PET SNAKE: A Special Purpose Architecture to Implement an Algebraic Attack in Hardware

Willi Geiselmann[1], Kenneth Matheis[2], and Rainer Steinwandt[2]

[1]Institut für Kryptographie und Sicherheit, Fakultät für Informatik,
Universität Karlsruhe (TH), Am Fasanengarten 5, 76128 Karlsruhe, Germany, geiselma@ira.uka.de
[2]Department of Mathematical Sciences, Florida Atlantic University, 777 Glades Road, Boca Raton, FL 33431,
{rsteinwa, kmatheis}@fau.edu

Abstract. In [19] Raddum and Semaev propose a technique to solve systems of polynomial equations over $\mathbb{F}_2$ as occurring in algebraic attacks on block ciphers. This approach is known as *MRHS*, and we present a special purpose architecture to implement MRHS in a dedicated hardware device. Our preliminary performance analysis of this **P**arallel **E**limination **T**echnique **S**upporting **N**ice **A**lgebraic **K**ey **E**limination shows that the use of ASICs seems to enable significant performance gains over a software implementation of MRHS. The main parts of the proposed architecture are scalable, the limiting factor being mainly the available bandwidth for interchip communication. Our focus is on a design choice that can be implemented within the limits of available fab technology. The proposed design can be expected to offer a running time improvement in the order of several magnitudes over a software implementation.

We do not make any claims about the practical feasibility of an attack against ciphers like AES or PRESENT with our design, however, as we do not see the necessary theoretical tools to be available: deriving reliable running time estimates for an algebraic attack with MRHS when being applied to a full-round version of block ciphers like AES or PRESENT is still an open problem.

Keywords: block cipher, algebraic attack, cryptanalytic hardware, MRHS

1 Introduction

Algebraic attacks have become an important cryptanalytic tool, and the security of major cryptographic algorithms relies on the infeasibility of solving certain systems of polynomial equations. Popular approaches for dealing with such systems of equations are based on the use of Gröbner basis techniques and SAT-solvers. Adding to the toolbox of algebraic cryptanalysis, in [19] Raddum and Semaev propose a technique known as *MRHS* (**M**ultiple **R**ight **H**and **S**ides) to handle polynomial systems of equations over $\mathbb{F}_2$. A full running time analysis of this algorithm is to the best of our knowledge not available, but the observed performance in software seems quite favorable, and in comparison to algebraic attacks involving the computation of a Gröbner basis, the required amount of memory seems easier to predict. Given arbitrarily large amounts of memory, MRHS should in principle be able to solve large systems of equations, but this is obviously not practical. Consequently, the hardware architecture we propose builds on an adaption of MRHS where the amount of memory is fixed. The specific design choices made are motivated by the limits of currently available fab technology, and the scalability of major components should facilitate the construction of small prototypes with technology that is available at moderate cost.

Our contribution. We propose an ASIC design for implementing MRHS, which according to our analysis enables significant performance gains compared to an MRHS implementation in software. Owing to the modular design and scalability, we think the proposed architecture to be of considerable interest when trying to mount algebraic attacks on relevant block ciphers. Building on a

45 nm manufacturing process, already a moderately sized network of chips of standard size seems capable of coping with rather non-trivial systems of equations. Our architecture is certainly far from optimal, and we hope that the promising results obtained so far stimulate further research along this line. Certain components of the architecture, specifically those for row reduction and multiplication of matrices over $\mathbb{F}_2$, might be of independent interest.

Related work. A first (unpublished) proposal for using dedicated hardware to implement MRHS has been developed by Semaev in 2007, and, after modifications, recently been published in [20, 21]. The architecture described below has been developed independently and uses a very different approach. The use of special hardware for attacking a specific symmetric cipher has been proposed in [3]. In addition, numerous special purpose architectures for cryptanalytic purposes have been devised and discussed in the research literature—some prominent examples being TWINKLE [22, 14], TWIRL [23] and their successors [9, 12] for factoring integers, or Deep Crack [8] and COPACOBANA [13] for attacking DES. As linear algebra over $\mathbb{F}_2$ plays an essential role in MRHS, it comes to no surprise that our design benefits from available work related to the Number Field Sieve: For the row reduction over $\mathbb{F}_2$ we modify the linear algebra design SMITH of Bogdanov et al. [5, 6] to enable a more efficient handling of sparse matrices as occurring in the context of MRHS. (Note that SMITH has enjoyed previous success in [3].) The resulting JONES (**J**ustifiable **O**ptimization **N**eatly **E**nhancing **S**MITH) device might be of independent interest for other applications involving sparse matrices over $\mathbb{F}_2$.

The overall data flow in our architecture is remotely reminiscent of the systolic linear algebra design in [10], a main difference being the emphasis on a two-dimensional data flow. Two-dimensional data flows are well-known from special purpose designs for the Number Field Sieve, like [2, 15, 11], but the organization of the data flow in the new design is quite different and explains the choice of the acronym PET SNAKE for our architecture.

Structure of the paper. We start with a brief discussion of MRHS where we detail the variant of the algorithm underlying our proposal. Section 3 gives a description of the overall architecture we use. The overall architecture uses several identical copies of a *main processing unit* whose various components are explained in Section 4. Further details on the individual components are given in the appendix. Finally, Section 5 analyzes the expected performance of the complete device, comparing it with a software implementation of MRHS.

2 Preliminaries: Multiple Right Hand Sides (MRHS)

For a detailed discussion of MRHS, we refer to Raddum and Semaev's work [19]. Here we restrict to an informal review of those aspects of the MRHS technique which are needed to explain the proposed hardware architecture. In particular, we do not discuss how to set up an MRHS system of linear equations to mount an algebraic attack on a block cipher like AES [16] and refer to [19, Section 6] for more details (cf. also [17]).

2.1 Basic Terminology

For our purposes, all matrices and vectors are assumed to have entries from $\mathbb{F}_2$, and it is helpful to fix some terminology:

Let $x := (x_1, \ldots, x_n)^{\mathrm{t}}$ be a column vector consisting of n Boolean variables, A a $k \times n$ matrix of rank k, and $b_1, b_2, \ldots, b_s$ column vectors of length k. An equation

$$Ax = b_1, b_2, \ldots, b_s \tag{1}$$

is called an *MRHS system of linear equations* with *right hand sides* $b_1, b_2, \ldots, b_s$. A *solution* to (1) is a vector in $\mathbb{F}_2^n$ satisfying one of the particular linear equation systems $Ax = b_i$. The set of *all solutions to* (1) is the union of solutions to the individual linear systems $Ax = b_i$ $(i = 1, \ldots, s)$. In an effort to manipulate the data contained in the above column vectors b_i, we write them side-by-side to form a matrix L and rewrite Equation (1) as $Ax = [L]$. The brackets around L emphasize that we are not working with a regular equation of matrices, and instead of the term *MRHS system of linear equations* the term *symbol* is often used.

Given a system of symbols

$$\begin{aligned} S_1 &: A_1 x = [L_1] \\ &\vdots \\ S_m &: A_m x = [L_m] \end{aligned}, \tag{2}$$

by a *solution to such a system* we mean a vector in $\mathbb{F}_2^n$ satisfying all of the underlying m MRHS systems of linear equations (where $x = (x_1, \ldots, x_n)$). The goal of the algorithm discussed next, and consequently of the PET SNAKE design below, is to find all solutions of (2).

2.2 Solving a System of Symbols

There are two main steps, commonly referred to as *agreeing* and *gluing*. The proposed PET SNAKE architecture tries to make use of similarities between these two algorithmic building blocks for reusing hardware components—therewith reducing the area complexity of the design.

Agreeing of symbols The basic approach is to remove some of the columns b in a right hand side L_i, if no one solution of $A_i x = b$ can be a solution to the System (2). The mechanism by which this is achieved is pairwise *agreeing* of symbols. Namely, let $S_i : A_i x = [L_i]$ and $S_j : A_j x = [L_j]$ be two symbols. Then S_i and S_j *agree* if for every $b \in L_i$, there exists a $b' \in L_j$ such that the linear system

$$\begin{pmatrix} A_i \\ A_j \end{pmatrix} x = \begin{pmatrix} b \\ b' \end{pmatrix} \tag{3}$$

is consistent, and, vice versa, for each $b' \in L_j$ there exists a $b \in L_i$ such that (3) is consistent.

When S_i and S_j do not agree, one removes those columns b from L_i for which the linear system $A_i x = b$ is inconsistent with $A_j x = [L_j]$. Dually, those columns b' from L_j are removed, for which $A_j x = b'$ is inconsistent with $A_i x = [L_i]$. Different strategies can be used for this approach, and for the design of PET SNAKE we follow the technique in Figure 1 (see [19, Section 3]) and realize it with a specialized hardware architecture.

It is important to note that if two symbols S_h and S_i agree, but S_i and S_j disagree, columns may be deleted in one or both of L_i and L_j. After this happens, it is possible for S_h to disagree with either of the modified symbols, and so S_h will have to be *re-agreed* with them. During that agreement, columns from L_h may have to be deleted, and so on. In this manner, a chain reaction of column deletions may occur. Hence, in order to ensure that a system of symbols gets to a pairwise-agreed state, in PET SNAKE we perform the *Agreeing1 Algorithm* in Figure 2 (see [19, Section 3.1]).

1. Produce a nonsingular transform matrix $U = U_{ij}$ of size $t \times t$ such that the product UA is a matrix with zeroes in its last $r = r_{ij}$ rows and of rank $t - r$. If $r = 0$, the symbols agree.
2. If $r > 0$, then compute the matrices UT_{ij} and UT_{ji}. Let Pr_{ij} denote the set of of UT_{ij}-column projections to the last r coordinates. If $Pr_{ij} = Pr_{ji}$, the symbols agree.
3. If $Pr_{ij} \neq Pr_{ji}$, first remove all columns from L_i whose UT_{ij}-associated column is such that its last-r-coordinate projection is not found in Pr_{ji}. Name the resulting matrix L'_i. Then similarly remove columns from L_j and name the resulting matrix L'_j. The symbols $A_i x = [L'_i]$ and $A_j x = [L'_j]$ agree.

Fig. 1. Agreeing two symbols $A_i x = [L_i]$ and $A_j x = [L_j]$, where $L_\eta \in \mathbb{F}_2^{k_\eta \times s_\eta}$. Here $A := \binom{A_i}{A_j}$ is the vertical concatenation of A_i and A_j, i. e., A has $t := k_i + k_j$ rows. Similarly $T_{ij} := \binom{L_i}{0}$ and $T_{ji} := \binom{0}{L_j}$ have t rows each.

While the symbols in a System (2) do not pairwise agree,

1. find S_i and S_j which do not agree
2. agree S_i and S_j with the agreeing procedure in Figure 1.

Fig. 2. Agreeing1 Algorithm.

Gluing of symbols After a system of symbols is in a pairwise-agreed state, we may choose to glue some symbols. The *gluing* of two symbols $S_i : A_i x = [L_i]$ and $S_j : A_j x = [L_j]$ is a new symbol $Bx = [L]$ whose set of solutions is the set of common solutions to $A_i x = [L_i]$ and $A_j x = [L_j]$. Once this new symbol is formed, it is inserted into the system and the two symbols S_i and S_j which formed it are no longer necessary and hence removed from the system. Obtaining the matrix B is easy: with the notation in Figure 1, B is just the submatrix of UA in its last $t-r$ nonzero rows. The matrix L has $t-r$ rows and the columns are formed by adding one column from UT_{ij} to one column from UT_{ji}. More specifically, we add a column from UT_{ij} and one from UT_{ji}, if they have the same projection to the last r coordinates. Reducing the sum to its first $t-r$ coordinates yields a column of L, and forming all such matching pairs yields the complete matrix L. Gluing two matrices L_i, L_j of width s_i and s_j may result in an L with as many as $s_i \cdot s_j$ columns. Consequently, we may not be able to afford to actually compute certain glues, and instead restrict to gluing only pairs of symbols where the number of columns in the resulting symbol does not exceed a certain *threshold*.

Once several pairs of symbols have been glued, the resulting system will usually not be in a pairwise-agreed state, so the Agreeing1 Algorithm in Figure 2 can be run again, initiating another round af agreeing and gluing. The eventual goal of successive agreements and gluings is to obtain a system of symbols consisting only of a single symbol.

Guessing Variables Owing to the chosen threshold, it may happen that a system is in a pairwise-agreed state and no pair of symbols can be glued anymore. In such a situation, one is forced to guess a value of a variable. Before a guess is committed, the system of symbols—to which we will refer as *state*—is stored. Then the guess is performed by constructing a new symbol whose A part is one row of all zeroes except for a single 1 in the position of the guessed variable, and whose L part is a single value, either 0 or 1, depending on the value of the guess. Such a symbol is inserted into the system, and then pairwise agreeing and gluing continues as normal. If after some steps the state, again, does not allow any further gluings, the state is again saved and another guess is committed.

Of course it is possible that in this process a guess for a variable is incorrect. This discovery manifests in the following manner: during the agreement of two symbols, all right hand sides of at least one of the symbols get removed. When this happens, the state must be rolled back to a previous state, and a different guess must be made. The practice of guessing variables, then, follows something akin to a depth-first search.

2.3 Implementation Choices

Raddum and Semaev [19] offer additional ideas to reduce the system of symbols to one symbol, including the building of an equation symbol. Such a symbol is obtained from computing *URHS* (**U**nique **R**ight **H**and **S**ide) equations. In PET SNAKE we do not provide such an improvement so far, but when considering further optimization such an option could certainly be considered.

For gluing symbols, we chose our threshold to be 2^{20} right hand sides. This seems a nice balance between the upper limits of software implementations and the upper limits of current hardware storage abilities. In light of the multiplicative nature of the growth of right hand sides during gluing, giving one or two more powers of 2 to the threshold does not seem to readily contribute to a significantly reduced running time.

