Accelerating the Whirlpool Hash Function using Parallel Table Lookup and Fast Cyclical Permutation

Yedidya Hilewitz[1], Yiqun Lisa Yin[2], and Ruby B. Lee[1]

[1] Department of Electrical Engineering,
Princeton University, Princeton NJ 08544, USA
{hilewitz, rblee}@princeton.edu
[2] Independent Security Consultant
yiqun@alum.mit.edu

Abstract. Hash functions are an important building block in almost all security applications. In the past few years, there have been major advances in the cryptanalysis of hash functions, especially the MDx family, and it has become important to select new hash functions for next-generation security applications. One of the potential candidates is Whirlpool, an AES-based hash function. Whirlpool adopts a very different design approach from MDx, and hence it has withstood all the latest attacks. However, its slow software performance has made it less attractive for practical use. In this paper, we present a new software implementation of Whirlpool that is significantly faster than previous ones. Our optimization leverages new ISA extensions, in particularly Parallel Table Lookup (PTLU), which has previously been proposed to accelerate block ciphers like AES and DES, multimedia and other applications. We also show a novel cyclical permutation algorithm that can concurrently convert rows of a matrix to diagonals. We obtain a speedup of 8.8× and 13.9× over a basic RISC architecture using 64-bit and 128-bit PTLU modules, respectively. This is equivalent to rates of 11.4 and 7.2 cycles/byte, respectively, which makes our Whirlpool implementation faster than the fastest published rate of 12 cycles/byte for SHA-2 in software.

1 Introduction

Hash functions form an important component in almost all security applications, e.g., digital signature schemes, to ensure the authenticity and integrity of data. Some of the most popular hash functions are MD5 [20] and SHA-1 [4]. Both have been widely deployed in practice and adopted by major security standards such as SSL/TLS and IPsec.

In the past few years, there have been major breakthroughs in the cryptanalysis of hash functions. New collision attacks on MD5 [24] and SHA-1 [23] have demonstrated serious weaknesses in their design. Built upon these attacks, researchers have also developed new attacks on hash-based security protocols such as X.509 digital certificate protocol [22]. While practical impact of these attacks is still debatable, it is obvious that new hash functions are needed. Indeed, NIST has already hosted two hash function workshops and has started an AES-like competition to select an Advanced Hash Standard (AHS) [17].

Whirlpool [1] is a hash function designed by Barreto and Rijmen in 2000. It is designed based on the AES with very similar structure and basic operations. It has been adopted by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) as part of the joint ISO/IEC international standard.

Since its publication, there have been some studies on fast implementation of Whirlpool [12, 19], mostly in hardware. A comprehensive comparative study on hash function implementation [16] shows that Whirlpool is several times slower than MD5 or SHA-1 in software. Due to its relative slow performance and the prevalence of MD5 and SHA-1 in existing implementations, Whirlpool has not attracted too much attention for practical use.

With the emergence of new hash proposals, there is some renewed interest in Whirlpool. Compared with most of the new proposals, Whirlpool stands out with its AES-based clean design. Its design approach is very different from the MDx hash family, and hence may resist existing attacks that are applicable to MDx. Also, since AES is now the NIST standard for block ciphers, there is intense interest in faster implementations of AES and its security analysis. Whirlpool's similarity to AES can leverage these fast implementation techniques and facilitate its security analysis.

In this paper, we present a new software implementation of Whirlpool that is significantly faster than previous implementations. Our optimization method takes advantage of the heavy use of table lookups and byte-oriented operations in Whirlpool by leveraging processor ISA (Instruction Set Architecture) extensions that are tailored to such operations. In particular, the Parallel Table Lookup module (PTLU) [6, 11] is a natural fit for the Whirlpool computation steps, thereby providing major speedup. In addition, a subword permutation instruction called `check` [14] is also useful for accelerating cyclical permutations, giving further performance enhancements. These ISA extensions have been defined previously for other purposes such as multimedia and cryptographic processing. Besides general-purpose microprocessors, these operations are even more suitable for crypto-processors and hardware ASIC (Application Specific Integrated Circuit) implementations, for fast software and hardware implementations of Whirlpool.

Our software implementation of Whirlpool attains a speedup of 8.8× with a 64-bit PTLU module and 13.9× with a 128-bit PTLU module, compared with a baseline single-issue processor. These performance results show that ISA extensions are much faster - with significantly simpler hardware - than using conventional micro-architectural performance-enhancing techniques such as superscalar execution. For example, 4-way superscalar execution achieves a speedup of only 3.3×. We also compare our Whirlpool performance with other 512-bit hash functions like SHA-2; we have a rate of 11.4 cycles/byte for the 64-bit PTLU module and 7.2 cycles/byte for the 128-bit PTLU module, while the best reported rate for SHA-2 is 12 cycles/byte on Intel Core 2 and AMD Opteron processors [21]. This suggests that Whirlpool is a viable hash function choice, providing excellent security and excellent performance.

