Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity

[Published in *IEEE Transactions on Computers* 53(6):760-768, 2004.]

Benoît Chevallier-Mames[1], Mathieu Ciet[2], and Marc Joye[1]

[1] Gemplus S.A., Card Security Group
La Vigie, Av. du Jujubier, ZI AThélia IV, 13705 La Ciotat Cedex, France
{benoit.chevallier-mames,marc.joye}@gemplus.com
http://www.gemplus.com/smart/
[2] UCL Crypto Group, Université catholique de Louvain
Place du Levant 3, 1348 Louvain-la-Neuve, Belgium
ciet@dice.ucl.ac.be — http://www.dice.ucl.ac.be/crypto/

Abstract. This paper introduces simple methods to convert a cryptographic algorithm into an algorithm protected against simple side-channel attacks. Contrary to previously known solutions, the proposed techniques are not at the expense of the execution time. Moreover, they are generic and apply to virtually any algorithm.
In particular, we present several novel exponentiation algorithms, namely a protected square-and-multiply algorithm, its right-to-left counterpart, and several protected sliding-window algorithms. We also illustrate our methodology applied to point multiplication on elliptic curves. All these algorithms share the common feature that the complexity is globally unchanged compared to the corresponding unprotected implementations.

Keywords. Cryptographic algorithms, side-channel analysis, protected implementations, atomicity, exponentiation, elliptic curves.

1 Introduction

According to Goldreich [1], cryptography deals with the conceptualization, definition and construction of computing systems that address security concerns. We would like to add that cryptography is also concerned with concrete implementations of such systems. This in turn implies that not only the systems but also *their implementations* must withstand any abuse or misuse.

Basically, there are two main families of implementation attacks: *faults attacks* [2] and *side-channel attacks* [3, 4]. This paper only deals with the second family of attacks and more precisely with *simple* (*i.e.* non-differential) side-channel attacks.

Suppose that (part of) an algorithm consists of a loop where the execution of a given set of instructions depends on certain input values. If from some side-channel information (*e.g.* timing or power consumption) one can distinguish which set of instructions is processed, then one can retrieve some secret data (if any) involved during the course of the algorithm. This is the basic idea behind simple side-channel attacks.

For example, imagine that, at a given step, a secret bit is used to select process Π_0 or Π_1. A straightforward counter-measure against simple side-channel attacks consists in making processes Π_0 and Π_1 indistinguishable. This is usually achieved by executing process Π_0 followed by a fake execution of process Π_1 when process Π_0 must be executed, and by executing a fake execution of process Π_0 followed by process Π_1 when process Π_1 must be executed. Such a solution is however unsatisfactory from a computational perspective because the running time can be increased by a non-negligible factor.

In a sense, our approach refines this obvious solution, as much as possible. By potentially inserting dummy (fake) operations, we divide each process so that it can be expressed as the repetition of instruction blocks which appear equivalent by side-channel analysis. Such a block is called a *side-channel atomic block*. Building on this, we develop several approaches for unrolling the *whole* code so that it appears as an *uninterrupted* succession of the processes. In other words, the whole code appears as a succession of blocks that are *indistinguishable* by simple side-channel analysis. Remarkably, contrary to previous solutions, the techniques we propose are *inexpensive* and present the additional advantage of being fully *generic*, *i.e.* they apply to a large variety of cryptographic algorithms.

The rest of this paper is organized as follows. In the next section, we define the notion of *side-channel atomicity*. We explain how to efficiently convert a cryptographic algorithm into an algorithm protected against simple side-channel attacks. Then, in Sections 3 and 4, we provide concrete applications to the RSA cryptosystem and to elliptic curve cryptosystems. Finally, we conclude in Section 5.

2 Side-Channel Atomicity

2.1 Side-channel atomic blocks

We view a process as a sequence of instructions. We say that two instructions (or a sequence thereof) are *side-channel equivalent* if they are indistinguishable through side-channel analysis. This relation is denoted by symbol "$\sim$". From this, we define what we call a *common side-channel atomic block*.

Definition 1 (Side-channel atomicity [5]). *Given a set of processes* $\{\Pi_0, \ldots, \Pi_n\}$, *a* common side-channel atomic block Γ *for* $\Pi_0, \ldots, \Pi_n$ *is a side-channel equivalent sequence of instructions so that each process* Π_j $(0 \leq j \leq n)$ *can be expressed as the repetition of this block* Γ, i.e. *there exist sequences* $\gamma_{j,i} \sim \Gamma$ *s.t.* $\Pi_j = \gamma_{j,1} \| \gamma_{j,2} \| \cdots \| \gamma_{j,\ell_j}$. *The instruction sequences* $\gamma_{j,i}$ *are called* side-channel atomic blocks.

A common side-channel atomic block Γ always exists by noticing that fake instructions can be artificially added to an existing process to make the different processes indistinguishable. The main difficulty resides in finding a block Γ which is small with respect to some metric (*e.g.* running time, code size, ...). We note that a possible rearranging and/or rewriting of the processes may shorten Γ or limit the number of dummy operations and thus improve the overall performances.

2.2 Illustration

Before going further, we quote a simple example: the square-and-multiply algorithm. On input of an element x in a (multiplicatively written) group $\mathbb{G}$ and the binary expansion of exponent d, $d = (d_{m-1}, \ldots, d_0)_2$, the square-and-multiply algorithm returns $y = x^d$.

Input: $x, d = (d_{m-1}, \ldots, d_0)_2$
Output: $y = x^d$

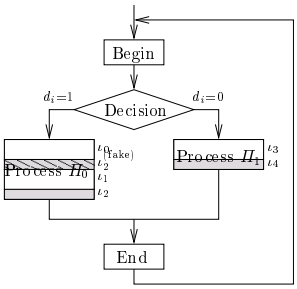
$R_0 \leftarrow 1$; $R_1 \leftarrow x$; $i \leftarrow m-1$
while $(i \geq 0)$ **do**
  $R_0 \leftarrow (R_0)^2$
  if $(d_i = 1)$ **then** $R_0 \leftarrow R_0 \cdot R_1$
  $i \leftarrow i - 1$
endwhile
return R_0

Fig. 1. (Unprotected) square-and-multiply algorithm.