3 Overall Architecture

A complete PET SNAKE architecture consists of a several interconnected boards with each board hosting several ***M**ain **P**rocessing **U**nits* (MPUs). Each MPU is comprised of a small group of chips wired in a particular way, and there are $p \times p$ such MPUs placed in a grid across the individual boards, where $p = 2^\lambda$ is a power of 2. Subsequently we use $\lambda = 5$, yielding a total of $2^5 \cdot 2^5 = 1024$ MPUs. Each MPU can communicate with its north, south, east, and west neighbor MPUs (with no wraparound).

For directing the action of the p^2 MPUs, a single ***M**aster **C**ontrol **P**rocessor* (MCP) is used. The MCP will make most of the decisions regarding which symbols to send where, which symbols to glue, and when to guess a variable. The MCP, which sits in a north corner, has *agents* which sit at the north end of the board, one per column. Each agent has a southbound bus that connects to each MPU in that column via 'hops' between MPUs, so each off-chip part of the bus is short. Each agent communicates to the MCP horizontally via 'hops' between agents. Figure 6 in Appendix A gives a schematic view of the overall architecture.

Remark 1. The proposed architecture scales within reasonable limits, and depending on the resources available, parameter values different from the choice we focus on may well be considered. For instance, reducing the number of MPUs to $p^2 = 2^8$ could be of interest. The specific parameter choices on which we focus aim at systems of symbols as occurring in an algebraic attack on a modern block cipher like AES [16] or PRESENT [4]. For AES, the initial system of symbols will consist of less than 400, and for PRESENT of less than 600 symbols. More generally, the handling of systems with no more than 2^{12} symbols still seems within the reach of PET SNAKE.

3.1 Initialization

The initial system of n symbols is derived from a particular known (plaintext, ciphertext)-pair, and a solution to the system of symbols yields a secret key for the attacked symmetric cipher that

is consistent with the particular (plaintext, ciphertext)-pair. The symbols are loaded onto the p^2 MPUs as evenly as possible. Let g be the number of symbols stored in each MPU—should the symbol count not be evenly divisible by the number of MPUs, we imagine empty symbols to fill in the gaps. Now imagine labelling each symbol in each MPU with a number in $\{1, \ldots, g\}$. We call all symbols labelled with the same number a *snake*. Hence we have g snakes. If at this point $g = 1$, we halve the number of MPUs to use, redistribute the symbols to this half, and try again; we continue this process until $g = 2$. The collection of MPUs now occupied with symbols is called the *active area* for this computation. Any inactive MPUs will be taken advantage of with parallelism, discussed later.

The MCP determines a Hamiltonian cycle through all MPUs, i. e., a path through all the MPUs such that one can move from one MPU to one of its neighbors in a closed circuit, without visiting the same MPU twice. The MCP will do the same for smaller groups of MPUs: $p \times \frac{p}{2}$, $\frac{p}{2} \times \frac{p}{2}$, $\frac{p}{2} \times \frac{p}{4}$, etc.—all the way down to 2×1. This data can be hardwired into the MCP, and we may assume that the MCP knows a path for each possible size active area.

3.2 Processing of Symbols

During a computation, it may happen that the symbol count n drops below the number of MPUs used to process them. If this happens, we move the symbols so that only half of the MPUs will be occupied with symbols. (This guarantees $g = 2$.) The active area is then halved. Any inactive MPUs will be taken advantage of with parallelism. Hence, at all points in the process, if g is not a power of 2, it will proceed as if g were the next highest power of 2 for board divisibility purposes. The overall algorithm run by PET SNAKE is summarized in Figure 3, and we discuss the agreement and glue phases separately.

1. Check to see if there is only one symbol. If so, we terminate, for all the keys are found.
2. Enter the agreement phase:
   - Each symbol is agreed to each other symbol until all symbols are pairwise-agreed.
   - If, in the agreement phase, we get a symbol whose L-matrix got all its columns deleted, then the current guess of the key variables is incorrect, so we roll back the symbols to a good state and go to (4).
3. Enter the glue phase:
   - If no two symbols can be glued such that the result's L matrix has no more than 2^{20} columns, save the state (that is, all the symbols) in the MPUs and then go to (4).
   - If necessary, move symbols so that any given pair of symbols to be glued appear in the same MPU. Different MPUs can be used for different pairs.
   - Pairwise glue the symbols whose resultant's L matrix has no more than 2^{20} columns. Delete the symbols which contributed to each glue.
   - If necessary, move symbols among the MPUs so that they have the same number of symbols. If there are less symbols than MPUs, move the symbols so that they only occupy half the MPUs. This halves the active area.
   - Go to (1).
4. Enter the guess phase: The MCP determines the next variable to guess and its value, and then directs an MPU to create a symbol to reflect this guess. Go to (2).

Fig. 3. Overall algorithm run by PET SNAKE.

3.3 PET SNAKE's Agreement Phase

The majority of activity on the board will be during the agreement phase. This is broken down into k stages, where $k = \lceil \log_g n \rceil$.

First Stage. In the first stage, the entire active area is used. All but one snake (i. e., snakes 1 through $g-1$) stay put on the MPUs. On each MPU, the symbol in the *motile snake* (i. e., snake g) is agreed to every other symbol on that MPU. When the last such agreement is taking place, the MPU sends the motile snake's updated symbol (that is, with deletions incorporated) to the next MPU in the active area's path. Since this is happening simultaneously for all MPUs in the active area, each MPU gets the next symbol in the motile snake. This continues q times, where q is the number of MPUs in the active area. If a deletion has occurred somewhere in any of this process, the MCP records the affected symbol's number, but otherwise continues normally.

Now, snake g will be fixed, and snake $g-1$ will move. The only difference here is that symbols from snake $g-1$ will not need to be agreed with those from snake g since that agreement has already been performed. After q times, snakes g and $g-1$ will be fixed, but snake $g-2$ will move. And so on. If a deletion has occurred for any of the g snakes, the MCP moves the affected symbols into larger-numbered snakes (e. g., g, $g-1$) and moves unaffected symbols into smaller-numbered snakes. Often this is just a renumbering inside an MPU, so no movement happens in these cases. Then the first stage is repeated again, noting that if all the symbols in a lower-numbered snake have no deletions in the previous run, it is not required to become motile. If a deletion occurs, the MCP repeats the process of moving affected symbols and starting the stage again.

Second Stage. At this point, all snakes are agreed to all other snakes, but the symbols within each snake still need to be addressed. The active area is split up into g *stage areas*, each with q/g MPUs. For each $1 \le j \le g$, symbols from snake j move to stage area j. After this move is complete, we relabel each symbol in each MPU so that different snakes are formed, but the snakes only move in their given stage area. Hence, each snake is $1/g$ the size it used to be. Now, the same process is performed as in the first stage, but with smaller snakes and smaller paths.

If a deletion has been recorded in this stage, the stage is allowed to complete, but not recur nor go into the next stage. Then the affected symbols (from *all* stage areas) are grouped together into one (or possibly more) q-sized snakes with large snake numbers, they are moved into appropriate positions, and the first stage is entered again.

Subsequent Stages. If the second stage records no deletions, we continue this process of dividing the snakes and the stage areas by g until the stage area is one MPU. (Deletions found in any subsequent stage are handled the same way as described in the second stage.) At the last stage, the g symbols comprise g snakes of size 1 each, and so they are simply agreed to each other inside that MPU.

Time Estimate. To get a handle on a time estimate, assuming that no deletions will occur, we note that per stage there are $g(g-1)/2$ agreements per MPU, and this happens q times in the first stage, q/g in the second, and so forth, up to 1 in the last. Since $g = n/q$, adding up the costs we have

$$\sum_{i=0}^{k-1} \frac{g(g-1)}{2} \cdot \frac{q}{g^i} = \frac{g(g-1)}{2} \cdot \frac{n}{g} \cdot \left(\frac{1 - \frac{1}{g^k}}{1 - \frac{1}{g}} \right) = \frac{(g-1)n}{2} \cdot \left(\frac{\frac{n-1}{n}}{\frac{g-1}{g}} \right) = \frac{g(n-1)}{2}$$

total agreements. Since we try to arrange things so that g is 2 as often as possible, this translates into $n-1$ agreements in these cases.

What is not included so far is the time of moving symbols between stages. Let the active area have dimensions $q_1 \times q_2 = q$ where $q_1 \leq q_2$, and suppose g is 2. After the first stage, a symbol moves along the longer dimension, but halfway so that it can find its new position. Another symbol from that position must get to where the first started, so they both must use those directions. This will introduce a factor two slowdown in all movement calculations. Hence, after the first stage it takes $2 \cdot \left(\frac{q_2}{2}\right)$ moves to get the symbols into their new positions, and the stage area then has dimensions $q_1 \times \frac{q_2}{2}$. We alternate which dimension we travel on in each stage, so the next stage cost is $2\left(\frac{q_1}{2}\right)$. Then $2 \cdot \left(\frac{q_2}{4}\right)$, then $2 \cdot \left(\frac{q_1}{4}\right)$, and so on. Presuming k is even, this gives a time estimate of

$$(q_1+q_2) \cdot \sum_{i=0}^{\frac{k}{2}-1} \frac{1}{2^i} = (q_1+q_2) \cdot \left(\frac{1-\left(\frac{1}{2}\right)^{k/2}}{1-\frac{1}{2}} \right) = 2 \cdot (q_1+q_2) \cdot \frac{2^{k/2}-1}{2^{k/2}} < 2 \cdot (q_1+q_2)$$

total moves for the whole agreement phase.

The situation for $g = 4$ is not as easy, since symbols have to move to different quadrants of the active area $q_1 \times q_2$. We observe that it must be the case that $q_1 = q_2$, since the only time we might have $g > 2$ is in the beginning, when we have the full board at our disposal.

Hence, we perform a sort of rotation, where each quadrant of symbols (one symbol per MPU per move) moves to the next clockwise (or counterclockwise) quadrant simultaneously. This is possible since all four directional buses of each MPU can be used simultaneously, and no directional bus needs to be used more than once at a time. After the first stage, in the first rotation the symbols whose target locations are in the diagonal quadrant move $\frac{q_1}{2}$ in one direction. In the second rotation, these same symbols move $\frac{q_1}{2}$ in the appropriate perpendicular direction to get to their target location. In the third rotation, symbols whose target quadrant are clockwise of them will move $\frac{q_1}{2}$ in that direction. The fourth rotation is similar to the third, but for counterclockwise-bound symbols. Thus, we have $4 \cdot \left(\frac{q_1}{2}\right) = 2 \cdot q_1$ moves for this stage. Subsequent stages are similar but the distance is half of the previous distance. Thus we have

$$\sum_{i=0}^{k-1} 2 \cdot \left(\frac{q_1}{2^i}\right) = 2 \cdot q_1 \cdot 2 \cdot \left(\frac{2^k-1}{2^k}\right) < 4q_1$$

total moves for the whole agreement phase.

Remark 2. The initial load's symbols will most likely have A parts whose 1s are in different positions, so any particular pair of symbols will likely be already agreed, so no deletions will occur. After the first glue, it is still likely no deletions will occur. After the second glue, however, things get less predictable, but by this point the symbol count will drop by a factor of 4. (In the case of AES, the threshold will take hold before the second glue, so we can only expect the symbol count to halve before guesses must be performed.)

After these initial turns, deletion prediction becomes much less predictable, and it is certainly possible to go through many agreement phases before considering a glue.

3.4 PET SNAKE's Glue Phase

Since the MCP knows which pairs of symbols will glue to produce a symbol with 2^{20} or less columns, it merely directs moves to get these pairs into MPUs, and then the MPUs glue them in parallel. The number of moves needed is not completely predictable, but it is expected to be a small constant multiple of $q_1 + q_2$. The glue time, however, is in general higher than an agreement time. With $g = 2$, we only pay the glue time once, since each MPU will be gluing all gluable pairs in parallel with none waiting to be glued. With $g = 4$, we pay the glue time at most twice; in general, the glue time is paid at most $g/2$ times.

3.5 Parallelism

Once the active area becomes half the original board (or less), and a guess is required, the MCP considers performing a parallel computation on the inactive area. The MCP will make a guess for a key variable in one area, and make the opposite guess for the same key variable in the other. Then both areas will be considered active areas, but their computations will be completely separated.

4 Main Processing Unit

The MPU is a collection of seven chips comprising five functional units, each with its own responsibilities and behavior. We discuss each functional unit in turn: the traffic controller, the row reducer, the multiplier, the hash table, and the adder. Each functional unit is connected to a 2048-bit-wide bus called the *MPU bus*.

4.1 MPU Data Flow

We describe the sequence of events that will occur inside each MPU when it is agreeing and when it is gluing. The particular details of each component are discussed in that component's section below. Figure 7 gives an overview of how most of the components are interconnected. (The traffic controller sits on the north end of the MPU bus, directing traffic between it and other traffic controllers of other MPUs.) The high level order of operations during an agreement between two symbols S_i and S_j is as given in Figure 4.

The procedure for gluing two symbols S_i and S_j is slightly different and described in Figure 5.

Subsequently we discuss the individual components of an MPU. To translate the size of a functional unit from its gate count to its transistor count, Table 3 is used. To calculate the surface area consumed by each component, we use the numbers in Table 4, which are obtained by scaling the 90 nm numbers in [24, Table 3.1] down to 45 nm by a factor of 4 resp. the 130 nm numbers there down to 45 nm by a factor 8 (cf. also [15, Table 2]).

4.2 Traffic Controller

The traffic controller is a collection of four chips responsible for receiving symbol data from neighbor MPUs, storing it, and pushing it across the MPU bus if need be. After the results of various computations from other functional units are complete, the traffic controller will store or forward to a neighbor MPU those results, depending on what is currently being done. This is the only functional unit that is connected to other MPUs and the MCP, as well as the MPU bus. Details on the architecture of the traffic controller and how it operates are given in Appendix C.