We remark that the use of the PTLU functional unit provides not only major performance advantages but also security advantages in preventing side-channel attacks. A new concern with software implementations of cryptographic algorithms based on table lookups is the leakage of the secret key due to cache-based *software* side channel attacks, which do not require additional equipment like power or timing *physical* side channel attacks. Our proposed fast implementation of Whirlpool, when it is used in keyed hash mode, is free from such cache-based software side channel attacks.

The rest of the paper is organized as follows. Section 2 provides a high-level overview of Whirlpool. Section 3 provides the motivation for our fast implementation of Whirlpool. Section 4 describes the parallel table lookup module and other ISA extensions. Section 5 explains how to use these ISA extensions to accelerate Whirlpool. Section 6 presents performance results and Section 7 considers security advantages. Section 8 is the conclusion.

2 Whirlpool

2.1 Algorithm Overview

Like most hash functions, Whirlpool operates by iterating a compression function that has fixed-size input and output. Its compression function is a dedicated AES-like block cipher that takes a 512-bit hash state M and a 512-bit key K. (Hence, both the state and the key can be conveniently represented as 8×8 matrices with byte entries.) The iteration process adopts the well-known Miyaguchi-Preneel construction [15].

In what follows, we provide a concise description of the compression function that is most relevant to our implementation. Technical details of the algorithm can be found in [1]. At a high level, each execution of the compression function can be divided into two parts:

a. expanding the initial key K into ten 512-bit round keys, and
b. updating the hash state M by mixing M and the round keys.

Part b consists of ten rounds, and each round consists of the following four steps (labeled W1 through W4 below) with byte-oriented operations:

W1. Non-linear substitution. Each byte in the state matrix M is substituted by another byte according to a predefined substitution, $S(x)$ (aka S-box).

W2. Cyclical permutation. Each column of the state matrix M is cyclic shifted so that column j is shifted downwards by j positions.

W3. Linear diffusion. The state matrix M is multiplied with a predefined 8×8 MDS matrix C.

W4. Addition of keys. Each byte of the round key is exclusive-or'ed (XOR) to each byte of the state.

The key expansion (part a) is almost the same as the above state update, except that the initial key K is treated as the state and some pre-defined constants as the key. Hence, both parts consist of ten similar rounds.

Note that Whirlpool differs from AES in that the rounds operate on 512-bit inputs rather than 128-bit inputs. Because of the larger block size, the design of the S-box and MDS matrix is also adjusted accordingly, but the general design philosophy remains the same.

2.2 A Useful Observation by the Designers

In [1], the designers of Whirlpool suggested a method to implement each round of the compression function using only table lookup and XOR operations on a 64-bit processor. We exploit this in our optimization.

Their idea is to first define a set of tables which combine the computation of the S-box S and MDS matrix C. For $0 \leq k \leq 7$, let C_k be the k-th row of the MDS-matrix C. Define eight tables of the following form:

$$T_k(x) = S(x) \cdot C_k,\ 0 \leq k \leq 7. \tag{1}$$

Note that each table T_k has $2^8 = 256$ entries, indexed by the input x. For each x, the entry $S(x) \cdot C_k$ has eight bytes (by multiplying $S(x)$ with each of the eight bytes in the row C_k) . Hence, each table T_k is 2^{11} bytes, and the total storage is 2^{14} bytes (16 KB) for the eight tables. Given these tables, one can rewrite the operations in Steps W1 through W3 as follows. Let $M_{i,j}$ denote the (i, j)th byte in the state matrix before Step W1, and let M'_i denote the ith row in the state matrix after Step W3. Then M'_i (which is 8 bytes) can be computed as

$$M'_i = \bigoplus_{k=0}^{7} T_k(M_{(i-k) \bmod 8,k}). \tag{2}$$

For example, the first output row M'_0 can be computed as

$$\begin{aligned} M'_0 = T_0(M_{0,0}) \oplus T_1(M_{7,1}) \oplus T_2(M_{6,2}) \oplus T_3(M_{5,3}) \oplus \\ T_4(M_{4,4}) \oplus T_5(M_{3,5}) \oplus T_6(M_{2,6}) \oplus T_7(M_{1,7}). \end{aligned} \tag{3}$$

Equation (3) produces the first row of the updated state matrix M'. It is repeated to generate all 8 rows of the new state matrix, M'_i, for $i = 0, 1, \ldots, 7$.

3 Motivation for our Fast Implementation

How fast can a software implementation of Whirlpool be? Considering Equation (3), each row of the updated matrix M' can be computed with 8 selections of byte-elements of the current 8×8 matrix M, 8 table lookup operations using these 8 selected bytes as indices, and 7 exclusive-or operations. Hence, this computation takes $8d + 8 + 7$ instructions, where d is the number of instructions needed to select a byte and place it in a register in a form that can be used by the next instruction for a load instruction (to perform the table lookup). In a typical RISC processor, $d = 3$ instructions: shift target byte to correct position, mask byte, and add to base address of table. Since Equation (3) is repeated for each of 8 rows, the number of instructions required is $8 \times (8d + 8 + 7) = 8 \times 39 = 312$ instructions. An additional 8 exclusive-or instructions are required for key addition, for a total of 320 instructions. Since this is performed for both the state and key matrices, the total number of instructions per round is $2 \times 320 = 640$ instructions.