As aforementioned, a valid choice for common side-channel atomic Γ consists of a squaring followed by a (possibly fake) multiplication and a counter decrementation. This algorithm is the well-known square-and-multiply *always* algorithm [6]. This is the classical way for preventing simple side-channel attacks in the square-and-multiply algorithm.

Such a choice for Γ is suboptimal. Assuming that (i) a squaring operation can be performed by calling the (hardware) multiplication routine, and (ii) instructions $R_0 \leftarrow R_0 \cdot R_0$ and $R_0 \leftarrow R_0 \cdot R_1$ are side-channel equivalent,[1] we can rewrite the previous algorithm to clearly reveal a shorter common side-channel block Γ (see Fig. 2-a). Remark the fake instruction $i \leftarrow i - 0$. Of course, we assume that this instruction is side-channel equivalent to $i \leftarrow i - 1$.
As depicted in Fig. 2-a, the algorithm is not balanced. There are two copies of Γ when $d_i = 1$ and only one when $d_i = 0$. However, as explained in the next section, it is easy to unroll the code so that a side-channel analysis only reveals a regular succession of copies of Γ without enabling to make the distinction

[1] These assumptions are discussed in Section 3.1.



(a) Synopsis.

Input: $x, d = (d_{m-1}, \ldots, d_0)_2$
Output: $y = x^d$

$R_0 \leftarrow 1$; $R_1 \leftarrow x$; $i \leftarrow m-1$
$k \leftarrow 0$
while $(i \geq 0)$ **do**
  $R_0 \leftarrow R_0 \cdot R_k$
  $k \leftarrow k \oplus d_i$; $i \leftarrow i - \neg k$
endwhile
return R_0

(b) Side-channel atomic square-and-multiply algorithm.

Fig. 2. Protected square-and-multiply algorithm.

amongst the processes being executed (*i.e.* Π_0 or Π_1). After simplification, we obtain the algorithm presented in Fig. 2-b.

It is worth noting that our protected algorithm (Fig. 2-b) only requires $1.5m$ multiplications, on average, for computing $y = x^d$, that is, the complexity of the usual, unprotected square-and-multiply algorithm (Fig. 1).[2]

2.3 General methodology

Given different processes $\Pi_0, \ldots, \Pi_n$, we first identify a common side-channel atomic block Γ. Next, we write the processes Π_j $(0 \leq j \leq n)$ as a repetition of Γ, *i.e.* $\Pi_j = \gamma_{c_j} \| \cdots \| \gamma_{c_j+\ell_j-1}$ where ℓ_j is the number of copies of Γ in Π_j,

$$\begin{cases} c_0 = 0 \\ c_j = c_{j-1} + \ell_{j-1} \quad \text{for } 1 \leq j \leq n \end{cases}$$

and with $\gamma_k \sim \Gamma$ for all $c_0 \leq k \leq c_n + \ell_n - 1$. Our strategy is to execute *exactly* ℓ_j times a sequence side-channel equivalent to Γ for process Π_j. As a result, denoting by t the running time for Γ, the time required for processing Π_j will only be $\ell_j \cdot t$ instead of $(\max_{0 \leq j \leq n} \ell_j) \cdot t$ for the trivial solution.

In order to chain the different processes, we use a bit, say s, to keep track when there are no more blocks $\gamma_k \sim \Gamma$ to be executed when processing Π_j. When process Π_j is terminated (and thus $s = 1$), we have to execute the next process according to the input values of the algorithm. Moreover, at the beginning of each loop, we update k, the number of the current sequence γ, as

$$\boxed{k \leftarrow (\neg s) \cdot (k+1) + s \cdot f(\text{input values})}$$

[2] As a side-effect, it also leaks the Hamming weight of the exponent. While this is generally not an issue, we note that the Hamming weight can be masked using standard techniques (*e.g.* blinding or splitting).

so that $f(\text{input values}) = c_{j'}$ if the next process to be executed is $\Pi_{j'}$. We see that when $s = 0$ then the value of k is incremented by 1. Of course, the above expression for k must be coded in such a way that no information about the input values is revealed from a given side-channel.

Alternatively, k can be defined as a counter in the current process; the updating step then becomes: $k \leftarrow (\neg s) \cdot (k + 1)$. The input values are used to make the distinction amongst the different atomic blocks.

The last step consists in expressing each atomic block γ_k:

- explicitly as the elements of a table, or
- implicitly as a function of k and s (and the input values).

3 Side-Channel Atomic RSA Exponentiation

The most widely used public-key cryptosystem is the RSA [7]. Its basic operation is the (modular) exponentiation, which is usually carried out with the square-and-multiply algorithm. A side-channel atomic version of the square-and-multiply algorithm is given in Fig. 2 (see also [8]). This section presents a protected version of the ω-bit sliding-window exponentiation algorithm for any $\omega > 1$.[3] It also presents a simplified version for $\omega = 2$ as well as a right-to-left variant for $\omega = 1$. All these new algorithms use the implicit approach.

3.1 Assumptions

Our methodology supposes that a simple side-channel analysis does not allow making the distinction between the different atomic blocks (cf. Definition 1). As a consequence, the atomic blocks are device-dependent. From most present-day smart cards equipped with an arithmetic co-processor, our experience shows that the following operations are side-channel equivalent:[4]

1. The (modular) multiplication of two large registers: $R_i \cdot R_j$, for all i, j. This includes the case $i = j$, provided that the squaring operation is carried out by a call to the hardware multiplication (not to the hardware squaring);
2. The (modular) addition/subtraction of two large registers: $R_i \pm R_j$, for all i, j;
3. The CPU operations, *i.e.* all arithmetical and logical operations manipulating the CPU registers. If the hardware does not satisfy the assumption, resistance against side-channel analysis can be obtained by a software implementation. For example, if the hardware evaluation of $b \cdot A$ behaves differently whether bit $b = 0$ or 1, a simple trick consists in evaluating bA as $(b+t)A - tA$ for a random t; similarly, the addition $A \pm b$ can be evaluated as $A \pm (b+t) \mp t$;

[3] The square-and-multiply algorithm corresponds to the case $\omega = 1$.
[4] This is even more true when hardware countermeasures are activated.

4. The equality testing of two CPU registers: $A \stackrel{?}{=} B$. Again, if this is not satisfied by the hardware, a software emulation could for example read the zero flag resulting from $A \oplus B$ or perform an OR on all bits of $A \oplus B$;
5. The loading/storing of values from different registers.

