Hardware Architecture and Cost/time/data Trade-off for Generic Inversion of One-Way Function

Arquitectura en Hardware y Compromiso de Costo, Tiempo y Datos para Inversiones Genéricas de Funciones Unidireccionales

Sourav Mukhopadhyay[1] **and Palash Sarkar**[2]

[1] Electronic Engineering Department
Dublin City University
Glasnevin, Dublin 9
Ireland
`msourav@eeng.dcu.ie`
[2] Applied Statistics Unit
Indian Statistical Institute
203 B.T. Road, Kolkata
India-700108
`palash@isical.ac.in`

Article received on March 1, 2008, accepted on October 3, 2008

Abstract
In many cases, a cryptographic algorithm can be viewed as a one-way function, which is easy to compute in forward direction but hard to invert. Inverting such one-way function amounts to breaking the algorithm. Time-Memory Trade-Off (TMTO) is a twenty five years old generic technique for inverting one-way functions. The most feasible implementation of TMTO is in special purpose hardware. In this paper, we describe a systematic architecture for implementing TMTO. We break down the offline and online phases into simpler tasks and identify opportunities for pipelining and parallelism. This results in a detailed top-level architecture. Many of our design choices are based on intuition. We develop a cost model for our architecture. Analysis of the cost model shows that 128-bit keys seem safe for the present. However, key sizes less than 96 bits do not provide comfortable security assurances.
Keywords: One-way function, generic method, time/meomry trade-off cryptanalysis.

Resumen
En muchos casos, un algoritmo criptográfico puede ser visto como una función de solo ida, la cual es fácil de calcular pero difícil de invertir. Invertir una función de sólo ida es equivalente a romper el algoritmo criptográfico. Compromisos de tiempo-memoria (TMTO por sus siglas en inglés) es una vieja técnica genérica concebida más de veinticinco años atrás para invertir funciones de sólo ida. La implementación más factible de TMTO es la de arquitecturas de hardware de propósito especial, y es así que en este artículo, describimos una arquitectura de ese tipo capaz de implementar dicho método. Subdividimos las fases fuera de línea y en línea del algoritmo en tareas simples e identificamos oportunidades para paralelizar y/o utilizar técnicas de tubería. Este proceso nos condujo a proponer una arquitectura de alto nivel muy detallada, en la cual muchas de las elecciones de diseño estuvieron basadas en la intuición. Asimismo, desarrollamos un modelo de costos para nuestra arquitectura. El análisis del modelo de costo sugiere que las llaves de 128 bits pueden ser consideradas seguras en la actualidad. Sin embargo, las llaves con longitudes menores de 96 bits no brindan garantías de seguridad suficientes.
Palabras Claves: Funciones de sólo ida, método genérico, cripto-análisis de compromiso tiempo memoria.

1 Introduction

Cryptographic algorithms such as block and stream ciphers require the use of a secret key to ensure confidentiality of transmitted messages. The basic goal of a cryptanalytic attack is to recover the secret key from publicly available information. Very often a successful attack exploits weaknesses in the design of the specific algorithm being considered. Two of the most popular attacks are: linear cryptanalysis (Matsui 1993; Matsui 1994; Borst, Preneel, and Vandewalle 1999; Shimoyama, Takenaka, and Koshiba 2002) and differential cryptanalysis (Biham and Shamir 1993; Lai 1994). There are several variants of differential attacks namely, truncated and higher order differential attack, impossible differential attack (Biham, Biryukov, and Shamir 1999a), boomerang attack (Wagner 1999). Related key attack (Biham 1994), miss in the middle attacks (Biham, Biryukov, and Shamir 1999b), slide attack (Biryukov and Wagner 1999), correlation attack (Shimoyama, Takeuchi, and Hayakawa 2002), statistical attacks (Gilbert, Handschuh, Joux, and Vaudenay 2000; Handschuh and Gilbert 1997) are examples of some other attacks on symmetric ciphers.

A generic approach for cryptanalysis views the encryption function as a black box, i.e., it does not utilize information about how the function is constructed. A simplest generic attack is to try every possible key until the correct one is found. This is called an exhaustive search attack. The importance of such an approach arises from the fact that if a cryptographic algorithm is not secure against exhaustive search, then it cannot be considered secure at all. Implementation of exhaustive search is most feasible in special purpose hardware. In 1998, a remarkable achievement was made (EFF 1998) when the Electronics Frontier Foundation built a machine *DES Craker* for cracking DES at a cost of US $200,000 and which cracked a DES problem in 3 and 1/2 days. Recently, Kumar et at. (Kumar, Paar, Pelzl, Pfeiffer, and Schimmler 2006) build the COPACOBANA (COPACOPANA 2006) machine to break DES. The cost of one machine is approximately US $10,000 and which cracked DES in less than a week.

The main disadvantage of using exhaustive search is that it has to be repeated separately for each target. To address this problem, Hellman (Hellman 1980) introduced *time/memory trade-off* (TMTO) attack that enables one to perform an exhaustive search *once* in an offline precomputation phase. The actual attack, i.e., *finding the key corresponding to a target* is done in an online phase with table lookup and is significantly faster than exhaustive search. Also, one can repeat the attack on different targets without going through the pre-computation each time. A TMTO attack is a generic attack which can be carried out against any one-way function f. The online target consists of an image y and the goal of the attack is to find a k, such that $f(k) = y$, k being the secret key (pre-image) from a key space of size N corresponding to the target y.

Since the publication of Hellman's result, there has been a lot of research on TMTO. Hellman's method can recover a key in time T using M memory with the trade-off curve $TM^2 = N^2$ for $1 \leq T \leq N$, N being the number of all possible keys. Rivest (Denning 1982) introduced the distinguished point (DP) property in TMTO attack to reduce the number of table lookups. Later, Biryukov and Shamir (BS) (Biryukov and Shamir 2000) showed how to modify Hellman's technique to take advantage of available multiple data. If D many y's are available, and the goal is to find a pre-image of any one of them, then BS obtain a trade-off curve $TM^2D^2 = N^2$. Later, Oechslin (Oechslin 2003) proposed the rainbow method to reduce runtime cost to one-half of Hellman's method with the same trade-off curve as Hellman's method. The problem has been investigated in a more theoretical setting by Fiat and Naor (Fiat and Naor 1991).

In 1988, Amirazizi and Hellman (Amirazizi and Hellman 1988) proposed *time/memory/processor trade-off* where several processors execute in parallel, sharing a large memory through a *switching/sorting* network. They assumed that the cost of the wires is less than $n \log n$ and left this as an open problem for further study. Wiener (Wiener 2004) investigated the problem and proved that if an algorithm has a very high memory access rate then the wiring cost is the dominating cost for any *switching/sorting* network and showed this cost to be $\Theta(n^{\frac{3}{2}})$ to connect n processors with n memory blocks. Quisquater and Standaert (Quisquater and Standaert 2005) provided a sketch of a generic architecture based on their two previous works (Quisquater and Delescaille 1989; Quisquater, Standaert, Rouvroy, David, and Legat 2002). They suggest a pipelined architecture for implementing a multi-round function f which is based on Wiener's design (Wiener 1996) of implementing DES in his exhaustive search attack on DES. Mentens et al. (Mentens, Batina, Preneel, and Verbauwhede 2005) propose a hardware architecture for key search based on rainbow method.

Following Wiener's (Wiener 2004) work, it is currently believed that the dominant cost of the hardware will be the interconnection cost of connecting a set of processors to a set of memory locations. However, this assumes a particular architecture, i.e., all the processors will actually be connected to all the memory locations. This is not the only possible architecture.

In this paper, we provide a pipelined architecture of the Hellman's algorithm with distinguished point (Hellman+DP) method. We systematically break down the offline and online phases into smaller computation tasks. For each task, we identify suitable opportunities for parallelism and pipelining. A fairly detailed register level architecture is provided for each individual component. Since TMTO is a generic algorithm, the top-level architecture is also quite generic. The hardware implementation of the particular one-way function to be inverted occurs at a lower level. We believe that the architecture presented in this paper can form a good starting point for a concrete implementation of TMTO method to invert a specific one-way function. There are several issues, such as power consumption, mean time between failures, which are not considered in this paper. These are important issues but can be judged effectively after efforts are made for actual implementation. We hope that our design will stimulate further work on this topic.

Based on our architecture, we develop a cost/time/data trade-off model. This is important, since it allows us to quantify statements like "with x many dollars, one can break the algorithm in y many days". Previously such statements have been discussed informally at several forums (such as the ecrypt forum on stream cipher primitives). To the best of our knowledge, no concrete trade-off cost model has appeared in the literature. Using the new cost model we analyze the effectiveness of exhaustive search and TMTO pre-computation for s-bit keys with $s \leq 128$. This analysis shows that $s \leq 96$ does not afford comfortable security while $s = 128$ appears to be secure in the foreseeable future. We apply our trade-off model to stream ciphers and find that the 80-bit stream ciphers does not provide adequate protection against TMTO attacks.

2 Preliminaries

Let $V_s = \{0,1\}^s$ be the set of all possible bit strings of length s. We take V_{s_1} and V_{s_2} to be the plaintext space and ciphertext space respectively. Let $K = V_s$ be the key space (set of all possible keys).
An s_1-bit block cipher is a function $E : V_{s_1} \times K \rightarrow V_{s_2}$ where $\mathsf{cpr} = E_k(\mathsf{msg})$ denotes the ciphertext cpr for msg under k. Let $R : V_{s_2} \rightarrow V_s$ be a function from ciphertexts to keys. If $s_2 > s$ (DES has $s_1 = s_2 = 64$ and $s = 56$), then we remove the first $(s_2 - s)$ bits. If $s_2 \leq s$ (AES has $s_1 = s_2 = 128$ and there are three allowable key lengths, $s = 128, 192$ and 256 bits), then we append $(s - s_2)$ constant bits.
For a fixed message msg, we define a function $f : V_s \rightarrow V_s$ as,

$$f(k) = R(E_k(\mathsf{msg})).$$

To get $y = f(k)$ from k we need to apply the encryption function under the known key k followed by a reduction function R, which is easy to compute. But to get k from $f(k)$ one has to decrypt the known plaintext msg under the unknown key k, which is equivalent to the chosen plaintext attack to the cipher. That is hard. Hence this function f can be viewed as a one-way function.
Other cryptographic primitives like stream cipher, hash function, modes of operation can also be viewed as a one-way function. See (Biryukov 2005; Hong and Sarkar 2005) for more details.
Problem Definition: Let $f : \{0,1\}^s \rightarrow \{0,1\}^s$ be the one-way function to be inverted. This function maybe obtained from a block cipher by considering the map from the keyspace to the cipherspace for a fixed message or from other crypto primitives like, stream cipher, hash function, modes of operation etc. Thus, our problem will be that given a string y, we will have to find a string x (*pre-image or key*) such that $f(x) = y$.

