Fast Hashing and Stream Encryption with PANAMA

Joan Daemen[1] and Craig Clapp[2]

[1] Banksys, Haachtesteenweg 1442, B-1130 Brussel, Belgium
Daemen.J@banksys.be
[2] PictureTel Corporation, 100 Minuteman Rd., Andover, MA 01810, USA
craigc@pictel.com

Abstract. We present a cryptographic module that can be used both as a cryptographic hash function and as a stream cipher. High performance is achieved through a combination of low work-factor and a high degree of parallelism. Throughputs of 5.1 bits/cycle for the hashing mode and 4.7 bits/cycle for the stream cipher mode are demonstrated on a commercially available VLIW micro-processor.

1 Introduction

PANAMA is a cryptographic module that can be used both as a cryptographic hash function and a stream cipher. It is designed to be very efficient in software implementations on 32-bit architectures.

Its basic operations are on 32-bit words. The hashing state is updated by a parallel nonlinear transformation, the buffer operates as a linear feedback shift register, similar to that applied in the compression function of SHA [6]. PANAMA is largely based on the STEPRIGHTUP stream/hash module that was described in [4].

PANAMA has a low per-byte work factor while still claiming very high security. The price paid for this is a relatively high fixed computational overhead for every execution of the hash function. This makes the PANAMA hash function less suited for the hashing of messages shorter than the equivalent of a typewritten page. For the stream cipher it results in a relatively long initialization procedure. Hence, in applications where speed is critical, too frequent resynchronization should be avoided.

A typical application for PANAMA might be the encryption or decryption of video-rate data in conditional access applications (e.g. pay-TV). Set-top boxes and future digital televisions will increasingly include media processors for decoding compressed video and for performing other computationally intensive image processing tasks. This is an application space where data rates are high, high-performance processors are increasingly likely to be present, and decryption must be done yet must not unduly burden an already heavily loaded processor.

After specifying the PANAMA hash function and stream cipher, we discuss the particular design strategy and the implementation aspects. We don't attempt to give a proof of security. However, a motivation for the design choices is given.

S. Vaudenay (Ed.): Fast Software Encryption – FSE'98, LNCS 1372, pp. 60–74, 1998.
© Springer-Verlag Berlin Heidelberg 1998

A C reference implementation of PANAMA and a PostScript and PDF versions of [4] are available from `http://www.esat.kuleuven.ac.be/~rijmen/daemen`.

2 Basic design principles

PANAMA is based on a finite state machine with a 544-bit *state* and a 8192-bit *buffer*. The state and buffer can be updated by performing an *iteration*. There are two modes for the iteration function. A *Push* mode, that allows to inject an input and generates no output, and a *Pull* mode that takes no input and generates an output. A *blank* Pull iteration is a Pull iteration in which the output is discarded.

The updating transformation of the state has high diffusion and distributed nonlinearity. Its design is aimed at providing very high nonlinearity and fast diffusion for multiple iterations. This is realised by the combination of four distinct transformations each with its specific contribution. There is one for nonlinearity, one for bit dispersion, one for inter-bit diffusion, and one for injection of buffer and input bits.

The buffer behaves as a linear feedback shift register that ensures that input bits are injected into the state over a wide interval of iterations. In the Push mode the input to the shift register is formed by the external input, in the Pull mode, by part of the state.

The PANAMA hash function is defined as performing Push iterations with message blocks as input. If all message blocks have been injected, a number of blank Pull iterations are performed to allow the last message blocks be diffused into the buffer and state. This is followed by a final Pull iteration to retrieve the hash result.

The PANAMA stream encryption scheme is initialised by doing two Push iterations to inject the key and diversification parameter followed by a number of blank Pull iterations to allow the key and parameter to be diffused into the buffer and state. After this initialisation, the scheme is ready to generate keystream bits at leisure by performing Pull iterations.

3 Specification

The state is denoted by a and consists of 17 (32-bit) words a_0 to a_{16}. The buffer b is a linear feedback shift register with 32 *stages*, each consisting of 8 words. An 8-word stage is denoted by b^j and its words by b^j_i. Both stages and words are indexed starting from 0.

The three possible *modes* for the PANAMA module are Reset, Push and Pull. In Reset mode the state and buffer are set to 0. In Push mode an 8-word input is applied and there is no output. In Pull mode there is no input and an 8-word output is delivered.