[The algorithms presented in this section and in Section 4 assume hardware implementations (or software emulations thereof) satisfying the above conditions.]

3.2 Generic sliding-window algorithm

When additional registers are available, the expected amount of multiplications for evaluating $y = x^d$ can be lowered by precomputing and storing the values of x^{2j+1} for $j \in \{1, \ldots, 2^{\omega-1} - 1\}$ and then by left-to-right scanning exponent bits with an ω-bit sliding window [9, Algorithm 14.85]. This is an efficient extension of the square-and-multiply algorithm [10, 11].

Input: $x, d = (d_{m-1}, \ldots, d_0)_2$, and an integer $\omega > 1$
Output: $y = x^d$
Precomputation: $R_{j+1} \leftarrow x^{2j+1}$ for $1 \leq j \leq 2^{\omega-1} - 1$

$R_0 \leftarrow 1$; $R_1 \leftarrow x$; $i \leftarrow m - 1$
for $j = 1$ **to** $\omega - 1$ **do** $d_{-j} \leftarrow 0$
$s \leftarrow 1$
while $(i \geq 0)$ **do**
  $k \leftarrow (\neg s) \cdot (k + 1)$
  $b \leftarrow 0$; $t \leftarrow 1$; $l \leftarrow \omega$; $u \leftarrow 0$
  for $j = 1$ **to** ω **do**
    $b \leftarrow b \vee d_{i-\omega+j}$; $l \leftarrow l - \neg b$
    $u \leftarrow u + t \cdot d_{i-\omega+j}$; $t \leftarrow b \cdot (2t) + \neg b$
  endfor
  $l \leftarrow l \cdot d_i$; $u \leftarrow [(u + 1) \text{ div } 2] \cdot d_i$
  $s \leftarrow (k = l)$
  $R_0 \leftarrow R_0 \cdot R_{u \cdot s}$
  $i \leftarrow i - k \cdot s - \neg d_i$
endwhile
return R_0

Fig. 3. Side-channel atomic ω-bit sliding-window algorithm.

3.3 Simplified algorithms

A larger value for ω in the ω-bit sliding-widow algorithm speeds up the computations but increases the memory requirements. A choice of particular interest for constrained devices is the case $\omega = 2$. The resulting algorithm is usually

Input: $x, d = (d_{m-1}, \ldots, d_0)_2$
Output: $y = x^d$

$R_0 \leftarrow 1$; $R_1 \leftarrow x$; $R_2 \leftarrow x^3$
$d_{-1} \leftarrow 0$; $i \leftarrow m-1$; $s \leftarrow 1$
while $(i \geq 0)$ **do**
  $k \leftarrow (\neg s) \cdot (k+1)$
  $s \leftarrow s \oplus d_i \oplus (d_{i-1} \wedge (k \bmod 2))$
  $R_0 \leftarrow R_0 \cdot R_{k \cdot s}$
  $i \leftarrow i - k \cdot s - \neg d_i$
endwhile
return R_0

(a) Side-channel atomic (M, M^3) algorithm.

Input: $x, d = (d_{m-1}, \ldots, d_0)_2$
Output: $y = x^d$

$R_0 \leftarrow 1$; $R_1 \leftarrow x$; $i \leftarrow 0$
$k \leftarrow 1$
while $(i \leq m-1)$ **do**
  $k \leftarrow k \oplus d_i$
  $R_k \leftarrow R_k \cdot R_1$
  $i \leftarrow i + k$
endwhile
return R_0

(b) Side-channel atomic right-to-left binary algorithm.

Fig. 4. Further simplified algorithms for constrained devices.

referred to as the (M, M^3) algorithm. The generic algorithm of Fig. 3 can then be simplified to the algorithm given in Fig. 4-a.

In some cases, it is easier to scan bits from the least significant position to the most significant one. There is a right-to-left analogue of the square-and-multiply algorithm for computing $y = x^d$. Analogously to Fig. 2, we can modify it into an algorithm preventing simple side-channel attacks. After simplification, we get the protected right-to-left exponentiation algorithm given in Fig. 4-b.

There are of course numerous possible variants that may be more efficient on a particular given architecture. What is remarkable is that our protected algorithms have roughly the *same* complexity (running time and memory requirements) as their respective unprotected versions.

4 Side-Channel Atomic Elliptic Curve Point Multiplication

Our methodology applies to virtually any algorithm. We show hereafter how to adapt it in the context of elliptic curve cryptography [12]. Two categories of elliptic curves are commonly used [13]: elliptic curves over large prime fields and non-supersingular elliptic curves over binary fields. The basic operation in elliptic curve cryptography consists in computing the multiple of a point, that is, given a point $\boldsymbol{P_1}$ on an elliptic curve, one has to compute $\boldsymbol{P_d} = d\boldsymbol{P_1}$. To ease the presentation, we assume that this is carried out with the (additive version of the) square-and-multiply algorithm. Other methods are discussed in [14]. Our methodology readily applies to those implementation choices as well.

4.1 Elliptic curves defined over large prime fields

Consider the elliptic curve E defined over a prime field $\mathbb{F}_p$ (with $p > 3$) given by the Weierstraß equation

$$E_{/\mathbb{F}_p} : y^2 = x^3 + ax + b \ .$$

To avoid field inversion, Jacobian coordinates are generally used [13] for representing points on E. With Jacobian coordinates, the doubling of $\boldsymbol{P_1}$ is $2(X_1, Y_1, Z_1) = (X_3, Y_3, Z_3)$ where

$$X_3 = M^2 - 2S,\ Y_3 = M(S - X_3) - T,\ Z_3 = 2Y_1Z_1$$

with $M = 3X_1^2 + aZ_1^4$, $S = 4X_1Y_1^2$ and $T = 8Y_1^4$. The sum of two (distinct) points $\boldsymbol{P_1} = (X_1, Y_1, Z_1)$ and $\boldsymbol{P_2} = (X_2, Y_2, Z_2)$ is (X_3, Y_3, Z_3) where

$$\begin{gathered} X_3 = W^3 - 2U_1W^2 + R^2, \\ Y_3 = -S_1W^3 + R(U_1W^2 - X_3), Z_3 = Z_1Z_2W \end{gathered}$$

with $U_1 = X_1Z_2^2$, $U_2 = X_2Z_1^2$, $S_1 = Y_1Z_2^3$, $S_2 = Y_2Z_1^3$, $W = U_1 - U_2$ and $R = S_1 - S_2$.

