

Cryptanalysis of PKP: A New Approach

Éliane Jaulmes and Antoine Joux

DCSSI
18, rue du Dr. Zamenhoff
F-92131 Issy-les-Mx Cedex
France
eliane.jaulmes@wanadoo.fr
Antoine.Joux@ens.fr

Abstract. Quite recently, in [4], a new time-memory tradeoff algorithm was presented. The original goal of this algorithm was to count the number of points on an elliptic curve, however, the authors claimed that their approach could be applied to other problems. In this paper, we describe such an application and show a new way to attack the Permuted Kernel Problem. This new method is faster than any previously known technique but still requires exponential time. In practice, we find that attacking PKP for the original size proposed by Shamir in [6] could be done on a single PC in 125 years.

1 Introduction

The Permuted Kernel Problem was introduced in cryptography by Shamir at Crypto 1989 [6]. This NP -complete problem can be stated as follows:

- Given a $m \times n$ matrix, a n vector V and a prime p
- Find a permutation π such that the permuted vector V_π is in the kernel of the matrix modulo p .

Any instance of the problem with this choice of parameters will be denoted as a $PKP_p(m, n)$ problem. Without loss of generality, the left part of $m \times n$ matrix can be turned into the identity sub-matrix, as explained in [6].

In [6], it was shown that this problem possesses a nice zero-knowledge proof and can thus be turned into an authentication scheme. Moreover, when used in practice the scheme offers a good level of security using only simple computations which can be efficiently implemented, even in small portable devices. Since PKP is so simple, and uses only basic linear algebra, it is extremely tempting to search for its weaknesses. This led to many papers [3,1,2,5], which all concluded that the original dimension proposed par Shamir are a bit too small, but the scheme still resists all known attacks. All the proposed attacks combine exhaustive search with some form of time-memory tradeoff. However, none of the classical time-memory tradeoff techniques seems to apply to this problem, and thus specific methods had to be developed in the previous papers. In this paper, we apply a

new time-memory tradeoff from [4] to the permuted kernel problem. This new technique was originally designed to replace the final baby-step/giant-step when counting points on elliptic curves using the Schoof-Elkies-Atkies algorithm.

2 General Description of the Algorithm

In this section, we reformulate the algorithm from [4] in a general setting, without any reference to the specific problem of point counting on elliptic curve. In our general setting, we want to solve the following problem:

- Given a n vector P whose entries are primes, four sets S_1, S_2, S_3 and S_4 of n vectors, and n sets D_1, \dots, D_n
- Find $v^{(1)} \in S_1, v^{(2)} \in S_2, v^{(3)} \in S_3, v^{(4)} \in S_4, d_1 \in D_1, \dots, d_{n-1} \in D_{n-1}$ and $d_n \in D_n$ such that:

$$\forall i \in [1 \dots n] : v_i^{(1)} + v_i^{(2)} + v_i^{(3)} + v_i^{(4)} \equiv d_i \pmod{P_i}$$

Clearly, this problem, which we note $\mathcal{4SET}$, can be solved by exhaustively trying the $N_1 N_2 N_3 N_4$ possible values of $v^{(1)}, v^{(2)}, v^{(3)}$ and $v^{(4)}$, where N_i denotes the cardinality of S_i . We propose here a time-memory tradeoff that allows to solve this problem faster than exhaustive search. Without loss of generality, we assume that:

$$\frac{|D_1|}{P_1} \leq \frac{|D_2|}{P_2} \leq \dots \leq \frac{|D_n|}{P_n},$$

where $|D_i|$ denotes the size of D_i . Then, let $\alpha_i = \frac{|D_i|}{P_i}$, choose k a positive integer smaller than n and let

$$\Psi = \prod_{i=1}^k \alpha_i \text{ and } \Phi = \prod_{i=1}^k P_i$$

The algorithm then consists of a precomputation phase and of a main loop containing two enumeration phases, one involving $v^{(1)}$ and $v^{(2)}$, the A -phase, and one involving $v^{(3)}$ and $v^{(4)}$, the B -phase.

Algorithm for solving $\mathcal{4SET}$

- **Precomputation step:** Sort the two sets S_2 and S_4 , according to the lexicographical order on the vector coordinates.

