# The Conjoined Microprocessor

Ehsan Aerabi[1], A. Elhadi Amirouche[2], Houda Ferradi[3], Rémi Géraud[3], David Naccache[2,3], and Jean Vuillemin[3]

[1] Hamedan University of Technology
Mardom Street, Shahid Fahmide Boulevard
IR-3733-1-65169, Hamedan, Iran
aerabi@hut.ac.ir
[2] Sorbonne Universités – Université Paris II
12 Place du Panthéon, F-75231, Paris, France
allel-elhadi.amirouche@etudiants.u-paris2.fr
[3] École normale supérieure, Département d'informatique
45, rue d'Ulm, F-75230, Paris CEDEX 05, France
given_name.family_name@ens.fr

**Abstract.** Over the last twenty years, the research community has devised sophisticated methods for retrieving secret information from side-channel emanations, and for resisting such attacks. This paper introduces a new CPU architecture called the Conjoined Microprocessor (C$\mu$P). The C$\mu$P can randomly interleave the execution of two programs at very low extra hardware cost. We developed for the C$\mu$P a preprocessor tool that turns a target algorithm into two (or more) separate queues like $\mathcal{Q}_0$ and $\mathcal{Q}_1$ that can run in alternation. $\mathcal{Q}_0$ and $\mathcal{Q}_1$ fulfill the same operation as the original target algorithm. Power-analysis resistance is achieved by randomly alternating the execution of $\mathcal{Q}_0$ and $\mathcal{Q}_1$, with different runs resulting in different interleavings. Experiments reveal that this architecture is indeed effective against CPA.

## 1 Introduction

Over the last twenty years, the research community has devised sophisticated methods for retrieving secret information from side-channel emanations and for resisting such attacks. We assume that the reader is familiar with side-channel attacks such as SPA, DPA [11] and CPA [2] that we do not re-describe here.

This paper introduces a new CPU architecture called the *Conjoined Microprocessor* (C$\mu$P). The C$\mu$P can be seen as the electronic equivalent of conjoined twins. In biology the term *conjoined twins* designates identical twins joined *in utero*. Conjoined twins usually share a few vital organs such as the heart or the liver. The C$\mu$P has one ALU (Arithmetic and Logical Unit) and one RAM space but two independent register banks and two independent stack areas. This enables the C$\mu$P to take two queues of codes that modify the same RAM space and alternate randomly their execution. C$\mu$P hardware relies on a preprocessor that transforms the target algorithm into (at least) two separate code queues like $\mathcal{Q}_0$

and $\mathcal{Q}_1$ run in alternation. $Q_0$ and $Q_1$ fulfill the same operation as the original target algorithm. Thus, $\mathcal{Q}_0$ and $\mathcal{Q}_1$ co-operate to alternatively modify the same RAM space and perform a specific computational task. The number of queues in this method is not limited and could be increased with low hardware overhead.

Attacks such as CPA or DPA exploit a device's power consumption at a specific clock cycle $t_0$. The statistical exploitation of data-related power variations usually requires a large number of re-runs. CPA and DPA are based on the observation that during $t_0$, secret data being processed and power consumption are correlated. Algorithms are often designed to run in constant time to thwart timing attacks [10], but constant-time implementations enable attackers to align the power traces of consecutive runs, focus on $t_0$, and perform statistical analysis. The opponent can then guess the secret data (usually a byte of it) and check whether this guess is correct. A correct guess will produce a power trace that resembles the real consumption at $t_0$.

Random execution of independent instructions or *shuffling* is a popular category of SCA prevention methods. There are different methods in hardware and software. Hardware-only solutions (*e.g.* [14]) generally do dependency checking and parallel execution using hardware at run-time which demands for significantly larger area and run-time costs for wastefully and repetitively checking a deterministic algorithm which remain fixed forever in a device live cycle (*e.g.* a smart card) while it can be done just once. Software-only solutions (*e.g.* [17, 18, 20]) mostly utilize shuffling approaches for a specific cryptographic algorithm and lack generality. They also considerably increase code size or diminish system performance.
The rationale of the C$\mu$P idea is to check dependency among instructions and code blocks at compile time, fill two queues $\mathcal{Q}_0$ and $\mathcal{Q}_1$ with independent instructions or code blocks which can run in parallel and randomly alternating between them on the fly. This can prevent SCA by changing the clock cycles at which sensitive data is being processed between executions. Even if an attacker succeeds in targeting a given clock cycle $t$, power consumption at $t$ also depends on the instructions executed before and after $t$. Thereby the C$\mu$P hinders the attacker's efforts at the expense of a small overhead on the device.
We used existing rich previous research in compilers and parallel computing to find independent instructions and blocks which can be run in parallel as well as a specific parallelization method that can be applied exclusively for C$\mu$P architecture. A similar method [19] was published when we developing and testing C$\mu$P idea that uses similar compiler approaches to partition a code into several independent code blocks $B_1$ , ..., $B_b$ all with equal size of $S$. It uses a custom "`shuffle b,S`" instruction which makes the cpu to identify blocks margins. Then a random number generator generates a sequence of $b$ random numbers between [1,$b$] which are used by cpu to randomly shuffle among blocks. This is a flexible approach as the number of blocks could easily change but this flexibility produce extra hardware cost as the RNG should be designed to work with variable number of blocks. In contrast C$\mu$P can use a very simple RNG which generates 0 or 1 and pick either $\mathcal{Q}_0$ or $\mathcal{Q}_1$ for execution. Code blocks in [19] should be equal

size which means compiler should fill up the shorter queues with dummy `nop` instructions. In comparison in C$\mu$P, $\mathcal{Q}_i$s could be at any size and there's no need for dummy instructions. More over [19] generally relies on block parallelization because when a block is chosen, it should be run entirely to its end before starting another block. But C$\mu$P can enjoy not only from block-level parallelization but also from fine grained instruction-level parallelization and on each instruction, cpu flips a coin and chooses either of $\mathcal{Q}_i$s. Finally, C$\mu$P has a specific architecture which helps to use an exclusive parallelization method which will be discussed in Section 3. All of the above mentioned characteristics helped C$\mu$P to demand for more than 2000 times traces to break as this value is 366 for [19].

The advantages of the C$\mu$P architecture are the following:

- An opponent trying to exploit side-channel leakage is faced with the problem of correctly partitioning operations in time. In addition, the power consumption at time $t$ does not only depend on $\texttt{opcode}_t$ but also on $\texttt{opcode}_{t-1}$. If these opcodes belong to two different $\mathcal{Q}_i$s, the power traces of the two processes will influence and blur each other.

- The insertion of random wait-states is a popular countermeasure (*e.g.* [4–6]) but wait-states impact system performance by slowing down computations (*i.e.* when the system marks a halt to mislead the opponent, time is inevitably lost). Alternating between two fully *useful* processes does not cause any time loss and preserves *global* system determinism. Previous works using shuffling methods focus on specific algorithms (*e.g.* [8,13,17]) or demand large runtime resources such as time and power [14].

- Because conjoining does not duplicate the ALU and the RAM (which are the most expensive chip parts in terms of surface) but only a few registers, conjoining is relatively cheap.