Since the only serial dependences are between generating the index for a table lookup, doing the table lookup, then combining this result with other results using an XOR, can we achieve a faster software implementation with appropriate new instruction primitives? By appropriate instruction primitives, we mean instructions that are reasonable in cost, and have general-purpose usage for a variety of applications. Reasonable cost also suggests that any new instruction should fit the datapath structure of general-purpose microprocessors, which implies that an instruction can have at most 2 source registers and 1 result register.

Equation (3) can also be described in two steps to generate each new row of the state matrix, M':

A1. Cyclical Permutation. Select all the 8 byte-elements in parallel, placing them in the appropriate order in a register. (Step W2)

A2. Substitution and Diffusion. Look up 8 tables in parallel, using the bytes in the register generated in step A1 as indices, and immediately combine these 8 results into a single result using an XOR tree. (Steps W1 and W3)

Fig. 1. Main steps in our optimized Whirlpool software implementation

Suppose Step A1 takes x instructions and Step A2 takes y instructions. Then, the total number of instructions taken for 8 rows, for the state and key matrices, is:

$$2 \times 8 \times (x + y). \tag{4}$$

Note that $x = 24$ instructions and $y = 15$ instructions in the above calculations for the basic RISC processor.

With the microprocessor datapath restriction described above where *an instruction can have at most 2 source registers and 1 result register*, Step A1 would require $x = 4$ instructions since it needs to read from 8 different registers. Step A2 could potentially be done in $y = 1$ instruction since it has only one operand and one result. It turns out that we can indeed achieve step A2 in $y = 1$ instruction using a powerful parallel table lookup instruction (Section 4). We can do better in Step A1 using effectively only $x = 3$ instructions rather than 4, by cyclically permuting all 8 rows of the matrix concurrently (Section 5).

4 ISA Extensions

Whirlpools's heavy use of table lookup and byte-oriented computations motivate us to pay special attention to ISA extensions that are related to such operations. We describe a parallel table lookup instruction (Section 4.1) and a subword permutation instruction (Section 4.2) previously proposed to accelerate multimedia, block ciphers and other applications.

In general, ISAs are extended when new applications emerge that require a set of operations that are not well supported by existing instructions. Emulation of these operations can take many tens or hundreds of existing instructions. Consequently, new instructions are added to perform the operations, yielding significant acceleration and, typically, reduced power consumption. For microprocessors, the goal is that the new operations are "general-purpose", meaning that they are useful in other applications beyond the initial motivating ones - the more applications the more likely the new operation will be supported in future generations of microprocessors. We show that two previously proposed operations are also useful for Whirlpool.

4.1 Parallel Table Lookup

Parallel table lookup was initially proposed to speed up block cipher execution, including AES [6], and other block ciphers including DES, 3DES, Mars, Twofish and RC4 [5]. It has also been used for fingerprinting and erasure codes to accelerate storage backup [11] and other algorithms that can employ table-lookup as an optimization.

An n-bit Parallel Table Lookup (PTLU) module consists of $n/8$ blocks of memory, each with its own read port. Fig. 2 shows a 64-bit PTLU with 8 parallel memory blocks. (A 128-bit PTLU will have 16 parallel memory blocks.) The inputs to the module are sourced from two general-purpose registers and the output is written to a single general-purpose register - hence fitting into the typical 2-source, 1-result datapath of processors. The $n/8$ blocks of memory are configured as a set of 256-entry tables, indexed by the $n/8$ bytes of the first source operand. The tables are read in parallel and the outputs from the tables are combined using a simple combinational logic function - a tree of XOR-Multiplexers (termed XMUXes). The result is then XORed with the second source operand and written to the result register.

The PTLU module is read using the following instruction:

```
ptrd.x1 r1, r2, r3
```

The bytes of r_2 are used as indices into the set of tables in the PTLU module, the outputs of which are XORed together into one value and then XORed with r_3 before being written to r_1. While a parallel table lookup only needs one source register, r_2, to supply the table indices, a second source register is available in processor datapaths, and so the XOR (or some other combination) with r_3 is essentially free in the above `ptrd.x1` instruction.

In the PTLU module proposed in [5, 6, 11], the XMUX's can also perform other operations like logical OR, or select the left (or right) input, in addition to the XOR operation. The "x1" in the ptrd instruction specifies that the XOR operation is selected and one 64-bit result is produced. (An "x2" subop is used for a 128-bit PTLU module to indicate that two 64-bit results are produced in the final XMUX stage.)

In [6], a fast instruction for loading the 8 tables in parallel is also proposed. A row across all 8 tables can be written from the contents of a data cache line in a single ptw instruction. Hence, only 256 instructions are needed to load 8 tables each having 256 entries, rather than 8×256 instructions. In [11], addressing multiple sets of tables is also described, to allow concurrent processing of different algorithms which use the parallel lookup tables, without the need for re-loading tables.