As the operations doubling or adding points are somewhat involved, we adopt the explicit approach. We refer the reader to the appendix for the detailed formulæ leading to the expression of atomic blocks γ_k as the rows of matrix:

$$(u^*_{k,l})_{\substack{0\le k\le 25\\ 0\le l\le 9}} = \begin{pmatrix}
4&1&1&5&4&4&3&4&4&5\\
5&3&3&1&1&1&3&1&1&3\\
5&5&5&1&1&3&3&1&1&3\\
5&0&5&4&4&5&3&5&2&2\\
3&3&5&1&1&3&3&1&1&3\\
2&2&2&2&2&2&4&1&1&3\\
5&1&2&1&1&5&5&1&1&5\\
1&4&4&1&1&5&4&1&1&5\\
2&2&2&2&2&2&3&5&1&5\\
4&4&5&2&2&4&2&4&4&5\\
4&9&9&5&1&5&5&5&1&5\\
1&1&4&5&1&5&5&5&1&5\\
4&4&9&5&1&5&5&5&1&5\\
2&2&4&5&1&5&5&5&1&5\\
4&3&3&5&1&5&5&5&1&5\\
5&4&7&2&2&5&5&5&1&5\\
4&3&4&2&2&5&6&6&5&6\\
4&4&8&6&5&6&4&4&2&4\\
3&3&9&6&5&6&6&6&5&6\\
3&3&5&6&5&6&6&6&5&6\\
6&5&5&6&3&6&3&6&3&6\\
1&1&6&1&1&4&4&1&1&4\\
5&5&6&6&1&2&2&6&2&6\\
1&4&4&1&1&5&6&1&1&6\\
2&2&5&1&1&6&3&6&1&6\\
4&4&6&2&2&4&6&6&1&6
\end{pmatrix}.$$

The resulting algorithm is given in the next figure.

Input: $\boldsymbol{P_1} = (X_1, Y_1, Z_1), d = (1, d_{m-2}, \ldots, d_0)_2$, and matrix $(u^*_{k,l})$ as above
Output: $\boldsymbol{P_d} = d\boldsymbol{P_1}$

$R_0 \leftarrow a$; $R_1 \leftarrow X_1$; $R_2 \leftarrow Y_1$; $R_3 \leftarrow Z_1$; $R_7 \leftarrow X_1$; $R_8 \leftarrow Y_1$; $R_9 \leftarrow Z_1$
$i \leftarrow m-2$; $s \leftarrow 1$
while $(i \geq 0)$ **do**
  $k \leftarrow (\neg s) \cdot (k+1)$
  $s \leftarrow d_i \cdot (k \text{ div } 25) + (\neg d_i) \cdot (k \text{ div } 9)$
  $R_{u^*_{k,0}} \leftarrow R_{u^*_{k,1}} \cdot R_{u^*_{k,2}}$; $R_{u^*_{k,3}} \leftarrow R_{u^*_{k,4}} + R_{u^*_{k,5}}$
  $R_{u^*_{k,6}} \leftarrow -R_{u^*_{k,6}}$; $R_{u^*_{k,7}} \leftarrow R_{u^*_{k,8}} + R_{u^*_{k,9}}$
  $i \leftarrow i - s$
endwhile
return (R_1, R_2, R_3)

Fig. 5. Side-channel atomic double-and-add algorithm for elliptic curves over $\mathbb{F}_p$.

Again, it is worth noting that, in terms of field multiplications, this algorithm is as efficient as the corresponding unprotected implementation (cf. [13]).

4.2 Elliptic curves defined over a binary field

Atomicity is a relative notion. In the previous examples, we considered addition and multiplication as basic operations. One could also imagine that division is an basic operation; for instance, if division is provided by a dedicated hardware routine. The more different (from a side-channel perspective) basic operations there are, the more difficult it is to exhibit a common side-channel atomic block Γ. We give hereafter an example involving a division (a rather costly operation) as basic operation.

For efficiency reasons, it is recommended to use affine coordinates for adding points on an elliptic curve defined over a binary field [15]. A (non-supersingular) elliptic curve E defined over the binary field $\mathbb{F}_{2^q}$ is given by the Weierstraß equation

$$E_{/\mathbb{F}_{2^q}} : y^2 + xy = x^3 + ax^2 + b \ .$$

In affine coordinates (cf. [13]), the doubling of point $\boldsymbol{P_1} = (x_1, y_1)$ is $2(x_1, y_1) = (x_3, y_3)$ where

$$x_3 = a + \lambda^2 + \lambda, \quad y_3 = (x_1 + x_3)\lambda + x_3 + y_1$$

with $\lambda = x_1 + (y_1/x_1)$. The sum of two (distinct) points $\boldsymbol{P_1} = (x_1, y_1)$ and $\boldsymbol{P_2} = (x_2, y_2)$ is (x_3, y_3) where

$$x_3 = a + \lambda^2 + \lambda + x_1 + x_2, \quad y_3 = (x_1 + x_3)\lambda + x_3 + y_1$$

with $\lambda = (y_1 + y_2)/(x_1 + x_2)$. We clearly see that doubling and addition of points can be made very similar. As a result, we choose for Γ a whole elliptic curve

doubling or addition (see Fig. 6-a). Only two extra (field) additions are needed for doubling a point compared to the unprotected version of [13].

From this, an efficient protected double-and-add algorithm can then be derived (see Fig. 6-b).