- Finally, by turning off one of the two processes, C$\mu$P runs code with total backward compatibility. Conversely, by just skipping the conjoining opcodes, C$\mu$P code is automatically serialized on-the-fly for a non-conjoined $\mu$P.

As we will subsequently see, the main challenge in designing the C$\mu$P is not that much the C$\mu$P's hardware design but the compilation of code for the C$\mu$P.

Section 2 provides technical background. Section 3 overviews the methods for alternating code used in the compilation process for the C$\mu$P. Section 4 provides implementation details. Section 5 illustrates the additional resistance to CPA and DPA brought by the C$\mu$P. We did not subject the C$\mu$P to higher-order or non-linear power analyses[4]. We also expect the C$\mu$P to increase resistance against other side-channels such as EM radiation or heat dissipation, although we did not perform such experiments as yet.

[4] That being said, such advanced methods are usually more sensitive to noise than CPA or DPA, and require more traces [7].

This paper is trimmed to comply with page limitations. A full version will be posted on the IACR ePrint server.

## 2 Technical Background

### 2.1 Reordering Constraints for Dependent Instructions

The C$\mu$P is designed for the interleaved non-deterministic execution of two (or more) instruction streams[5]. This requires all possible code interleaving configurations to be equivalent. To guarantee the equivalence of two instances of an algorithm, we use Theorem 1 of [9]:

**Theorem 1.** *Any reordering transformation $\phi$ that preserves every dependence in a program preserves the meaning of that program.*

In this theorem the word *dependence* refers to Write-after-Write (WaW), Write-after-Read (WaR) or Read-after-Write (RaW) relationships between instructions. Recall that WaW, WaR and RaW are *data hazards* that occur when instructions exhibiting data dependence modify data concurrently. We illustrate these hazards with simple examples in Table 1 assuming that, to perform correctly a given computational task, $\texttt{opcode}_1$ must *precede* $\texttt{opcode}_2$. The problematic variable in the following three examples is `x`.

| | RaW | WaR | WaW |
|---|---|---|---|
| $\texttt{opcode}_1$ | `x := a` | `a := x` | `x := a` |
| $\texttt{opcode}_2$ | `b := x` | `x := b` | `x := b` |

Table 1: Different types of dependence

There are three situations in which a data hazard can occur:

– **RaW (true dependence):** $\texttt{opcode}_2$ tries to read a variable before $\texttt{opcode}_1$ could write it. *i.e.* $\texttt{opcode}_2$ refers to a result that has not been calculated yet.

  In Table 1. $\texttt{opcode}_1$ saves a value in `x` and $\texttt{opcode}_2$ is expected to move this value into `b`. Hence, $\texttt{opcode}_1$ and $\texttt{opcode}_2$ do not commute. This is a data dependence because $\texttt{opcode}_2$ *depends* on the successful completion of $\texttt{opcode}_1$.

[5] In the following sections, instruction streams will also be referred to as "queues", "programs" or "codes" and will be denoted by $\mathcal{Q}_0$ and $\mathcal{Q}_1$.

– **WaR (anti-dependence):** $\texttt{opcode}_2$ tries to write a variable (`x`) before $\texttt{opcode}_1$ could read it. A brief look at the WaR column of Table 1 shows that $\texttt{opcode}_1$ and $\texttt{opcode}_2$ do not commute.

– **WaW (output dependence):** In this configuration $\texttt{opcode}_2$ tries to write a variable (`x`) before $\texttt{opcode}_1$ writes it. Here as well, opcodes do not commute[6].

Conversely, a transformation $\phi$ does not yield an equivalent program ($\phi$ is an *invalid transformation*) if $\phi$ changes the order of two $\texttt{opcode}_i$s referring to a same memory location when at least one of these $\texttt{opcode}_i$s is a write. Otherwise $\phi$ is defined as *valid*.

Kennedy *et al.* [9] introduced a set of methods for loop dependence testing and parallelization which are complete and adequate for our purpose. Kotha *et al.* [12] applied a subset of these methods to parallelize binary codes. Both approaches are integrated in the C$\mu$P compilation toolkit.

## 3 Parallelization and Alternation

To recompile a program $\mathcal{P}$ into the two alternatable queues of codes $\mathcal{Q}_0$ or $\mathcal{Q}_1$, we had plenty of previous research around code compilation methods for parallel computing except that in our method, parallel codes run interleaved not simultaneous. We adapted compilation tools [9] to generate code that can be run safely in alternation, and combined these tools with optimization strategies in [12] to overcome the lack of symbolic information, missing function indices and implicit induction variables.

[9] is comprised of various methods for dependency checking in loop iterations[7]. For example in AddRoundKey function in AES, all of the 16 iterations are independent exclusive-or's between a byte of the key and data. Therefore, first 8 iterations could be run in $\mathcal{Q}_0$ and other 8 iterations in $\mathcal{Q}_1$. But loop iterations are not always independent. In this cases methods in [9] generally fails and we use a specific algorithm to obtain more in parallelization at instruction-level. This algorithm can be used exclusively for C$\mu$P because of its specific architecture and we will discuss it later in this Section. We integrated all of these methods in a preprocessor tool.

### 3.1 Synchronizing Instruction Queues

It is sometimes necessary to pause the execution of one $\mathcal{Q}_i$, to make sure that executing instructions in two queues never violates dependence. This requires

[6] The reader may question the usefulness of overwriting `x`. Such a situation may occur if `x` is an I/O port for instance.

[7] *e.g.* SIV and ZIV tests.

adding new synchronization opcodes to the C$\mu$P's instruction-set. Instruction queues are synchronized using the (new) *barrier* (`brr`) and *carrier* (`crr`) instructions. In the following, `brr` and `crr` instructions will be referred to by the generic notation `xrr`.

- `brr` is used to enforce the correct order of execution of two instructions belonging to different $\mathcal{Q}_i$s. Each `brr` is followed by a memory address (`brr M`). The queue issuing a `brr` instruction will be stalled until the other queue executes the instruction at address `M`. Table 2 gives a 68HC05 example for `brr`. Assume the `lda mem1` in $\mathcal{Q}_0$ must be processed before the `sta mem1` in $\mathcal{Q}_1$. `brr` stalls $\mathcal{Q}_1$ until $\mathcal{Q}_0$ reaches `LABEL1`.

- `crr` is used to transfer a register value from one queue to another. Each `crr` is followed by a memory address `M` and a register name `Y` (`crr M,Y`). The queue $\mathcal{Q}_i$ issuing the `crr` instruction will be stalled until $\mathcal{Q}_{1-i}$ executes the instruction at address `M`, and then the contents of register `Y` are carried (copied) from $\mathcal{Q}_{1-i}$ to the $\mathcal{Q}_i$. Table 3 illustrates the use of `crr`. Suppose that the `add` in $\mathcal{Q}_1$ requires the value loaded into register `A` from `mem1` in $\mathcal{Q}_0$. `crr` waits until `lda` finished, then it transfers the contents of $Q_0$'s `A` into $\mathcal{Q}_1$'s `A`. Then the `add` instruction is processed. Passing values between $\mathcal{Q}_i$s could be a source of power information leakage and we will cover that later in Section 4.

| $\mathcal{Q}_0$ | $\mathcal{Q}_1$ |
|---|---|
| ... | ... |
| lda mem1 | brr LABEL1 |
| LABEL1: | sta mem1 |
| ... | ... |

Table 2: Barrier synchronization instruction `brr`.

| $\mathcal{Q}_0$ | $\mathcal{Q}_1$ |
|---|---|
| ... | ... |
| lda mem1 | crr LABEL2,A |
| LABEL2: | add 0x20 |
| ... | ... |

Table 3: Carrier synchronization instruction `crr`.