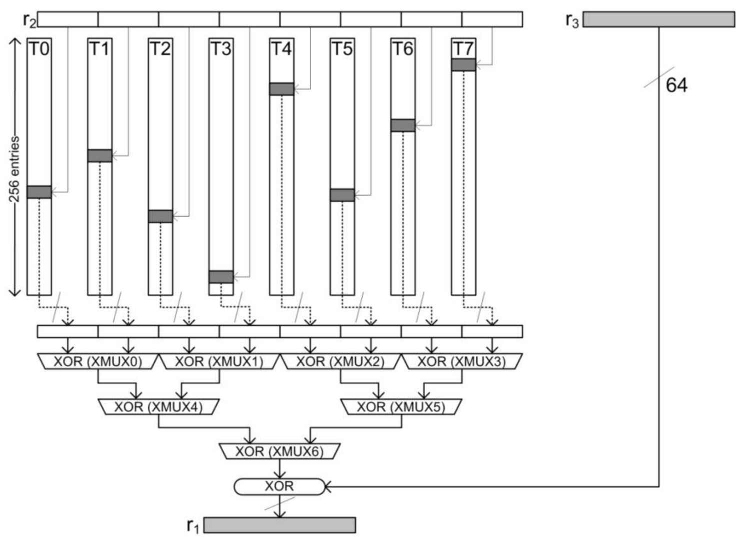

Fig. 2. PTLU module

4.2 Byte Permutations

Multimedia applications often require operations on subwords, or data smaller than the processor word (or register) size, down to a byte. ISAs have been extended with instructions that perform standard arithmetic or logical operations on these subwords in parallel as well as with instructions to efficiently rearrange these subwords in a register and between registers [13, 14]. For example, the check instruction was defined by Lee [14] as one of a small set of subword permutation instructions for rearranging the elements of matrices in processing two-dimensional multimedia data like images, graphics and video. We propose re-using this to accelerate Whirlpool. The `check` instruction is defined as follows:

```
check.sw r1, r2, r3
```

The subwords of size sw bytes are selected alternately from the two source registers, r_2 and r_3, in a checkerboard pattern, and the result is written to r_1. In Fig. 3, each register is shown as 8 bytes, and the `check` instruction is shown for 2-byte subwords. The IBM AltiVec `vsel` instruction [9], which, for each bit position, conditionally selects from the bits of the two source operands depending on the value of the bit in a third source operand, can also be used to perform `check` when executed with the appropriate masks in the third operand. Similarly, the Intel SSE4 `pblend` instructions [10], which conditionally select subwords from two operands depending on the value of an immediate or a fixed third source operand, can also be used to perform `check`.

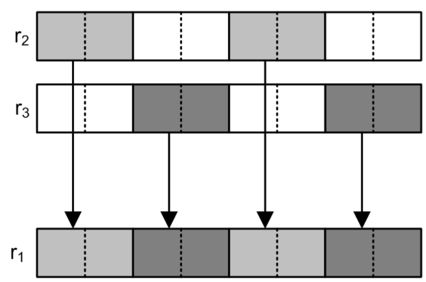

Fig. 3. check.2 r1, r2, r3 (for 64-bit registers)

5 Fast Software Implementation of Whirlpool

We now show in detail how we use the two instructions defined in Section 4 to implement the two steps in our optimized Whirlpool algorithm shown in Fig. 1. We will focus on 64-bit processors - the same techniques can be easily extended to processors with 128-bit registers, with minor variations.

Fig. 4 shows our optimized pseudocode for one round of the state update of the Whirlpool compression function on a 64-bit processor using PTLU. The 64 bytes of key are held in 8 general purpose registers (RK0-RK7) and the 64 bytes of state are held in 8 general purpose registers (RM0-RM7). The eight PTLU tables contain the eight tables from Equation (1), which combine steps W1 and W3 (Section 2) of the Whirlpool algorithm, also labeled step *A2* (Section 3) . A further optimization with PTLU is that step W4 is also combined with steps W1 and W3 by a single `ptrd` instruction. Step W2, also labeled step *A1* (Section 3), is performed in the Cyclical Permute function described in Section 5.2.

```
# RM0-RM7 are the 8 state registers
# RK0-RK7 are the 8 key registers

Cyclical_Permute(RM0-RM7)
ptrd.x1 RM0, RM0, RK0
ptrd.x1 RM1, RM1, RK1
ptrd.x1 RM2, RM2, RK2
ptrd.x1 RM3, RM3, RK3
ptrd.x1 RM4, RM4, RK4
ptrd.x1 RM5, RM5, RK5
ptrd.x1 RM6, RM6, RK6
ptrd.x1 RM7, RM7, RK7
```

Fig. 4. Pseudocode for one round of the state update of Whirlpool compression.