3 TMTO Methodology

In 1980, Hellman (Hellman 1980) presented a cryptanalytic time/memory trade-off attack which can viewed as a generic one-way function ($f : \{0,1\}^s \rightarrow \{0,1\}^s$) inverter. Hellman's attack consists of two steps: precomputing the tables and searching (table lookups) in the tables. In a precomputed table, we generate a chain of length t from a start point k_0 as,

$$k_0 \xrightarrow{f} k_1 \xrightarrow{f} k_2 \rightarrow \ldots \rightarrow k_{t-2} \xrightarrow{f} k_{t-1}.$$

For an $m \times t$ table, m chains of length t are generated. We store start and end points in the table, sorted in the increasing order of end points. Using matrix stopping rule, we choose m and t such that $mt^2 = N$, where $N = 2^s$. So one table can cover only a fraction $\frac{mt}{N} = \frac{1}{t}$ of N. Hence, we need t different (unrelated) tables to cover all N keys. For the i^{th} table, we choose a function $f_i(k) = \phi_i(f(k))$, which is a simple output modification of $f(k)$. The functions f_i, $i = 1, 2, \ldots, t$ are unrelated. ϕ_i's are also called masking functions. In the i^{th} table, we randomly select m distinct keys from the key space, generate m chains taking each key as a start point with the same function f_i.

Given a target $y = f(k)$, we need to find its pre-image k. Suppose k is in one of the constructed tables. For all $i = 1, 2, \ldots, t$, we repeatedly apply f_i to $y' = \phi_i(y)$ at most t times, each time we check whether it reaches an end point of i^{th} table. The number of table lookups for this is at most t. If it reaches an end point, we have the position of k. Then we come to the corresponding start point and repeatedly apply the function until it reaches y. The previous value it visited is k. Hence, the total number of f invocations $= t^2 + t \approx t^2$. The total number of table lookups required is t^2. The Hellman method can recover a key in time T (total number of f invocations) using M memory such that $TM^2 = N^2$.

Rivest introduced the distinguished point (DP) property in time/memory trade-off attack. We can define a DP property on the key space K as follows: a key k satisfies the DP property if its first p bits are zero. In the Hellman + DP method, we generate r tables with maximum chain length t in the precomputation phase as follows. We choose r different functions $f_1, \ldots, f_r$, where each f_i is a simple output modification of the function f, i.e. $f_i(x) = \psi_i(f(x))$, where ψ_i is the i^{th} output modification function. For each table, we choose m start points uniformly at random from the key space. In the i^{th} table, for each start point we generate a chain by repeatedly applying f until we reach a DP or until length of the chain is t. If a DP is encountered in the chain, then we store the tuple (start point, DP point, length of the chain) in the table, otherwise the chain will be discarded. We sort the tuple in the increasing order of the end points (DP). If the same DP occurs in two different tuples, then the tuple with maximum chain length will be stored. Sort the tuples in the increasing order of the end points. Given a cipher text, in the search phase we generate a chain, until we reach a DP. After reaching a DP, we perform a table lookup, and so the number of table lookups reduces from t^2 (for t Hellman tables) to t. As mentioned earlier, Biryukov-Shamir showed how to exploit the availability of multiple data to obtain a new trade-off curve $TM^2D^2 = N^2$.

4 Notational Convention and Abbreviation

We provide below the notations used in the architecture.

– SCC: a two-bit register used in the table preparation stage

– sgc1 and sgc2 are the completion signals of the chain computation and sorting unit respectively.

– start signal indicates that the assembly line movement is complete in the Table Preparation stage.

– P_i: $i = 1, 2, \ldots, n$ are the processors used to generates start point, end point pairs for the table.

– PMS_i:$i = 1, 2, \ldots, n$ are the processor memory space for P_i.

– R is n-bit register used to store the completion signal of all the processors.

– SC is sequential circuit with n-bit input to check whether all the input bits are 1.

– L is s-bit LFSR corresponding to a primitive polynomial whose internal stage are used for output modifications.

– CT: a one bit tag to control write blocks and movement of the assembly line.

– T: one bit tag to control the execution of the processor unit, if $T = 0$ then the unit will be idle until $T = 1$.

– SPG: the start point generator.

– C_1 and C_3 are both $r_1(= \log t)$ bits counter. C_2 is $r_2(= \log \frac{t}{n})$ bits counter.

– SC_1 and SC_2 are sequential circuits with r_1-bit input to check whether all the input bits are 1.

– R_i: $i = 1, 2, 3, 4, 5$ are s-bit registers.

– RF_i is the i^{th} round function.

– SPR_i: $i = 1, 2, \dots q$ are s-bit registers used to store the start points.

– CQR is r_1-bit counter to count the number of start point generated by SPG.

– DB: data block

– R_{2j}: $j = 1, 2, \dots q$ are s-bit intermediate registers to store the output values for different rounds.

– $SPC_{i,}$: $j = 1, 2, \dots q - 1$ are r_1-bit counters.

– WB: write block; RB: read block and DB: data block.

– CQ: k-bit register.

– SCQ is sequential circuit with k-bit input to check whether all the input bits are 1.

– y : data point

– DP : distinguished point

– SP : start point

– OMB : output memory block

– MR and DR are both s-bit register.

– PC_1 is r_3-bits ($r_3 = \log z$) counter.

– $BUF1$ and $BUF2$ are buffer queues.

5 Precomputing Stage

The precomputing stage consists of two phases: *chain computation* and *sorting*. Figure 1 describes architecture of the precomputing stage and the tables are computed one by one. To generate a table, a high speed memory is used as an input of the chain computation phase. In the chain computation phase, chains are generated until it reaches to a DP and then storing the start point and end point pairs into the memory. After storing t number of pairs into the table, the chain computation unit sends completion signal sgc1 (1 bit value) to the register SCC and terminates execution until the start signal received.

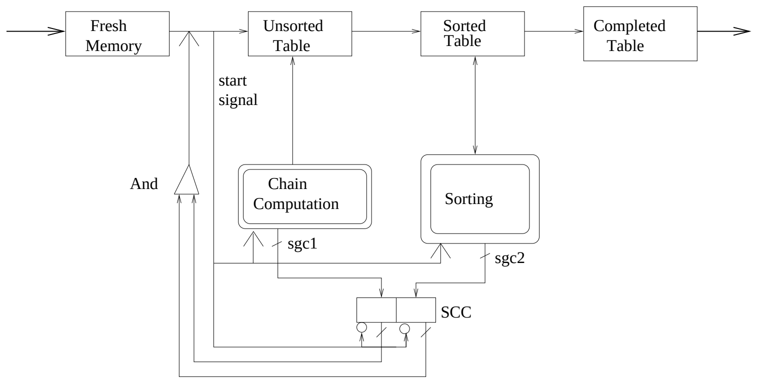

Fig. 1. Table Preparation

At the sorting phase, the previous table (unsorted) is to be sorted into increasing order of end points. Both the chain computation unit and the sorting unit run in parallel, i.e., while the chain computation unit computes the ith table, the sorting unit performs sorting on the $(i-1)$th table. After completion of sorting phase, a completion signal sgc2 is sent to SCC and the execution is stopped until a start signal is received. The assembly line will shift (i.e., the fresh memory, unsorted table and sorted table will be copied into unsorted memory unit, sorted memory unit and completed table unit respectively) when SCC receives both the signals sgc1 and sgc2 (i.e., when both the chain computation unit and sorting unit will report completion). After completion of assembly line movement, start signal will be sent to both the chain computation unit and sorting unit and the SCC is set to zero.

There are several issues to be considered.

- *Parallel sorting:* Chain computation and sorting hardware are to be designed so that they complete simultaneously. Depending on the design and speed of the chain computation stage it is required to determine whether parallel in-place sorting is required. The other issue is the type of table memory being used and whether random access is supported. In case parallel sorting is to be used, one can use mesh sort which requires a 2-d table structure. Then the chain computation phase will be required to access a 2-d memory.
- Both chain computation and sorting phase will require memory writes. For the chain computation stage, batching can be used to reduce number of memory accesses. Also chain computation and memory access can be pipelined to some extent.
- We are using four blocks of high speed memory while keeping the actual tables into DVDs. The completed table in a high speed memory will be written to a DVD and then the high speed memory will be cycled back. The time to copy from high speed memory to DVD will be overlapped with the chain computation and sorting phases.

5.1 Chain Computation Phase

Suppose there are n processor units $P_1, P_2, \ldots, P_n$ available at the chain computation phase. In Figure 2, we describe the architecture of the chain computation unit. The given memory block (fresh memory) is partitioned into n separate Processor Memory Space (PMS) units $PMS_1, PMS_2, \ldots, PMS_n$. Each processor P_i will store $\frac{t}{n}$ (start point, end point) pairs into PMS_i through a write block (WB) unit WB_i. Each PMS_i has $\frac{t}{n}$ memory locations to store the pairs and its starting address add_i (address of the first memory location) is stored in WB_i. Hence to access j^{th} memory location of PMS_i, the offset j is to be added with add_i to get the exact address. Processors execute the

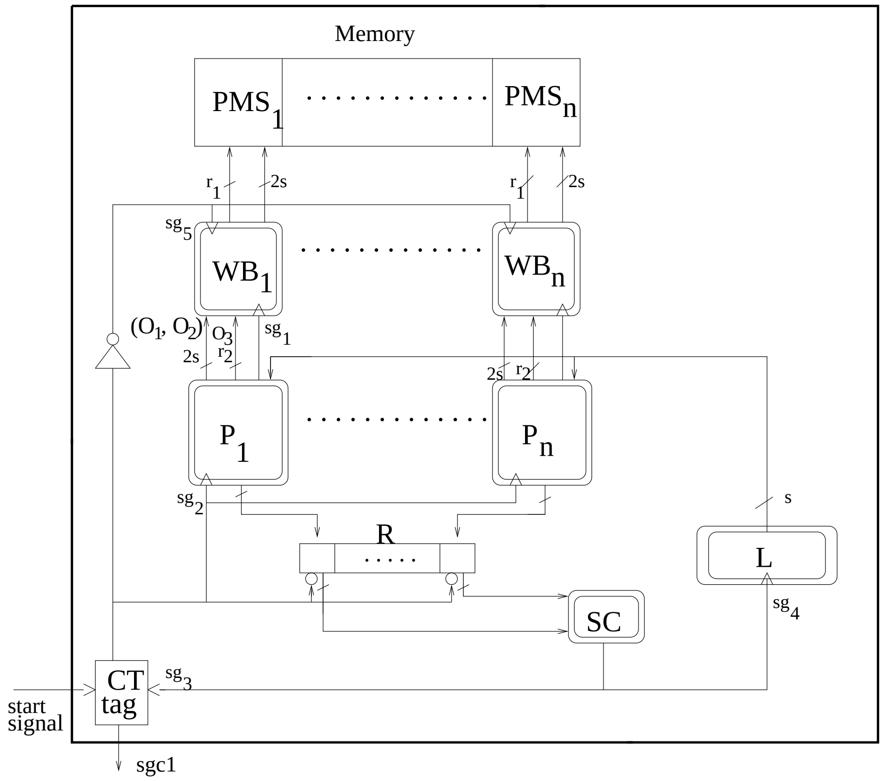

Fig. 2. Architecture of chain computation phase. i/p : start signal; o/p : sgc1

chains with different start points which are coming from *start point generator*, with each processor having its own start point generator. Let O_3 (offset) be the next free location of the corresponding PMS. After encountering a DP, the processor enables the *write block* unit by the signal sg_1 and passes the address O_3. Then the corresponding WB unit goes to the exact address of the free location by adding the offset with the starting address of the PMS and storing the pair (O_1, O_2) into the location. Processors run in parallel and after generating $\frac{t}{n}$ number of DPs, the corresponding processor passes completion signal (1-bit value) to the n-bit register R and stops the execution until it receives a start signal sg_2 from the CT (see Figure 1). L is an s-bit LFSR which is used as function generator and its internal state value passes to each of the processors to do the output modification of the function f. SC is a sequential circuit to check whether all the values R are 1. If yes, then the table has completed and SC sends a signal sg_4 to enable L to generate the next state (for the next table) and sg_3 to set CT to 1. Then CT will send a signal sgc1 to SCC (see Figure 1) requesting to move the table, a signal sg_5 to disable write block, a signal sg_2 to the processor and clear the contents of R. After the movement of the assembly line, the start signal (see Figure 1) sets the value of CT to zero and the write blocks will be enabled to write the pairs for the next table.