Input: $(T_1, T_2) = \boldsymbol{P_1}$, $(T_3, T_4) = \boldsymbol{P_2}$ Output: $\boldsymbol{P_1} + \boldsymbol{P_2}$ or $2\boldsymbol{P_1}$	
Addition: $\boldsymbol{P_1} \leftarrow \boldsymbol{P_2} + \boldsymbol{P_1}$	Doubling: $\boldsymbol{P_1} \leftarrow 2\boldsymbol{P_1}$
$T_1 \leftarrow T_1 + T_3$ $(= x_1 + x_2)$	$T_6 \leftarrow T_1 + T_3$ (fake)
$T_2 \leftarrow T_2 + T_4$ $(= y_1 + y_2)$	$T_6 \leftarrow T_3 + T_6$ $(= x_1)$
$T_5 \leftarrow T_2/T_1$ $(= \lambda)$	$T_5 \leftarrow T_2/T_1$ $(= y_1/x_1)$
$T_1 \leftarrow T_1 + T_5$	$T_5 \leftarrow T_1 + T_5$ $(= \lambda)$
$T_6 \leftarrow {T_5}^2$ $(= \lambda^2)$	$T_1 \leftarrow {T_5}^2$ $(= \lambda^2)$
$T_6 \leftarrow T_6 + a$ $(= \lambda^2 + a)$	$T_1 \leftarrow T_1 + a$ $(= \lambda^2 + a)$
$T_1 \leftarrow T_1 + T_6$ $(= x_3)$	$T_1 \leftarrow T_1 + T_5$ $(= x_3)$
$T_2 \leftarrow T_1 + T_4$ $(= x_3 + y_2)$	$T_2 \leftarrow T_1 + T_2$ $(= x_3 + y_1)$
$T_6 \leftarrow T_1 + T_3$ $(= x_2 + x_3)$	$T_6 \leftarrow T_1 + T_6$ $(= x_1 + x_3)$
$T_5 \leftarrow T_5 \cdot T_6$	$T_5 \leftarrow T_5 \cdot T_6$
$T_2 \leftarrow T_2 + T_5$ $(= y_3)$	$T_2 \leftarrow T_2 + T_5$ $(= y_3)$
return (T_1, T_2)	

(a) Side-channel atomic elliptic curve addition[5] for elliptic curves over $\mathbb{F}_{2^q}$.

Input: $\boldsymbol{P_1} = (x_1, y_1), d = (1, d_{m-2}, \ldots, d_0)_2$
Output: $\boldsymbol{P_d} = d\boldsymbol{P_1}$

$R_1 \leftarrow x_1$; $R_2 \leftarrow y_1$; $R_3 \leftarrow x_1$; $R_4 \leftarrow y_1$
$i \leftarrow m-2$; $s \leftarrow 1$
while $(i \geq 0)$ **do**
  $k \leftarrow (\neg s) \cdot (k+1)$; $s \leftarrow k \vee (\neg d_i)$
  $R_{6-5k} \leftarrow R_1 + R_3$; $R_{6-4k} \leftarrow R_{3-k} + R_{6-2k}$
  $R_5 \leftarrow R_2/R_1$
  $R_{5-4k} \leftarrow R_1 + R_5$
  $R_{1+5k} \leftarrow (R_5)^2$
  $R_{1+5k} \leftarrow R_{1+5k} + a$
  $R_1 \leftarrow R_1 + R_{5+k}$
  $R_2 \leftarrow R_1 + R_{2+2k}$; $R_6 \leftarrow R_1 + R_{6-3k}$
  $R_5 \leftarrow R_5 \cdot R_6$; $R_2 \leftarrow R_2 + R_5$
  $i \leftarrow i - s$
endwhile
return (R_1, R_2)

(b) Side-channel atomic double-and-add for elliptic curves over $\mathbb{F}_{2^q}$.

Fig. 6. Side-channel atomic elliptic curve algorithms.

5 Conclusion

This paper introduced the notion of common side-channel atomicity. Based on this, novel solutions towards resistance against side-channel attacks were presented. The proposed solutions are generic and apply to a large variety of cryptographic systems. In particular, they apply to exponentiation-based systems for which they lead to protected algorithms having roughly the same efficiency as their straightforward (*i.e.* unprotected) implementations. Finally, it should be noted that our methodology nicely combines with countermeasures against the more sophisticated differential analysis.

[5] In order to save a register, we take advantage of commutativity by computing $\boldsymbol{P_1} \leftarrow \boldsymbol{P_2} + \boldsymbol{P_1}$ instead of $\boldsymbol{P_1} \leftarrow \boldsymbol{P_1} + \boldsymbol{P_2}$ for the elliptic curve addition.

Acknowledgements

We would like to thank the anonymous referees for their useful comments. Part of this work was performed while the second author was visiting Gemplus. Thanks go to David Naccache, Philippe Proust and Jean-Jacques Quisquater for making this arrangement possible.

References

1. O. Goldreich. *Foundations of Cryptography – Basic Tools*. Cambridge University Press, 2001.
2. D. Boneh, R.A. DeMillo, and R.J. Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology – EUROCRYPT '97*, vol. 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, 1997.
3. P.C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.
4. P.C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO '99*, vol. 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.
5. B. Chevallier-Mames and M. Joye. Procédé cryptographique protégé contre les attaques de type à canal caché. Demande de brevet français, FR 28 38 210, April 2002.
6. J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems (CHES '99)*, vol. 1717 of *Lecture Note in Computer Science*, pages 292–302. Springer-Verlag, 1999.
7. R.L. Rivest, A. Shamir, and L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* **21**(2):120–126, 1976.
8. M. Joye. Recovering lost efficiency of exponentiation algorithms on smart cards. *Electronics Letters* **38**(19):1095–1097, 2002.
9. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
10. L.-C.-K. Hui and K.-Y. Lam. Fast square-and-multiply exponentiation for RSA. *Electronics Letters* **30**(17):1396–1397, 1994.
11. K.-Y. Lam and L.-C.-K. Hui. Efficiency of $SS(l)$ square-and-multiply exponentiation algorithms. *Electronics Letters* **30**(25):2115–2116, 1994.
12. I. Blake, G. Seroussi, and N.P. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.
13. IEEE Std 1363-2000. *IEEE Standard Specifications for Public-Key Cryptography*. IEEE Computer Society, August 29, 2000.
14. D.M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms* **27**:129–146, 1998.
15. E. De Win, S. Mister, B. Preneel, and M. Wiener. *On the performance of signature schemes based on elliptic curves*. In *Algorithmic Number Theory Symposium*, vol. 1423 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 1998.
16. M. Joye and C. Tymen. Protections against differential analysis for elliptic curve cryptography: An algebraic approach. In *Cryptographic Hardware and Embedded Systems (CHES 2001)*, vol. 2162 of *Lecture Notes in Computer Science*, pages 377–390. Springer-Verlag, 2001.
17. C.D. Walter. MIST: An efficient randomized exponentiation algorithm for resisting power analysis. In *Topics in Cryptology – CT-RSA 2002*, vol. 2271 of of *Lecture Notes in Computer Science*, pages 53–66. Springer-Verlag, 2002.