5.1 Using PTLU for Substitution and Diffusion (step A2)

A single PTLU read instruction updates a row of the state matrix, performing the eight table lookups of Equation (2) at once. For example, the instruction

```
ptrd.x1 RM0, RM0, RK0
```

corresponds to Equation (3), which details the state transformation of row 0. The eight bytes in row 0 of M: $M_{0,0}, M_{7,1}, \ldots, M_{1,7}$, are stored in RM0 after the cyclical permutation step. These 8 bytes are used as the indices into the set of eight tables defined by Equation (1) which are stored in the PTLU module (Fig. 2). The eight 64-bit table entries read out, $T_0(M_{0,0}), T_1(M_{7,1}), \ldots, T_7(M_{1,7})$, are XORed together by the XMUX tree. At this point, the PTLU module has performed Equation (3). The output of the XMUX tree is also XORed with the first row of the key matrix stored in RK0, completing the state transformation of row 0. The updated row 0 of M is then written back to RM0. Seven more `ptrd` instructions update the remaining 7 rows of M.

5.2 Novel Algorithm for Cyclical Permutation

The state matrix at the start of a round is shown in Fig. 5(a). The transformed matrix, used in the table lookup, is shown in Fig. 5(b). This transformation is the columnar cyclical permutation of the Whirlpool compression function, accomplished by rotating the jth column down by j positions. We propose a novel algorithm that accomplishes this in a logarithmic number of steps. First, move columns 1, 3, 5 and 7 down by 1 row. Second, move columns 2 and 3, 6 and 7 down by 2 rows. At this point, columns 0 and 4 have been moved down by 0 rows, columns 1 and 5 by 1 row, columns 2 and 6 by 2 rows, and columns 3 and 7 by 3 rows. Third, move columns 4, 5, 6 and 7 down by 4 rows. This achieves the desired result, where column j has been moved down by j rows.

The transformation by cyclical permutation from Fig. 5(a) to Fig. 5(b) turns rows of the matrix into (wrapped) diagonals. In [14], Lee showed how two

R0	00	01	02	03	04	05	06	07
R1	10	11	12	13	14	15	16	17
R2	20	21	22	23	24	25	26	27
R3	30	31	32	33	34	35	36	37
R4	40	41	42	43	44	45	46	47
R5	50	51	52	53	54	55	56	57
R6	60	61	62	63	64	65	66	67
R7	70	71	72	73	74	75	76	77

(a)

R0'	00	71	62	53	44	35	26	17
R1'	10	01	72	63	54	45	36	27
R2'	20	11	02	73	64	55	46	37
R3'	30	21	12	03	74	65	56	47
R4'	40	31	22	13	04	75	66	57
R5'	50	41	32	23	14	05	76	67
R6'	60	51	42	33	24	15	06	77
R7'	70	61	52	43	34	25	16	07

(b)

Fig. 5. (a) 8×8 matrix at start of round; (b) 8×8 matrix after cyclical permutation

R0	00	01	02	03	04	05	06	07
R7	70	71	72	73	74	75	76	77

$\Downarrow$

R0'	00	71	02	73	04	75	06	77

Fig. 6. check.1 R0', R0, R7

`check.1` instructions can be used to rotate one column of each 2×2 matrix mapped across 2 registers. We propose using the `check.sw` instructions, doubling the subword size (`sw`) at each step, to turn the eight rows of the 8×8 matrix of bytes (in eight 64-bit registers) into eight diagonals.

First, we execute a `check.1` instruction on each row and its neighbor one row above (Fig. 6), which selects one byte alternately from the two registers. This has the effect of rotating columns 1, 3, 5 and 7 down by one position (Fig. 7(a)). Second, we execute a `check.2` instruction on each row and its neighbor two rows above, which selects 2 bytes alternately from the two registers. This has the effect of rotating columns 2, 3, 6 and 7 down by an additional two positions (Fig. 7(b)). Third, we execute a `check.4` instruction on each row and its neighbor four rows above, which selects 4 bytes alternately from the two registers. This results in rotating columns 4, 5, 6 and 7 down an additional four positions to yield the final permutation (Fig. 5(b)).

5.3 Register Usage and Instruction Counts

Register usage: Most RISC processors have only 32 General Purpose Registers. Our software implementation requires only 24 registers, 8 each for key, state and scratch space, plus a few registers for memory pointers. The first step of the cyclical permutation writes its result to 8 scratch registers, the second step writes back to the original 8 registers, the third step writes to the scratch registers, and the `ptrd` instruction writes back to the original registers. Thus our implementation is not constrained by register allocation.

Instruction Counts: Updating the state matrix takes 32 instructions total as $8 \times \lg(8) = 24$ `check` instructions are needed to cyclically permute the matrix

R0’	00	71	02	73	04	75	06	77
R1’	10	01	12	03	14	05	16	07
R2’	20	11	22	13	24	15	26	17
R3’	30	21	32	23	34	25	36	27
R4’	40	31	42	33	44	35	46	37
R5’	50	41	52	43	54	45	56	47
R6’	60	51	62	53	64	55	66	57
R7’	70	61	72	63	74	65	76	67

(a)

R0	00	71	62	53	04	75	66	57
R1	10	01	72	63	14	05	76	67
R2	20	11	02	73	24	15	06	77
R3	30	21	12	03	34	25	16	07
R4	40	31	22	13	44	35	26	17
R5	50	41	32	23	54	45	36	27
R6	60	51	42	33	64	55	46	37
R7	70	61	52	43	74	65	56	47

(b)

Fig. 7. (a) State matrix with columns 1, 3, 5 and 7 rotated down by 1 position; (b) State matrix with columns 1 and 5 rotated down by one position, columns 2 and 6 by two positions and columns 3 and 7 by three positions

and 8 `ptrd` instructions are needed to complete the update (see Fig. 4). The key matrix undergoes a similar update with the only difference being an additional load instruction to retrieve the round constant. Thus one round of the Whirlpool compression function takes 65 instructions with PTLU-64.