In the sequel, this will permit to quickly find vectors in one of these sets given its first k coordinates.

- **Main loop:**

For $M_1 \in [0 \dots P_1 - 1], M_2 \in [0 \dots P_2 - 1], \dots, M_k \in [0 \dots P_k - 1]$ do:

- **A phase:**

- * For each $\Theta \in D_1 \times \dots \times D_k$,

- * For $v^{(1)} \in S_1$ and $v^{(2)} \in S_2$ such that the first k coordinates¹ $v_i^{(1)} + v_i^{(2)}$ match $\Theta_i - M_i$ modulo P_i ,

¹ Thanks to the precomputation step, such $v^{(2)}$ can be accessed quickly by computing $\Theta_i - M_i - v_i^{(1)} \pmod{P_i}$ before searching the matching entries in the sorted set S_2 .

* For all $\ell > k$ compute and store the following set:

$$H_{\Theta, v^{(1)}, v^{(2)}, \ell} = \{\theta - v_\ell^{(1)} - v_\ell^{(2)} \mid \theta \in D_\ell\}.$$

• **B phase:**

- * For each $v^{(3)} \in S_3$ and $v^{(4)} \in S_4$ such that the first k coordinates $v_i^{(3)} + v_i^{(4)}$ match M_i ,
- * If there exists $\Theta \in D_1 \times \dots \times D_k$, $v^{(1)} \in S_1$ and $v^{(2)} \in S_2$ such that for every $\ell > k$, $v_\ell^{(3)} + v_\ell^{(4)} \pmod{P_\ell}$ is in $H_{\Theta, v^{(1)}, v^{(2)}, \ell}$,
- * Then $v_i^{(1)} + v_i^{(2)} + v_i^{(3)} + v_i^{(4)}$ is a solution of the *4SET* problem.

Terminate

2.1 Practical Considerations

In practice, building the sets $H_{\Theta, v^{(1)}, v^{(2)}, \ell}$ in the *A*-phase and checking their intersections in the *B*-phase can be done very efficiently. Indeed, all these sets can be stored in a single array of bits. This array has $\sum_{\ell=k+1}^n P_\ell$ lines and one column for each pair $(v^{(1)}, v^{(2)})$. Each line of this array can also be seen as a bit string $B_{\ell, \tau}$ where $\tau \in \{0, \dots, P_\ell - 1\}$. During the *A*-phase, we store a 1 in $B_{\ell, \tau}$ in the position corresponding to $(v^{(1)}, v^{(2)})$ if $\tau \in H_{\Theta, v^{(1)}, v^{(2)}, \ell}$ and a 0 otherwise. Note that all strings $B_{\ell, \tau}$ have the same length, however this length may vary from one round of the main loop to the next. On average, this length is $\Psi N_1 N_2$.

During the *B*-phase, to check whether $\tau_\ell = v_\ell^{(3)} + v_\ell^{(4)} \pmod{P_\ell}$ is in $H_{\Theta, v^{(1)}, v^{(2)}, \ell}$ for every ℓ and some pair $(v^{(1)}, v^{(2)})$, we simply perform a logical AND between the strings B_{ℓ, τ_ℓ} . If the resulting string is non-nil we have a solution, since any bit equal to 1 in this string corresponds to a pair $(v^{(1)}, v^{(2)})$ such that $v^{(1)} + v^{(2)} + v^{(3)} + v^{(4)}$ is a solution of the *4SET* problem.

Note that, when the expected number of solutions of a *4SET* problem is much smaller than 1, it is worthwhile not to test the last conditions. Indeed, in that case, one can simply remove the useless components and build a similar problem with fewer conditions. In fact, this approach was implicitly used in [4] since some of the conditions found by the SEA algorithm were discarded for the final step. On the contrary, PKP problems are usually built in such a way that all conditions are useful and cannot be discarded (see section 4).

2.2 Analysis of the Algorithm

- **Precomputation step :** The number of operations required to sort S_2 is $O(N_2 \log(N_2))$ and to sort S_4 it is $O(N_4 \log(N_4))$. Thus the time needed is $O(\max(N_2, N_4) \log(\max(N_2, N_4)))$. The total memory required in this precomputation step is $O(\max(N_1, N_2, N_3, N_4) \sum_i \log(P_i))$ because S_1 and S_3 must also be stored, and because each vector can be represented with $\sum_i \log(P_i)$ bits.