### 3.2 Algorithm for Instruction-Level Parallelization

The $k$-th instruction of a program is denoted by `opcode`$[k]$. We denote by $D[k]$ and $S[k]$ the sets of all *destination* instructions (an instruction that must be executed *after* `opcode`$[k]$) and *source* instructions (an instruction that must be executed *before* `opcode`$[k]$), respectively; $\mathcal{Q}_i[k]$ is the queue that `opcode`$[k]$ belongs to ; $n[k]$ is the number of instructions that must be executed before instruction `opcode`$[k]$ can be assigned[8].

Finally, we introduce $R[k] \in \{0, 1\}$, the *recommended queue index*, which is determined from the instructions in $S[k]$.

Figure 1 illustrates this process for a short program: seven instructions affect registers `A` (*accumulator*) and `X` (*index*). Their dependence graph is illustrated in Figure 2a. Each node is an instruction and each directed edge is a dependence between two instructions. The `incx` at line 4 has the attribute $D[4] = \{(\texttt{X}, 6), (\texttt{X}, 7)\}$ because its writing of register `X` must be executed before line 6 and 7 and the `lda` at line 1 has the attribute $D[1] = \{(\texttt{mem1}, 6), (\texttt{A}, 3)\}$.

```
1   lda mem1 ;      A ← mem1
2   ldx mem2 ;      X ← mem2
3   inca     ;      A ← A+1
4   incx     ;      X ← X+1
5   sta mem2 ;      mem2 ← A
6   stx mem1 ;      mem1 ← X
7   mul      ;      X:A = 256*X+A ← A*X
```

Fig. 1: Sample program and opcode attributes.

Now each instruction must be assigned to a queue for the program to be effectively executed. The assigning process is described in Algorithm 1. Each iteration begins by selecting a node having no input edge, or having all its adjacent source nodes assigned. We refer to this node as the *current* node. The algorithm then assigns the current node to a queue and ends when all nodes are assigned. Choosing the appropriate queue is based on two conditions:

1. Already assigned instructions can assign a queue to their destinations. This is called *recommendation*.
2. If the node has no recommendation, it is assigned to the shorter queue.

After determining the appropriate queue, and assigning the current node to the queue, Algorithm 1 updates the nodes that depend on the current node.

[8] $n[k]$ is initialized by $\#S[k]$ and decremented whenever one of the sources of `opcode`$[k]$ is assigned to a queue. When $n[k]$ reaches zero, `opcode`$[k]$ can be safely assigned.

To find out if a synchronizing instruction `xrr` is required, the algorithm checks whether a source instruction of the current node has been assigned to the other queue or not; if so, it inserts an appropriate `brr` or `crr`.
It can be easily proven that if a program can be sequentially run, then a deadlock in its parallel version is not possible.

### 3.3 A Toy Example

To show how Algorithm 1 works, we apply it to the code of Figure 1. In Figure 2, bold rectangles show the instruction processed by Algorithm 1 at each step. Plain arrows represent the analyzed code's dependences. Bold arrows denote recommendation paths and gray nodes represent assigned opcodes that will not be processed again.

**Algorithm 1:** Generating queues from straight-line target code.

**Data**: Program $\mathcal{P}$
**Result**: Queues $\mathcal{Q}_0$ and $\mathcal{Q}_1$
*// Initialization*
**for** $k \in \{1, 2, ...,$ *size of program* $\mathcal{P}\}$ **do**
  $D[k] \leftarrow$ destinations of `opcode`$[k]$
  $S[k] \leftarrow$ sources of `opcode`$[k]$
  $n[k] \leftarrow \#S[k]$
  $\mathcal{Q}_i[k] \leftarrow -1$ *//assigned queue*
  $\mathcal{R}_i[k] \leftarrow -1$ *//recommendation queue*
**end for**
*// Assigning nodes*
**while** *there is an unassigned instruction in* $\mathcal{P}$ **do**
  find $k$ s.t. $n[k] = 0$ and $\mathcal{Q}_i[k] = -1$
  **if** $\mathcal{R}[k] \neq -1$ **then**
    $\mathcal{Q}_i[k] \leftarrow \mathcal{R}[k]$
  **else**
    Assign `opcode`$[k]$ to the shortest queue
  **end if**
  **if** $\exists (R, J) \in S[k]$ *s.t.* $\mathcal{Q}_i[J] \neq \mathcal{R}[k]$ **then**
    Insert required `xrr` instructions before `opcode`$[k]$ in $\mathcal{R}[k]$
  **end if**
  Recommend $\mathcal{Q}_i[k]$ to appropriate nodes in $D[k]$
  **for** $(r, \ell) \in D[k]$ **do**
    $n[\ell] \leftarrow n[\ell] - 1$
  **end for**
**end while**

The procedure starts by considering the instruction at line 1, which is assigned to $\mathcal{Q}_0$, while queue $\mathcal{Q}_0$ is recommended to the 3$^{\text{rd}}$ instruction (Figure 2b). The algorithm then proceeds to the next sourceless node, which is the 2$^{\text{nd}}$ instruction

(Figure 2c). Now the 3rd and 4th instructions have a recommended queue equal to 0 and 1 respectively, so the algorithm will assign the 3rd instruction to $\mathcal{Q}_0$ and the 4th to $\mathcal{Q}_1$. Note that the 3rd instruction and its source (1st inst.) have been assigned to the same queue, so that no `xrr` is required (Figure 2d); the same remark applies to the 4th instruction (Figure 2e).

However, instruction 5 has been sent to $\mathcal{Q}_0$, but has a source (2nd inst.) that is assigned $\mathcal{Q}_1$. A `brr` is therefore inserted to ensure instruction 2 has been processed on $\mathcal{Q}_1$ (Figure 2f). The same reasoning applies to instruction 6 (Figure 2g).

The algorithm proceeds to the 7th and last instruction, which is assigned to $\mathcal{Q}_0$. This instruction requires the content of registers `A` and `X`. Since X is affected by $Q_1$, a `crr` is necessary to synchronize its value (Figure 2h).

Table 4 illustrates the queues once this procedure has completed. The instruction `brr LABEL_1_1` on $\mathcal{Q}_0$ ensures that this queue will be stalled until `ldx` is processed by $\mathcal{Q}_1$. The `brr LABEL_0_1` instruction on $\mathcal{Q}_1$ has a comparable effect on $\mathcal{Q}_1$. The `crr LABEL_0_2,A` instruction will transfer the contents of register `A` from $\mathcal{Q}_0$ after `inca` has finished.

| Instruction | $\mathcal{Q}_0$ | $\mathcal{Q}_1$ |
|---|---|---|
| 1 | `lda mem1` | `ldx mem2` |
| 2 | `LABEL_0_1:` | `LABEL_1_1:` |
| 3 | `inca` | `incx` |
| 4 | `LABEL_0_2:` | `brr LABEL_0_1` |
| 5 | `brr LABEL_1_1` | `stx mem1` |
| 6 | `sta mem2` | `crr LABEL_0_2,A` |
| 7 | | `mul` |

Table 4: Queues after applying Algorithm 1.