The following are some of the rationales for our design decisions:

- *Utility of having separate memory spaces:* Each processor P_i uses separate processor memory spaces PMS_i to store the (start point, end point) pairs. This avoids multiple access of same memory space and it is possible to use this idea since sorting is done separately.

- *Each processor generates $\frac{t}{n}$ DPs:* Since DPs are generated at different time points and a processor may have to consider different number of chains, the time taken by processor will be different (though the expected time

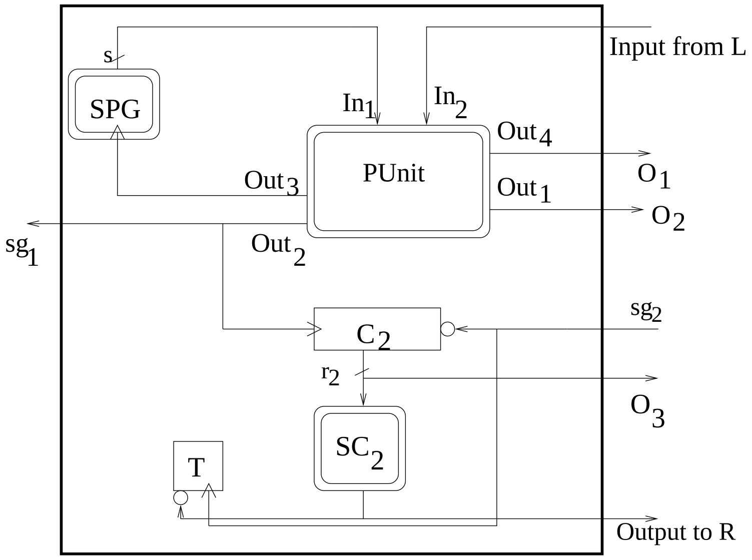

Fig. 3. Architecture of a processor P_i. $i/p : sg_2$, function mask; $o/p : sg_1, O_1, O_2, O_3$

will be same for all processor). Consequently, it may happen that one processor may complete ahead of others and hence will be idle for some time. On the other hand allowing each processor to generate the same number of DPs considerably simplifies the design and the expected delay is zero.

- *No overlap of processing between tables:* At no point of time, two processors will be handling chains of different tables. This again simplifies design.

Description of a Processor: Figure 3 describes the architecture of a processor. Each processor takes two inputs, a signal sg_2 and s-bit output modification value from L. The 1-bit register T is the control unit of the whole processor unit; the processor will stop if T is set to be zero and start running if the value of T is 1. C_2 is the counter to count the number of DPs encountered and it is incremented after encountering a DP. SC_2 checks whether number of DPs encountered reaches $\frac{t}{n}$. If yes, then the value of T will be set to zero and the whole processor unit will stop until the signal sg_2 resets C_2 to zero. A start point is generated by the *start point generator (SPG)* unit and passes to the $PUnit$ as the input In_1. Then $PUnit$ takes other input In_2 from L, which is the internal state of L (i.e., function masking) and starts executing the chain with the start point until it reaches a DP or the chain length reaches t; if yes then it outputs a signal Out_3 to SPG to generate a new start point, loaded into the register R_1 and passes to $PUnit$ as an input (In_1) for the next chain. If a DP is encountered, then $PUnit$ outputs a signal Out_2 to increase the counter C_2 by 1 and enables (the signal sg_1) the WB unit to load the (start point, end point) pair (O_1, O_2) and the offset address O_3.

A suggestion for SPG to be implemented using an LFSR where each P_i has its own SPG as opposed to a global SPG for all the P_i's. See (Mukhopadhyay and Sarkar 2006) for parallel start points generation using LFSRs sequences. This simplifies the design considerably while retaining the pseudo-random characteristic of start points.

Description of $PUnit$: Figure 4 describes the $PUnit$ where inputs In_1 is a new start point which is loaded into the register R_1 and In_2 is stored into R_4 for function masking. The counter C_3 is set to zero through the multiplexers when a new start point is loaded into R_2. The function f is applied on R_2 and the output is loaded into the register R_3 followed by function masking (xoring R_3 and R_4). The result is stored into the register R_5 to check for DP. If DP is encountered, then the multiplexers select the second line so that a new start point is loaded into R_2 and the counter C_2

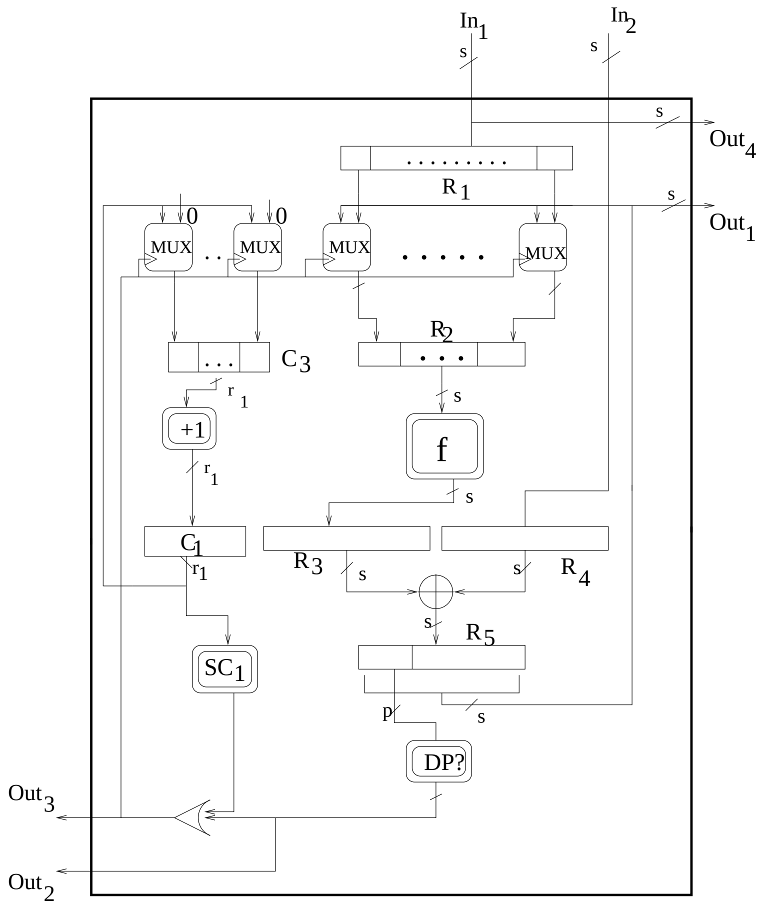

Fig. 4. Architecture of $PUnit$. $i/p : In_1, In_2$; $o/p : Out_1, Out_2, Out_3, Out_4$

will set to zero. Otherwise R_5 and C_1 will be copied (in a synchronized operation) into R_2 and C_3 respectively for the next iteration in the chain. The increment of C_3 and copying to C_1 will be synchronized with the application of f on R_2 and output to R_3. The result of one operation will not be used until the other one is completed.

Note that in our design we use chain length counter (i.e., C_1) which adds complexity to the circuit. Removing the chain length counter gives rise to the possibility that a DP in some chain occurs after a very long time or does not occur at all. This will stall the operation. While this will be rare event, it cannot be ignored. Counter chain length is one way of handling this. There may be other ways. Also note that we do not store chain length in the table. This reduces memory requirement but will increase online search time for false alarms.

Description of a processor when f is a multi-round function: Let us consider the case when the function f is a multi-round function, i.e.,

$$f = RF_q \circ RF_{q-1} \circ \cdots \circ RF_2 \circ RF_1$$

where q is the number of rounds. For example DES and AES are multi-round block ciphers and A5/3 (ETSI/SAGE 2002) is an example of a stream cipher whose design is based on the 8-round block cipher KASUMI (3GPP). We apply q-stage pipeline strategies to deal with q-different chains in parallel within a processor as follows (this idea has been earlier used in (Quisquater and Standaert 2005) and (Wiener 1996)). In the architecture of a processor unit (Figure 3), the $PUnit$ is replaced by $PUnitRound$ (the description of PUnitRound is given below). For each table,

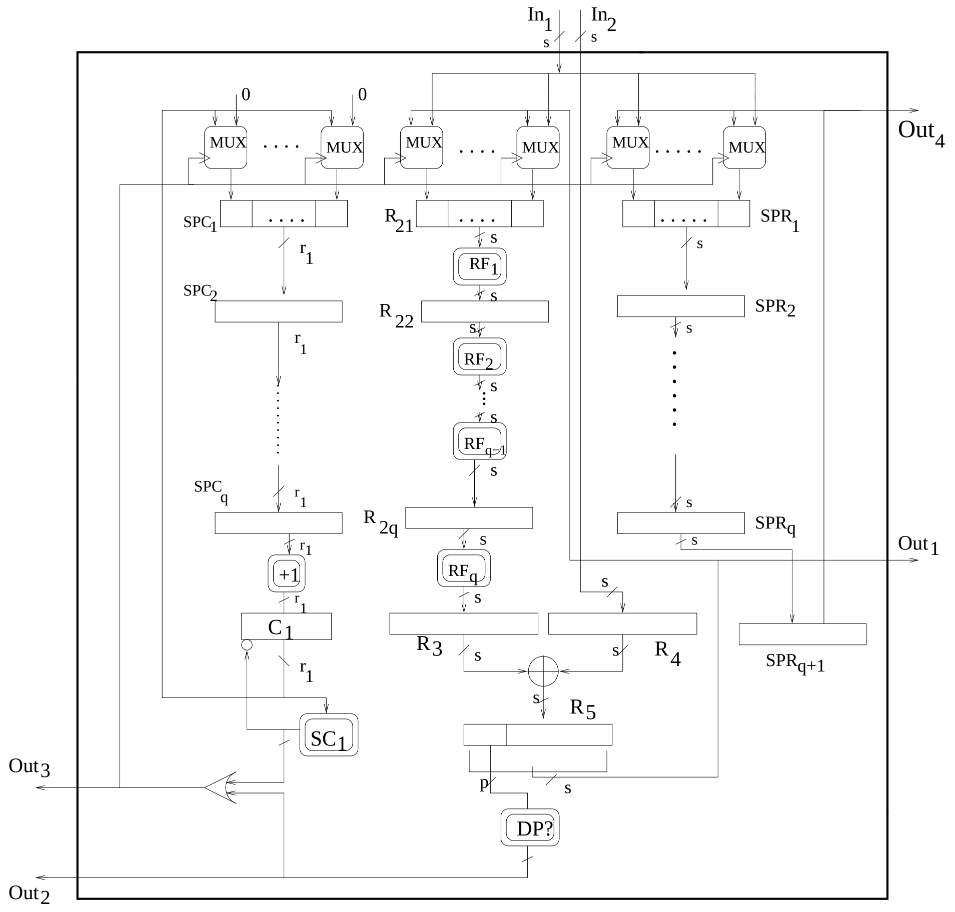

Fig. 5. Architecture of PunitRound. $i/p : In_1, In_2; o/p : Out_1, Out_2, Out_3, Out_4$

the SPG unit generates q many start points initially.