A Matrix Representation for the Side-Channel Atomic Double-and-Add Algorithm for Elliptic Curves over $\mathbb{F}_p$

This appendix details how matrix $(u^*_{k,l})$ used in the double-and-add algorithm of Fig. 5 was obtained.

From the point addition formulæ given in Section 4.1, we see that doubling a point requires 10 multiplications and adding two points requires 16 multiplications. In addition to multiplications, adding or doubling points also involve (field) additions/subtractions. Consequently, a common side-channel atomic block, Γ, must at least include one multiplication and one addition (a subtraction can be considered as a special case of negation followed by an addition). Since 1) the formula for adding two (distinct) points requires more multiplications than (field) additions, and 2) the formula for doubling a point requires 11 (field) additions/subtractions, we choose to express Γ with 1 (field) multiplication and 2 (field) additions (along with a negation to possibly perform a subtraction).

We now express the point doubling and point addition as a repetition of blocks side-channel equivalent to Γ. A '$\star$' indicates that any register that does not disturb the course of the algorithm can be selected.

Replacing the '$\star$' by appropriate choices, process Π_0 (doubling followed by an addition) and process Π_1 (doubling) in the double-and-add algorithm can be defined by matrix

$$(u_{k,l})_{\substack{0\le k\le 35\\ 0\le l\le 10}} = \begin{pmatrix}
4&1&1&5&4&4&3&4&4&5&0\\
5&3&3&1&1&1&3&1&1&3&0\\
5&5&5&1&1&3&3&1&1&3&0\\
5&0&5&4&4&5&3&5&2&2&0\\
3&3&5&1&1&3&3&1&1&3&0\\
2&2&2&2&2&2&4&1&1&3&0\\
5&1&2&1&1&5&5&1&1&5&0\\
1&4&4&1&1&5&4&1&1&5&0\\
2&2&2&2&2&2&3&5&1&5&0\\
4&4&5&2&2&4&2&4&4&5&0\\
4&9&9&5&1&5&5&5&1&5&0\\
1&1&4&5&1&5&5&5&1&5&0\\
4&4&9&5&1&5&5&5&1&5&0\\
2&2&4&5&1&5&5&5&1&5&0\\
4&3&3&5&1&5&5&5&1&5&0\\
5&4&7&2&2&5&5&5&1&5&0\\
4&3&4&2&2&5&6&6&5&6&0\\
4&4&8&6&5&6&4&4&2&4&0\\
3&3&9&6&5&6&6&6&5&6&0\\
3&3&5&6&5&6&6&6&5&6&0\\
6&5&5&6&3&6&3&6&3&6&0\\
1&1&6&1&1&4&4&1&1&4&0\\
5&5&6&6&1&2&2&6&2&6&0\\
1&4&4&1&1&5&6&1&1&6&0\\
2&2&5&1&1&6&3&6&1&6&0\\
4&4&6&2&2&4&6&6&1&6&1\\
\hline
4&1&1&5&4&4&3&4&4&5&0\\
5&3&3&1&1&1&3&1&1&3&0\\
5&5&5&1&1&3&3&1&1&3&0\\
5&0&5&4&4&5&3&5&2&2&0\\
3&3&5&1&1&3&3&1&1&3&0\\
2&2&2&2&2&2&4&1&1&3&0\\
5&1&2&1&1&5&5&1&1&5&0\\
1&4&4&1&1&5&4&1&1&5&0\\
2&2&2&2&2&2&3&5&1&5&0\\
4&4&5&2&2&4&2&4&4&5&1
\end{pmatrix}$$

Point doubling

$T_0 \leftarrow a,\ T_1 \leftarrow X_1,\ T_2 \leftarrow Y_1,\ T_3 \leftarrow Z_1$

1. $T_4 \leftarrow T_1 \cdot T_1 \quad (= X_1^2)$
   $T_5 \leftarrow T_4 + T_4 \ (= 2X_1^2)$
   $\star$
   $T_4 \leftarrow T_4 + T_5 \ (= 3X_1^2)$
2. $T_5 \leftarrow T_3 \cdot T_3 \quad (= Z_1^2)$
   $T_1 \leftarrow T_1 + T_1 \ (= 2X_1)$
   $\star$
   $\star$
3. $T_5 \leftarrow T_5 \cdot T_5 \ (= Z_1^4)$
   $\star$
   $\star$
   $\star$
4. $T_5 \leftarrow T_0 \cdot T_5 \quad (= aZ_1^4)$
   $T_4 \leftarrow T_4 + T_5 \ (= M)$
   $\star$
   $T_5 \leftarrow T_2 + T_2 \ (= 2Y_1)$
5. $T_3 \leftarrow T_3 \cdot T_5 \ (= Z_2)$
   $\star$
   $\star$
   $\star$
6. $T_2 \leftarrow T_2 \cdot T_2 \quad (= Y_1^2)$
   $T_2 \leftarrow T_2 + T_2 \ (= 2Y_1^2)$
   $\star$
   $\star$
7. $T_5 \leftarrow T_1 \cdot T_2 \ (= S)$
   $\star$
   $T_5 \leftarrow -T_5 \quad (= -S)$
   $\star$
8. $T_1 \leftarrow T_4 \cdot T_4 \quad (= M^2)$
   $T_1 \leftarrow T_1 + T_5 \ (= M^2 - S)$
   $\star$
   $T_1 \leftarrow T_1 + T_5 \ (= X_2)$
9. $T_2 \leftarrow T_2 \cdot T_2 \quad (= 4Y_1^4)$
   $T_2 \leftarrow T_2 + T_2 \ (= T)$
   $\star$
   $T_5 \leftarrow T_1 + T_5 \ (= X_2 - S)$
10. $T_4 \leftarrow T_4 \cdot T_5 \quad (= -Y_2 - T)$
    $T_2 \leftarrow T_2 + T_4 \ (= -Y_2)$
    $T_2 \leftarrow -T_2 \quad (= Y_2)$
    $\star$

These formulæ assume that multiplication by parameter a (cf. Step 4) behaves identically as a multiplication with another value. If this multiplication can be distinguished by side-channel analysis, elliptic curve operations can be performed on a randomly chosen isomorphic curve [16]. Provided that multiplication by $-a$ cannot be distinguished, another way to prevent side-channel leakage is to replace above Steps 2, 3 and 4 by

2. $T_5 \leftarrow T_3 \cdot T_3$
   $T_1 \leftarrow T_1 + T_1$
   $T_0 \leftarrow -T_0$
   $\star$

3. $T_5 \leftarrow T_5 \cdot T_5$
   $\star$
   $T_5 \leftarrow -T_5$
   $\star$

4. $T_5 \leftarrow T_0 \cdot T_5$
   $T_4 \leftarrow T_4 + T_5$
   $T_0 \leftarrow -T_0$
   $T_5 \leftarrow T_2 + T_2$

.