# 4 Implementation

## 4.1 The C$\mu$P Architecture

The C$\mu$P (Figure 3) has two separated register banks, an *Alternator* and a common core including an ALU and control logic.

Each register bank duplicates registers `A`, `X`, the Flags, Stack Pointer and Program Counter. The *Multiplexer* can change the data path between the two banks based on the queue being run. The Alternator is also composed of a *Shuffler Switch*

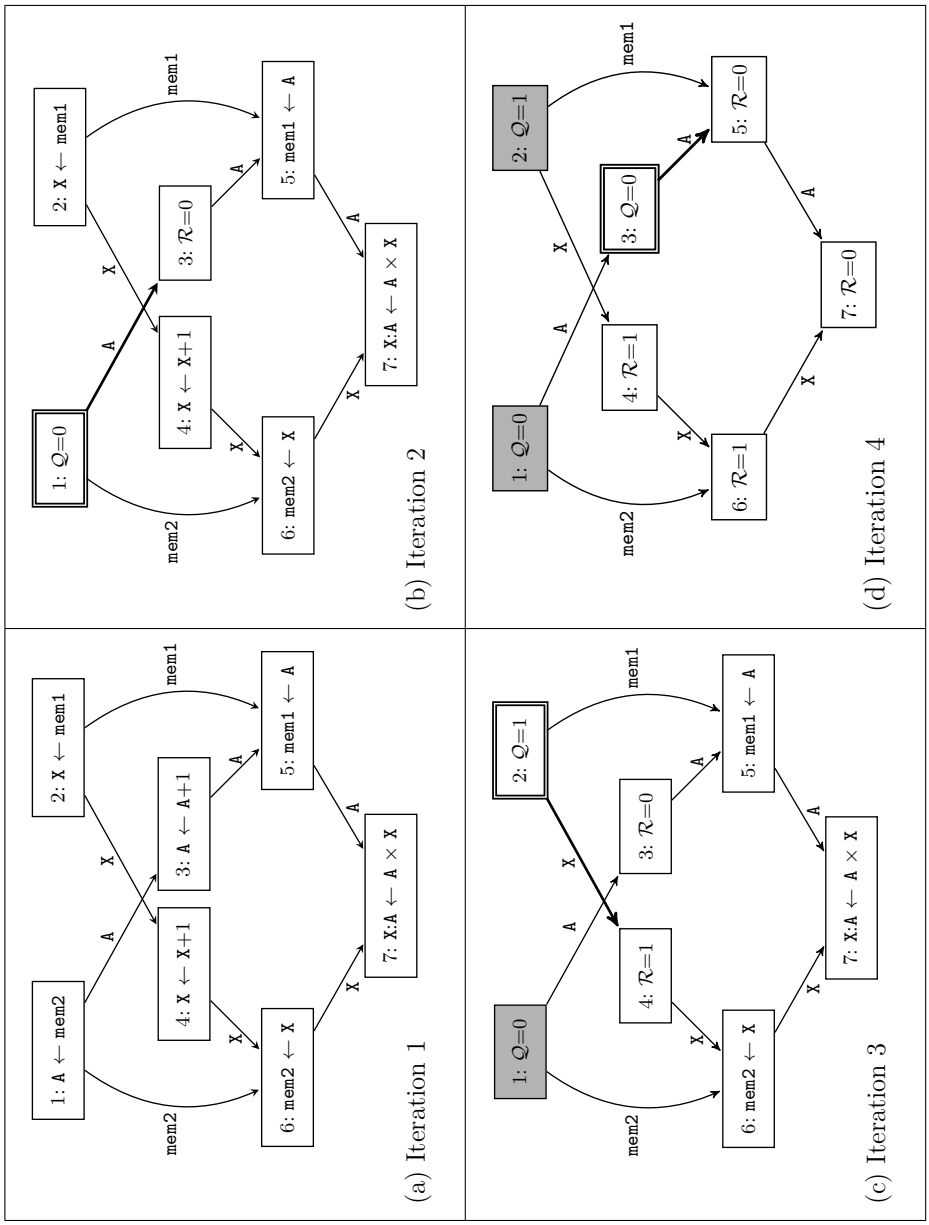


Fig. 2: Iterations of Algorithm 1 on Figure 1.

that randomly shuffles two instruction streams and a *Thread Synchronizer* that checks if the fetched instruction is an `xrr`. The Alternator controls that $\mathcal{Q}_0$ and $\mathcal{Q}_1$ are run in correct order using `xrr`s or carries a register value from one queue to another. This value might be a secret information which we aim to protect and