Figure 5 describes the $PUnitRound$. We use $q+1$ counters $SPC_1, SPC_2, \ldots, SPC_q, C_1$ of r_1-bit each. Initially, the SPG generates q many start points. At each time, the start point in register SPR_i will be copied into the next register SPR_{i+1} to keep track of it, since after getting a DP, we need to get the corresponding start point to return. Pipelining strategy is applied in the execution of round function and whenever a DP is encountered, the processor outputs the DP and the corresponding start point which is available at the register SPR_{q+1}. The following are synchronized operations:

- Copying SPC_i to SPC_{i+1}, SPR_i to SPR_{i+1} and R_{2i} to R_{2i+1} for $i = 1, 2, \ldots, q-1$.
- Copying SPC_q to C_1, SPR_q to SPR_{q+1} and R_{2q} to R_3.
- Copying C_1/"0" to SPC_1, SPR_{q+1} to SPR_1 and R_5 to R_{21}.

5.2 Sorting Phase

We do not describe details of sorting hardware but discuss the various issues that need to be considered. The sorting hardware so that the sorting and chain computation should complete simultaneously. In the chain computation phase,

for a table with size $m \times t$, the total number of f invocations required is mt whereas the sorting phase could be done in $m \log m$ comparison using a single processor and the sorting should be *in-place*. If we have t many processors available at the chain computation phase, then total number of f invocations will be reduced from mt to m by running the processors in parallel. But for significantly large t, t-many processors may be expensive. Also, one f invocation takes more time than one comparison operation for sorting. So sorting with a single processor will not take more time than chain computation. But the chain computation requires memory write which is done in parallel and sorting requires both memory read and write. Hence depending on the memory speed one may have to perform parallel sorting (including memory read and write) so that the sorting and chain computation phase complete simultaneously.
Note that, at the sorting phase if there is a collision (i.e., common DP in different chains), then we randomly select one chain to store and remove others, but it is desirable to select the maximum length chain for getting more coverage. In our design we are not storing the individual chain length in the table, so we cannot take the maximum length chain for the collision. Also since the sorting phase starts after completion of chain computation phase for a table, we may need to remove some of the chains at the sorting phase due to the collision. Thus to get a constant coverage, more chains need to be computed in the chain precomputation phase. On the whole, our design is simpler and requires less amount of memory since we do not take the extra overhead of storing individual chain length.

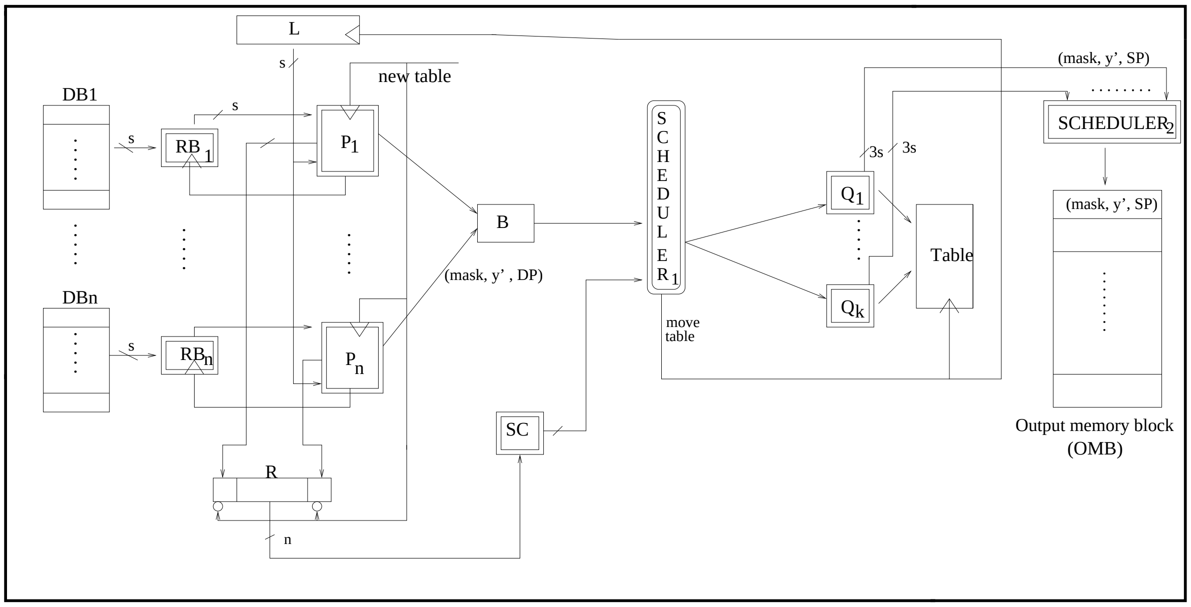

Fig. 6. Architecture for the matching phase for a single table when D is large. i/p : Data points; o/p : OMB

6 Online Search

The online stage consists of two phases–*matching* and *find key*. In the matching phase, table lookup is performed when a DP is encountered during an iteration (execution of the chain starting with the given value y). If the encountered DP is not in the table, then we will not be able to find the key by iterating further and can skip the current search in the rest of this table for that given value y. To search the key in the i^{th} table, we need to execute the following chain.
$y \xrightarrow{\phi_i} k_{i+0} \xrightarrow{f}\xrightarrow{\phi_i} k_{i+1} \xrightarrow{f}\xrightarrow{\phi_i} k_{i+2} \rightarrow \ldots \rightarrow k_{i+t-1} \xrightarrow{f}\xrightarrow{\phi_i} k_{i+t}$.
After each iteration (f application + masking (ϕ_i)) DP is checked and if found we stop the chain.

Suppose D points $y_1, \ldots, y_D$ are available at the online stage and we have to find the pre-image of any one of these points where y_i, $i = 1, 2, \ldots, D$ is viewed as unrelated random points. This enables us to perform independent search for different data points. Suppose we have processors $P_1, P_2, \ldots, P_n$ which are dedicated to perform the f invocation and $Q_1, Q_2, \ldots Q_k$ are I/O processors to perform the table lookup. We now describe the matching phase architecture for the following cases depending on the value of D.

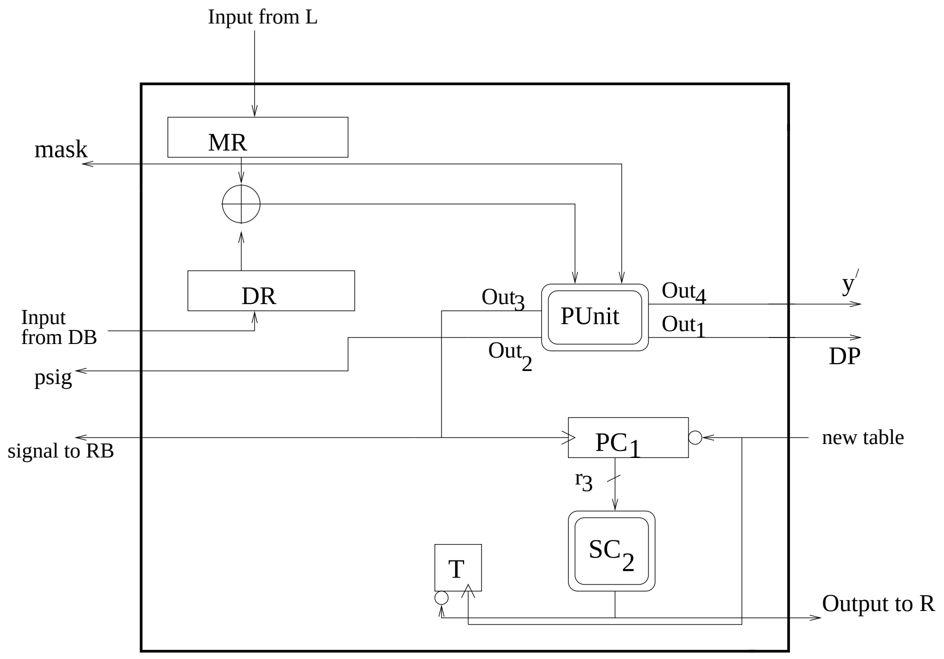

Fig. 7. Architecture of a p-processor in the matching phase architecture when D is large. i/p : new table signal, y, function mask; o/p : mask, y', DP, psig, load signal to RB, completion signal to R

6.1 For Many Data Points

Let there be sufficient amount of data available to the attacker. We partition the data points into n separate data blocks $DB_1, DB_2, \ldots, DB_n$ with z data points in each. Then $D = z \times n$. We apply the search technique for all the data within a single table and after completion of the search for all data points we move to the next table. *Since table load is expensive, we complete the search on one table before moving onto the next table*. Figure 6 illustrates the architecture to perform the parallel searching for a single table and for all data points. The processor P_i stores a data point from DB_i into its internal register DR and the mask value corresponding to the table (coming from L) in the register MR (see Figure 7 that describes a P-processor). The counters PC_1 keeps track of the number of data points already covered. The P-processors and Q-processors are connected through a common buffer queue B and a scheduler SCHEDULER$_1$. To search in a table, the processors $P_1, P_2, \ldots P_n$ are assigned to perform the DP search technique for different data in parallel where the data are coming from its data block through the read block unit. So these processors are essentially executing n different chains corresponding to n different data in parallel. The table lookup is needed whenever a DP is encountered during the execution of the chain while searching the key in a table.

After encountered a DP during the execution of the chain, the corresponding processor generates a signal $psig$. It then passes the tuple (mask, y', DP) to B and a load signal where "mask" is generated by LFSR L and $y' = masking(y)$. The processor also passes a signal to RB to read the next data point from data block. After this, the processor starts executing the chain for the next data. In this way, each P-processors keeps on executing until it finishes the search for all z data points. After completion it sends a completion signal to the register R. Processor waits until it receives the new table signal. Job of SCHEDULER$_1$ is to check the buffer queue. If there is any tuple in the buffer queue then it searches for a free I/O processor and if there is any free I/O processor, it assigns the DP into the (the ordering is $Q_1, Q_2, \ldots, Q_k$ circularly) free I/O processor for table lookup. The queue size is to be chosen such that it will not be full until there is a free I/O processor. During the table lookup, if a match occurs, then the corresponding Q-processor passes the tuple $(mask, y', SP)$ (SP is the start point for the data y) to the SCHEDULER$_2$ to store the tuple into OMB to get the key. After receiving completion signal from all P-processors the SCHEDULER$_1$ checks whether B is empty and all the Q-processors have finished the table lookup. After completion of all table lookup, a new table will be loaded and continue.