Without PTLU, a round takes approximately 640 instructions on a basic RISC processor (Section 3). Thus, using PTLU reduces the instruction count by an order of magnitude. In Section 6, we consider cycle counts of the full Whirlpool hash function.

5.4 Extending the Techniques to PTLU-128

For a processor with 128-bit registers, a PTLU-128 module with 16 parallel memory blocks can be used (Fig. 2 shows PTLU-64 with 8 memory blocks). In a PTLU-128 version of the parallel lookup instruction, `ptrd.x2`, the 16 bytes of the first source register are used as indices into the 16 tables, the outputs of which are XORed into 2 parallel 64-bit values, which are each XORed with the second source register before being written to the 128-bit destination register.

For 128-bit registers, the cyclical permutation step also requires an instruction to rearrange the bytes within a word, as two rows are contained within a single processor register. We use a `byteperm` instruction, also defined in [6]. In this instruction, the first source register holds the data to be permuted and the second source register lists the new ordering for the bytes of the data. This instruction is similar to the IBM AltiVec `vperm` instruction [9] or the IA-32 `pshufb` instruction [10] and is only needed for the 128-bit PTLU module, not for the 64-bit PTLU module.

In total, 8 `check` instructions and 8 `byteperm` instructions are needed to cyclically permute the matrix (held in only 4 128-bit registers) in a 128-bit processor; the precise sequence of instructions is omitted for brevity. Only 4 `ptrd.x2` instructions are needed for each of the key and state matrix transformations in a round as two iterations of Equation (2) are done in parallel with `ptrd.x2`.

Hence, a Whirlpool round takes only $2 \times 20 + 1 = 41$ instructions with a 128-bit PTLU.

Commodity microprocessors have 128-bit register files for their multimedia instructions like SSE for Intel x86 processors [10] and AltiVec for PowerPC processors [9]. Hence, it is not unreasonable to add a 128-bit PTLU unit to the multimedia functional units using the 128-bit registers already present.

6 Performance Analysis

Table 1 summarizes the performance improvement for Whirlpool over the basic 64-bit RISC processor for single-issue 64-bit and 128-bit processors with PTLU and for 64-bit superscalar execution (evaluated using the SimpleScalar Alpha simulator [2]). We compare our performance using ISA extensions to superscalar execution, because the latter is the technique typically used by processor designers to increase performance by executing multiple instructions each processor cycle. k-way superscalar means the execution of k instructions per cycle. In general, the hardware cost of superscalar execution increases exponentially with k, while the performance increases less than linearly with k.

While Whirlpool scales well with superscalar execution, ranging from 1.65× to 5.97× for 2-way to 8-way superscalar, adding a PTLU module (and using the `check` and `byteperm` instructions) yields even better results: 8.79× with a 64-bit PTLU and 13.90× with a 128-bit PTLU. The latter can be compared to the 1.65× speedup of a 2-way superscalar processor, as both perform the equivalent of two instructions per cycle - the processor with 128-bit PTLU is 8.42× faster. Even the 64-bit PTLU with 1 instruction per cycle is faster than the very complex 8-way superscalar processor.

In Table 2, we compare our results (PTLU-64 and PTLU-128) with the performance of Whirlpool on some existing processors [16, 21], and with the performance of the SHA-2 512-bit hash function [16, 21, 8]. The single-issue 64-bit processor with PTLU greatly outperforms more complex 3- and 4-way superscalar processors like the AMD Opteron or the Intel Core 2.

We also estimated the performance of Whirlpool on the Intel Core 2 hypothetically enhanced with a single PTLU-128 module using its 16 128-bit SSE registers. The performance result is slightly slower than that of our single issue RISC processor with PTLU-128. This is due to the Core 2 machine having a `byteperm` (implemented by `pshufb`) with a 3 cycle latency and 2 cycle pipelined instruction issue. (Note that later Core 2 processors have a "super shuffle engine" with a 1 cycle `pshufb`.) Performance was also impacted by extra copy instructions due to IA-32 instructions overwriting one of the source operands, and limited superscalar speedup due to the single PTLU module and serialization restrictions on the byte permutation instructions. Nevertheless, due to the tremendous performance boost provided by the PTLU-128 module, our Whirlpool implementation still has better performance than SHA-2 on the complex Intel Core 2 microprocessor.