1. A_i is sent across the MPU bus and the row reducer picks it up.
2. A_j is sent across the MPU bus and the row reducer picks it up.
3. The row reducer calculates both B and U.
4. The row reducer determines if r is 0. If $r = 0$, terminate with agreement signal. Otherwise,
5. The row reducer sends the left $\text{cols}(L_i)$ part of U across the MPU bus to the multiplier.
6. For each column c of L_i:
   - c is sent across the MPU bus and the multiplier picks it up.
   - The multiplier sends its r-part to the hash table.
   - The hash table stores an indicator that that r-part has been created.
7. The row reducer sends the right $\text{cols}(L_j)$ part of U across the MPU bus to the multiplier.
8. For each column d of L_j:
   - d is sent across the MPU bus and the multiplier picks it up.
   - The multiplier sends its r-part to the hash table.
   - If the r-part had been formed by L_i, the hash table stores an indicator for this.
   - If not, the hash table reports the column index of d across the MPU bus to be deleted.
9. For each entry in the hash table's buffer DRAM, if the entry is not found in the table itself, the column index is reported across the MPU bus to be deleted.
10. If no deletions have been recorded, the hash table sends the value of its glue counter across the MPU bus to the traffic controller.

Fig. 4. High level order of operations during an agreement.

4.3 Row Reducer

The row reducer is comprised of a chip named `A/U`, which is connected to the MPU bus. Each part of its name will refer to a separate processing area inside this chip. The row reducer has two responsibilities: compute a row-reduced version of A (i. e., the vertical concatenation of A_i and A_j when they are received), and compute the matrix U such that UA yields the row-reduced matrix that will appear in the `A` part. During a glue, the data stored in the `A` part will be sent back across the MPU bus. (This corresponds to B in the MRHS gluing algorithm.) During both agreeing and gluing, the data stored in the `U` part will be sent across the MPU bus to the multiplier. Details on the architecture of `A` and `U` and how it operate are given in Appendix D. The JONES element used in `A` and `U` builds on ideas from SMITH [5, 6] and may be of independent interest.

4.4 Multiplier

The multiplier occupies one part of a chip named `M/HT`. If the MPU is agreeing two symbols, the multiplier receives data from `A/U` and stores it in a processing area called `Ur`. If the MPU is gluing two symbols, the multiplier will also receive additional data from `A/U` and store it in a separate processing area called `Us`. It then receives the L-part of a symbol one column at a time, and multiplies it with the contents in `Ur` and (if gluing) `Us`. Once this multiplication is complete for the received L-column, the multiplier will send the result from `Ur` (called an *r-part*) to the hashtable. If gluing, it will also send the result from `Us` (called an *s-part*) to the adder across the MPU bus. Details on the architecture and working of the multiplier are discussed in Appendix E. Similarly like the row reducer, this architecture might be of independent interest.

4.5 Hash Table

The hash table is used in both PET SNAKE's agreeing and PET SNAKE's gluing phase, and it is designed to process one write query per clock cycle—similarly, for look-ups, one look-up query

1. A_i is sent across the MPU bus and the row reducer picks it up.
2. A_j is sent across the MPU bus and the row reducer picks it up.
3. The row reducer calculates both B and U, determines if r is 0, and sends B across the MPU bus to be stored.
4. The row reducer sends the left $\text{cols}(L_i)$ part of U across the MPU bus to the multiplier.
5. For each column c of L_i:
   - c is sent across the MPU bus and the multiplier picks it up.
   - The multiplier sends its s-part to the adder for storage.
   - If $r \neq 0$, the multiplier sends its r-part to the hash table, and the hash table stores the L_i column index that gave rise to the r-part.
6. The hash table re-examines its DRAM buffer, possibly sending pairs of data across the MPU bus to the adder.
7. The row reducer sends the right $\text{cols}(L_j)$ part of U across the MPU bus to the multiplier.
8. For each column d of L_j:
   - d is sent across the MPU bus and the multiplier picks it up.
   - The multiplier sends its s-part s to the adder for adding.
   - If $r \neq 0$, the multiplier sends its r-part to the hash table.
   - If $r \neq 0$, the hash table sends all indices from L_i that match the r-part across the MPU bus to the adder. For each such index i,
     - The s-part at index i is looked up in the adder.
     - The s-part is retrieved, added to s, and sent across the MPU bus.
   - If $r = 0$, the adder runs through all its contents. For each such index i,
     - The s-part at index i is looked up in the adder.
     - The s-part is retrieved, added to s, and sent across the MPU bus.

Fig. 5. High level order of operations during gluing.

per clock cycle can be coped with. Elements to be stored or looked up in the hash table are r-parts with a (zero padded) size of $r_{max} = 135$ bit, and the hash table is designed to store up to 2^{20} such r-parts. Details on the architecture and the inner working of the hash table are discussed in Appendix F.

Remark 3. Having no more than 2^{20} columns, identifying each column with a 135 bit hash value seems a safe choice: taking the hash values for being uniformly distributed, the probability that no collision occurs is $\geq \prod_{i=0}^{2^{20}-1}(1 - \frac{i}{2^{135}}) \geq 1 - 2^{-90}$.

4.6 Adder

The adder is comprised of its own chip, which is largely a memory storage device. The adder is only used during a glue. During a glue, while the columns of L_i are being processed, `M/HT` will send out s-parts across the MPU bus. These will be picked up by the adder and stored in a collection of 256 DRAMs. Later, for each column in L_j that is being processed, the adder first acquires an s-part and stores it in a separate row of flip-flops called the *adding register*. Then the hash table will send across the MPU bus either a series of indices in L_i that match to that particular L_j column (i. e., whose Pr_{ij} columns are the same), or a popularity number of the resulting r-part. In the first case, the adder will look up the indices in its DRAM collection. In the second case, it will use the popularity number to find indices in its own table, and look those up in its DRAM collection. The resulting s-parts are then added to the adding register, and the sum is sent back across the MPU bus. More details on the architecture and the internal working of the adder are given in Appendix G.

5 Performance Analysis

5.1 Total Chip Area and Cost

With the area estimates in Appendix C–G, the total chip area of the (seven) chips comprising one MPU computes to $1.1 + 0.84 + 2.4 + 4 \cdot 3.9 < 20$ cm^2. For a PET SNAKE architecture with $p^2 = 2^5 \times 2^5$ MPUs, this results into a total chip area of about 2.05 m^2. For actually placing the chips more space will be required to enable the necessary wiring, cooling, etc. Obviously this is a non-trivial size requirement, but it is important to note that none of the involved chips is larger than 3.9 cm^2, and the resulting device should be capable of hosting a system of symbols as needed to attack a modern block cipher like AES or PRESENT.

One MPU uses some 20 cm^2 of silicon. If we assume a 30 cm wafer to cost \$5000, the pure silicon for one MPU calculates to some \$150. If we apply a factor 4 for the full design, including the board and some safety margin, one MPU is about the price of one PC. Therefore we compare the performance of one MPU with one PC.

5.2 Timing and Comparison with Software

A more detailed discussion of timings and a comparison with software is given in Appendix H–K. Specifically, Appendix H presents a simplified model to analyze the running time in software, and in Appendix I software measurements of our MRHS implementation in C++ on an Intel E2180 processor (single core, 2 GHz) when working with 4 rounds of PRESENT are documented. For PET SNAKE we assume a 1 GHz clocking rate: with each component of our architecture having a gate depth of four or less, we believe such a clocking rate not to be implausible

The overall running time of MRHS is dominated by the time spent to agree symbols, and basing on our experiments with four rounds of PRESENT, a speed-up by a factor of $\gg 2000$ of PET SNAKE over our (reasonable optimized) software implementation is plausible. Actually, when looking at full round versions of AES or PRESENT, we expect symbols to be involved in the computation, where the performance advantage of PET SNAKE becomes more drastic. As documented in Appendix I, here expected improvement factors in agreeing timings might well be in the range of 5 digit factors. Thus, even more conservative PET SNAKE clocking rates than 1 GHz still can be expected to realize several magnitudes of improvement over software.

Another advantage of our device to a software implementation is the huge amount of fast memory available. On one MPU we have 4 GByte of DRAM and a very fast communication to the other MPUs. So the communication does not slow down the processing speed. A PC has no option but to write data to a secondary storage like a hard disk, for the complete task needs the handling of TBytes of data. This will add another significant slowdown. To cope with a cipher like AES or PRESENT, the only plausible option seems to use a cluster of PCs, but here the communication cost between these PCs will add another significant factor to the overall running time of the algorithm: A standard PC network is a factor 1000 slower than the connection between MPUs (cf. Appendix K), and the connection between the MPUs is busy nearly all the time. To cope with this dramatic slowdown in software, one would have to come up with a very elaborated network between the PCs.

6 Conclusion

In this paper we propose a dedicated hardware design to implement an algebraic attack, based on MRHS, against block ciphers. We think that our analysis gives ample evidence that PET SNAKE

is an architecture of significant cryptanalytic interest. Lacking a reliable running time estimate of an MRHS-based algebraic attack, we cannot give a reliable estimate on the running time of our design when being applied to a modern block cipher like AES or PRESENT. Notwithstanding this, the above discussion gives ample evidence that the practical feasibility of (MRHS-based) algebraic attacks can be improved significantly through the use of a dedicated hardware design: substantial performance improvements over software implementations can be achieved, and owing to the scalability of PET SNAKE, exploring small prototypes seems a plausible next step in research along this line. Some of the building blocks of PET SNAKE, like the JONES design for the linear algebra part, might be of independent interest.

References

1. Gregory V. Bard. *Algorithms for solving linear and polynomial systems of equations over finite fields with applications to cryptanalysis*. PhD thesis, University of Maryland at College Park, Applied Mathematics and Scientific Computation, 2007.
2. Daniel J. Bernstein. Circuits for Integer Factorization: a Proposal. At the time of writing available electronically at `http://cr.yp.to/papers/nfscircuit.pdf`, 2001.
3. Andrey Bogdanov, Thomas Eisenbarth, and Andy Rupp. A Hardware-Assisted Realtime Attack on A5/2 Without Precomputations. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems; CHES 2007 Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 394–412. Springer, 2007.
4. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J.B. Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES 2007,*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, Cryptographic Hardware and Embedded Systems – CHES 2007.
5. Andrey Bogdanov, Marius C. Mertens, Christof Paar, Jan Pelzl, and Andy Rupp. A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2). In *IEEE Symposium on Field-Programmable Custom Computing Machines — FCCM 2006, Napa, CA, USA*, 2006.
6. Andrey Bogdanov, Marius C. Mertens, Christof Paar, Jan Pelzl, and Andy Rupp. SMITH - A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2). In *2nd Workshop on Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2006*, 2006. Available at `http://www.crypto.ruhr-uni-bochum.de/imperia/md/content/texte/publications/conferences/sharcs2006_matrix.pdf`.
7. Wieb Bosma, John J. Cannon, and Catherine Playoust. The Magma Algebra System I: The User Language. *Journal of Symbolic Computation*, 24:235–265, 1997.
8. Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. OReilly & Associates, July 1998.
9. Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, and Colin Stahlke. SHARK: A Realizable Special Hardware Sieving Device for Factoring 1024-Bit Integers. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems; CHES 2005 Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 119–130. Springer, 2005.
10. Willi Geiselmann, Adi Shamir, Rainer Steinwandt, and Eran Tromer. Scalable Hardware for Sparse Systems of Linear Equations, with Applications to Integer Factorization. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems; CHES 2005 Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2005.
11. Willi Geiselmann and Rainer Steinwandt. Yet Another Sieving Device. In Tatsuaki Okamoto, editor, *Topics in Cryptology — CT-RSA 2004*, volume 2964 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 2004.
12. Willi Geiselmann and Rainer Steinwandt. Non-wafer-Scale Sieving Hardware for the NFS: Another Attempt to Cope with 1024-bit. In Moni Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Compute Science*, pages 466–481. Springer, 2007.
13. Tim Güneysu, Timo Kasper, Martin Novotný, Christof Paar, and Andy Rupp. Cryptanalysis with COPA-COBANA. *IEEE Transactions on Computers*, 75(11):1498–1513, November 2008.
14. Arjen K. Lenstra and Adi Shamir. Analysis and Optimization of the TWINKLE Factoring Device. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2000.

15. Arjen K. Lenstra, Adi Shamir, Jim Tomlinson, and Eran Tromer. Analysis of Bernstein's Factorization Circuit. In Yuliang Zheng, editor, *Advances in Cryptology — ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2002.
16. National Institute of Standards and Technology. *Federal Information Processing Standards Publication 197. Specification for the ADVANCED ENCRYPTION STANDARD (AES)*, November 2001.
17. Håvard Raddum. MRHS Equation Systems. In Ali Miri Carlisle Adams and Michael Wiener, editors, *Selected Areas in Cryptography – SAC 2007*, volume 4876 of *Lecture Notes in Computer Science*, pages 232–245. Springer-Verlag, 2007.
18. Håvard Raddum and Igor Semaev. Solving MRHS linear equations. Cryptology ePrint Archive, Report 2007/285, 2007. Available at `http://eprint.iacr.org/2007/285`.
19. Håvard Raddum and Igor Semaev. Solving Multiple Right Hand Sides linear equations. *Designs, Codes and Cryptography*, 49:147–160, 2008. Preprint available in [18].
20. Igor Semaev. Sparse Boolean equations and circuit lattices. Presentation at International Workshop on Coding and Cryptography WCC 09, Ullensvang (Norway), May 2009.
21. Igor Semaev. Sparse Boolean equations and circuit lattices. Cryptology ePrint Archive, Report 2009/252, 2009. Available at `http://eprint.iacr.org/2009/252`.
22. Adi Shamir. Factoring Large Numbers with the TWINKLE Device. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems. First International Workshop, CHES'99*, volume 1717 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 1999.
23. Adi Shamir and Eran Tromer. Factoring Large Numbers with the TWIRL Device. In Dan Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2003.
24. Eran Tromer. *Hardware-Based Cryptanalysis*. PhD thesis, Weizmann Institute of Science, May 2007. Available at `http://people.csail.mit.edu/tromer/phd-dissertation/`.