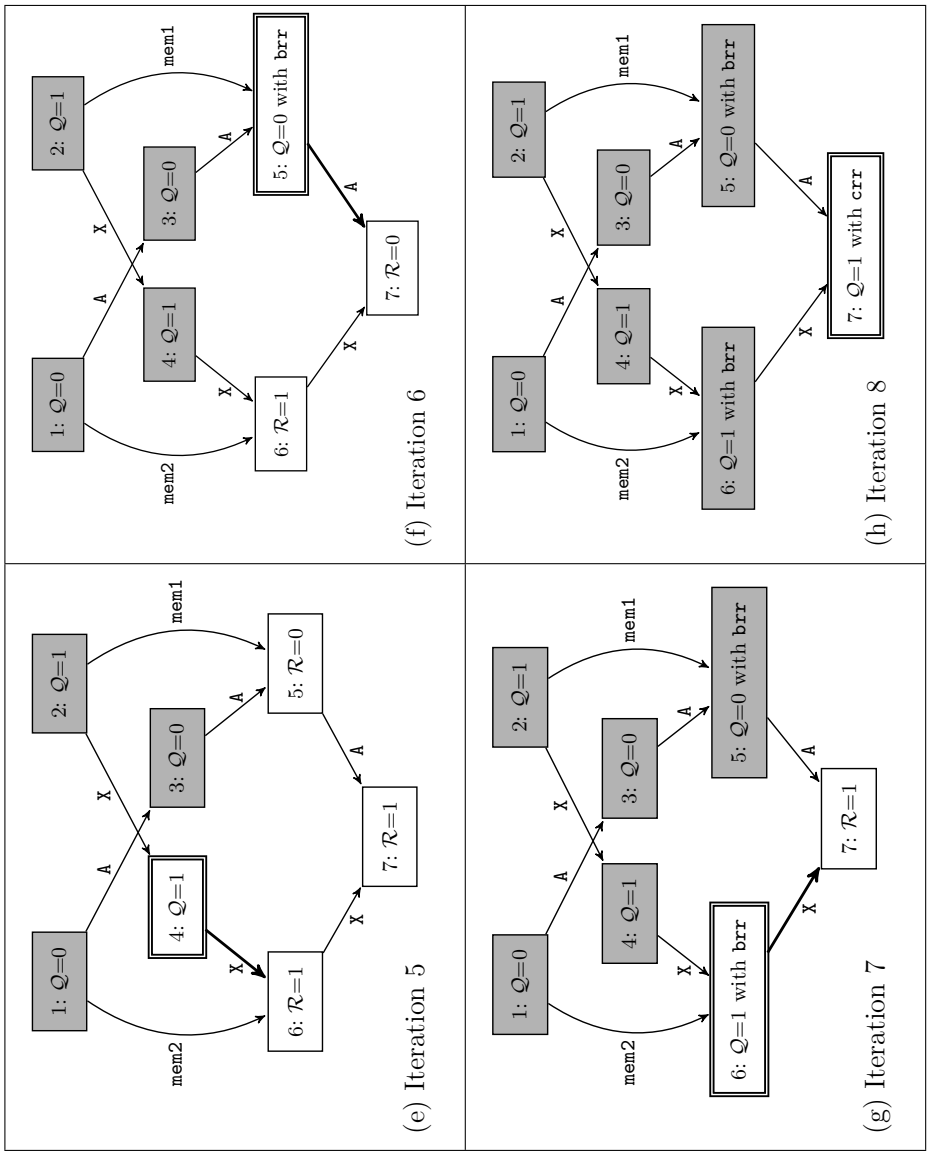


Fig. 2: Iterations of Algorithm 1 on Figure 1 (cont'd).

moving it may lead to information leakage through the device power. In order to neutralize mentioned effect, we used a leakage model [2, 15] based on Hamming distance which provides a linear estimation of power consumption. The model consists in measuring the number of bit flips (transitions from 0 to 1 or *vice versa* at the transistor level) from an original (unknown) rest state. Therefore the power consumption in changing a register from its old value $R_{old}$ to a new

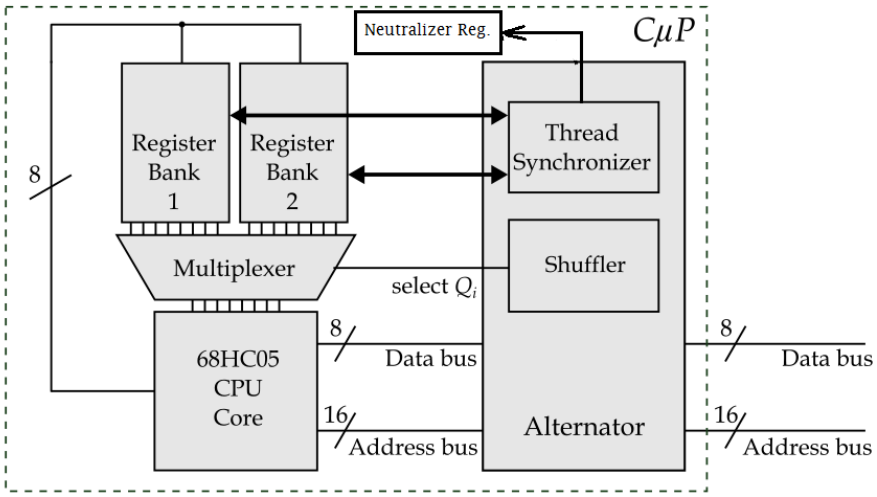


Fig. 3: C$\mu$P architecture.

value $R_{new}$ is related to:

$$\text{HammingDist}(R_{old}, R_{new}) \tag{1}$$

We tried to keep all Hamming distances or bit flips constant in every changes, regardless of the old and new values. This need a simple circuitry shown in figure 4. In this figure register R is the register which accept a new value. It is accompanied by another neutralizer register N. New and old values of R are connected to an exclusive-or gate and the output is connected to another exclusive-nor gate with the old value of N. This circuit grantees to keep all accumulated bit changes in R and N to be constant. In C$\mu$P we need only one neutralizer register which is used to neutralize all values are exchanged by `crr`.

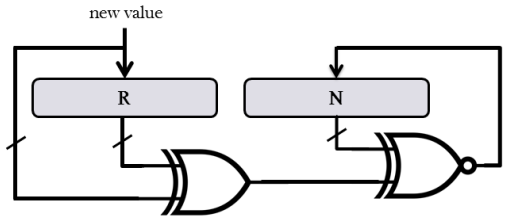


Fig. 4: Neutralizing effect of value exchange in `crr` instruction.

### 4.2 Proportional Shuffling

Algorithm 1 decides to assign instructions with no dependencies to the shorter queue to strive to keep queues equal in size. But dependencies may affect the symmetry in queue sizes. So the shuffler must be capable of shuffling in proportion of queue sizes. Proportional shuffling results in two queues with fair execution time shares based on their sizes so as to minimize the idle time of short queues. To that end a random number generator outputs a queue index $i \in \{0, 1\}$ with probabilities $L_0/(L_0+L_1)$ and $L_1/(L_0+L_1)$ respectively, where $L_i$ is the size of $Q_i$. To construct this dynamically biased generator we used the hardware permutation technique of [3] and [1]. These keyed hardware permutation generators are fast enough to produce an element $i \in \{0, 1\}$ at each clock cycle. Using a permutation function $\Pi(k, L)$ where $k$ is the key and $L = L_0 + L_1$ is the length of permutation, it is possible to generate a random permutation $\Pi$ of $(1, 2, \ldots, L_0 + L_1)$ and then construct a switch $S(i, \Pi)$ that returns 0 if $i$ appears in $\Pi$ at a position $\ell_i$ such that $1 \leq \ell_i \leq L_i$, and returns 1 if $L_0 + 1 \leq \ell_i \leq L_0 + L_1$. In other words $S(i, \Pi)$ acts as an oracle informing the C$\mu$P which queue must be unleashed to run instruction $i$.

### 4.3 Choosing a CPU Core

The C$\mu$P was implemented in VHDL. The design includes a 68HC05 core[9], the Alternator, ROM, two clone register banks and a 32-bit LFSR (as a simple Shuffler). Since we have simulated the C$\mu$P on ModelSim 6.3, no I/O ports were required. All input and output vectors are stored in files by the VHDL simulator. The implementation was used to generate the simulated power traces necessary for side-channel evaluation. We used the Hamming distance variation of all registers to model dynamic power consumption [16].

We also developed a code splitting tool in C++. The tool accepts a program in 68HC05 assembly, extracts all dependences into several graphs, applies the alternation algorithms to these graphs and outputs two binary code streams. The binary code is then imported into ROM. Our method requires the C$\mu$P to implement the two new `xrr` opcodes. Luckily the 68HC05 instructions table has several unused opcodes (*e.g.* `0x90`, `0x91` and `0x92`). We hence assigned `0x90`, `0x91`, `0x92` and `0x93` to `brr`, `crr A`, `crr X` and `crr flags` instructions respectively. If $\mathcal{Q}_0$ contains an instruction such as `brr 0x1234` then $\mathcal{Q}_0$ should be stalled until $\texttt{PC}_2$ reaches `0x1234`. Likewise, if $\mathcal{Q}_0$ contains an instruction such as `crr A,0x1234` then $\mathcal{Q}_0$ should be stalled until $\texttt{PC}_2$ reaches `0x1234` and then the contents of register `A` of $\mathcal{Q}_1$ would be transferred to the register `A` of $\mathcal{Q}_0$.