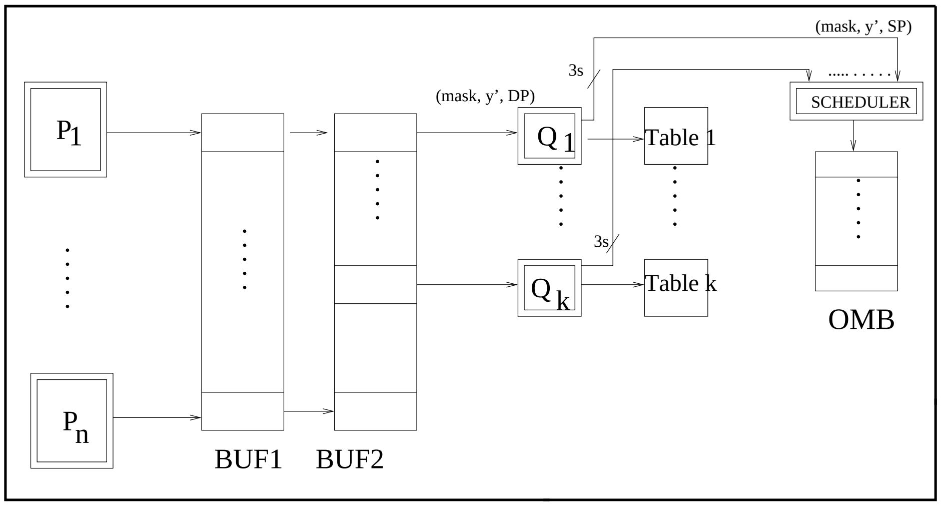

Fig. 8. Architecture for matching phase when $D = 1$

Analysis: Let 2^{-p} be the probability of a point being a DP. Hence, we can expect one DP in a random collection of 2^p points. In our parallel execution, n processors are executing in parallel and generating n many random points each time. Assuming, $n < 2^p$, after each $\lfloor \frac{2^p}{n} \rfloor = t_1$ (say) iterations we can expect one DP. At each of time $T = it_1$ for $i = 1, 2, \ldots$, we can expect a DP. The encountered DP will be assigned to the I/O processors for table lookups. Thus at time $T = it_1$, the corresponding DP will be assigned to the processor Q_i for $i = 1, 2, \ldots, k$. The next DP will be encountered (expected) at time $T = (k+1)t_1$, but at that time the I/O processor Q_1 may not be free, since the table lookup time (γ) is quite significant. Let us consider the following cases.
Case 1: When $kt_1 = \gamma$, i.e., $k2^p = \gamma n$, then at time $T = (k+1)t_1$, the I/O processor Q_1 will be free (since the time difference between the present time and the time when the processor Q_1 was assigned the DP is $(k+1)t_1 - t_1 = kt_1 = \gamma$, the table lookup time). So the corresponding DP will be assigned to Q_1 for table lookup. In this way the next DP will be assigned to Q_2 and so on. So in this case all the processors will remain busy at all the time. For a table with size $m \times t$, the total number of f invocations will be reduced from tD to $\lfloor \frac{tD}{n} \rfloor$. Then the total runtime for a single table is $\frac{tD}{n} + \gamma$. Hence in this case the total number of f invocations is reduced by a factor of n and the *effective* number of table lookup required is only one for a single table.
Case 2: When $kt_1 < \gamma$, then we need to use the buffer queue. Note that the DPs are coming at the following expected times:

$$T = t_1, 2t_1, 3t_1, \ldots, kt_1, (k+1)t_1, \ldots$$

So up to time kt_1, we keep on assigning the DP's into the I/O processors. But after that the next (circular ordering) I/O processor, i.e., Q_1 will be free at time $t_1 + \gamma$. Thus the next generated DP's need to store into the waiting queue upto time $T = (k + \jmath)t_1$, where $\jmath$ is the integer such that, $(k + \jmath - 1)t_1 < \gamma < (k + \jmath)t_1$. Hence the *optimal* size of the buffer queue is $\jmath$ so that all the processors will remain busy at all the time. Hence in this case also the total number of f invocations is reduced by a factor of n and *effective* number of table lookups required is $\jmath$. Hence, total runtime for a single table is $\frac{tD}{n} + \jmath\gamma$.
Case 3: When $kt_1 > \gamma$, this case is similar to the case 1, except that in this case not all the k I/O processors will be busy, some of the I/O processors will always be idle which is not desirable. In this case, one can use fewer number of I/O processor if some of them are busy.
In Figure 6, k many I/O processors are randomly accessing the table (memory block). Hence the memory block need to have multiple data and address bus to support this multiple access. The table lookup time $\gamma = \delta \log m$ where δ is the memory access time. Note that in the above analysis, we have assumed that the Q-processors will have finished their table lookups in the same ordering which may not be true. More than one match can occur at the same time for the Q-processors and that is the reason we need to have SCHEDULER$_2$.

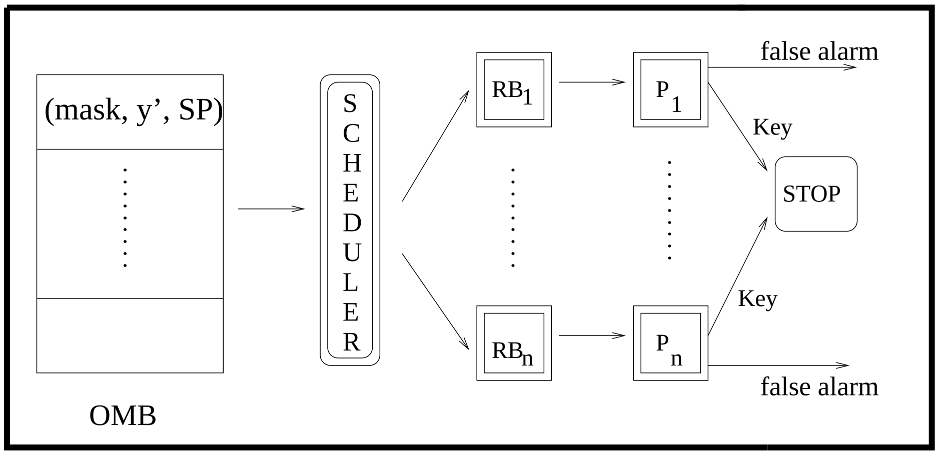

Fig. 9. Architecture for parallel key find strategy

6.2 For a Single Data Point

Suppose the attacker has a single data point at the online stage, i.e., $D = 1$. We perform the parallel search strategy by grouping tables $GT_1 = \{Table_1, Table_2, \ldots, Table_n\}; GT_2 = \{Table_{n+1}, Table_{n+2}, \ldots, Table_{2n}\}; \ldots$ such that each group contains n many tables. In Figure 8, we describe the architecture where P_i's are the processor units running in parallel to search for DP for n tables from the same group in the increasing order of the table number. After encountering a DP, the processor passes the quadruple $(mask, y', DP)$ to BUF_1 and waits until the other processors finish their DP search for the same group. After completion of DP search for all the tables in the group, BUF_1 will be stored into BUF_2 in increasing order of table numbers and the P-processors start searching the DP for the next group of tables. The processor P_i is similar to the processor which is used in the many data points case (see Figure 7) expect for the following. (1) register DR will always contains the given data point. (2) For each group of tables, the LFSR (L) will clock n times to get the corresponding mask value for the tables and store it to the register MR for each table.

For table lookup, k many Q-processors are connected to the first k position of BUF_2. Parallel table lookup is performed for first k tables in the group and after completion of all this k table lookup, we pop the first k tuples and push the next k tuple in the first k places of BUF_2. Then we load next k tables from the same group for table lookup. This technique can also be used when D is small.

Analysis: We can expect one DP in a random collection of 2^p points. In our parallel execution, n processors are executing in parallel and generating n many random points each time. Hence after 2^p time, expected number of DPs is n which completes the f invocation for a group of tables. Let $T_1 = 2^p$. The time required to complete table lookups for a group = $\frac{\gamma n}{k}$, since k many Q-processor are running in parallel. Let $T_2 = \frac{\gamma n}{k}$. Let us consider the following cases.
Case 1: When $T_1 = T_2$, i.e., the total expected time required to complete f invocations is same as the total time required to complete table lookups for a group. Then the following will be done simultaneously: (1) i^{th} group of tables, i.e., GT_i completes the f invocation stage and (2) $(i-1)^{th}$ group of tables, i.e., GT_{i-1} has completed the table lookups stage. There are total number of $\lfloor \frac{r}{n} \rfloor$ group of tables. Hence the expected runtime required to complete the matching phase in this case = total time required to complete f invocation stage for $\lfloor \frac{r}{n} \rfloor$ group of tables + time required to complete the table lookup step for the last group ($GT_{\lfloor \frac{r}{n} \rfloor}$) of table = $\lfloor \frac{r}{n} \rfloor \times 2^p + \frac{\gamma n}{k} = \left(\lfloor \frac{r}{n} \rfloor + 1\right) 2^p$.
Case 2: When $T_1 < T_2$, i.e., table lookup time dominates the total time. The total table lookup time = $\frac{r}{n} \times \frac{\gamma n}{k} = \frac{\gamma t}{k}$, which is independent of n.

7 Finding the Key

After a match is found in table lookup step, we come to the corresponding start point and repeatedly apply the function (f + masking) until it reaches $masking(y)$. The previous value it visited is k. Hence, to get the key from the given $(mask, y', SP)$, the following chain is executed. $SP \xrightarrow{f} \xrightarrow{masking} k_1 \rightarrow \ldots \rightarrow k_i \xrightarrow{f} y \xrightarrow{masking} k_{i+1} \rightarrow \ldots \rightarrow k_{i+t-1} \xrightarrow{f} \xrightarrow{masking} DP.$ (1)
Figure 9 describes the architecture for parallel key find strategy where n processors $P_1, P_2, \ldots, P_n$ are running in parallel taking input tuples from OMB.

Finding a matching endpoint when searching the key in a table does not necessarily imply that the key is in the table, since the key may be a part of a chain that has the same end point but is not in the table. It is called a *false alarm.*

7.1 Description of a Processor

Figure 10 describes the P-processor in the parallel key find architecture. Each processor takes the input $(mask, y, SP)$ from the OMB and stores $mask, y', SP$ respectively into the registers R_8, R_6 and R_1. Then the processor executes chain (1). Every time it checks for equality with y' and stops if it finds the match and returns the previous point as the key. If it finds no match after executing complete chain (length t), it returns a false alarm.

7.2 Analysis

Finding the key can require "substantial" time compared to finding a table match due to false alarms. The number of false alarms can be as large as half the total number of f required for online phase. However, table match requires memory access, whereas finding key does not.