A PET SNAKE Layout (High Level)

The following figure gives a schematic view on the overall architecture of PET SNAKE.

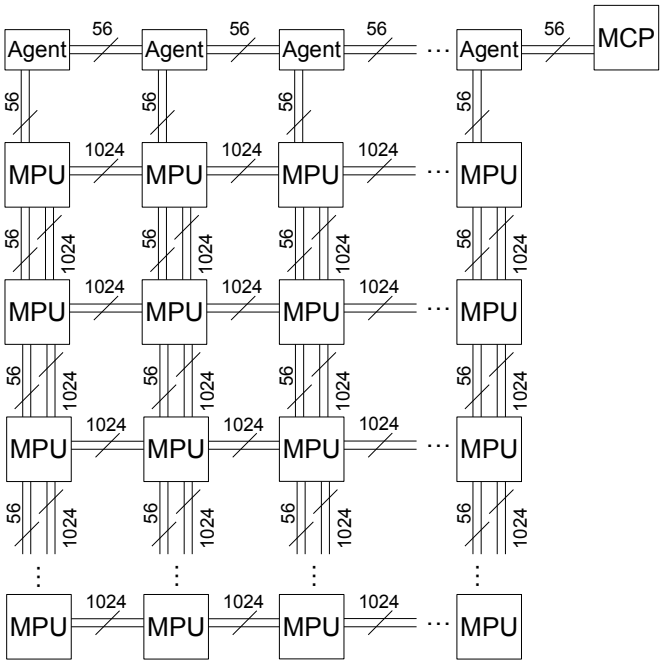

Fig. 6. Overall architecture of PET SNAKE.

B MPU Busing

The following figure gives a high level view on the interconnections in an MPU.

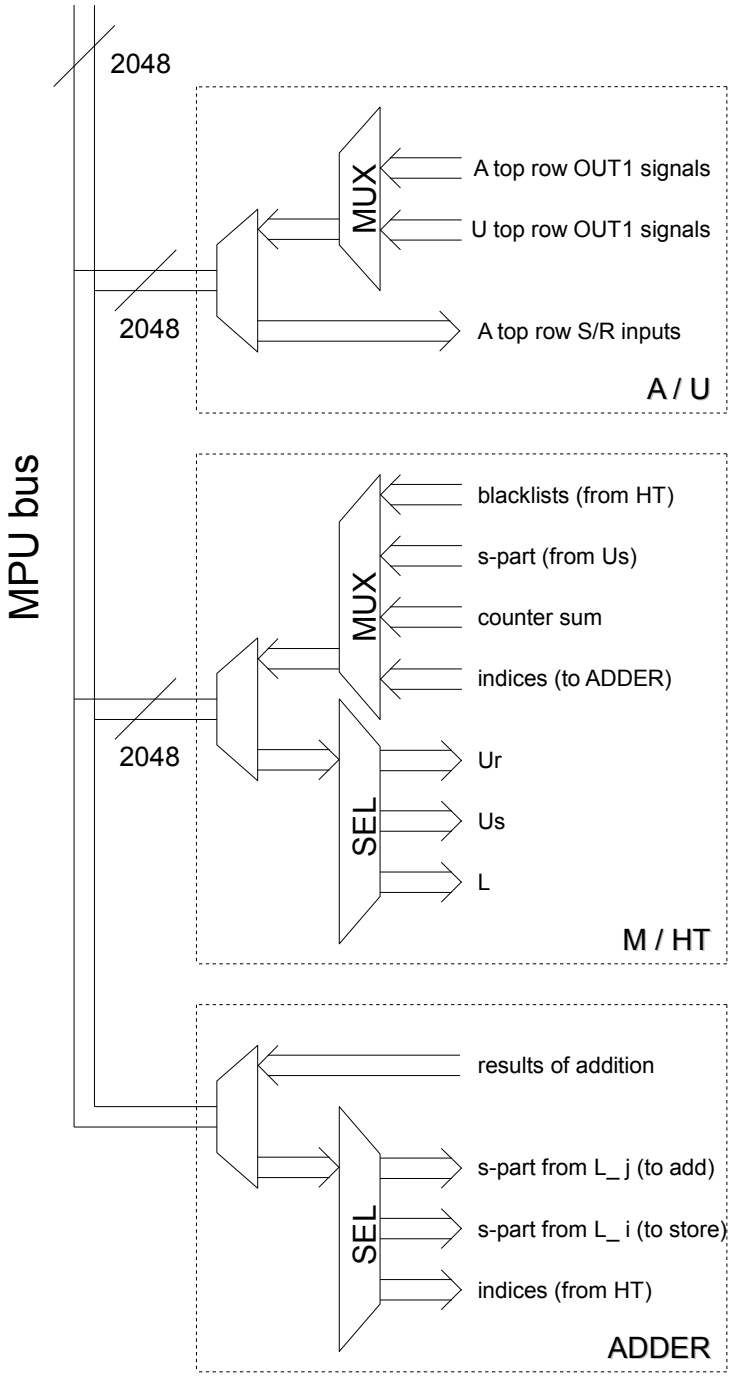

Fig. 7. MPU Busing Diagram (High Level).

C Traffic Controller

Architecture. Each of the four chips has 256 pins in each cardinal direction, plus 512 pins for the MPU bus. This means that, together, buses between MPUs are 1024 wide, but the MPU bus is 2048 wide. Hence, each of the four chips is responsible for a quarter of each symbol the MPU stores. In addition, there are 56 pins in and 56 pins out to pass data across the MCP bus. It is through this bus that the MCP will direct the actions of the symbols and the MPU.

Each chip has ten DRAM areas. Five are considered the 'active' DRAMs: one for receiving a new symbol in the motile snake, one for sending a symbol in the motile snake, and three for storing the currently-fixed $g - 1$ symbols. These DRAMs are 65 MByte each, enough to store a quarter of a full symbol. The other five are the 'passive' DRAMs used to store a state before a guess is committed. Each is 234 MByte (enough for 18 quarters of symbols when combined) and connected to a corresponding 'active' DRAM. In addition, for each active DRAM there is a 1024×1024 flip-flop grid with two decoders used to record deletions. Finally, there is some control logic to handle MCP commands.

Just in Time Process. These chips operate on a 'just in time' basis during the agreement phase, specifically during last agreement that must be performed before the snake must move. Label the two symbols that must be agreed S and T, where T is part of the motile snake, and suppose they are of roughly equal size. The MPU will process T second, that is, T provides the A_j and L_j in the MPU data flow description.

Just after the last deletion for L_j (if any) is sent from the hash table, the traffic controller begins to transmit A_j and L_j (taking care not to send the columns marked for deletion) to the next MPU in the path. During this time, it is receiving the A and L matrices from the next symbol in this snake; label them A_y and L_y. Also during this time, the MPU is also processing the columns from L_i the second time, getting deletions for that symbol. These three actions are done in parallel since they are using different buses and different active DRAMs. By the time L_i has gotten all its marks for deletion, at a minimum only half of the data for the incoming/outgoing symbols will be received/sent, since the transmission buses are only 1024 wide in either direction. Since A parts get transmitted before L parts, A_y will be received.

Now, the traffic controller will continue receiving/sending symbol data while it sends the A part of a fixed snake (call it A_x; if $g = 2$, $x = i$) to the row reducer to begin agreement with the incoming symbol. Once done, A_y will be sent to the row reducer (and transmissions of symbols will temporarily be halted). Then the row reducer will do its processing (at which point transmissions of symbols will resume), followed by L_x being sent to the multiplier. By the time L_x is done being processed (the first time), L_y will be completely received, and so L_y is now sent to the multiplier for processing.

In situations where the symbols have mismatching sizes, there must naturally be halts in processing until the data is received. Time calculations are performed supposing all symbols are full maximal size, so actual performance can be better than these calculations.

Deletion Handling. Each active DRAM has attached to it a flip-flop grid to register deletions. When a column index to be deleted is received on the MPU bus, this grid will handle it, leaving the DRAM to continue sending columns. The handling works as follows: each flip-flop's value is determined by a different AND gate. The most significant 10 bits of the received index are decoded

into 1024 horizontal lines out. Each such line attaches to a row of 1024 of these AND gates. The least significant 10 bits of the index are also decoded into 1024 vertical lines out. Each such line attaches to a column of 1024 of these AND gates. Hence, each index corresponds to exactly one flip-flop, and the received index's flip-flop will be set to 1.

When it is time to transmit the L part of a symbol, the flip-flop grid shifts its values in the top row (cyclically) over so that there is a 0 or 1 corresponding to the column to be sent. If the upper-right flip-flop has 1, the column is not sent; else, it is. After the send (or no-send), the row shifts again. After 1024 shifts, the whole grid performs a cyclical shift up.

Note that the two grids for motile snake symbols are reset before the new symbol is agreed, but the grids for stationary snakes are not. Hence, their grids will cumulatively store deletion marks until it is time for them to move for the next agreement stage. In addition their values will be used in determining if a column will be sent to the multiplier when it is called for in future agreements in the current stage. (This doesn't impact timing concerns, but it does make the hash table's life easier.) After a full agreement phase has been completed, all symbols are moved one MPU so that deletions are incorporated into DRAM. Of course, if in the last agreement stage a deletion has occurred, the whole agreement phase starts over.

Area Calculation. Each chip has $((65 + 234) \times 5)$ MByte of DRAM, yielding 1495 MByte, or 3.13 cm^2. In addition there are 5×2^{20} flip-flops, for 0.22 cm^2 (since we manufacture these chips using the 45 nm DRAM process). Each decoder can be realized with two 3-8 decoders (outs inverted), one 4-16 active low decoder (outs inverted), and 1024 3-input NOR gates (with latches at the inputs so as to form a two-stage pipeline) for a total of < 0.001 cm^2 for all decoders. The control logic will not grow above 0.50 cm^2, and a complete traffic controller thus fits onto 3.9 cm^2.

D Row Reducer

Architecture of `A`. The `A` part's main workhorse is 4096×2048 JONES elements connected in roughly the same way that SMITH elements usually are. Readers familiar with SMITH in [5] will undoubtedly recognize the similarities; indeed, JONES is meant as an improvement to SMITH to handle sparse matrices. This improvement is helpful because in the early and middle stages of processing, many symbols' A-parts are sparse, and the vertical concatenation of such things will still be sparse. The connections between JONES elements are essentially those of SMITH, taking care to send the `OUT3` signal to the leftward element's `IN3` input, wrapping cyclically around.

We see from Figure 8 that a JONES element is similar to a SMITH element, but there are three additional lines added: `IN3`, `OUT3`, and `live_col`. A row of 2048 flip-flops called the *LC row* will be set so that a given flip-flop holds 1 only when there is a 1 in that column of the JONES matrix, and 0 otherwise. (`U` does not require these flip-flops; `live_col` will be set in a different manner in them.) The leading LC flip-flop, along with some control logic, will be attached to every element's `live_col` signal. If there is a 1 in the leading column, `live_col` will be 1 for the entire array, and JONES processes its data exactly as SMITH does. However, if there is not a 1 in the leading column, `live_col` is 0, and we cyclically shift the entire matrix leftward.

To bring about this change in behavior, the circuitry for SMITH was modified. The logical changes can be seen in Figure 9. The gate depths for each path leading to the flip-flop are shorter than those in SMITH, so JONES can withstand faster clocking rates.

Turning our attention to some helper control circuits, we remark that `A`'s LC row will be initialized after both A_i and A_j are done being loaded into the elements in the following way: the

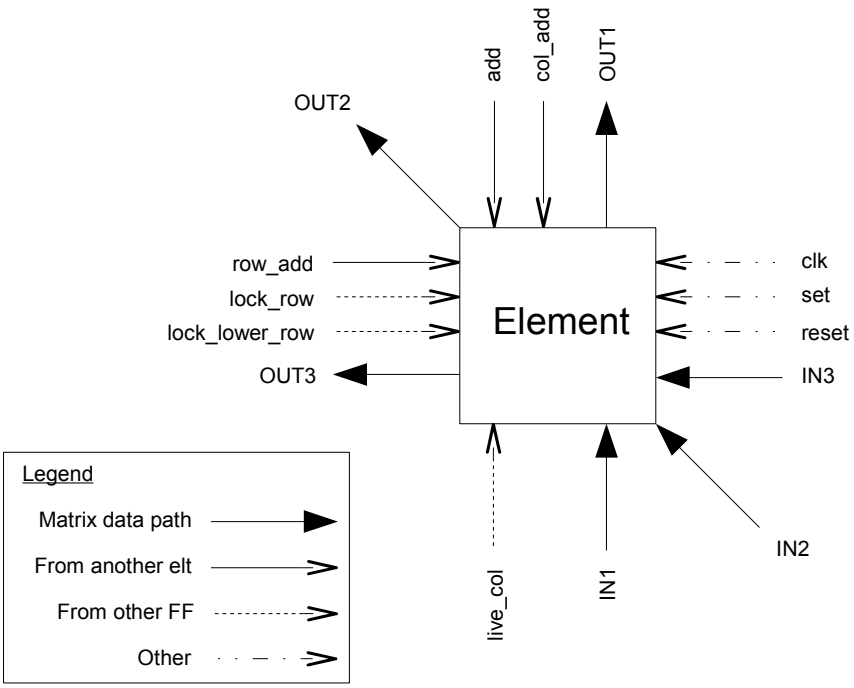

Fig. 8. JONES Element (High Level).