[9] Recall that the Motorola 68HC05 core has an 8-bit accumulator `A`, an 8-bit index register `X`, a 16-bit Program Counter `PC` and a few basic flags[10]. The CPU can address $\ell \leq 2^{16} = 64$k one-byte memory locations denoted $M[0], ..., M[\ell - 1]$.

### 4.4 Area and Speed Penalty

Table 5 compares the simulation results of a standard 68HC05 with those of its corresponding C$\mu$P on Xilinx Spartan-6 FPGA. Adding an extra register bank and the code alternation circuits increased area by about 2% in terms of registers and 7% in LUTs and decreased speed by about 4%.

| **Design** | **Registers** | **LUTs** | **Frequency** |
|---|---|---|---|
| Standard 68HC05 | 137 | 1288 | 120 MHz |
| C$\mu$P 68HC05 | 165 | 1375 | 115 MHz |
| **Ratio between designs** | 1.02 | 1.07 | 0.96 |

Table 5: Performance and Area on a Xilinx xc6slx4-3tqg144 FPGA

## 5 AES Experiments

To benchmark our C$\mu$P design, we developed a naive 128-bit AES in 68HC05 assembly and implemented the cpu core and memory on a Xilinx evaluation board. AES was recompiled as a parallel program which comprised of 660 instructions including 64 `brr`s and no `crr`s. As AES and other block ciphers usually work separately on sub-bytes of input data, they are easily parallelized by splitting round loops into two and hence don't need any `crr` instruction. `brr` is a fast two-cycle instruction and its overall execution time was only %0.047 of an AES operation.

We then mounted a simple Correlation Power Attack (CPA) [2] that targets the AddRoundKey subroutine during the final AES encryption round. In AddRoundKey, the secret key is exclusive-ored (`eor` instruction) with data. Knowing implementation details, spotting this `eor` is easy. Since the 68HC05 is an eight-bit microprocessor, the 128-bit final round key is processed by a 16-round loop, each iteration of which performs an 8-bit `eor`. We attack the first 8 bits of the key in the first iteration.

To ensure that the 68HC05 AES implementation is vulnerable to CPA, we performed the attack with the C$\mu$P working as a single queue CPU. We ran 10000 encryptions using random inputs with a fixed key and gathered all power traces.

The correct key causes a significant spike in correlation coefficient as is shown in Figure 5 where first four subkeys are shown in different colors. Each spike represents the most correlated key value for one of the four (8-bit) subkeys.

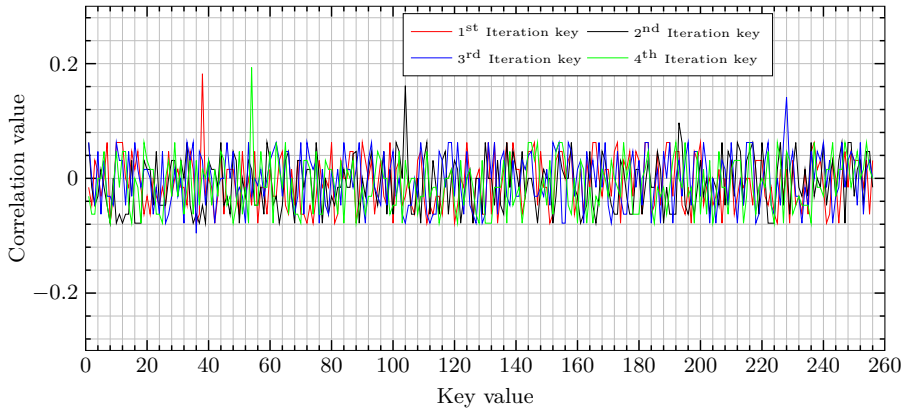


Fig. 5: CPA: Correct key guess for four 8-bit subkeys (standard 68HC05)

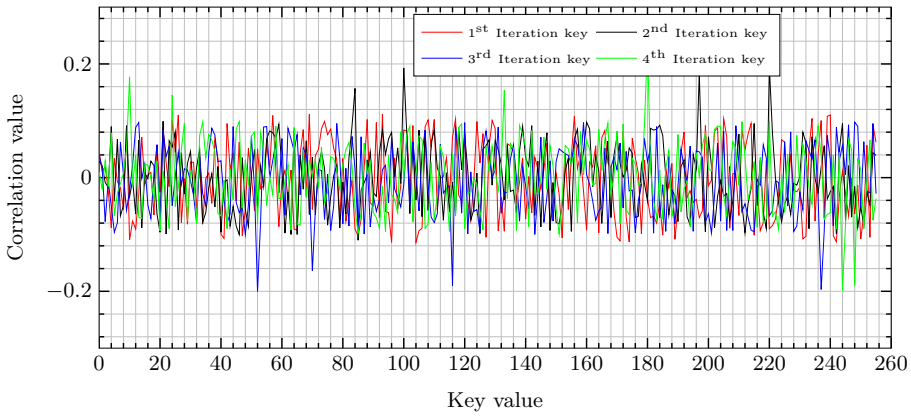


Fig. 6: CPA: Correct key attempts for four 8-bit subkeys (C$\mu$P 68HC05)

Next, we have run the same attack on the C$\mu$P. Figure 6 shows the result of CPA targeting four bytes of the final sub-keys. Figure 7 illustrates how three `eor`s in the final round of AES (1$^{st}$, 2$^{nd}$ and 8$^{th}$) were scattered over the CPU cycles for 200 execution instances. In this experiment, CPA fails to identify the correct key.

### 5.1 Analysis for ultimate resistance of C$\mu$P

C$\mu$P's effectivness is based on two observations. First, in many runs of AES, `eor` take places on cycles other than the target clock cycle $t_0$. Second, in those executions, other instructions might run on $t_0$ including other `eor`s from other rounds of the algorithm. These two observations would reduce the correlation between the data being processed and device power and cause masking noise on

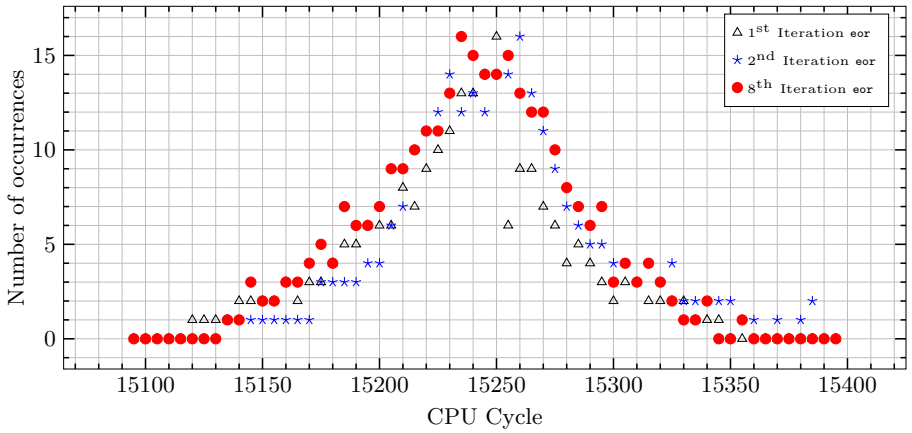


Fig. 7: Temporal distribution of three `eor`s in final AES round. 200 Executions.