8 Cost Analysis

We would like to perform a cost-time analysis of TMTO and exhaustive search attacks. To do this, we need to identify the dominant components of both the attack time and the costs. This is relatively easy to do for exhaustive search. The function f has to be applied on every possible input in the domain. Hence, the dominant component of the time is the time required to apply f a total of N times; for parallel implementation, this time is scaled down by the number

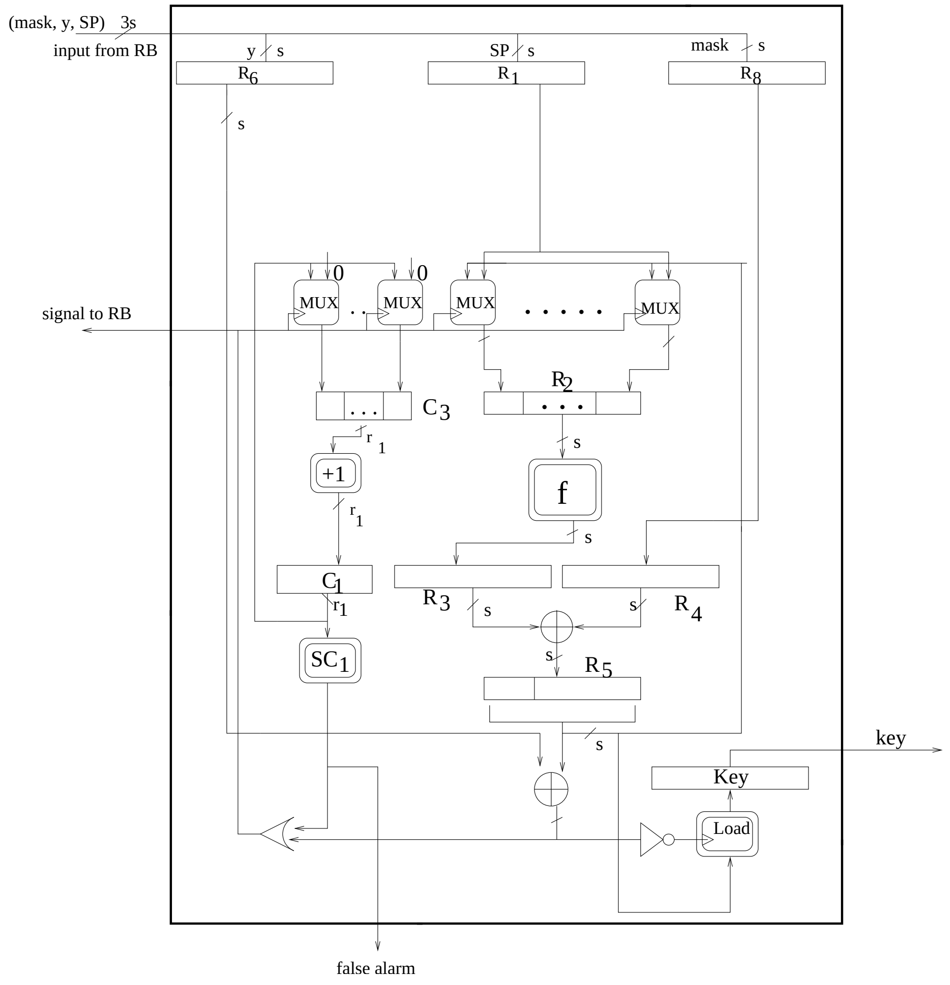

Fig. 10. P-processor in the parallel key find architecture

of processors used. The dominant cost component is the cost of implementing the parallel f-invocation units (or processors). The cost should also include the manpower cost, but this is harder to estimate.

A TMTO algorithm is more complex than exhaustive search and deriving an appropriate cost model is more difficult. The precomputation phase of the TMTO algorithm has several time components – time required to obtain the (start-point, end-point) pairs; memory access time required to store these pairs into the table; and the time required to sort the tables. The online time has two major components – time to obtain the end-points; and the time for table look-up. Similarly, the cost has several components – the cost of the parallel f-invocation units; and the cost of storage media. In the online stage, the wiring cost of connecting processors to memory can also be substantially high (Wiener 2004).

To a large extent, the appropriate choice of the cost model depends on the underlying architecture used for the implementation. Below we provide a top level description of our proposed architecture. This top level view makes understanding of the cost analysis easier.

Pre-Computation Phase: Let us consider the tasks performed in the pre-computation phase. At a top level this consists of the following two separate tasks for each table.

1. Compute the chains and write the (start-point, end-point) pairs to the table.
2. Sort the table.
3. Write the table into a DVD

Let us call the first task, chain-computation, the second task sorting, and third task DVDwrite. In the Hellman+DP method, a total of r tables are to be prepared. Let us denote the tables by $Tab_1, \ldots, Tab_r$. Consider the following algorithm.

```
1. Perform chain-computation for $Tab_1$;
2. do in parallel
      perform chain-computation for $Tab_2$;
      perform sorting for $Tab_1$;
3. for $i = 3$ to $r$ do in parallel
      perform chain-computation for $Tab_i$;
      perform sorting for $Tab_{i-1}$;
      initiate DVDwrite for $Tab_{i-2}$ ;
4. end do;
5. do in parallel
      perform sorting for $Tab_r$;
      perform DVDwrite for $Tab_{r-1}$;
6. perform DVDwrite for $Tab_r$.
```

This algorithm pipelines the chain computation for Tab_i with the sorting of Tab_{i-1} and DVD writing for Tab_{i-2}. Under the reasonable assumption that the sorting time is at most the chain computation time, the major time component is at most the time required for chain-computation of r tables plus the time required to sort a table and write to DVD. The chain-computation itself has two tasks – parallel f-invocations and writing to high speed memory. These two tasks can also be pipelined as we discuss below.

Suppose n many f-invocation units are available. Each table has a total of m many (s-p, e-p) pairs. These are divided into m/n blocks $B_1, \ldots, B_{m/n}$, where each block contains n pairs. The n many f-invocation units will be operating in parallel to produce one block.

```
1. Generate block $B_1$;
2. for $i = 2$ to $m/n$ do in parallel
      Generate block $B_i$;
      Write block $B_{i-1}$ to the table;
3. end do;
4. Write block $B_{m/n}$ to the table;
```

Producing each block B_i requires $n \times t$ many f-invocations. We may assume that the time for nt many f-invocations is more than the time to write a block of n pairs to the table. Hence, the dominant time is the time required to compute all the chains in a table, which is time required for $m \times t$ many f invocations.

Let us consider the time required to prepare all the tables. Using the above two algorithms, the total time will essentially be mrt many f-invocations done in parallel by n many f-invocation units. The cost has several components–cost of the f-invocation units; cost of input/output (I/O) units to write the blocks B_i's to the table; cost of storing r tables; and cost of the sorting unit. The dominant cost components are the cost of the f-invocation units and the cost of storage (memory).

On-Line Phase: We would like to avoid the lower bound on the wiring cost obtained by Wiener (Wiener 2004). Our architecture can be described as follows. There is a set of n many f-invocation units, which produce DPs and write them to a buffer. There is another set of k many I/O processors, which read from this buffer and perform look-up into the tables.

At a time, the k I/O processors are connected to k tables. Once look-up on k tables are completed, the tables are moved out and a new set of k many tables are moved into place. Thus, the system operates as follows: Look-up on $Tab_1, \ldots Tab_k$ are completed, then look-up on $Tab_{k+1}, \ldots, Tab_{2k}$ are completed, and so on. Once a table is replaced, it is never loaded again for this data set. Thus, if we have D targets, then the look-up into table Tab_i for all these targets are completed before Tab_i is replaced.

In the above scenario, the following two tasks are performed in parallel.

- Apply f-invocations to the D targets and write the final DPs to the buffer.
- Read from the buffer; perform look-up in the k tables; and then replace the tables.

With a suitable design and choice of the parameters k and n, we can make the assumption that the above two tasks require approximately the same time. Under this assumption, the total time required in the online phase can be taken to be the total time for all the f-invocations. Further, in this architecture, the wiring cost is minimal and the dominant cost is the cost of implementing the f-invocation units. The task of an I/O processor is relatively simple and also we will have k to be much less than n. Hence, the cost of implementing k I/O processors can be ignored with respect to the cost of implementing the n many f-invocation units.

We summarize the above discussion with respect to the cost and time measures.

Pre-computation phase:

- Time: time required for rmt many f-invocations;
- Cost: cost of implementing n many parallel f-invocation units and cost of storing r many tables.

Online phase:

- Time: time required for rtD many f-invocations;
- Cost: cost of implementing n many parallel f-invocation units.

8.1 Approximate Cost Analysis

In CHES 2005, Good and Benaissa (Good and Benaissa 2005) proposed a new FPGA design for AES using Xilink Spartan-III (XC3S2000). The cost of a Xilinx Spartan-III FPGA device whose cost is around 12 USD (see (Quisquater and Standaert 2005)). The speed of encryption of the design in (Good and Benaissa 2005) is 25Gbps=0.2×2^{32} AES-128 encryption/sec. Under the assumption that the cost and time scale linearly as we move from one processor to n processors, the total processor cost for n processor units is $H_p = 12n$ USD and the speed is $n \times 0.2 \times 2^{32}$ AES-128 encryptions/sec. Let T_{sec} be the pre-computation time in seconds. In T_{sec} time, the number of encryptions will be, $T_{sec} \times n \times 0.2 \times 2^{32}$.

For a general s-bit ($s \leq 128$) cipher, attacking $D = 2^d$ online data points, the number of encryptions required at the pre-computation stage is 2^{s-d}. *We assume that for an s-bit cipher with $s \leq 128$, the throughput and chip area will remain same as for the best AES-128 implementation.* Hence, in T_{sec} time, the number of encryptions will be, $T_{sec} \times n \times 0.2 \times 2^{32}$ and we get,

$$T_{sec} \times n \times 0.2 \times 2^{32} = 2^{s-d}. \tag{1}$$

Using $H_p = 12n$, we get $T_{sec}H_p = 60 \times 2^{s-d-32}$, or

$$2^{32}T_{sec}H_pD = 60N. \tag{2}$$

Table 1. Trade-off for different values of s with $D = 1$

s	r	m	t	T_{sec}	n	H_p	H_m	k	H_w	τ_{sec}
56	2^{19}	2^{19}	2^{19}	$2^{16.5}$	2^{10}	$2^{13.6}$	2^{19}	$2^{8.5}$	2^{15}	0.31
64	2^{21}	2^{21}	2^{21}	$2^{16.5}$	2^{18}	$2^{21.6}$	2^{21}	2^{12}	$2^{18.5}$	0.03
80	2^{27}	2^{27}	2^{27}	2^{25}	2^{25}	$2^{28.6}$	2^{27}	2^{8}	$2^{14.5}$	0.62
86	2^{29}	2^{29}	2^{29}	2^{25}	2^{31}	$2^{34.6}$	2^{29}	2^{10}	$2^{16.5}$	0.61
96	2^{32}	2^{32}	2^{32}	$2^{38.3}$	2^{28}	2^{32}	2^{32}	1	$2^{6.5}$	80
128	2^{32}	2^{64}	2^{32}	$2^{70.3}$	2^{28}	2^{32}	2^{32}	1	$2^{6.5}$	80

This gives a new type of trade-off involving pre-computation time T_{sec}, processor cost H_p and data D whereas usual trade-off curve involves online time (number of f invocations), data and memory.