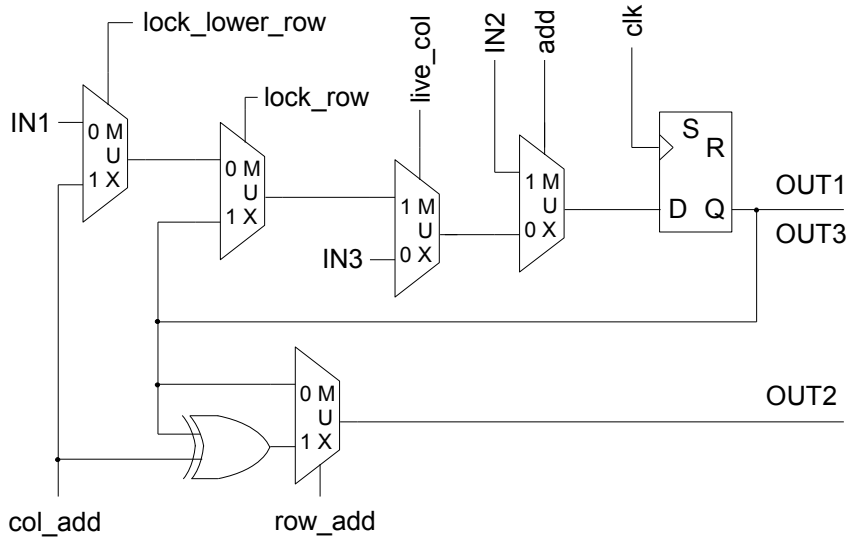

Fig. 9. JONES Element (Low Level).

LC row will be initialized to all zeroes, but OR gates will hang on them and the input lines (one per flip-flop). As A_i and A_j are loading, the LC row flip-flop for that column will constantly be updated with the cumulative OR of the current column value and the current flip-flop value. Hence, when the data load is complete and before processing begins, the LC row will be populated as described earlier. The LC row will also be wired to cyclically left shift so that, during JONES processing, such a cyclic left shift will occur upon any add or left-shift in the JONES elements.

`A` also uses a column of 4097 flip-flops called the *LR column*, which will feed the `lock_row` and `lock_lower_row` inputs. When a new matrix is to be row-reduced, this column is initialized to all zeroes, with a 1 in the bottom position. A given flip-flop will feed its value to all `lock_lower_row` inputs in its row, and it will feed its value to all `lock_row` inputs in the row below it. As the elements perform their processing, when they detect that an add must be performed, the LR column will shift upwards, inserting a 1 in the bottommost position.

Architecture of `U`. The `U` part's main workhorse is 4096×4096 JONES elements, connected in roughly the same way that the `A` part's are. The data lines `IN1`, `IN2`, `IN3`, and their `OUT` counterparts (collectively referred to as the *matrix data lines*) will be connected in the usual way so that the matrix data will follow a closed path at all times, but the other lines will be connected differently. Since the data in `A` determines which operations to perform on the matrix, we must direct the `U` elements to perform the same operations `A` does. To this end, every `add` input in `U` will be connected to `A`'s upper left `OUT1` signal. Every `row_add` input in `U` will be connected to `A`'s leftmost `OUT1` signal in the corresponding row. Every `live_col` signal in `U` will be connected to the LC row's leading flip-flop (the same one `A` uses). Every `lock_row` and `lock_lower_row` input in `U` will be connected to the LR column in the same way `A`'s are.

Before row-reducing a matrix, `U`'s JONES elements will be initialized to the identity matrix. This can be accomplished by using a row of 4096 flip-flops (called the *identity row*), initialized with all zeroes except for a 1 in the leading column. During a clock cycle, the identity row stuffs its values in the JONES top row elements, and then on the next clock, the identity row cyclically shifts to the right as the JONES elements shift upward.

Process. When A_i comes across the MPU bus, `A` will stuff its JONES top row elements with these values while cumulative-ORing them with the LC row (which was initialized to all zeroes). Simultaneously, `U` will begin stuffing itself with the identity matrix. This continues when A_j comes across the MPU bus. When this is complete, row reduction can begin.

Since we have introduced a new operation (the left-shift), in Figure 10 we detail the processing algorithm, which is realized with the JONES matrix as well as some additional control logic attached to the non-matrix-data inputs. We write two quantities, *locks* and *overs*, which represent the number of adds and left-shifts, respectively, that have occurred in `A`. (The same counts will apply to `U`.) One may realize their sum with actual counters, or with a long row of flip-flops with all zeroes save for one 1. We also have a counter z for zeroes found in the current leading column of the `A` matrix.

After completion of the while loop in Figure 10, `A` is finished, and the resulting r all-zero rows (if any) will appear on the top. `U` is not finished, however, so we must rotate it into proper position as shown in Figure 11. At this point, `U` is in proper position, and the r rows on top, when multiplied with the original values in `A`, will produce zero rows.

```
while (locks + overs < cols(A))
      if (live_col = 1)
         set z to 0
         while (z < rows(A) − locks and 0 is in upper left element of A)
               shiftup (both A and U)
         end while
         if (1 is in upper left element of A)
               add (both A and U)
         else
               shiftover (both A and U)
         end if
      else
         shiftover (both A and U)
      end if
end while
```

Fig. 10. Processing algorithm.

```
while ((locks + overs)  mod cols(U) ≠ 0)
      shiftover (U)
end while
```

Fig. 11. Rotating U into proper position.

Remark 4. Even though live_col may be 1, the 1s that were in this column beforehand may have been eliminated by a previous add, and so a 1 left in this column would belong to the row responsible for eliminating the others; such a 1 would be in a locked row, where it will do us no good. Hence, z is still necessary.

Life may be made substantially easier by using a tree of ORs (along with a column of ANDs and inverters attached to the LR column and the leading column's OUT1 signals) attached to the leading column's OUT1 signals to compute an accurate, real-time value for live_cols, but since there are 4096 rows, such a tree would require a column of interstitial flip-flops in the middle, which would impose a factor two slowdown in the linear algebra. With such a choice, z would no longer be necessary, and the first while loop above can be simplified. Further, the LC row would not be needed.

We now turn our attention to sending U's data to the multiplier. Determining that the top row of A is a zero row is fairly easy: a tree of ORs branching from the 2048 JONES top row elements of A (with the required interstitial flip-flops in the middle to alleviate gate depth concerns) can be used. This means that data can only be sent every other clock cycle. In the event that there are no zero rows to begin with, a signal is sent decreeing that the two symbols this MPU is processing are agreed, and no further work needs to be done. Otherwise, row data (detailed below) is sent across the MPU bus to the multiplier until A no longer has a zero row on top. Then (if gluing) row data will be sent to the multiplier again, but the multiplier will put this data in a different place.

We determine which data needs to be sent at which time by observing the following. The MRHS algorithm calls for the multiplications $U\binom{L_i}{0} = UT_{ij}$ and $U\binom{0}{L_j} = UT_{ji}$, which produces matrices whose columns' bottom r values we must later compare. Because of the positions of the zero blocks,

we only need the left rows(L_i) columns of U to produce Pr_{ij}, and the right rows(L_j) columns of U to produce Pr_{ji}. If we rotated the rows U cyclically downward r times, we would have the current contents of `U`. Since the top r rows in `U` contain the data we are immediately interested in, for r iterations we send the first rows(L_i) entries in the top row across the MPU bus and shift `U` and `A` up. If we are gluing, for rows(L_i) + rows(L_j) $- r$ iterations, we again send the first rows(L_i) entries in the top row across the MPU bus and shift `U` up. (Otherwise we just perform the shiftups without sending this data.) We remark that, since `A` has only 2048 columns, the maximum rank of A is 2048, and so rows(L_i) + rows(L_j) $- r$ must be less than or equal to 2048.

While the multiplier and hashtable are working to compute parts of UT_{ij}, `U` and `A` will perform the necessary shiftups to get them back to their original positions, and then `U` will perform rows(L_i) shiftovers so that it is in position to send the data needed to compute parts of UT_{ji}. Data will be sent in a similar way once it is time to begin computing UT_{ji}.

Area Calculation. Figure 9 is provided so that the logic is easy to follow. Five multiplexers, an XOR gate, and a flip-flop yield 48 transistors per element. Since `A` has $2^{12} \times 2^{11}$ elements, we use Table 4 and the 45 nm logic process to obtain an area of roughly 1.2 cm^2 for the elements. Since we only have a few more groups of 2048 flip-flops with some associated control logic, the area of `A` does not expand much from this value. The area of `U`, however, is roughly twice of that of `A`, say 2.4 cm^2, since it has twice the number of JONES elements that `A` does. Hence, `A` and `U` together really fit on a chip.

E Multiplier

Architecture. `Ur` and `Us` are two grids of flip-flops with some nontrivial logic attached. To make the process of comparing Pr_{ij} and Pr_{ji} as fast as possible, we do not compute these full matrices. Instead, we compute the first $r_{max} = 135$ rows of these matrices and ignore the rest. Hence, `Ur` is comprised of a grid of $2048 \times r_{max}$ flip-flops. `Us` however must be 2048×2048 flip-flops. They have otherwise identical architectures.

Figure 12 illustrates the following description. Each flip-flop has an AND gate hanging off of it. All the AND gates in a given row will take their second input from that row's corresponding wire in the L bus. Then, each column of AND gates sprouts an XOR tree off of the AND outputs; this is to add these bits. (Successive pairs of AND outputs in a column are XORed together. Successive pairs of these XOR outputs in the column are again XORed. And so on until we get down to one XOR.)

Remark 5. To make transistor counts smaller, we observe that in reality, we could use NANDs instead of ANDs since their results are all XORed together, and XOR produces the same result if both inputs are inverted.

Since there are 2048 rows in `Ur` (and `Us`), the signal from a flip-flop would have to travel through 12 gates. This gate depth can induce a slower clocking rate, so we insert interstitial latches after the signals have traveled through 4 gates. This means that 256 such latches are inserted after 4 gates, and another 16 are inserted after another 4 gates, per column. Then the signal can travel through the remaining parts of the XOR tree (4 more gates) until it gets to the *final row* of flip-flops. This arrangement forms a small pipeline. The data from this final row will be sent to the hash table. Since the hash table can process this row every clock cycle, we sent L_i and L_j columns through on each clock, and the interstitial latches merely induce a two-clock latency.

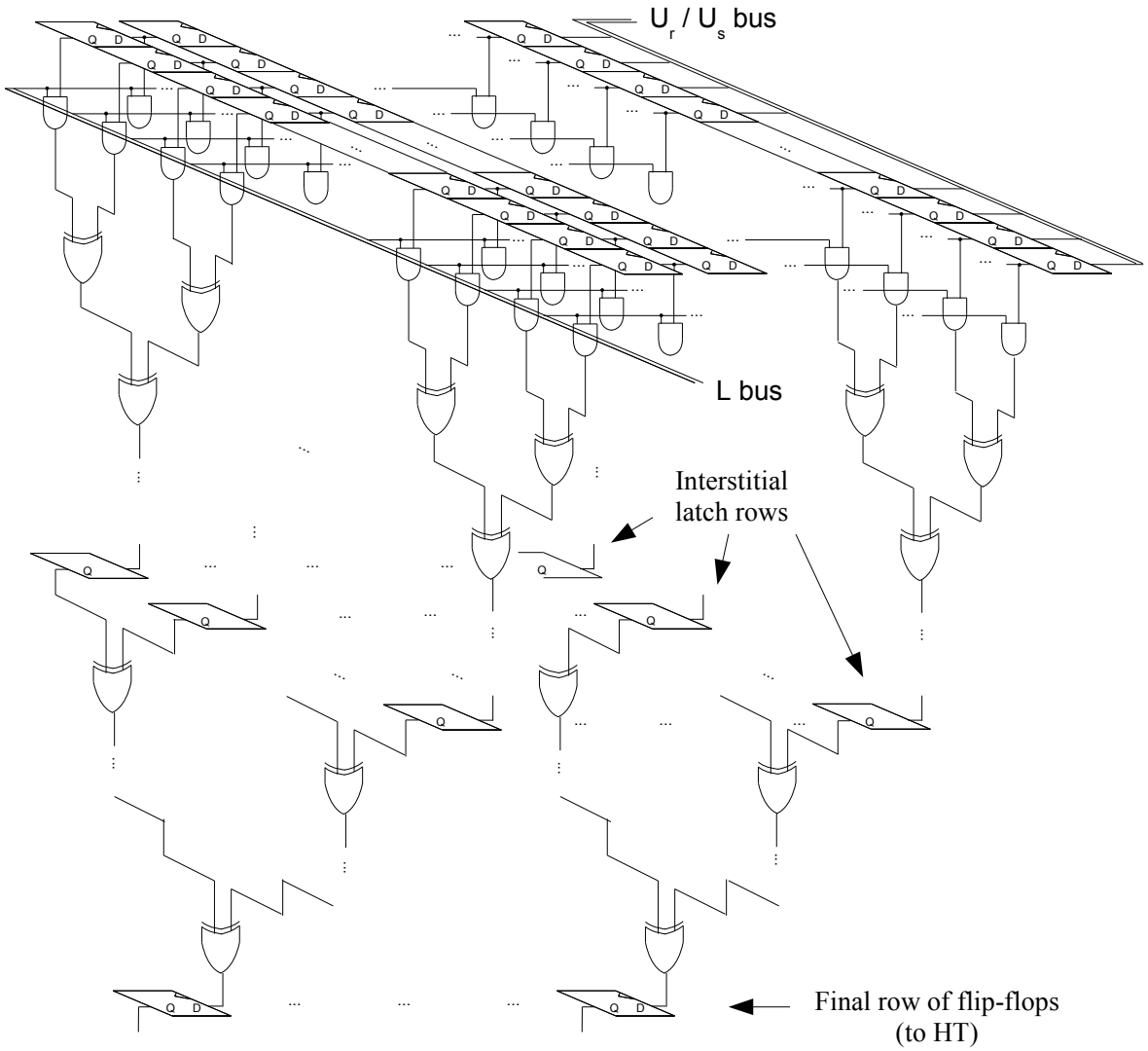

Fig. 12. `Ur` and `Us` (Low Level).