$t_0$ to hinder the attacker's effort.
In order to understand ultimate resistance of C$\mu$P architecture, we give an analysis here which is applicable to any other shuffling approach.
First we define device's power consumption components based on [15]:

$$P_{total} = P_{exp} + P_{sw.noise} + P_{el.noise} + P_{constant} \tag{2}$$

$P_{exp}$ is the exploitable power consumption which is produced by processing secret data and is attacker's source of information for key hypothesis testing. It obviously has a correlation with the hamming distance of data changes. $P_{sw.noise}$ is the switching noise of the logical operations in chip which has no dependency with the secret data being processed. $P_{el.noise}$ is device's power leakage independent from the logical operations and data changes and finally $P_{constant}$ which is constant device's leakage has no advantage for the attack process. Following equation describes how signal to noise ratio can negatively affect correlation coefficient between hamming distances and exploitable power consumption on $t_0$:

$$\rho(HD, P_{total}) = \frac{\rho(HD, P_{exp})}{\sqrt{1 + \frac{1}{\mathrm{SNR}}}} \tag{3}$$

which $\rho\left(HD, P_{total}\right)$ is the correlation between the hamming distance of plain and encrypted data and the device's total power usage at the target moment $t_0$. In this equation, $SNR$ can be calculated as follows:

$$\mathrm{SNR} = \frac{\sigma^2 P_{exp}}{\sigma^2 \left(P_{sw.noise} + P_{el.noise}\right)} \tag{4}$$

$P_{exp}$,$P_{sw.noise}$ and $P_{el.noise}$ could be described as a normal distribution [15]. So we tried to find couple of mean and standard deviation $(\mu, \sigma)$ for all of them. By

capturing device power in idle time we obtained $P_{el.noise} \sim \mathcal{N}(0, 12.02\text{mV})$ and by capturing mean values when it was processing random values we obtained $P_{sw.noise} \sim \mathcal{N}(180.0\text{mV}, 2.85\text{mV})$. To estimate $P_{exp}$, device's power on $t_0$ was observed by taking mean values of 200 operations. Table 6 illustrates device power consumption along with hamming distance of the bit flips as well as number of occurrences. Based on the values in Table 6, $P_{exp}$ can be described by $\mathcal{N}(180.0\text{mV}, 1.72\text{mV})$.

| **HD** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| **Occurrences** | 4 | 6 | 19 | 45 | 51 | 42 | 22 | 8 | 3 |
| **Power** | 173.1 | 174.8 | 176.5 | 178.3 | 180.0 | 181.8 | 183.5 | 185.1 | 186.8 |

Table 6: Distribution of bit flips hamming distance and power usage

Now we can analyse minimum required power traces to distinguish correct key guess for both unprotected and C$\mu$P system. Based on [15] minimum needed power trace to mount a successful correlation attack can be calculated as follows:

$$n = \frac{28}{\rho^2\left(HD, P_{total}\right)} \tag{5}$$

For unprotected system all the `eor`s are run on $t_0$ hence we can assume $\rho^2(HD, P_{exp}) \approx 1$ and $P_{sw.noise} \approx 0$ then using (3) and (4):

$$\rho_{unprot.}(HD, P_{total}) = \frac{1}{\sqrt{1 + \frac{12.02^2}{1.72^2}}} \approx 0.141 \tag{6}$$

This value confirms the correlation coefficient which experimentally obtained in Figure 5. It also implies that at least 1394 traces were enough to achieve the same result in Figure 5 using (5).

To conduct the same analysis on C$\mu$P, we denote exploitable power consumption of the device by $P_{exp.C\mu P}$ . In order to calculate $\rho_{C\mu P}\left(HD, P_{total}\right)$ from (3( we should recalculate $\rho_{C\mu P}(HD, P_{exp})$ and SNR. Hence we need mean values and variances for exploitable power and switching noise $E(P_{exp.C\mu P})$, $E(P_{sw.C\mu P})$, var$(P_{exp.C\mu P})$ and var$(P_{sw.C\mu P})$ on $t_0$. Assume that proportion of traces which `eor` run on $t_0$ in C$\mu$P be $\alpha$. This case is like combining two different sets of events with proportion of $\alpha$ and $1 - \alpha$. If we denote mean value and variance of exploitable or switching power consumption for both sets with $(\mu_\alpha, \sigma_\alpha)$ and $(\mu_{1-\alpha}, \sigma_{1-\alpha})$, we would have:

$$
\begin{aligned}
E(P_{exp.C\mu P}) &= \frac{\alpha.n.\mu_\alpha + (1-\alpha).n.\mu_{1-\alpha}}{\alpha.n + (1-\alpha).n} = \alpha.\mu_\alpha + (1-\alpha).\mu_{1-\alpha} \\
\mathrm{var}(P_{exp.C\mu P}) &= \frac{\alpha.n.\left[\sigma_\alpha^2 + \left(\mu_\alpha - E(P_{exp.C\mu P})\right)^2\right]}{\alpha.n + (1-\alpha).n} \\
&\quad + \frac{(1-\alpha).n.\left[\sigma_{1-\alpha}^2 + \left(\mu_{1-\alpha} - E(P_{exp.C\mu P})\right)^2\right]}{\alpha.n + (1-\alpha).n} \\
&= \alpha.\left(\sigma_\alpha^2 + \left(\mu_\alpha - E(P_{exp.C\mu P})\right)^2\right) \\
&\quad + (1-\alpha).\left(\sigma_{1-\alpha}^2 + \left(\mu_{1-\alpha} - E(P_{exp.C\mu P})\right)^2\right)
\end{aligned}
\tag{7}
$$

It is obvious that in $\alpha.n$ traces, $P_{exp.C\mu P} \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha)$ holds the properties of $P_{exp} \sim \mathcal{N}(180.0\text{mV}, 1.72\text{mV})$ and the $(1-\alpha).n$ remainder traces have no exploitable information so $\mu_{1-\alpha} = 180\text{mV}$ and $\sigma_{1-\alpha} = 0\text{mV}$. Regarding to the Figure 7, in 16 traces, `eor` is run on $t_0$ which mean $\alpha = 0.08$. Hence $E(P_{exp.C\mu P}) = 180.0\text{mV}$ and $\mathrm{var}(P_{exp.C\mu P}) = 0.236\text{mV}^2$ using (7). With similar reasoning, in $(1-\alpha).n$ instructions, $P_{sw.C\mu P} \sim \mathcal{N}(\mu_{1-\alpha}, \sigma_{1-\alpha})$ holds the properties of $P_{sw.noise} \sim \mathcal{N}(180.0\text{mV}, 2.85\text{mV})$ and the $\alpha.n$ traces have no switching noise. Using similar equations $E(P_{sw.C\mu P}) = 180.0\text{mV}$ and $\mathrm{var}(P_{sw.C\mu P}) = 7.472\text{mV}^2$. $\mathrm{SNR}_{C\mu P}$ is obtained as follows regarding the independence of $P_{el.noise}$ and $P_{sw.noise}$:

$$
\mathrm{SNR}_{C\mu P} = \frac{\mathrm{var}(P_{exp.C\mu P})}{\mathrm{var}(P_{sw.C\mu P} + P_{el.noise})} = \frac{\mathrm{var}(P_{exp.C\mu P})}{\mathrm{var}(P_{sw.C\mu P}) + \mathrm{var}(P_{el.noise})} = 0.0015
\tag{8}
$$