Table 1. Relative Performance of Whirlpool

baseline	Speedup with Superscalar			Speedup with PTLU, 1-way	
	2-way	4-way	8-way	64-bit	128-bit
1	1.65	3.26	5.97	8.79	13.90

Table 2. Performance of Whirlpool and SHA-2

Algorithm	Processor	Cycles per Byte
Whirlpool	**PTLU-64**	11.41
	PTLU-128	7.22
	Pentium III (asm)	36.52 [16]
	Core 2 (C)	44 [21]
	Opteron (C)	38 [21]
SHA-2 512	Pentium III (asm)	40.18 [16]
	Core 2 (C)	12 [21]
	Opteron (C)	12 [21] / 13.4 [8]

7 Security Advantages

In Section 1, we discussed the security advantages of the Whirlpool algorithm, in light of recent advances in finding collisions in MD-5 and SHA-1 hash functions. We now discuss the additional advantages of using PTLU in our Whirlpool implementation in thwarting side-channel attacks as well.

Cache side-channel timing attacks [18] have recently been shown to be viable against cryptographic algorithms that use lookup tables stored in cache, such as AES. One such attack forces part of the lookup table out of the cache and then measures the time of a subsequent encryption. If the encryption takes longer than the baseline time, it implies that the part of the table that was evicted from the cache had to be refetched from main memory. This provides information about the key bytes. The general idea can also be applied to keyed hash functions that use lookup tables.

Using PTLU to perform the table lookups precludes these timing attacks from taking place, as the tables do not reside in cache. Table access time is always a constant for all tables in the PTLU module. Multiple processes can use the same Whirlpool PTLU tables without impacting each other. If another process needs the PTLU module, either multiple sets of tables may be implemented in hardware or the OS is responsible for fully replacing and restoring the table contents during context switch. Consequently, the use of PTLU for Whirlpool not only provides tremendous performance improvements but also increases the security of the implementation when Whirlpool is used in keyed mode such as for MACs.

In general, the use of PTLU can protect crypto algorithms from cache-based side-channel attacks. This would allow table lookup to continue to be an effective *non-linear component* in ciphers and hash functions. For the MDx hash family, the linear relation between the hash state and the input message has proved to be a major weakness that made these functions vulnerable to the so-called message modification techniques [23, 24]. Whirlpool, with its heavy use of table lookup, provides excellent resistance against this line of new attacks on hash functions.

8 Conclusions

We have presented a fast software implementation of the hash function Whirlpool, based on ISA extensions that permit parallel table lookup and a novel algorithm that performs the cyclical permutation of the columns of the state (or key) matrix in parallel. We show that the PTLU (parallel table lookup) module, together with `check`, a subword permutation instruction, can greatly improve the performance of Whirlpool. More specifically, on a single-issue 64-bit processor, our software implementation provides an 8.79× speedup, more than the 5.97× speedup gained from the much more complex hardware technique of 8-way superscalar execution. With our speedup, Whirlpool is faster than SHA-512, both of which produce 512-bit hash results.

Our optimization approach is somewhat different from existing ones. While most research in fast software implementations has focused on how to optimize given existing ISA, we also try to address the problem from the other direction. That is, what ISA extensions are most useful to speed up existing algorithms? The ISA extensions used in our implementation have already been defined and applied earlier to accelerate multimedia and cryptographic processing. Our new results on Whirlpool, together with the earlier work, support the inclusion of more powerful ISA extensions in both general-purpose processors and crypto-processors. In particular, the fact that many crypto algorithms make heavy use of table lookups make the PTLU module and associated instructions very attractive for future CPUs. Additionally, the use of PTLU inoculates these crypto algorithms against cache-based software side channel attacks.

Due to Whirlpool's initial performance problem, its designers have proposed the Maelstrom-0 hash function [7] as a replacement. This new hash function changes the key schedule, but uses the same compression function for updating the hash state. Consequently, the techniques presented in this paper will speed up Maelstrom-0 as well.

Designing and selecting new hash functions is a hot subject for both the crypto research community and the security industry. Our new implementation results suggest that, in addition to its security, Whirlpool can also have great performance. Therefore, Whirlpool can be a viable hash function choice for next-generation security applications.

Acknowledgments. Y. Hilewitz is supported by NSF and Hertz Foundation Fellowships.