Memory Cost: We assume that one table will fit into one memory block. This simplifies the table management and in particular the design of the sorting algorithm. The latest cheap high density storage is DVD with storage capacity between 4 and 20 Gbyte. In the near future, SONY will launch the paper disk with capacity of 100 Gbytes. At present, we consider 4Gbyte ($= 4 \times 2^{32}$ bytes) DVD with cost around 1 USD. Since, for a table we need $\frac{2sm}{8}$ bytes storage, so $\frac{2sm}{8} \leq 4 \times 2^{32}$, or,

$$sm \leq 2^{36}. \tag{3}$$

DVD write time: At present, we consider the writing time for a 4GB DVD is 1min[3] ($\approx 2^6$sec). The total number of f-invocations required for a single table is mt and the time required for this is $t_1 = \frac{mt}{n \times 0.2 \times 2^{32}}$. Let $W_1, \ldots W_k$ be the DVD writers which are running in parallel. At each of time $T = it_1$ for $i = 2, \ldots r+1$, one table will be ready for DVD write. At time $T = (i+1)t_1$, the table T_i will be assigned to W_i for $i = 1, 2, \ldots, k$. The next table T_{k+1} will be ready for DVD write at time $T = (k+2)t_1$. If we choose $kt_1 \geq 2^6$, then at time $T = (k+2)t_1$, W_1 will be free (since the time difference between the present time and the time when W_1 was assigned the table is $(k+2)t_1 - 2t_1 = kt_1 \geq 2^6 =$ DVD write time). So the table T_{k+1} will be assigned to W_1 for DVD write. In this way the next table will be assigned to W_2 and so on. So in this case all the processors and DVD writers will remain busy at all the time. Hence from the above discuss we have $kt_1 \geq 2^6$, or,

$$k \times \frac{mt}{n \times 0.2 \times 2^{32}} \geq 2^6. \tag{4}$$

or,

$$k \geq \frac{n2^{36}}{mt}. \tag{5}$$

Note the there are r table to be written into r DVDs and each DVD write takes 2^6 seconds. The total time required for DVD write is $\frac{r2^6}{k}$ while k many DVD writers are running in parallel. This time must be less than or equal to the pre-computation time, i.e., $\frac{r2^6}{k} \leq T_{sec}$, or,

$$k \geq \frac{r2^6}{T_{sec}}. \tag{6}$$

We take $k = max\left(\frac{n2^{36}}{mt}, \frac{r2^6}{T_{sec}}, 1\right)$. Then k satisfies both the inequalities 5 and 6. At present, we consider the DVD writer cost is 100 USD each. The total DVD writer cost is $H_w = 100k$ USD. For r tables, memory cost is $H_m = r$ USD and total hardware cost $C = H_p + H_m + H_w = (12n + r + 100k)$ USD. Let us consider the following cases.

[3]For example writing speed of Samsung SH-W162 is 21.6MB/sec (16X).

Table 2. Trade-off for different values of s and $d = \frac{s}{4}$

s	r	$m = t$	T_{sec}	n	H_p	H_m	k	H_w	τ_{sec}
80	$2^{6.7}$	$2^{26.7}$	$2^{16.5}$	2^{14}	$2^{17.6}$	$2^{6.7}$	1	$2^{6.5}$	845
86	$2^{6.7}$	$2^{28.6}$	$2^{16.5}$	2^{18}	$2^{21.6}$	$2^{6.7}$	1	$2^{6.5}$	776
96	2^{8}	2^{32}	$2^{16.5}$	2^{26}	$2^{29.6}$	2^{8}	1	$2^{6.5}$	320
96	2^{8}	2^{32}	2^{25}	2^{17}	$2^{20.6}$	2^{8}	1	$2^{6.5}$	$2^{17.3}$
128	2^{11}	2^{43}	2^{25}	2^{41}	$2^{44.6}$	2^{11}	1	$2^{6.5}$	$2^{15.3}$
128	2^{32}	2^{32}	2^{25}	2^{41}	$2^{44.6}$	2^{32}	2^{13}	$2^{19.5}$	$2^{25.3}$
128	2^{32}	2^{32}	2^{38}	2^{28}	2^{32}	2^{32}	1	$2^{6.5}$	$2^{38.3}$

Case 1: $D = 1$ **(**$d = 0$**).** We choose the Hellman table parameters as: $r = m = t = N^{1/3} = 2^{s/3}$. The total number of f invocations required at the online stage $= r \times t$ and the time required for this is $\tau_{sec} = \frac{r \times t}{n \times 0.2 \times 2^{32}}$, running n processors in parallel with the speed of 0.2×2^{32} encryptions/sec. Suppose we want to finish the pre-computation within a day, then $T_{sec} = 2^{16.5}$ (the number of seconds in one day). From Equation 1, we get, $n = 5 \times 2^{s-48.5}$. For 1 year pre-computation time, i.e., $T_{sec} = 2^{25}$ (the number of seconds in one year) we need the number of processors, $n = 5 \times 2^{s-57}$. In Table 1, we summarize some of the trade-offs with different values of s.

Case 2: $D > 1$**.** The memory cost increases with the number of tables. We consider the following table parameters as in (Biryukov and Shamir 2000): $r = \frac{N^{1/3}}{D} = 2^{\frac{s}{3}-d}$ and $m = t = N^{1/3} = 2^{s/3}$. The total number of f invocations required for online search $= rtD$ and the time required for this is $\tau_{sec} = \frac{r \times t \times D}{n \times 0.2 \times 2^{32}}$, running n processors in parallel with speed of 0.2×2^{32} encryption/sec. From Equation 1 we get, $n = \frac{5 \times 2^{s-d-32}}{T_{sec}}$. Table 2 summarizes some of the trade-offs with different values of s and $d = \frac{s}{4}$. The rows of the tables were calculated by fixing some of the parameters as mentioned below.

- Table 1 ($d = 0$)

  –*rows 1 and 2*: Fix T_{sec} to be one day.

  –*rows 3 and 4*: Fix T_{sec} to be one year.

  –*rows 5 and 6*: Fix $H_p = H_m = 2^{32}$.

- Table 2 ($d = s/4$)

  –*rows 1, 2 and 3*: Fix T_{sec} to be one day.

  –*rows 4 and 5*: Fix T_{sec} to be one year.

  –*row 6*: Fix T_{sec} to be one year and $H_m = 2^{32}$.

  –*row 7*: Fix $H_p = H_m = 2^{32}$.

Discussion: From Tables 1 and 2, we conclude the following.

- 56-bit and 64-bit f's are completely insecure.

- For $d = 0$, with one year pre-computation time and around 500M USD investment it is possible to crack 80-bit f in online time less than one second. For multiple targets (data) with $d = s/4$, attacking 80-bit becomes easier.

- For $s = 96$, and with a single data point, pre-computation time is more than 4000 years. This is at a cost of around 1 billion USD. It is possible to bring down the pre-computation time to a few years by increasing the cost to around 1 trillion dollar. Another problem is that the size of single table becomes large and barely fits in a single storage unit (see the bound 3). In the presence of multiple data of the order of 2^{24} ($d = s/4$), the attack becomes reasonable. Hence, 96-bit f also does not provide comfortable security

- For $s = 128$, and with a single data point ($d = 0$), at least one of the parameters among (T_{sec}, H_p, H_m) become infeasible. Also even with $d = s/4 = 32$, one of the above parameters continue to remain infeasible. Increasing d beyond 32 is not practical. Hence, 128-bit can be considered to provide adequate security margin, at least until a new technological revolution invalidates the analysis performed here.

8.1.1 General Case For the general case, let us assume that C_1 and C_2 are the costs of one search unit and one storage unit respectively and ρ, δ are the rate of encryption and size of one storage unit in Gbyte respectively. Then Equation 1 becomes,

$$T_{sec} \times n \times \rho = 2^{s-d} \tag{7}$$

and, $H_p = C_1 n$ and $H_m = C_2 r$. Using $H_p = C_1 n$ in Equation 7, we get $T_{sec} \times H_p \times \rho = 2^{s-d} C_1$, or $\rho T_{sec} H_p D = C_1 N$. Since for a table we need $\frac{2sm}{8}$ bytes storage, so $\frac{2sm}{8} \leq \delta \times 2^{32}$, or,

$$sm \leq \delta 2^{34}. \tag{8}$$

This constraint is required because we are fitting one table into one storage unit. Let ϵ be the DVD (storage) writing time. Then equation 4 becomes, $k \times \frac{mt}{n \times \rho} \geq \epsilon$, or, $k \geq \frac{n \times \rho \times \epsilon}{m \times t}$ and equation 6 becomes, $k \geq \frac{r\epsilon}{T_{sec}}$. Thus we take $k = max\left(\frac{n \times \rho \times \epsilon}{m \times t}, \frac{r\epsilon}{T_{sec}}, 1\right)$. Let C_3 be the cost of one DVD writer, then $H_w = kC_3$USD.

8.2 Cost of Exhaustive Search

Cost analysis of exhaustive search is same as the cost analysis for TMTO pre-computation except the memory cost and DVD writer cost. Note that the processor cost H_p is required for both exhaustive search and TMTO pre-computation. The factor H_m is additionally required for TMTO. Hence, the trade-off for exhaustive search is same as Equation 2, i.e.,

$$2^{32} THD = 60N \tag{9}$$

where T denotes the time in seconds required for exhaustive search, H is the total processor cost and D is the number of data points. The general equation is the following.

$$\rho THD = C_1 N \tag{10}$$

8.3 Rainbow Method

The rainbow method replaces t Hellman table of size $m \times t$ into a single rainbow table with size $m' \times t$, where $m' = mt$. Let us consider the case when $s = 56$ (DES). Then $N = 2^{56}$, taking $m = t = N^{1/3}$, we get $m' = 2^{36}$, i.e. $sm' = 56 \times 2^{36} > 2^{36}$. This violates the constraint 3 ($sm' \leq 2^{36}$). Hence a single large rainbow table need to stored into more than one memory block (the number of memory block will increase with the value of s). Then the sorting algorithm becomes much more complicated since it has now to sort the table which is split into different memory blocks. On the other hand, if we break the large single rainbow table into several number of small mutually disjoint rainbow tables the online time increases by a factor of r, where r is the number of rainbow tables. In view of this, rainbow method is not a good choice for hardware implementation.

9 Application to Stream Ciphers with IV

Application of TMTO to stream ciphers with IV was analysed in (Hong and Sarkar 2005). For a k-bit stream cipher using an l-bit IV, consider the following $(k+l)$-bit one-way function f:

$$(k\text{-bit key}, l\text{-bit IV}) \mapsto (k+l)\text{-bit keystream prefix.} \tag{11}$$

Table 3. Trade-off of GSM for different values of D

D	r	$m = t$	T_{sec}	n	H_p	H_m	k	H_w	τ_{sec}
1	2^{29}	2^{29}	2^{25}	2^{31}	$2^{34.6}$	2^{29}	2^{10}	$2^{16.5}$	0.61
2^8	2^{21}	2^{29}	2^{25}	2^{23}	$2^{26.6}$	2^{21}	2^2	$2^{8.5}$	32
2^{16}	2^{13}	2^{29}	$2^{16.5}$	2^{24}	$2^{27.6}$	2^{13}	2^3	$2^{9.5}$	16
2^{22}	2^7	2^{29}	$2^{16.5}$	2^{18}	$2^{21.6}$	2^7	1	$2^{6.5}$	2^{10}

As pointed out in (Hong and Sarkar 2005), inverting this one-way function f will provide the secret key. Since many IVs are used with the same key, and since IVs are public, one can apply multiple data TMTO to f, using D many publicly available IVs. It has been shown in (Hong and Sarkar 2005), that if IV length is less than key length, then this the online time of TMTO is less than exhaustive key search. (This has resulted in the recent Ecrypt call for stream ciphers, to mandate IV length to be at least equal to the key length.) However, the pre-computation time becomes 2^{k+l} which is more than exhaustive key search. On the other hand, the importance of IV in a TMTO attack matters more than its length. The effective length of IV is also crucial and has been pointed out in (Hong and Sarkar 2005). Let us consider this point in more details.