If we denote set of $(1-\alpha).n$ shuffled instructions which has no exploitable information by $S_{1-\alpha}$ and $\alpha.n$ unshuffled ones by $S_{\alpha.n}$ then $\rho_{C\mu P}(HD, P_{exp})$ can be calculated by Pearson's correlation formula:

$$
\begin{aligned}
\rho_{C\mu P}(HD, P_{exp.C\mu P}) &= \frac{\sum_{i=1}^{n} HD_i.P_{i.exp} - n.\overline{HD}.\overline{P_{exp}}}{(n-1).\sigma_{HD}.\sigma_{Pexp}} \\
&= \frac{\sum_{i\in S_\alpha} HD_i.P_{i.exp} - \alpha.n.\overline{HD}.\overline{P_{exp}}}{(n-1).\sigma_{HD}.\sigma_{Pexp}} \\
&\quad + \frac{\sum_{i\in S_{1-\alpha}} HD_i.P_{i.exp} - (1-\alpha).n.\overline{HD}.\overline{P_{exp}}}{(n-1).\sigma_{HD}.\sigma_{Pexp}} \\
&= \frac{\sum_{i\in S_\alpha} HD_i.P_{i.exp} - \alpha.n.\overline{HD}.\overline{P_{exp}}}{\alpha.(1/\alpha).(n-1).\sigma_{HD}.\sigma_{Pexp}} \\
&= \alpha.1 = \alpha
\end{aligned}
\tag{9}
$$

Now $\rho_{C\mu P}(HD, P_{total})$ can be calculated using 8 and 9:

$$\rho_{C\mu P}(HD, P_{total}) = \frac{\alpha}{\sqrt{1 + \frac{1}{\mathrm{SNR}}}} = \frac{0.08}{\sqrt{1 + 0.0015}} = 0.0031 \tag{10}$$

which means at least 2813303 power traces is needed to mount same attack on C$\mu$P using (5). In other words, CPA resistance for C$\mu$P is 2016 times more than unprotected version.

Another try to hinder attacker's effort is to increase the number of code queues to achieve more parallelization and obfuscation which reduces $\alpha$ and adds more switching noise. Adding more queue requires a new more register bank and also modification in shuffler.

## 6 Conclusion

This paper introduced the use of randomized instruction interleaving as a side-channel countermeasure. The new CPU architecture interleaves randomly the execution of two instruction queues. These queues are generated from a single target algorithm, whose functionality is preserved.

Since the instructions' execution order varies between runs, fixed patterns in power consumption vanish, and the device resists side-channel attacks such as SPA and CPA.

## References


1. V. Bagini and G. Morgari. *Keyed permutations for fast data scrambling.* European Transactions on Telecommunications, volume 17, issue 1, pp. 1-9, 2006.
2. E. Brier, C. Clavier, and F. Olivier. *Correlation power analysis with a leakage model.* Cryptographic Hardware and Embedded Systems - CHES 2004, LNCS 3156, pp. 16-29. Springer, 2004.
3. E. Brier, H. Handschuh, and C. Tymen. *A fast primitives for internal data scrambling in tamper resistant hardware.* Cryptographic Hardware and Embedded Systems - CHES 2001, LNCS 2162, pp. 16-28, Springer, 2001.
4. C. Clavier, J-S. Coron, and N. Dabbous. *Differential power analysis in the presence of hardware countermeasures.* Cryptographic Hardware and Embedded Systems - CHES 2000, LNCS 1965, pp. 252-263. Springer, 2000.
5. J-S. Coron and I. Kizhvatov. *An efficient method for random delay generation in embedded software.* Cryptographic Hardware and Embedded Systems - CHES 2009, LNCS 5747, pp. 156-170. Springer, 2009.
6. J-S. Coron and I. Kizhvatov. *Analysis and improvement of the random delay countermeasure of CHES 2009.* Cryptographic Hardware and Embedded Systems - CHES 2010, LNCS 6225, pp. 95-109. Springer, 2010.
7. B. Gierlichs, L. Batina, and P. Tuyls. *Mutual Information Analysis – A Universal Differential Side-Channel Attack.* Cryptology ePrint Archive, Report 2007/198.

8. A. A. Kamal and A. M. Youssef. *An area-optimized implementation for AES with hybrid countermeasures against power analysis.* International Symposium on Signals, Circuits and Systems - ISSCS 2009, pp. 1âĂŞ4. 2009.
9. K. Kennedy and J. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., 2002.
10. P. Kocher. *Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems.* Advances in Cryptology - CRYPTO'96, LNCS 1109, pp. 104-113. Springer, 1996.
11. P. Kocher, J. Jaffe, and B. Jun. *Differential power analysis.* Advances in Cryptology - CRYPTO'99, LNCS 1666, pp. 388-397. Springer-Verlag, 1999.
12. A. Kotha, K. Anand, M. Smithson, G. Yellareddy and R. Barua. *Automatic parallelization in a binary rewriter.* IEEE/ACM International Symposium on Microarchitecture - MICRO '43, pp. 547-557. 2010.
13. F. Madlener, M. Stoettinger, and S. A. Huss. *Novel hardening techniques against differential power analysis for multiplication in GF(2n).* International Conference on Field-Programmable Technology - ICFPT 2009. pp. 328-334. 2009.
14. D. May, H. L. Muller, and N. P. Smart. *Non-deterministic processors.* Information Security and Privacy - ACISP 2001, Vol. 2119, pp. 115-129. 2001.
15. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards.* Springer, 2007.
16. K. Tiri and I. Verbauwhede. *Simulation models for side-channel information leaks.* Design Automation Conference âĂŞ DAC'05, pp. 228-233. 2005.
17. S. Tillich, C. Herbst, and S. Mangard. *Protecting AES software implementations on 32-bit processors against power analysis.* Applied Cryptography and Network Security (ACNS), pp. 141-157. 2007.
18. Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. *An aes smart card implementation resistant to power analysis attacks.* In Jianying Zhou, Moti Yung,and Feng Bao, editors, ACNS, volume 3989 of Lecture Notes in Computer Science, pages 239-252, 2006.
19. A. Galip Bayrak, N. Velickovic, P. Ienne and W. Burleson. *An architecture-independent instruction shuffler to protect against side-channel attacks.* In TACO, 8(4), 2012.
20. Matthieu Rivain, Emmanuel ProuK, and Julien Doget. *Higher-order masking and shuffing for software implementations of block ciphers.* In Christophe Clavier and Kris Gaj, editors, CHES, volume 5747 of Lecture Notes in Computer Science, pages 171-188. Springer, 2009.