References

1. Barreto, P.S.L.M., Rijmen, V.: The Whirlpool Hashing Function. http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html
2. Burger, D. Austin, T.: The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Report #1342 (1997)
3. CACTI 4.2. HP Labs. http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html
4. Federal Information Processing Standards (FIPS) Publication 180-1. Secure Hash Standard (SHS). U.S. DoC/NIST, (1995)
5. Fiskiran, A.M.: Instruction Set Architecture for Accelerating Cryptographic Processing in Wireless Computing Devices. PhD Thesis, Princeton University (2005)
6. Fiskiran, A.M., Lee, R.B.: On-Chip Lookup Tables for Fast Symmetric-Key Encryption, Proceedings of the IEEE 16th International Conference on Application-Specific Systems, Architectures and Processors (ASAP) pp. 356–363, IEEE (2005)
7. Gazzoni Filho, D.L., Barreto, P.S.L.M., Rijmen, V.: The Maelstrom-0 Hash Function. VI Brazilian Symposium on Information and Computer Systems Security (2006)
8. Gladman, B.: SHA1, SHA2, HMAC and Key Derivation in C. http://fp.gladman.plus.com/cryptography_technology/sha/index.htm
9. IBM Corporation. PowerPC Microprocessor Family: AltiVec Technology Programming Environments Manual. Version 2.0 (2003)
10. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Vol. 1-2 (2007)
11. Josephson, W., Lee, R.B., Li, K.: ISA Support for Fingerprinting and Erasure Codes. Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), IEEE (2007)
12. Kitsos, P., Koufopavlou, O.: Whirlpool Hash Function: Architecture and VLSI Implementation. Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS '04), pp. 23–36 (2004)
13. Lee, R.B.: Subword Parallelism with MAX-2. IEEE Micro, vol. 16, no. 4, pp. 51–59, IEEE (1996)
14. Lee, R.B.: Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures. Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors, pp. 3–14, IEEE (2000)
15. Menezes, A., van Orschot, P., Vanstone, S.: Handbook of applied cryptography. CRC Press, (1997)
16. Nakajima, J., Matsui, M.: Performance Analysis and Parallel Implementation of Dedicated Hash Functions. EUROCRYPT 2002, LNCS, vol. 2332, pp. 165–180, Springer (2002)
17. NIST. Hash Function Main Page. http://www.nist.gov/hash-competition
18. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. Cryptology ePrint Archive, Report 2005/271, (2005)

19. Pramstaller, N., Rechberger, C., Rijmen, V.: A Compact FPGA Implementation of the Hash Function Whirlpool. Proceedings of 14th International Symposium on Field Programmable Gate Arrays, pp. 159–166, (2006)
20. Rivest, R.L.: The MD5 message-digest algorithm. Request for comments (RFC) 1321, Internet Activities Board, Internet Privacy Task Force, (1992)
21. St. Denis, T.: LibTomCrypt Benchmarks. http://libtomcrypt.com/ltc113.html
22. Stevens, M., Lenstra, A., de Weger, B.: Chosen-prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. EUROCRYPT 2007, LNCS, vol. 4515, pp. 1–22, Springer (2007)
23. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. CRYPTO 2005, LNCS, vol. 3621, pp. 17-36, Springer (2005)
24. Wang, X. Yu, H.: How to Break MD5 and Other Hash Functions. EUROCRYPT 2005, LNCS, vol. 3494, pp. 19–35, Springer (2005)

Appendix A: Hardware Cost Analysis

We estimate the cost in terms of area and latency of adding PTLU, `byteperm` and `check`. For PTLU we used CACTI [3] to estimate the latency and area of the tables and we synthesized the XMUX tree using a TSMC 90nm library. We compare the access time latency and area of our 64-bit PTLU module with a cache of the same capacity (i.e., 16 Kilobyte cache), and also compare our 128-bit PTLU with a 32 Kilobyte cache. The 64-bit PTLU module, which has 16KB of tables, has 88% of the latency and 92% of the area of a 16KB 2-way associative cache with 64 byte lines. The 128-bit PTLU module has 75% of the latency and 79% of the area of a 32KB 2-way associative cache with 64 byte lines. In each case, we find that the PTLU module is faster and smaller than a typical data cache of the same capacity. Still, the two modules have larger latencies than an ALU, so we conservatively estimate the `ptrd` instruction to take two processor cycles. Since the results of the table lookups are not needed right away (Fig. 4), this has no impact on performance.

For `byteperm` and `check`, in an ISA such as IA-32 or IA-64 that has a multimedia subword permutation unit, the cost of adding these instructions, if they do not already exist, is negligible. For other ISAs, support for the `byteperm` instruction can be added to the shifter unit with minimal impact to area and without affecting the cycle time [6]. The `check` instruction can be implemented by a set of $n/8$ 8-bit 2:1 multiplexers with the control bit pattern selected from a small set of fixed bitstrings: $(0^k1^k)^{n/2k}$, $k = 1, 2, 4, \ldots$ and n the register width in bytes. Thus, it can also be easily added without area or cycle time impact.

Appendix B: Related Work

Byte permutation instructions such as the `byteperm` instruction described (or the PowerPC AltiVec `vperm` [9] or Intel SSSE3 `pshufb` [10] mentioned above), can be used as a limited PTLU instruction. For example, in the `vperm` instruction, which uses three 128-bit registers, the bytes of the third source operand are indices that select bytes in the first two source operands. The latter can be considered a single 32-entry table, with byte entries. With `byteperm` or `pshufb`, which only have 2 source registers, the first operand functions as a 16-entry table. These instructions can be used for the S-box non-linear substitutions, which map a byte to a byte, in AES or Whirlpool implementations that explicitly perform all four steps (W1, W2, W3, W4) of the state transformation (Section 2). However, the PTLU instruction used in this paper is much more capable.