decided to use the cryptographic algorithm PRESENT for our case study, for the reasons we exposed in the introduction. We decided to use an 8-bit

	cycle count	code size*	RAM words*
state-of-the-art	11342	1000	18
bitsliced	6473	1194	144
DPL protected	182572	2674	192

* The state-of-the-art code size and RAM words are given for encryption + decryption, while ours are for encryption only. Code size and RAM words are given in bytes.

Table 1: Comparison of implementations of PRESENT.

Atmel AVR device as the target architecture so we could compare our results to the state-of-the-art which was presented at AfricaCrypt 2012 [6]. The first step was to write a bitsliced implementation of the PRESENT cipher in AVR assembly code. This was done by hand and the author of this code was learning AVR assembly while writing it so there must be a lot of possible optimizations. Despite this fact, the resulting implementation is faster than the state-of-the-art. The code actually requires 4.5 times more clock cycles, but due to the bitslicing it can cipher 8 blocks at the same time and is thus 1.75 times faster. However, our implementation uses 8 times more memory. Table 1 sums up these results.

The code was then automatically translated to abstractASM. This step does not change the code size or anything since it's a direct mapping from AVR assembly to abstractASM.

The obtained abstractASM code was then DPLized according to the method presented in Section 3. This transformation multiplied the code size by a little more than 2 and the number of clock cycles by approximately 3.5. It also multiplied the RAM needs by a factor of 1.3. The additional memory is used for the DPL lookup tables, as described in Section 3. Table 1 sums up these numbers.

The resulting protected code was then executed on the abstractCPU. The execution confirmed that the protected code was leak-free using the techniques developed in Section 4.

The code was then mapped back from abstractASM to AVR assembly which compiled properly after a few hand tweaking: the DPL macros expanded the code sufficiently to break small-jump branching (*i.e.*, the target labels were more than 64 instructions away after the code transformation), which was easy to fix.

The resulting protected AVR assembly code ran successfully on the Atmel AVR ATMega163.

6 Conclusion

We have automatically protected an assembly code against side-channel attack by transforming it so it has the same semantic and behavior as before, but respect the dual-rail with precharge logic (DPL) protocol, which is a balancing

countermeasure against side-channel attacks.

Independently, we have certified an assembly code against side-channel attacks by proving it to be leak-free using formal methods directly on the implementations after having modeled the leakage (as the Hamming distance of updates of sensitive values). By doing so, we have shown that formal methods (*e.g.*, symbolic evaluation) can help proving non-functional security properties on critical cryptographic code.

These methods have been put in pratice with the PRESENT algorithm running on an AVR micro-controller. The result of this case study are very good with a provably leak-free implementation that is at least 2 times faster than the state-of-the-art of protected (with a 2nd order masking scheme) implementations of PRESENT.

Our approach to proving resistance to side-channel analysis is also an enabler for optimizations on protected code as it can be used as non-regression test for the leak-free security property.

Our results are promising and we believe formal methods have a bright future concerning side-channel attacks countermeasures certification for trustable cryptosystems. The method we developed would also be beneficial to other code level protections if we can model timing leakage, correctness of masking schemes, or countermeasures to fault injection.

References

[1] 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.

[2] 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.

[3] Zhimin Chen, Ambuj Sinha, and Patrick Schaumont. Implementing virtual secure circuit using a custom-instruction approach. In Vinod Kathail, Reid Tatge, and Rajeev Barua, editors, *CASES*, pages 57–66. ACM, 2010.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] Paul C. Kocher. From Proof to Practice: Real-World Cryptography (invited talk). In *CHES*, volume 3156 of *Lecture Notes in Computer Science*. Springer, August 11-13 2004. Cambridge, MA, USA.

[12] 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.

[13] 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/`.

[14] 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.

[15] 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.

[16] 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.

[17] Francesco Regazzoni, Alessandro Cevrero, François-Xavier Standaert, Stéphane Badel, Theo Kluter, Philip Brisk, Yusuf Leblebici, and Paolo Ienne. A Design Flow and Evaluation Framework for DPA-Resistant Instruction Set Extensions. In *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 6-9 September 2009. Lausanne, Switzerland.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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 AbstractASM's BNF

```
Prog    ::= ( Inst? [ ';' <comment> ] '\n' )*
Inst    ::= Opcode0
          | Opcode1 Val
```

```
          | Opcode2 Dst Val
          | Opcode3 Dst Val Val
          | Branch3 Val Val Val
Opcode0 ::= 'nop' | 'dbg'
Opcode1 ::= 'jmp'
Opcode2 ::= 'not' | 'mov'
Opcode3 ::= 'and' | 'orr' | 'xor'
          | 'shl' | 'shr'
          | 'add' | 'mul'
Branch3 ::= 'beq' | 'bne'
Loc     ::= 'r' <register-number>
          | '@' <memory-address>
Imm     ::= '#' <immediate-value>
Rel     ::= '$' <relative-code-address>
Dst     ::= Loc | '!' ( Loc | Imm ) ( ',' <offset> )?
Val     ::= Dst | Imm | Rel
```