Point addition

$T_1 \leftarrow X_1,\ T_2 \leftarrow Y_1,\ T_3 \leftarrow Z_1,\ T_7 \leftarrow X_2,\ T_8 \leftarrow Y_2,\ T_9 \leftarrow Z_2$

1. $T_4 \leftarrow T_9 \cdot T_9 \ (= Z_2^2)$
   $\star$
   $\star$
   $\star$
2. $T_1 \leftarrow T_1 \cdot T_4 \ (= U_1)$
   $\star$
   $\star$
   $\star$
3. $T_4 \leftarrow T_4 \cdot T_9 \ (= Z_2^3)$
   $\star$
   $\star$
   $\star$
4. $T_2 \leftarrow T_2 \cdot T_4 \ (= S_1)$
   $\star$
   $\star$
   $\star$
5. $T_4 \leftarrow T_3 \cdot T_3 \ (= Z_1^2)$
   $\star$
   $\star$
   $\star$
6. $T_5 \leftarrow T_4 \cdot T_7 \quad (= U_2)$
   $\star$
   $T_5 \leftarrow -T_5 \quad (= -U_2)$
   $T_5 \leftarrow T_1 + T_5 \ (= W)$
7. $T_4 \leftarrow T_3 \cdot T_4 \ (= Z_1^3)$
   $\star$
   $\star$
   $\star$
8. $T_4 \leftarrow T_4 \cdot T_8 \quad (= S_2)$
   $\star$
   $T_4 \leftarrow -T_4 \quad (-S_2)$
   $T_4 \leftarrow T_2 + T_4 \ (= R)$
9. $T_3 \leftarrow T_3 \cdot T_9 \ (= Z_1 Z_2)$
   $\star$
   $\star$
   $\star$
10. $T_3 \leftarrow T_3 \cdot T_5 \ (= Z_3)$
    $\star$
    $\star$
    $\star$
11. $T_6 \leftarrow T_5 \cdot T_5 \ (= W^2)$
    $\star$
    $\star$
    $\star$
12. $T_1 \leftarrow T_1 \cdot T_6 \ (= U_1 W^2)$
    $\star$
    $T_4 \leftarrow -T_4 \quad (= -R)$
    $\star$
13. $T_5 \leftarrow T_5 \cdot T_6 \quad (= W^3)$
    $T_6 \leftarrow T_1 + T_2 \ (= S_1 + U_1 W^2)$
    $T_2 \leftarrow -T_2 \quad (= -S_1)$
    $T_6 \leftarrow T_2 + T_6 \ (= U_1 W^2)$
14. $T_1 \leftarrow T_4 \cdot T_4 \quad (= R^2)$
    $T_1 \leftarrow T_1 + T_5 \ (= R^2 + W^3)$
    $T_6 \leftarrow -T_6 \quad (= -U_1 W^2)$
    $T_1 \leftarrow T_1 + T_6 \ (= X_3 + U_1 W^2)$
15. $T_2 \leftarrow T_2 \cdot T_5 \quad (= -S_1 W^3)$
    $T_1 \leftarrow T_1 + T_6 \ (= X_3)$
    $\star$
    $T_6 \leftarrow T_1 + T_6 \ (= X_3 - U_1 W^2)$
16. $T_4 \leftarrow T_4 \cdot T_6 \quad (= Y_3 + S_1 W^3)$
    $T_2 \leftarrow T_2 + T_4 \ (= Y_3)$
    $\star$
    $\star$

Fig. 7. Expressing point doubling and point addition as a repetition of blocks $\sim \Gamma$.

whose k^{th} row represents sequence γ_k, which reads as

$$\gamma_k = [R_{u_{k,0}} \leftarrow R_{u_{k,1}} \cdot R_{u_{k,2}}; R_{u_{k,3}} \leftarrow R_{u_{k,4}} + R_{u_{k,5}}; \\ R_{u_{k,6}} \leftarrow -R_{u_{k,6}}; R_{u_{k,7}} \leftarrow R_{u_{k,8}} + R_{u_{k,9}}; i \leftarrow i - u_{k,10}] \ .$$

So, a direct application yields the following implementation of the double-and-add algorithm.

Input: $\boldsymbol{P_1} = (X_1, Y_1, Z_1), d = (1, d_{m-2}, \ldots, d_0)_2$, and matrix $(u_{k,l})$ as above
Output: $\boldsymbol{P_d} = d\boldsymbol{P_1}$

$R_0 \leftarrow a$; $R_1 \leftarrow X_1$; $R_2 \leftarrow Y_1$; $R_3 \leftarrow Z_1$; $R_7 \leftarrow X_1$; $R_8 \leftarrow Y_1$; $R_9 \leftarrow Z_1$
$i \leftarrow m-2$; $s \leftarrow 1$
while $(i \geq 0)$ do
  $k \leftarrow (\neg s) \cdot (k+1) + s \cdot 26(\neg d_i)$
  $(u_0, u_1, \ldots, u_9, s) \leftarrow (u_{k,0}, u_{k,1}, \ldots, u_{k,9}, u_{k,10})$
  $R_{u_0} \leftarrow R_{u_1} \cdot R_{u_2}$; $R_{u_3} \leftarrow R_{u_4} + R_{u_5}$; $R_{u_6} \leftarrow -R_{u_6}$; $R_{u_7} \leftarrow R_{u_8} + R_{u_9}$
  $i \leftarrow i - s$
return (R_1, R_2, R_3)

Fig. 8. A [simple] side-channel atomic double-and-add algorithm for elliptic curves over $\mathbb{F}_p$.