- **Phase A :** Clearly, the average number of pairs $(v^{(1)}, v^{(2)})$ constructed in each execution of phase *A* is $\Psi N_1 N_2$. To do this construction, we enumerate all possible values of Θ and $v^{(1)}$ and search for matching values of $v^{(2)}$. This requires $O(N_1 \log(N_2) \Psi \Phi)$ operations. Then for each valid pair $(v^{(1)}, v^{(2)})$, $n - k$ bits are to be set, the total number of operations for this step is $O((n - k) \Psi N_1 N_2)$. All in all, the number of operations required is:

$$O(\max(N_1 \log(N_2) \Psi \Phi, (n - k) \Psi N_1 N_2)).$$

The total memory needed to store all the sets is $O(\Psi N_1 N_2 \sum_{i=k}^n P_i)$.

- **Phase B :** In each execution of phase *B*, $N_3 N_4 / \Phi$ pairs $(v^{(3)}, v^{(4)})$ are constructed. This construction requires $O(N_3 \log(N_4))$ operations. Then for each pair the logical AND of $n - k$ of the strings constructed in phase *A* is computed. Since on average the length of the strings is $\Psi N_1 N_2$ the number of operations is $O((n - k) \Psi N_1 N_2 N_3 N_4 / \Phi)$. All in all, each iteration of phase *B* costs :

$$O(\max(N_3 \log(N_4), (n - k) \Psi N_1 N_2 N_3 N_4 / \Phi)).$$

In term of memory complexity, phase *B* does not require any memory not already used in the precomputation or in phase *A*.

When the choice of parameters is reasonable, the time complexity is dominated by phase *B* and can be expressed as :

$$O((n - k) \Psi N_1 N_2 N_3 N_4).$$

Without going too far into the analysis of the parameters, let say that a choice is reasonable if all the N_i are of the same order N , if $\Psi \approx 1/N$ and if Φ does not become too large. Moreover, in that case the memory needed is:

$$O(N \sum_{i=k}^n P_i).$$

Note: With the algorithm as presented here Φ should not become larger than $N^{3/2}$. However, if we slightly modify it by transferring half of Θ from phase *A* to phase *B*, Φ can grow up to N^2 . Moreover, this transformation reduces the amount of memory needed, by shortening the sets stored during phase *A*. Since we still need to store the sets, the memory requirement becomes:

$$O(N \sum_{i=1}^n \log(P_i)).$$

3 Application to PKP

In order to apply the algorithm of the previous section to PKP, we need to build sets S_1, S_2, S_3, S_4 and D_1, D_2, \dots, D_n from a PKP instance. Before doing

that, we will slightly transform the PKP instance. Following [3], we can add one more linear equation to the PKP instance. This new equation stems from the simple fact that the sum σ of the coordinates of the solution vector does not depend on the permutation π . Applying Gaussian elimination to the extended linear system, we find that the solution vector V_π must verify:

$$\left(A_0 \ I_{m+1} \right) V_\pi = \begin{pmatrix} \sigma \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

where A_0 is a $(m+1) \times (n-m-1)$ matrix and I_{m+1} is the $(m+1) \times (m+1)$ identity matrix. Clearly, in order to find a solution to the permuted kernel problem thus written, it suffices to try all the possible values of the components of V_π that enters A_0 , to find the remaining components by Gaussian elimination and to check that the vector found is indeed a permutation of V . This algorithm requires $n!/(n-m-1)!$ trials. In the sequel, we will refer to it as being the exhaustive search technique for PKP, and we will completely forget the simple minded search where one tries all possible values for π , which requires $n!$ trials.

We can now divide A_0 into four roughly equal parts, and we find:

$$\left(A_1 \ A_2 \ A_3 \ A_4 \ I_{m+1} \right) V_\pi = \begin{pmatrix} \sigma \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

where A_i is a $(m+1) \times n_i$ matrix and $n_1 + n_2 + n_3 + n_4 = n - m - 1$.