Process. When (the left cols(L_i) components of) one of the top r rows of U gets sent from `A/U` across the MPU bus, it is picked up and stored vertically in the rightmost `Ur` flip-flop column. Then all the `Ur` flip-flops get shifted leftward. This continues for each of the r rows, up through r_{max} rows. If more than r_{max} rows are sent, they are ignored by `M/HT`. If gluing, then rows(L_i) + rows(L_j) $- r$ more rows of data are sent by `A/U`, and the multiplier picks these up and stores them vertically (in a similar fashion) in `Us`.

Then, the columns of L_i will be sent. For each such column, the multiplier will pick it up and put in on the L bus, where it will be multiplied to the contents of both `Ur` (and `Us`, simultaneously, if gluing). This method of multiplication mimics the *Method of Four Russians* technique (illustrated in [1]), only without the T-storage component. The result on the final flip-flops of `Ur` is a column of Pr_{ij}, which then gets sent to the hash table. If gluing, the result on the final flip-flops of `Us` is a

column of UT_{ij} without its last r components; this gets sent to the adder across the MPU bus for storage. Note that, because of this use of the MPU bus, gluing will take at least twice as long as agreeing (possibly longer due to the hashtable).

After the columns of L_i are processed, `A/U` will send (the right $\mathrm{cols}(L_j)$ components of) the top r rows of U, which is followed by, if gluing, $\mathrm{rows}(L_i) + \mathrm{rows}(L_j) - r$ more rows of data. `Ur` and `Us` will be flushed and reloaded with this data in the same way as before.

Then the columns of L_j will be sent, and they will be multiplied to the contents of `Ur` (and if gluing, `Us`) in the same way as before. The contents of the final flip-flops of `Ur` will once again be sent to the hash table, and the contents of the final flip-flops of `Us` will once again be sent across the MPU bus to the adder (though the adder will use them differently this time).

Area Calculation. We count 2049 flip-flops per column, plus 272 interstitial latches, plus 2047 XORs, plus 2048 NANDs. Conservatively counting 10 transistors per XOR gate (which may happen in dense tree arrangements), this gives us $2049 \cdot 12 + 272 \cdot 6 + 2047 \cdot 10 + 2048 \cdot 4 = 54\,882$ transistors per column. Since there are $2048 + r_{max} = 2183$ such columns, we use the 45 nm DRAM process to get an area of 0.43 cm^2 for `Ur` and `Us` combined, including the amplification of the data on the long buses. (Since the multiplier shares a chip with the memory intensive hash table, we elect to use the DRAM process for `M/HT`.)

F Hash Table

Architecture. The hash table internally splits into two identical separate subtables HT_0 and HT_1. Each HT_μ ($\mu = 0, 1$) contains $n_{DRAM} = 256$ DRAM blocks DB_ν ($\nu = 0, \ldots, n_{DRAM} - 1$), and is designed to process one (write or look-up) query per clock cycle. Elements to be stored or looked up in the hash table have a size of $r_{max} = 135$ bit, and each of the two subtables HT_μ has to store up to $n_{entries} = 2^{20}$ entries of $r_{max} + \log_2(n_{entries}) = 135 + 20$ bit each. To ensure a reliable collision resolution, each DRAM block will offer space for $4 \cdot n_{entries}/n_{DRAM} = 2^{14}$ table entries.

The hash table is used in both PET SNAKE's agreeing and PET SNAKE's gluing phase. Subsequently, we describe the operations of the hash table for agreeing a pair of symbols. For the gluing phase, the usage of the hash table differs slightly, and we elaborate on this at the end of this section.

Storing a value in a DRAM block DB_ν. A small input logic uses a demultiplexer to select, based on the least significant $\log_2(n_{DRAM}) = 8$ bit of an incoming r-part r, one particular DRAM block DB_ν where the value r is to be stored. As the number of the DRAM implicitly encodes 8 bits of r already, only $135 - 8 = 127$ bits actually have to be stored in a table entry. Along with each table entry we store a $\log_2(n_{entries}) = 20$ bit *multiplicity* that is to count how often this value has been stored in the hash table. The 5 remaining bits to fill up 19 byte are used as flags, e. g., one flag indicates that the entry is *not empty*, the other bits are used for gluing.

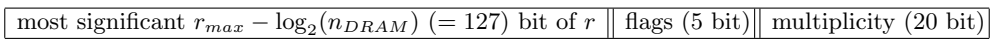

Fig. 13. Hash table entry for a value r.

With a size of 19 byte, hash table entries respect byte boundaries, and the address in the DRAM block where to store an incoming r-part r can be obtained by reading off bits no. 8–21 from r. When storing an r-part the multiplicity counter is increased. When the *not empty* flag is set, two cases can occur:

- The r-part in this table entry is identical to the value to be stored: in this case the multiplicity counter is increased.
- The r-part in this table entry differs from the value to be stored: in this case subsequent table entries are searched until a free spot or a table entry with identical r-part is found.

The basic idea of distributing each subtable HT_μ across several DRAM blocks is to ensure that a single DRAM has about $n_{DRAM} = 256$ clock cycles, before a new element to be stored arrives. So simple linear probing should suffice to resolve collisions. To handle irregularities in the distribution of incoming r-parts, we provide safety mechanisms for storing and looking-up elements. These mechanisms are detailed below in a separate subsection.

Looking up values in a DRAM block DB_ν. Once the first subtable HT_0 has been filled with the entries corresponding to a matrix L_i, read queries with r-parts corresponding to a second matrix L_j are received by the hash table. In addition to doing the look-up of these r-parts in HT_0 as described next, all these incoming r-parts are also forwarded to the second subtable HT_1 as a write query: the latter will be filled with these r-parts in exactly the same way as HT_0 has been filled with the r-parts belonging to L_i. This ensures that once the complete list of look-up queries for the second matrix L_j have been processed, the hash table is ready to answer read queries for table HT_1. As the values to be looked-up in HT_1 are exactly the values that have been written in HT_0, when filling HT_0 we also store all incoming r-parts in the order they are arriving in a *buffer DRAM* of size $n_{entries} \cdot r_{max}$ bit.

Remark 6. The buffer DRAM is realized as a collection of 16 DRAM blocks that are accessed cyclically, therewith accounting for the slower access time of DRAMs.

To check if a queried r-part is stored in a particular DRAM block of a subtable, first the memory address is computed by reading off bits no. 8–21 from the incoming r-part, just as when storing r-parts. Starting at the address obtained in this way, hash table entries are read sequentially until either a read entry is marked as unused—meaning that the queried element is not in the table—or the queried element is found as a table entry. To perform the necessary comparisons for our parameters, we can use a 128-bit comparer which checks in two clock cycles if a stored r-part coincides with the queried r-part. In case a queried value is found in the table, the stored multiplicity is added to a 20 bit *glue size counter*. If the latter counter overflows, this will be reported (see below).

Dealing with a non-uniform input distribution. One could consider a set-up where incoming r-parts are preprocessed to smoothen the bit distribution, e. g., by multiplication with a fixed random matrix. Judging from our software experiments, this does not seem to be necessary, and for simplicity we do without such a preprocessing. Nevertheless, judging from our software experiments, it can in particular happen that identical r-parts are sent to the hash table in rapid succession. This could be problematic as then a single DRAM block DB_ν might be overwhelmed with a large number of write queries arriving in a (too) short time frame.

Caching repeated write queries. To cope with the problem just mentioned, we place a comparer pipeline of length $n_{duplicates} = 32$ *before* the above-mentioned demultiplexer which directs incoming r-parts to a particular DB_ν. Every pipeline stage can store one *new* r_{max}-bit entry, one *old* r_{max}-bit entry that has already been forwarded to the demultiplexer, an 8-bit multiplicity counter and hosts two r_{max}-bit comparison units, either of which can compare the stored value with an incoming r_{max}-bit entry within two clock cycles. These comparison units are realized as a pipeline of length two and thus can handle a new input every clock cycle. When writing entries into the hash table, and an incoming r-part coincides with the *old* value currently stored in the pipeline stage, this means that the incoming r-part has been sent to the hash table during the last $n_{duplicates}$ queries already. In this case the internal multiplicity counter of the pipeline stage is increased by one, but the r-part is not forwarded to a DRAM block (yet). By setting a simple discard flag, we can ensure that such an r-part is discarded at the end of the pipeline instead of being forwarded to a DB_ν. The *new* entries are forwarded to the next pipeline stage, and the *old* values (including the counters) are shifted into the opposite direction if one value is handed over to the demultiplexer. If an entry with a stored counter value does not equal zero, the respective entry is handed over to the buffer into the demultiplexer and will be forwarded to the appropriate DB_ν, along with the appropriate counter value.

When looking up entries in the hash table, the above-mentioned discard signal is ignored and all incoming r-queries are forwarded to the individual DRAM blocks DB_ν, which will locally handle repeated r-queries as discussed next.

Local buffering. To cope with repeated read queries, in front of each DRAM block we have another short pipeline of $n_{cache} = 4$ registers, each of which is capable of holding an $r_{max} - \log_2(n_{DRAM}) =$ 127-bit value and a 20-bit multiplicity counter: after each read query actually executed by the DRAM, the respective r-part and its multiplicity is stored in one of the pipeline stages. The $n_{cache} = 4$ registers are overwritten cyclically, after each read query executed by the DRAM. If an incoming read query is found in one of the $n_{cache} = 4$ pipeline stages, this means that the answer to the read query is known already and no actual access to the DRAM block is needed. This protects the hash table from being overwhelmed with repeated read queries with identical r-parts. Finally, for the case that different read or write queries reach an individual DRAM block DB_ν in short succession, between the cache registers just mentioned and the actual DRAM block we place $n_{buffer} = 10$ registers, each of which can hold an $r_{max} - \log_2(n_{DRAM}) = 127$-bit entry, a $\log_2(n_{entries}) = 20$ bit counter representing the corresponding column index and an 8-bit counter indicating the number of times this element has been queried in a write-query already while still being held in a pipeline stage. If all n_{buffer} entries in a DRAM run full, a throttle signal is set, indicating that the hash table cannot process further queries at the moment.

Reporting results. When writing entries into one of the subtables, by default no return value from the hash table needs to be sent, except there is a danger of buffer overrun and the throttle signal is used to prevent the transmission of further write queries.

In the lookup phase no output is produced if a queried r-part is found in the list. The glue size counter is increased only. Once this counter overflows, a warning signal "glue exceeds threshold" is sent. If a queried r-part is not found, then the corresponding column index, which is a $\log_2(n_{entries}) = 20$ bit value, is returned. To keep track of the column indices, a local $\log_2(r_{max})$-bit counter is used that is increased by one whenever a read query is submitted to the hash table.

Gluing phase. The first step in the gluing phase is identical with the agreeing phase—namely, the r-parts of L_i are stored in the buffer DRAM and sent to HT_0.

In the next step, the r-parts of L_i will be sent to HT_0 again, including the column index, and this time as a look-up query. Of course, each r-part will be found and we store the corresponding 20 bit column index next to the r-part, provided the stored multiplicity for r is not greater than $\sigma_{popular} = 255$. If the r-part is hit for the first time in the second run, the place for the multiplicity counter (in Figure 13 the value on the right) is used to store the column index, and a flag is set to indicate that this value no longer represents a *multiplicity*. If the *multiplicity* has already been overwritten, the next free space in the hash table is used to store the column index. In case the next entry is not free, the 4 bit flags are used to indicate how many entries to skip. At this next address, there is space for 7 more numbers, and a new forward pointer, if necessary.

In the rare cases, that the multiplicity counter of an r-part is greater than $\sigma_{popular} = 255$, this technique would fill up the DRAM. In this case, the multiplicity is reported through the output control of the chip to the adder when this r-part r is looked up for the first time. Now, at the adder chip, memory for *multiplicity* many counters can be allocated. To simplify the communication, a *popularity number* is assigned to this r by the output control. This number is stored in the DRAM at the place where *multiplicity* has been, and from now on, this *popularity number* is used to identify this particular r. The column index is reported directly to the control unit of the adder when such an r is hit.

Finally, in the third step, the r-parts of L_j, including the column index, will be sent to HT_0 for being looked up. The column index and the list of the column indices of L_i that are stored in HT_0 (or the popularity number) will be reported to the control unit of the adder.

Area Calculation. Part of the incoming data is stored in the buffer DRAM; this can hold up to 2^{20} values with 136 bit. To cope with non-uniform distributions, the inputs are first stored and checked for duplicates in a pipeline with 32 stages. Each stage stores one 156-bit value and one 143-bit values in latches and holds two comparers for 135-bit values (NANDs followed by a tree of XORs). The comparers have a row of 9 latches after stage 4 of the XOR-tree to keep the critical path short. The distribution to the 256 DB_ν requires a tree of 255 demultiplexers (1 to 2 bit) of width 156 and depth 8. Each DB_ν has 10 buffers of width 148 bit to hold the incoming data, realized as latches. Three registers with 148 flip-flops each to store the last and the running requests and three 128-bit compare units and a 20 bit adder. Besides some control logic, there remains the DRAM to store $4 \times 2^{12} \times 152$ bit in each DB_ν.

The buffer DRAM fits in 4 mm^2. The pipeline for checking for duplicates and the demultiplexing can be realized with some 430 000 transistors, 200 000 for the pipeline, 190 000 for the demultiplexing and 40 000 for additional logic. This is an area of approx 0.15 mm^2. Each of the 256 DRAM blocks in a subtable HT_μ requires some 26 000 transistors for buffers (17 500), comparers (6 500) and logic (2000). This corresponds to an area of 0.009 mm^2 for the transistors of each subtable, the DRAM of one block requires 0.063 mm^2. The complete hash table thus fits on an area of 41 mm^2.