The usual requirement on IV is that it should be a nonce, i.e., no value should be repeated. Thus, for example, one can fix a key and use the numbers $1, 2, \ldots,$ as IVs for different messages. Suppose at most 2^λ messages are encrypted before a key change. The above appears to be a valid protocol for using stream cipher. The problem is that in this approach, only the last λ bits of the IV ever change. If we put the (arbitrary) restriction that at most 1000 messages are encrypted before a key change, then $\lambda \approx 10$.

Suppose, for a particular key we have access to the keystream segment for about $32 = 2^5$ messages. This gives $D = 2^5$. Since we know all the IVs, we can apply TMTO to a search space of size $N = 2^{k+10}$ with $D = 2^5$. The precomputation time is $N/D = 2^{k+10}/2^5 = 2^{k+5}$ and the online time then comes to around $2^{2(k+5)/3}$. If $k = 80$, then the precomputation can be completed in one year at a cost of 2^{32} USD and the online time is around a minute. While the cost is quite high, it is not out of reach of powerful organizations.

We interpret this situation as indicating that to resist TMTO, it is *not* sufficient to have IV length to be equal to key length. The protocol must ensure that the entire IV length is actually used. One simple way of doing this can be to choose a random nonce as IV for the first msg encrypted using a particular key and then use nonce + 1, nonce + 2, . . . as IVs for subsequent msg.

9.1 GSM

For the GSM mobile phones (3GPP 2003), A5/3 stream cipher is used which is based on the iterated block cipher KASUMI. The cipher A5/3 uses 64-bit key and 22-bit effective IV size (others bits of IV are fixed). The following one-way function f from 86-bit to 86-bit has been considered in (Hong and Sarkar 2005):

$$(\text{64-bit key, 22-bit effective IV}) \mapsto \text{86-bit keystream prefix.} \tag{12}$$

The size of the search space for exhaustive search attack is 2^{64}. From Table 1 (see row 2), we have the time for exhaustive search attack which is same as the pre-computation time for TMTO to be $2^{16.5}$ sec with a 2^{21} USD investment.

This is certainly doable and hence GSM mobile phone communications cannot be considered secure for more than a day. However, can we consider such communications to be secure for a shorter duration such as an hour. For example, a stock order is placed over a phone and the order is executed within an hour. Once the order is executed, there is no need for secrecy. Thus, it is enough to ensure secrecy from the point of the order being placed and it being executed, which is at most an hour. If we consider only exhaustive search attacks, then such communication over GSM phones appears to be secure. However, if we apply TMTO to the search space of the function f defined in (12), then this might not be true.

The size of the search space f is $N = 2^{86}$. From Equation 1 we get, $n = \frac{5\times 2^{86-d-32}}{T_{sec}}$ where 2^d is the number of data points available to the attacker. Table 3 summarizes some of the trade-offs with different values of D where the table parameters are taken as: $r = \frac{N^{1/3}}{D} = 2^{\frac{s}{3}-d}$ and $m = t = N^{1/3} = 2^{s/3}$. From Table 3, we conclude that the A5/3 algorithm of GSM provides inadequate security.

10 Conclusion

In this paper, we have provided a detailed top-level description of an architecture for inverting a one-way function using TMTO. Based on our architecture, we have developed a new cost/time/data trade-off model and have used it to analyse the security of different key sizes. Our future work is to validate the different hardware architectures through a hardware synthesis tool. Such a simulation will provide estimates of the number of gates, the clock frequency, routing costs, power consumption, mean time between failures and other relevant parameters. This may lead to possible alterations of the design as well as provide a better understanding of different implementation issues.

Acknowledgments

Authors would like to thank anonymous reviewers for providing constructive and generous feedback. Despite their invaluable assistance any error remaining in this paper is solely attributed to the author.

References

3GPP. 3rd generation partnership program. http://www.3gpp.org/.

3GPP (2003). 3gpp ts 55.215 v6.2.0 (2003-09), a5/3 and gea3 specifications. http://www.gsmworld.com.

Amirazizi, H. and M. Hellman (1988). Time-memory-processor trade-offs. *IEEE Transactions on Information Theory 34*(3), 505–512.

Biham, E. (1994). New types of cryptanalytic attacks using related keys. *Journal of Cryptology 7*(4), 229–246.

Biham, E., A. Biryukov, and A. Shamir (1999a). Cryptanalysis of skipjack reduced to 31 rounds using impossible differentials. In *Eurocrypt 1999, Proceedings*, Volume 1592 of *Lecture Notes in Computer Science*, pp. 12–23. Springer.

Biham, E., A. Biryukov, and A. Shamir (1999b). Miss in the middle attacks on idea and khufu. In *FSE 1999, Proceedings*, Volume 1636 of *Lecture Notes in Computer Science*, pp. 124–138. Springer.

Biham, E. and A. Shamir (1993). *Differential Cryptanalysis of the Data Encryption Standard*. SpringerVerlag.

Biryukov, A. (2005). Some thoughts on time-memory-data tradeoffs. http://eprint.iacr.org/2005/207.

Biryukov, A. and A. Shamir (2000). Cyptanalytic time/memory/data tradeoffs for stream ciphers. In *Asiacrypt 2000, Proceedings*, Volume 1976 of *Lecture Notes in Computer Science*, pp. 1–13. Springer.

Biryukov, A. and D. Wagner (1999). Slide attack. In *FSE 1999, Proceedings*, Volume 1636 of *Lecture Notes in Computer Science*, pp. 245–259. Springer.

Borst, J., B. Preneel, and J. Vandewalle (1999). Linear cryptanalysis of rc5 and rc6. In *FSE 1999, Proceedings*, Volume 1636 of *Lecture Notes in Computer Science*, pp. 16–30. Springer.

COPACOPANA (2006). A codebreaker for des and other ciphers.

Denning, D. (1982). *Cryptography and data security*. Addison Wesley.

EFF (1998). *Electronics Frontier Foundation: Cracking DES*. O'Reilly and Associates.

ETSI/SAGE (2002). Specification of the a5/3 encryption algorithms for gsm and edge, and the gea3 encryption algorithm for gprs, document 1: A5/3 and gea 3 specifications.

Fiat, A. and M. Naor (1991). Rigorous time/space tradeoffs for inverting functions. In *STOC 1991*, pp. 534–541.

Gilbert, H., H. Handschuh, A. Joux, and S. Vaudenay (2000). A statistical attack on rc6. In *FSE 2000, Proceedings*, Volume 1978 of *Lecture Notes in Computer Science*, pp. 64–74. Springer.

Good, T. and M. Benaissa (2005). Aes on fpga from the fastest to the smallest. In *CHES 2005, Proceedings*, Volume 3659 of *Lecture Notes in Computer Science*, pp. 427–440. Springer.

Handschuh, H. and H. Gilbert (1997). χ^2 cryptanalysis of the seal encryption algorithm. In *FSE 1997, Proceedings*, Volume 1267 of *Lecture Notes in Computer Science*, pp. 1–12. Springer.

Hellman, M. (1980). A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory 26*, 401–406.

Hong, J. and P. Sarkar (2005). New applications of time memory data tradeoffs. In *Asiacrypt 2005, Proceedings*, Volume 3788 of *Lecture Notes in Computer Science*, pp. 353–372. Springer.

Kumar, S., C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler (2006). Breaking ciphers with copacobana-a cost-optimized parallel code breaker. In *CHES 2006, Proceedings*, Volume 4249 of *Lecture Notes in Computer Science*, pp. 101–118. Springer.

Lai, X. (1994). Higher order derivatives and differential cryptanalysis. *Communication and Cryptography*, 227–233.

Matsui, M. (1993). Linear cryptanalysis method for des cipher. In *Eurocrypt 1993, Proceedings*, Volume 765 of *Lecture Notes in Computer Science*, pp. 386–397. Springer.

Matsui, M. (1994). The first experimental cryptanalysis of the data encryption standard. In *Crypto 1994, Proceedings*, Volume 839 of *Lecture Notes in Computer Science*, pp. 1–11. Springer.

Mentens, N., L. Batina, B. Preneel, and I. Verbauwhede (2005). Cracking unix passwords using fpga platforms. In *SHARCS 2005, Proceedings*.

Mukhopadhyay, S. and P. Sarkar (2006). Application of lfsrs for parallel sequence generation in cryptologic algorithms. In *Applied Cryptography and Information Security 2006 (ACIS'06) in conjunction with ICCSA 2006, Proceedings*, Volume 3982 of *Lecture Notes in Computer Science*, pp. 426–435. Springer.

Oechslin, P. (2003). Making a faster cryptanalytic time-memory trade-off. In *Crypto 2003, Proceedings*, Volume 2729 of *Lecture Notes in Computer Science*, pp. 617–630. Springer.

Quisquater, J. and J. Delescaille (1989). How easy is collision search? application to des. In *Eurocrypt 1989, Proceedings*, Volume 434 of *Lecture Notes in Computer Science*, pp. 429–434. Springer.

Quisquater, J. and F. Standaert (2005). Exhaustive key search of the des: Updates and refinements. In *SHARCS 2005, Proceedings*.

Quisquater, J., F. Standaert, G. Rouvroy, J. David, and J. Legat (2002). A cryptanalytic time-memory tradeoff: First fpga implementation. In *FPL 2002, Proceedings*, Volume 2438 of *Lecture Notes in Computer Science*, pp. 780–789. Springer.

Shimoyama, T., M. Takenaka, and T. Koshiba (2002). Multiple linear cryptanalysis of a reduced round rc6. In *FSE 2002, Proceedings*, Volume 2365 of *Lecture Notes in Computer Science*, pp. 76–88. Springer.

Shimoyama, T., M. Takeuchi, and J. Hayakawa (2002). Correlation attack to the block cipher rc5 and simplified variants of rc6. In *3rd AES Candidate Conference*.

Wagner, D. (1999). The boomerang attack. In *FSE 1999, Proceedings*, Volume 1636 of *Lecture Notes in Computer Science*, pp. 156–170. Springer.

Wiener, M. (1996). Efficient des key search. In *Crypto 1993 (rump session presentation)*. Reprint in Practical Cryptography for Data Internetworks, William Stallings editor,IEEE Computer Society Press, pp. 31-79, 1996.

Wiener, M. (2004). The full cost of cryptanalytic attacks. *Journal of Cryptology 17*(2), 105–124.

Sourav Mukhopadhyay *completed his B.Sc (Honours in Mathematics) in 1997 from University of Calcutta, India. He has done M.Stat (in statistics) and M.Tech (in computer science) from Indian Statistical Institute, India, in 1999 and 2001 respectively. He received his Ph.D. degree in the area of Cryptology (Computer Science) from Indian Statistical Institute, India in 2007. Currently, he is working as a full time post-doctoral research fellow and part time Lecturer with School of Electronic Engineering, Dublin City University, Ireland. His research and teaching interests include network security, cryptology, mathematics, statistics and computer science*

Palash Sarkar *received his Bachelor of Electronics and Telecommunication Engineering degree in the year 1991 from Jadavpur University, Kolkata and Master of Technology in Computer Science in the year 1993 from Indian Statistical Institute, Kolkata. He completed his Ph.D. from Indian Statistical Institute in 1999. Since June 2005 he has been a professor at Indian Statistical Instute. His research interests include cryptology, discrete mathematics and computer science.*