We then build the sets S_i by computing the product of A_i by all possible choices of the corresponding n_i bits. Clearly, the size of S_i is $N_i = n!/(n-n_i)!$. Once these sets are constructed, we can apply the algorithm from section 2.

Note: In fact, the algorithm from section 2 can be further refined in the case of PKP. The idea is that while merging together an element of S_1 and an element of S_2 in phase A or an element of S_3 and an element of S_4 in phase B , one should check their compatibility, i.e. verify that put together they form a correct subset of V , which is true if and only if they have no nontrivial intersection (assuming that V contains no double). This reduces the term $N_1 N_2$ and $N_3 N_4$ in the complexities respectively to $n!/(n-n_1-n_2)!$ and $n!/(n-n_3-n_4)!$.

4 Asymptotical Analysis

In order to make an asymptotical analysis of our algorithm, we first need to describe an asymptotical version of PKP. For defining this version, we will follow the two following criteria:

- Building a strong instance of PKP, should be easy. More precisely, this means that for any random matrix, finding a kernel vector with distinct coordinates should be easy. Since kernel vectors are chosen at random when building a PKP problem, this implies that the probability of a random kernel vector to have all distinct coordinates should not be too low. Taking in account the birthday paradox, this means that n should be no larger than $O(\sqrt{p})$.
- As explained in [6], the expected number of solutions of a PKP instance should be as near to 1 as possible. This leads to the condition $p^m \approx n!$.

Following these criteria let $p = O(n^2)$, then:

$$\begin{aligned} m &\approx n \log n / \log p \\ &\approx n/2 \end{aligned}$$

Using these two criteria, we propose $PKP_p(n, \lfloor n/2 \rfloor)$ as a reasonable asymptotic choice, where p is a prime near n^2 . With this choice of parameters, an exhaustive search attack on PKP takes roughly $n!/m! = O((2n/e)^{n/2})$ trials. For the attack described in section 3, we need to choose the parameter k and thus the value of Ψ . A particularly interesting choice is to use the same amount of storage for the sets S_i and the strings. Assuming that $N_1 = N_2 = N_3 = N_4$, this leads to $\Psi \approx 1/N_1$. Since $N_1 = O(((8/7)^7)(n/e)^{n/8})$, we find that the time complexity of the algorithm is

$$O(((8/7)^7)(n/e)^{3n/8+\varepsilon})$$

and the space complexity is

$$O(((8/7)^7)(n/e)^{n/8+\varepsilon}).$$

The value ε in the exponent offers a simple replacement for the non exponential terms that should appear in these two formulas, and permits a simpler expression.

In fact, if we further take into account the note at the end of section 3, we can somewhat reduce the constant $(8/7)^7$ appearing in the time complexity.

5 Practical Results

In practice, it turns out that the previous ideas lead to a faster attack against PKP, than all previously known techniques. The best previous theoretical attacks against PKP are those from [2] and an implementation of these attacks is described in [5]. In the rest of this section we compare the available data for this attack with our results.

In [2], the following results are found:

Results from [2]	Time needed	Memory needed (in tuples)
$PKP_{251}(16, 32)$	2^{54}	2^{17} 6-tuples
	2^{52}	2^{24} 10-tuples
$PKP_{251}(37, 64)$	2^{123}	2^{27}
	2^{119}	2^{52}
	2^{116}	2^{65}

In order to make the same kind of evaluation for our algorithm, we first need to compute the size of the sets S_1, S_2, S_3, S_4 . Starting with $PKP(16, 32)$, we take $n_1 = n_2 = n_4 = 4$ and $n_3 = 3$, we find $N_1 = N_2 = N_4 = 863040$ and $N_3 = 29760$. In order to have $\Psi \approx 1/N_1$, we take $k = 6$ and find $\Psi = (24/251)^6$. With these choice, the space needed is dominated by the storage of the four sets S_i , and $32 \times (N_1 + N_2 + N_3 + N_4) \approx 2^{26}$ bytes are needed. This may seem larger than the 2^{24} in the above table, however this size was not in bytes but in 10-tuples, and thus both sizes are equivalent. The basic time estimate is $\Psi N_1 N_2 N_3 N_4 \approx 2^{54}$. However, recalling the note from section 3 it becomes $\Psi n!^2 / ((n - n_1 - n_2)!(n - n_3 - n_4)!) \approx 2^{52}$. Once again, this does not seem better than the value 2^{52} in the above table. However, remember than our basic operation is a bit operation and that on most computers we can pack 32 or even 64 bit operations in a single word operation, thus lowering the complexity to 2^{46} .