G Adder

Architecture. The DRAM collection is comprised of 256 DRAMs, each 1 MByte in size. Five 2060-bit buffers are placed at the inputs of each DRAM so as to protect it from multiple fast read/write operations, and five 2048-bit buffers are placed at the outputs so their results can be collected for

adding. In front of this arrangement is a small station where a 20-bit value is multiplied to a 20×20 random, fixed, invertible matrix M so as to randomize the input (which in turn randomizes which DRAM will be selected by the input). In all, this comprises the *s-lookup chain*.

Popular r-parts Preprocess. In the glue phase, a matrix L_i is first processed by the hash table and its DRAM buffer is filled with pairs of r-parts and indices. Then it processes its DRAM buffer to assign the numbers of the queries to the r values, and to find *popular r-parts*; these are those r-parts which occur more than 255 times in Pr_{ij}. Since there can be at most 4096 of them, such a popular r-part is assigned a *popularity number* (label it *pid*) from 0 through 4095. When this popularity number is assigned, the multiplicity is known and the appropriate amount of memory is allocated. To keep track of the memory for the *popular r-parts*, a list of starting point and current write position is needed for each of the up to 4096 popular numbers. The memory for the counters has to be big enough to store up to 2^{20} numbers with 20 bit. This memory is realized as an SRAM with 4 MByte. During the processing of the DRAM buffer, if it encounters a popular r-part, it sends both the popularity number *pid* and the associated index across the MPU bus to the adder. The index is stored at the appropriate memory address, and the current write position for the popularity number is increased. This continues for the duration of the DRAM buffer processing.

DRAM Collection Preprocess. At the same time, the preprocessing of popular r-parts takes place, the adder will also receive the s-part for the current L_i column. The adder will use the column number register and divert that and the s-part to the DRAM collection. The column number will be multiplied to M. The least significant 8 bits of this result are used to select which DRAM to store the s-part in, and the other 12 bits are used to determine its address. Then the s-part is stored. Because of the buffering and randomization, this will in general not slow down the preprocessing, but should an input buffer be filled, a throttle line will signal the traffic controller to stop sending columns in L_i until the writes are processed.

Process. Now columns of L_j are processed. The adder will receive the s-part of the current column of L_j, and it will store it in the adding register. The hash table will send either a collection of indices of where to find the associated s-part of L_i, or a popularity number *pid*. In the first case, the indices are simply marched along the s-lookup chain, multiplied, diverted to the DRAM collection, and the corresponding s-parts will be output. Each is then added to the value in the adding register, and the result is sent to traffic control for storage. In the second case, the adder takes *pid* and determines the memory address where the numbers start and end. These values are output.

As the s-lookup chain receives indices, it multiplies them to M, obtains results, and uses them in the same way as before to obtain s-parts from the DRAM. These are added to the value in the adding register, and the results are sent to traffic control for storage.

Area Calculation. The DRAM collection consumes 0.54 cm^2. The buffers surrounding it contain $256 \times (5 \cdot 2060 + 5 \cdot 2048) = 5\,258\,240$ flip-flops consuming 0.22 cm^2. The SRAM to hold the popular r values consumes some 0.12 cm^2—here we assume that 1T-SRAM can be used and estimate the area equivalent of 1 bit to be 1 transistor. The remaining control logic including the access to the SRAM, updating memory addresses, and the adder (3×2048 flip-flops and 2048 XORs) will not consume more than 0.20 cm^2. Together, this yields an upper bound of 1.1 cm^2 for the adder.

H Timing Comparison with Software

We turn our attention to the time cost of an MPU versus software. To measure this, the MPU's time is measured in clock cycles, but software's time is given in number of processor *steps*. Factors which relate to the software moving data in and out of memory, cache, and so forth can be captured via a constant α (i. e., each step takes α clocks on average), so a step count serves as a sort of best case scenario for software.

Suppose we are agreeing two symbols S_i and S_j. Let A_i have dimensions $w_i \times y$, A_j have dimensions $w_j \times y$, L_i have dimensions $w_i \times c_i$, and L_j have dimensions $w_j \times c_j$. Note that y then is the number of variables in the cryptosystem. Let β be the number of bits of a value that the processor can perform arithmetic on at once; in modern machines, $\beta \in \{32, 64\}$.

H.1 Linear Algebra

Let A be the vertical join of A_i and A_j. Then A has size $(w_i + w_j) \times y$. We suppose that each row will rarely have more than one 1 in A; this is usually true in the middle and later stages of a run. Let γ be the chance a second 1 exists in a column of A provided a 1 exists already in that column. Note that γ will change from symbol to symbol, but $0 \leq \gamma \leq 1$.

Hardware. JONES has two advantages over software: if a zero column exists, we dispense with it in one step, and if an add is to be performed, this also takes one step. Further, the modifications to U are done in parallel to A.

Let h be the number of columns of A that have more than one 1. Then we have that $\gamma = \frac{h}{w_i+w_j-h}$, and so $h = \frac{\gamma}{1+\gamma}(w_i + w_j)$. Thus, the number of columns of A that have exactly one 1 are $w_i - h + w_j - h$, which yields $\frac{1-\gamma}{1+\gamma}(w_i + w_j)$. Label this value t. Adding h and t gives the total number of populated columns of A. So, if we let z be the number of columns of A which are all zero, then $y - z = h + t = \frac{1}{1+\gamma}(w_i + w_j)$.

Now, since the matrices A_i and A_j are already row-reduced prior to this process, we have some reasonable expectations on where to find a 1 if it exists in a column at all; that is, if it is not near the main diagonal of A_i, it is near the main diagonal of A_j. It could happen that $h = 0$ and we are extremely unlucky with 1 placement, in which case JONES will take $y + \frac{1}{2}(w_i + w_j)^2$ clocks.

This will almost never happen, however. If there are two ones in the leftmost column of `A`, one of them will be near or at the top. If there is only one 1, it will either be at or near the top, or it will be roughly halfway down. If there are none, we just shiftover without further examining the column. So, for the h columns, we won't have to shift the rows of `A` up, and for about $\frac{1}{2}t$ columns, we still won't. For the other $\frac{1}{2}t$ columns, we can expect to perform shiftups equal to about half of the unlocked rows.

After an add, another locked row is created, so the number of unlocked rows is lessened. Further, we can expect at least two such adds to be performed between times we have to shiftup half of the unlocked rows. Hence, the first time we encounter such a column we shiftup $\frac{1}{2}(w_i+w_j)$ rows, but the next time we encounter such a column we will shiftup $\frac{1}{2}(w_i+w_j-2) = \frac{1}{2}(w_i+w_j)-1$ rows. Hence, we have a truncated triangular sum of shiftups to count. Since the number of unlocked rows starts at $w_i + w_j$, we expect a total shiftup count of $\frac{1}{2}(\frac{1}{2}(w_i + w_j))^2 - \frac{1}{2}\left[\frac{1}{2}(w_i + w_j) - \frac{1}{2}t\right]^2$, which yields $\frac{1}{8}\left(1 - \frac{4\gamma^2}{(\gamma+1)^2}\right)(w_i + w_j)$ shiftups. Hence, our total clock count is $y + \frac{1}{8}\left(1 - \frac{4\gamma^2}{(\gamma+1)^2}\right)(w_i + w_j) = y + \frac{1}{8}\frac{(1-\gamma)(1+3\gamma)}{(1+\gamma)^2}(w_i + w_j)$.

Software. Different choices for the algorithm can be made, and here we consider a situation where Gauß elimination is used to perform the row reduction. For the matrix sizes at hand, this seems a plausible option. Then software must examine $w_i + w_j$ elements in the first column. It first must find a 1, and if successful, it scans the rest of the column looking to add a row. If it finds such a row (i. e., with a 1 in this column), it performs an add of the two rows which takes y/β steps.

It then proceeds to the next column, examining the bottommost $w_i + w_j - 1$ elements, and addition of rows costs $(y-1)/\beta$ steps. And so on. We note that any additions that are performed in A are also performed in the U that is being built, and U has dimensions $(w_i + w_j) \times (w_i + w_j)$, though we do not explicitly count them.

If $y \geq w_i + w_j$, then in total there are $\frac{1}{2}(w_i + w_j)^2$ locations to visit, with a truncated triangular sum of addition steps in A equal to $\frac{\gamma}{\beta}\left[\frac{1}{2}y^2 - \frac{1}{2}(y - (w_i + w_j))^2\right] = \frac{\gamma}{\beta}(w_i + w_j)(y - \frac{1}{2}(w_i + w_j))$. In these cases we expect γ to be closer to 0 than to 1, and so hardware offers at least a factor 4 improvement in clocks over steps.

If $y \leq w_i + w_j$, then we have a truncated triangular sum of locations to visit equal to $\frac{1}{2}(w_i + w_j)^2 - \frac{1}{2}(w_i + w_j - y)^2 = y(w_i + w_j - \frac{1}{2}y)$. The addition steps total $\frac{\gamma}{\beta}\frac{1}{2}y^2$. In these cases we expect γ to be closer to 1 than to 0, and we expect few, if any, zero columns. Hence we use $y = \frac{1}{1+\gamma}(w_i + w_j)$, and putting just the locations expression over the clocks expression, we have a factor improvement equal to

$$\frac{\frac{1}{1+\gamma}(w_i + w_j)\left((w_i + w_j) - \frac{1}{2}\frac{1}{1+\gamma}(w_i + w_j)\right)}{\frac{1}{1+\gamma}(w_i + w_j) + \frac{1}{8}\frac{(1-\gamma)(1+3\gamma)}{(1+\gamma)^2}(w_i + w_j)^2} = \frac{\frac{1+2\gamma}{2+2\gamma}(w_i + w_j)}{\frac{1}{8}\frac{(1-\gamma)(1+3\gamma)}{1+\gamma}(w_i + w_j) + 1}$$

As γ increases towards 1, this expression will tend towards a factor $\frac{3}{4}(w_i + w_j)$ improvement (i.e. JONES takes linear time). This does not come as a surprise, for when γ gets closer to 1, there is less and less need to perform shiftups to find 1s.

H.2 Matrix Multiplication and Recording Deletions

Hardware. Once `Ur` and `Us` are loaded, their multiplications to L_i occur in parallel; similarly for L_j. Because of the pipeline structure of the multiplier, all the columns of UT_{ij} (similarly, UT_{ji}) are computed at a rate of one clock per column, plus a few clocks of latency in the beginning. The hash table then picks up the resulting r-parts and processes them at a rate of one clock per r-part, and it is also structured in a pipeline fashion.

Hence, processing L_i takes c_i clocks, plus a few clocks of latency. Then, processing L_j also takes c_j clocks, plus a few clocks of latency. Since the MPU bus must be used to report a deletion, it will take one clock per deletion, up to a maximum of c_j clocks to report all of L_j's deletions. Finally, L_i is processed again from the hash table's DRAM buffer, and those entries are looked up (for deletions) at the same rate. Since the hash table can report a deletion at the same time as looking up the next value, we count c_i clocks to report any deletions for L_i.

Since the traffic controller can record a deletion in a pipeline fashion and send a column at the same time, no additional overhead is counted for this. Finally, because of the 'just in time' nature of symbol transmission, it takes no additional time for a deletion to actually take hold in a symbol.

Thus, two symbols will have their deletions processed in $2c_i + 2c_j$ clocks, plus some small latency. (At the very end of an agreement phase, an additional c_i clocks will also be spent for one symbol. This is a one-time latency cost.)

Software. Using a Method of Four Russians approach in software is certainly helpful in constructing Pr_{ij}. The T-storage matrix is set up on each pass. Arranging the data the same way the hardware handles it, this T matrix has 2^k rows of r entries each, where k is the storage constant (typically $k = 8$, but can be increased), and $r = \text{rows}(A) - \text{rank}(A)$. It is built in $2^k \frac{r}{\beta}$ steps. Then, for (the given k bits of) each L_i column, the appropriate entry in the T matrix is read off and stored (taking $\frac{r}{\beta}$ steps), waiting to be added later. This continues for the entire pass. Hence, a pass takes $2^k \frac{r}{\beta} + c_i \frac{r}{\beta}$ steps. Afterwards, a new T matrix will need to be built. Since there are $\frac{w_i}{k}$ passes, all passes total comprise $\frac{w_i}{k}(2^k + c_i)\frac{r}{\beta}$ steps.

After all passes are complete, the subresults are added together to produce the final result of the multiplication. We can use $\log \frac{w_i}{k}$ additions of matrices, each addition taking $c_i \frac{r}{\beta}$ steps. This gives a total step count of

$$\frac{w_i}{k}(2^k + c_i)\frac{r}{\beta} + c_i \frac{r}{\beta} \log \frac{w_i}{k} = \frac{r}{\beta}\left(\frac{w_i}{k}2^k + c_i(\frac{w_i}{k} + \log \frac{w_i}{k})\right)$$

to construct Pr_{ij}. A similar expression will result when constructing Pr_{ji}.

One could try to optimize by increasing k to 16 or so, but $k = 32$ is troublesome as the 2^k term starts to dominate.

The situation gets worse for software; it still has to search through the data to find matching r-parts. Sorting Pr_{ij} will take at least $c_i \log c_i$ steps and as many as $\frac{r}{\beta} c_i \log c_i$, should many r-parts become popular. Similar expressions result when sorting Pr_{ji}. Finally, a bilinear search taking $\frac{r}{\beta}(c_i + c_j)$ more steps must be performed to find matching r-parts. Once the mismatches are found, columns have to be deleted from L_i and L_j; this takes $\frac{r}{\beta}(c_i + c_j)$ steps. Hence, total sorting and searching for both matrices takes $\frac{r}{\beta}\left(c_i(2 + \log c_i) + c_j(2 + \log c_j)\right)$ steps.