Matrix $(u_{k,l})$ is highly redundant: except for variable s (last column), the first 10 rows are exactly the same as the last 10 ones. This is not too surprising since these rows correspond to the same operation (namely an elliptic curve doubling). It is fairly easy to remove the redundancy. Since, except for s, the 10 rows representing a doubling in matrix $(u_{k,l})$ are equivalent, they can be shared. It suffices then to express s as a function of d_i and k in the optimized matrix $(u^*_{k,l})$ given by

$$(u^*_{k,l})_{\substack{0 \leq k \leq 25 \\ 0 \leq l \leq 9}} = \begin{pmatrix} 4&1&1&5&4&4&3&4&4&5 \\ 5&3&3&1&1&1&3&1&1&3 \\ 5&5&5&1&1&3&3&1&1&3 \\ 5&0&5&4&4&5&3&5&2&2 \\ 3&3&5&1&1&3&3&1&1&3 \\ 2&2&2&2&2&2&4&1&1&3 \\ 5&1&2&1&1&5&5&1&1&5 \\ 1&4&4&1&1&5&4&1&1&5 \\ 2&2&2&2&2&2&3&5&1&5 \\ 4&4&5&2&2&4&2&4&4&5 \\ 4&9&9&5&1&5&5&5&1&5 \\ 1&1&4&5&1&5&5&5&1&5 \\ 4&4&9&5&1&5&5&5&1&5 \\ 2&2&4&5&1&5&5&5&1&5 \\ 4&3&3&5&1&5&5&5&1&5 \\ 5&4&7&2&2&5&5&5&1&5 \\ 4&3&4&2&2&5&6&6&5&6 \\ 4&4&8&6&5&6&4&4&2&4 \\ 3&3&9&6&5&6&6&6&5&6 \\ 3&3&5&6&5&6&6&6&5&6 \\ 6&5&5&6&3&6&3&6&3&6 \\ 1&1&6&1&1&4&4&1&1&4 \\ 5&5&6&6&1&2&2&6&2&6 \\ 1&4&4&1&1&5&6&1&1&6 \\ 2&2&5&1&1&6&3&6&1&6 \\ 4&4&6&2&2&4&6&6&1&6 \end{pmatrix} .$$

Since, when $d_i = 1$ we have $s = 0$ if $0 \leq k \leq 24$ and $s = 1$ if $k = 25$, and when $d_i = 0$ we have $s = 0$ if $0 \leq k \leq 8$ and $s = 1$ if $k = 9$, we may for example define s as

$$s = d_i \cdot (k \text{ div } 25) + (\neg d_i) \cdot (k \text{ div } 9) \ .$$

The expression for k must also be modified accordingly: k is always incremented unless when $s = 1$, in which case it must be set to 0. So, $k \leftarrow (\neg s) \cdot (k + 1)$ is a valid expression for updating k. Doing so, we obtain the algorithm of Fig. 5, which is very similar to the one above but with a smaller matrix representation (*i.e.* matrix $(u^*_{k,l})$).

B Side-Channel Atomic Implementation of the MIST Exponentiation Algorithm

MIST is a randomized exponentiation algorithm for preventing DPA-like attacks. However, as presented in [17], the MIST algorithm is susceptible to SPA-like analysis. We give hereafter a [simple] side-channel atomic version of the MIST algorithm[6] as an additional illustration of the genericity of our methodoly for preventing SPA-like attacks.

Input: $x, d = (d_{m-1}, \ldots, d_0)_2$, and matrices $(F_{\delta,r})$ and $(G_{k,l})$ (see below)
Output: $y = x^d$

$R_1 \leftarrow m$; $R_3 \leftarrow 1$; $i \leftarrow 0$; $s \leftarrow 1$
while $(d > 0)$ **do**
  $\rho \leftarrow_R \{2, 3, 5\}$; $\delta \leftarrow \neg s \cdot \delta + s \cdot \rho$; $r \leftarrow d \bmod \delta$
  $k \leftarrow \neg s \cdot (k + 1) + s \cdot F_{\delta,r}$
  $(u_1, u_2, u_3, s) \leftarrow (G_{k,0}, G_{k,1}, G_{k,2}, G_{k,3})$
  $R_{u_3} \leftarrow R_{u_1} \cdot R_{u_2}$
  $d \leftarrow \neg s \cdot d + s \cdot (d \text{ div } \delta)$
endwhile
return R_3

Fig. 9. Side-channel atomic MIST exponentiation algorithm.[7]

[6] To avoid register rewritting, the divisor subchain corresponding to the divisor/residue pair $[2, 1]$ is replaced with $\{(133), (111)\}$. The original MIST algorithm uses subchain $\{(112), (133)\}$; cf. [17, Table 3.1].

[7] Again, we assume that all involved operations are side-channel equivalent (and if not, are made so by an appropriate software emulation).

with matrices:

$$(F_{\delta,r})_{\substack{2\le\delta\le 5\\ 0\le r\le 4}} = \begin{pmatrix} 0 & 1 & \star & \star & \star \\ 3 & 5 & 8 & \star & \star \\ \star & \star & \star & \star & \star \\ 11 & 14 & 18 & 22 & 26 \end{pmatrix} \quad \text{and} \quad (G_{k,l})_{\substack{0\le k\le 29\\ 0\le l\le 4}} = \begin{pmatrix} 1\,1\,1\,1 \\ \hline 1\,3\,3\,0 \\ 1\,1\,1\,1 \\ \hline 1\,1\,2\,0 \\ 1\,2\,1\,1 \\ \hline 1\,1\,2\,0 \\ 1\,3\,3\,0 \\ 1\,2\,1\,1 \\ \hline 1\,1\,2\,0 \\ 2\,3\,3\,0 \\ 1\,2\,1\,1 \\ \hline 1\,1\,2\,0 \\ 1\,2\,1\,0 \\ 1\,2\,1\,1 \\ \hline 1\,1\,2\,0 \\ 1\,3\,3\,0 \\ 1\,2\,1\,0 \\ 1\,2\,1\,1 \\ \hline 1\,1\,2\,0 \\ 2\,3\,3\,0 \\ 1\,2\,1\,0 \\ 1\,2\,1\,1 \\ \hline 1\,1\,2\,0 \\ 1\,2\,1\,0 \\ 1\,3\,3\,0 \\ 1\,2\,1\,1 \\ \hline 1\,1\,2\,0 \\ 2\,2\,2\,0 \\ 2\,3\,3\,0 \\ 1\,2\,1\,1 \end{pmatrix}.$$