As the size increases, the advantage of the new algorithm becomes much clearer. Indeed, for $PKP(37, 64)$, we can take $n_1 = n_3 = 6$, $n_2 = n_4 = 7$ and $k = 17$. Then $\Psi = (48/251)^k$, the space needed becomes $2^{48.5}$ and the time needed 2^{106} . However, while better than the estimates from [2], these values are completely unreachable. The following table shows the results of the new attack for various dimensions of PKP.

New Results	k	Time needed	Memory needed
$PKP_{251}(16, 32)$	6	2^{46}	2^{26} bytes
$PKP_{251}(15, 32)$	6	2^{51}	2^{27} bytes
$PKP_{251}(24, 48)$	12	2^{85}	2^{35} bytes
$PKP_{251}(37, 64)$	17	2^{106}	$2^{48.5}$ bytes

In [5], the attack described in [2] was truly implemented, and experiments were made. At that time, a single workstation would have taken 2000 years for $PKP(16, 32)$. In a private communication, the author from [5] told us than on current machines, experiment showed that this estimate was lowered to 700 years. The ratio between the two figures is much worse than expected because all these computations heavily rely on memory usage. Since the speed of memory access did not increase as quickly as the speed of processors, this accounts for the low ratio. By comparison, on the same machine (Pentium II, 400MHz), the new attack would take 125 years (at most) to find the secret key of $PKP(16, 32)$. Quite strangely, in a practical implementation, phase A takes proportionally much longer than phase B because in the former case we make random memory access (on single bits) while in the latter we read the memory in sequential order. Consequently, phase B is cache friendly while phase A isn't. Moreover, we cannot use the theoretically optimal choice for k , because the code that controls the loop then becomes predominant. Thus our practical choices were $n_1 = 3$, $n_2 = 4$, $n_3 = 3$, $n_4 = 5$ and $k = 4$. With these choices, each iteration of the main loop took just under a second, and the total memory needed was 250 megabytes. Since 251^4 iterations of the main loop are needed, a total running time of 125 years is expected.

6 Conclusion

In this paper, we showed that the time-memory tradeoff technique for [4] could be applied to the PKP problem. Very curiously, this leads to an algorithm which presents similarities with the algorithm from [2]. In practice, this new algorithm can attack PKP(16,32) about five times faster than all previous attacks. However, this attack would still require 125 years on a 450MHz PC. Since the algorithm is straightforward to parallelized, this computation is feasible and PKP(16,32) can no longer be considered as secure. Moreover, PKP(15,32) which takes about 24 times as long, is potentially endangered and should no longer be used for long-term applications. However, slightly larger problems such PKP(24,48) or PKP(37,64) are completely out of reach.

References

1. T. Baritaud, M. Campane, P. Chauvaud, and H. Gilbert. On the security on the permuted kernel identification scheme. In *CRYPTO92*, volume 740 of *LNCS*, pages 305–311, 1992.
2. P. Chauvaud and J. Patarin. Improved algorithms for the permuted kernem problem. In *CRYPTO93*, volume 773, pages 391–402, 1994.
3. J. Georgiades. Some remarks on the security of the identification scheme based on permuted kernels. *Journal of Cryptology*, 5:133–137, 1992.
4. A. Joux and R. Lercier. “Chinese & Match”, an alternative to atkin’s “match and sort” method used in the SEA algorithm. *Mathematics of Computation*, 1999. To appear.
5. G. Poupard. A realistic security analysis of identification schemes based on combinatorial problems. *European transactions on telecommunications*, 8:471–480, 1997.
6. A. Shamir. An efficient identification scheme based on permuted kernels. In *CRYPTO89*, volume 435 of *LNCS*, pages 606–609, 1989.