In total, we have

$$\frac{r}{\beta}\left(\frac{w_i + w_j}{k}2^k + c_i(\frac{w_i}{k} + 2 + \log c_i \frac{w_i}{k}) + c_j(\frac{w_j}{k} + 2 + \log c_j \frac{w_j}{k})\right)$$

steps to agree the symbols S_i and S_j.

The MPU has a very clear and obvious advantage. Aside from the additional terms the software induces in its step count, it is important to stress that the hardware does not rely on the values of r, w_i, or w_j at all. Hence, large r (whose maximum value is 2^{11}) will dramatically slow down the software, but the hardware will be unaffected. Since r will steadily increase over the entire run, hardware's advantage will grow over time.

H.3 Gluing

Both hardware and software must pay the linear algebra times and the multiplication times as described earlier. From there the situation changes slightly. At this point we know that we may only construct a symbol whose L-part has no more than 2^{20} columns, so we label the number of such columns d.

Hardware. During the matrix multiplication of L_i, r-parts are being stored in the hash table at the same time, so we do not count this cost again. However, s-parts are being sent to the adder at the same time, so the adder's DRAM collection is filled for free.

Afterwards, the hash table will go through a preprocessing of its c_i entries. It may happen that these values hit the SRAM of the adder entirely too quickly, at which point we must pay upwards of an 8-clock penalty per such index. In the worst case this takes $8c_i$ clocks in total, but is expected to average to more like $2c_i$ over the course of an entire run.

Then, L_j is processed. We get an s-part in one clock (after some latency), and at the same time, its r-part is examined for matches in the hash table. If the hash table has the matching indices, it simply sends them, one per clock. If the adder has them, the adder uses its SRAM to produce them to the s-lookup chain. Since the SRAM produces values 128 bits at a time (that is, 6 indices per 8 clocks), the penalty of multiple fast read requests is mitigated.

Hence, we have worst case behavior of $8c_i + \frac{8}{6}d$ and best case behavior of $c_i + d$ clocks to finish all additions.

Software. It is plain that the software will suffer tremendously if it has to re-match r-parts to find corresponding s-parts to add, so we give it a fighting chance by allowing it to store the matching indices during agreement. (This gets expensive in memory with a state of several hundred symbols, but can nonetheless be theorized.)

Then it merely performs lookups of its storage data. Since there are d pairs of s-parts to be added, software takes $\frac{r}{\beta}d$ steps to finish all additions. Again, as r steadily increases over a run, software becomes vastly inferior to hardware, which does not rely on the value of r.

I Software Measurement

It should be noted that, in the above derivations, the linear algebra is almost always dominated by matrix multiplication and recording deletions, both in hardware and in software.

In order to get a handle on performance metrics, four rounds of PRESENT were cryptanalyzed in software ($k = 8$, $y = 308$) using MRHS with the above options, and this entire session's timing values were recorded. The platform was an Intel E2180 processor, $\beta = 32$, on a single core of 2 GHz, with 2 GByte of RAM. Out of the nearly 10,000 agreements that took place, the vast majority took less than two seconds. We removed these from consideration since fractions of seconds were not measured. Many calculations were made on the remaining 350 or so agreements using the above step count formulas, some results of which are illustrated in Table 1. We see no problem using just these $\sim$350 values since in a full cryptosystem operated on by PET SNAKE, there will commonly be high w_i, w_j, r, c_i, and c_j values, and these data points are more reflective of this scenario. It should be noted that we calculated steps using $\gamma = 0.5$; varying γ in either direction does not adversely affect our overall results.

An average of the $\sim$350 time improvement factors gives an average improvement of 2281 for four rounds of PRESENT. As noted above, as r gets larger, we suspect PET SNAKE will only improve from there.

To get a better feeling for just how much more favorable PET SNAKE will be, we see that an average of the $\sim$350 α data points gives $\alpha = 66.068$, where α is the metric of steps per processor clock. Some things are not included in the step count, such as loop counter variables incrementing, allocation space instructions, and low-level memory management.

Once we have a good handle on the α that a given processor exhibits, we can predict software behavior for larger systems. For example, if PET SNAKE runs an MRHS attack on AES or PRESENT, it won't be uncommon for $y > 1500$, $w_i > 1024$, and $r > 1024$. Modeling such systems

w_1	c_1	w_2	c_2	r	time (s)	total steps	pentium clocks	α	PS clocks	PS time (s)	improvement
208	32768	236	524144	192	4	185372686.4	8000000000	43.15630395	1127822	0.001127822	3546.658959
211	98304	236	196796	192	2	94214489.28	4000000000	42.45631463	604383.625	0.000604384	3309.156498
211	98304	236	524144	192	4	205688510.1	8000000000	38.89376221	1259079.625	0.00125908	3176.923779
229	121856	236	196796	192	2	103100923.9	4000000000	38.79693654	652627.625	0.000652628	3064.534695
213	14336	237	196796	190	2	68614875.44	4000000000	58.29639673	436634.5	0.000436635	4580.49009
207	32768	229	248832	190	3	89847645.61	6000000000	66.77971314	576709.1111	0.000576709	5201.929261
207	32768	229	248832	190	2	89847645.61	4000000000	44.51980876	576709.1111	0.000576709	3467.95284
212	16384	229	248832	189	2	85166930.02	4000000000	46.96658667	544245.625	0.000544246	3674.81135
217	16384	229	248832	189	2	85273467.77	4000000000	46.90790822	544553.6111	0.000544554	3672.732967
212	16384	229	497664	189	3	168450194.4	6000000000	35.61883689	1041909.625	0.00104191	2879.328425
212	16384	229	786432	189	5	266482279.2	10000000000	37.52594743	1619445.625	0.001619446	3087.476308
213	28672	236	524144	189	3	184351734.1	6000000000	32.54647986	1119940.069	0.00111994	2678.714765
213	57344	229	497664	189	6	180808589.4	12000000000	66.36852841	1123890.944	0.001123891	5338.596267
213	57344	229	497664	189	3	180808589.4	6000000000	33.1842642	1123890.944	0.001123891	2669.298133
210	98304	229	248832	189	2	110163469.2	4000000000	36.30967713	707963.4028	0.000707963	2825.004784
210	98304	229	497664	189	3	193446733.7	6000000000	31.01629005	1205627.403	0.001205627	2488.330966
212	12288	229	248832	187	2	83962706.02	4000000000	47.64019872	536053.625	0.000536054	3730.970013
212	12288	229	497664	187	3	167245970.4	6000000000	35.8753038	1033717.625	0.001033718	2902.146512
212	12288	229	786432	187	5	265278055.2	10000000000	37.69629565	1611253.625	0.001611254	3103.173779
213	57344	229	497664	185	3	180808589.4	6000000000	33.1842642	1123890.944	0.001123891	2669.298133
213	57344	229	786432	185	5	278840674.1	10000000000	35.86277372	1701426.944	0.001701427	2938.709779
213	16384	236	524144	184	3	180691970.9	6000000000	33.20568131	1095364.069	0.001095364	2738.815416
134	1048576	147	131072	102	5	202949079.1	10000000000	49.27344359	2365087.403	0.002365087	2114.086775
125	4096	141	450816	101	2	78081960.56	4000000000	51.22822187	915045.6111	0.000915046	2185.683397
125	4096	137	1048576	101	4	185889011.7	8000000000	43.03643301	2110418.944	0.002110419	1895.35827
122	8192	137	1048576	101	4	186478052	8000000000	42.90049105	2118502.403	0.002118502	1888.126251
129	16384	137	524288	101	2	93036403.87	4000000000	42.99392317	1086565.611	0.001086566	1840.661972
129	65536	140	450816	101	3	87773276.82	6000000000	68.35793555	1038037.069	0.001038037	2890.070199
129	65536	137	1048576	101	6	195580582.3	12000000000	61.35578419	2233445.611	0.002233446	2686.432107
135	1048576	152	131072	101	5	202950785.3	10000000000	49.27302934	2365324.069	0.002365324	2113.875246
134	1048576	139	1048576	101	7	366057846.9	14000000000	38.24532138	4199787.625	0.004199788	1666.750947

Table 1. Some measured values of software performance ($k = 8$, $\beta = 32$, $\gamma = 0.5$, $y = 308$)

in software directly is problematic owing to the lack of sufficient on-board memory at the time of this writing, but we can predict step counts for software under these conditions.

Table 2 gives the relevant predictions fixing $\alpha = 66.068$. In the later stages of a given attack of a full cipher of something like AES or PRESENT, we'll see symbol sizes listed in this table. The relative improvement of PET SNAKE is now even clearer, touching a six-digit improvement.

Finally, it is worth noting that other software methods may be used to multiply large matrices; it is certainly possible that some of them may be more efficient than the Method of Four Russians, and so the improvement factor may be reduced. However, PET SNAKE's time is still unaffected by these large symbols, processing each pair in less than half of a hundredth of a second. We feel that such absolute speed is too compelling to be dismissed.

w_1	c_1	w_2	c_2	r	time (s)	total steps	pentium clocks	PS clocks	PS time (s)	improvement
1000	1048576	1000	1048576	500	170.7644319	5169289689	3.41529E+11	4474081.778	0.004474082	38167.48115
750	1048576	750	1048576	300	84.95722911	2571779869	1.69914E+11	4352054	0.004352054	19521.17991
500	1048576	1000	1048576	200	59.44129911	1799375263	1.18883E+11	4352304	0.004194304	14171.91007
500	1048576	1000	1048576	400	110.3584658	3340712542	2.20717E+11	4352304	0.004194304	26311.50861
500	1048576	1000	1048576	600	161.2756325	4882049821	3.22551E+11	4352304	0.004194304	38451.10715
500	1048576	1000	1048576	800	212.1927992	6423387100	4.24386E+11	4352304	0.004194304	50590.70569
1500	1048576	1500	1048576	1000	482.5320576	14606952758	9.65064E+11	4821304	0.004194304	115044.6075

Table 2. Some projected values of software performance ($k = 8$, $\beta = 32$, $\gamma = 0.5$, $y = 308$, $\alpha = 66.068$)

J Storage and Parallelism

PET SNAKE has a storage capacity of 4.792 TB (not including the 'active' DRAMs in the traffic control chips), or enough to store 18000 full-size symbols. This number was chosen based on the following observations: the symbol count for AES will drop to 180 before threshold takes over, and it is possible we may have to guess up to 100 key variables before we find the key; hence up to 100 states may need to be stored along the way. Very rarely will a state actually be comprised of nothing but full-size (that is, 257 MB) symbols, so it will almost always be possible to store more than 100 states; 400 or more states are not unlikely.

PET SNAKE will, in its depth-first search of keys, eventually guess enough keys so that either the system is found to be inconsistent or the key is correct. This number of keys we refer to as δ. So that it may make appropriate use of parallelism, PET SNAKE will eventually guess enough keys discovering δ, and then make note of its available storage. Then the MCP will be able to determine how high in the guess tree it can fork a new guess into another area of the board, while having the ability to store the states required for a sub-branch of this new guess as well as for the original branch. The idea here is that PET SNAKE will use all of its MPUs to finish off a branch of a guess tree as quickly as possible. If more MPUs become available, more guesses can potentially be forked.

Should the MCP determine that storage will run out, it will delete some states higher in the guess tree. Any such state which needs to be recovered later can always be recalculated based on the next-highest state in the guess tree, and the remaining key guess symbols to affect the deleted state's guess. It is true that these (possibly several) guesses will need to be re-performed in one series of agrees and glues, increasing the overall running time, but PET SNAKE at least has recovery

options should storage requirements vary wildly across parallel branches of the guess tree. For this reason, PET SNAKE will never delete the highest state in the guess tree, that is, the state which was arrived at before any guesses were committed.

K PET SNAKE versus Multiple PCs

In addition to an expected improvement in the order of several magnitudes of an MPU versus a PC for agreeing, PET SNAKE also offers additional expected time savings versus networked PCs. Connecting networked PCs in the same way as PET SNAKE connects its MPUs will introduce additional time spent. Suppose that a grid of PCs is connected so each can talk to its neighbor in each cardinal direction using gigabit Ethernet, and suppose that this network actually communicates perfectly (i.e., 1 gigabit/sec).

PET SNAKE's connections are 1024 wires clocked at 1 GHz, so it can transmit 1000 gigabit/sec between MPUs. This makes the PC network 1000 times as slow. With the observation that a PC agrees $\gg$1000 times slower than an MPU, the PCs could also implement a 'just in time' delivery method to reduce agreement communication times. However, when symbols need to be moved between agreement stages or to prepare for a glue, we see that the movement time for a PC is a little over 2.15 sec per symbol per hop (over 4.5 minutes per agreement phase, assuming no deletions), whereas for PET SNAKE it is 0.00215 sec per symbol per hop. Hence, a faster network between PCs will need to be established, which in turn adds to the cost of such a solution.

Finally, for multiple PCs to provide the same storage as PET SNAKE, a single PC has to store 4680 MB, not including active memory of at least 325 MB. This is slightly larger than 4 GB per PC, and so more expensive motherboards that can provide larger memory will need to be acquired. (Slower storage solutions like hard drives can be used instead, but given their notorious relative slowness, the times for loading and storing would start to dominate an overall time estimate, and this would make finding a key infeasible.)

L Gate Counts and Surface Area for an ASIC Implementation

gate/component	AND	OR	NOT	XOR	NAND	NOR	D flip-flop	Latch	2-to-1 Multiplexer
transistors	6	6	2	6	4	4	12	6	6

Table 3. Transistor counts of logic gates and components.

	45 nm logic process	45 nm DRAM process
transistor	$0.2975\mu\text{m}^2$	$0.35\mu\text{m}^2$
DRAM bit	$0.0875\mu\text{m}^2$	$0.025\mu\text{m}^2$

Table 4. Average surface area of hardware components.