The buffer update operation is denoted by λ. We have (with $d = \lambda(b)$):

$$\begin{aligned} d^j &= b^{j-1} \text{ if } j \notin \{0, 25\}, \\ d^0 &= b^{31} \oplus q \ , \\ d_i^{25} &= b_i^{24} \oplus b_{i+2 \bmod 8}^{31} \text{ for } 0 \leq i < 8 \ . \end{aligned} \tag{1}$$

In Push mode q is the input block p, in Pull mode it is part of the state a, with its 8 component words given by

$$q_i = a_{i+1} \text{ for } 0 \leq i < 8 \ . \tag{2}$$

The state updating transformation is denoted by ρ. It is composed of a number of specific transformations:

$$\rho = \sigma \circ \theta \circ \pi \circ \gamma \ . \tag{3}$$

Here $\circ$ denotes the (associative) composition of transformations where the rightmost transformation is executed first.

θ is an invertible linear transformation defined by:

$$c = \theta(a) \Leftrightarrow c_i = a_i \oplus a_{i+1} \oplus a_{i+4} \text{ for } 0 \leq i < 17 \ , \tag{4}$$

with the indices taken modulo 17. The invertibility of θ follows from the fact that $1 \oplus x \oplus x^4$ is coprime to $1 \oplus x^{17}$.

γ is an invertible nonlinear transformation defined by:

$$c = \gamma(a) \Leftrightarrow c_i = a_i \oplus (a_{i+1} \text{ OR } \overline{a_{i+2}}) \text{ for } 0 \leq i < 17 \ , \tag{5}$$

with the indices taken modulo 17. A proof for the invertibility of γ can be found in [4].

The permutation π combines cyclic word shifts and a permutation of the word positions. If we define τ_k to be a rotation over k positions from LSB to MSB, we have:

$$c = \pi(a) \Leftrightarrow c_i = \tau_k(a_j) \ , \tag{6}$$

with

$$\begin{aligned} j &= 7i && \bmod 17 \quad \text{and} \\ k &= i(i+1)/2 && \bmod 32 \ . \end{aligned} \tag{7}$$

The transformation σ corresponds with bitwise addition of buffer and input words. It is given by (let $c = \sigma(a)$):

$$\begin{aligned} c_0 &= a_0 \oplus 00000001_{\text{hex}} \ , && \\ c_{i+1} &= a_{i+1} \oplus \ell_i && \text{for } 0 \leq i < 8 \ , \\ c_{i+9} &= a_{i+9} \oplus b_i^{16} && \text{for } 0 \leq i < 8 \ . \end{aligned} \tag{8}$$

In the Push mode ℓ corresponds with the input p, in the Pull mode $\ell = b^4$.

In the Pull mode the output z consists of 8 words given by

$$z_i = a_{i+9} \text{ for } 0 \leq i < 8 \ . \tag{9}$$

The transformation ρ is illustrated in Fig. 1, the Push and Pull modes of the Panama module are illustrated in Fig. 2.

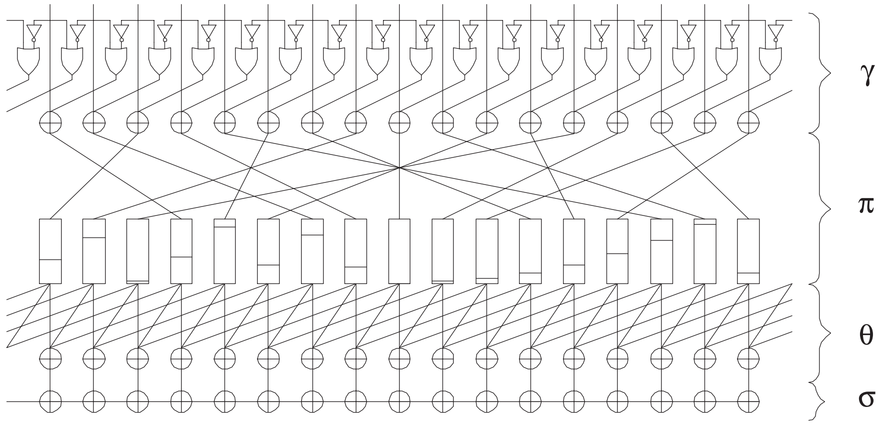

Fig. 1. The state updating transformation ρ.

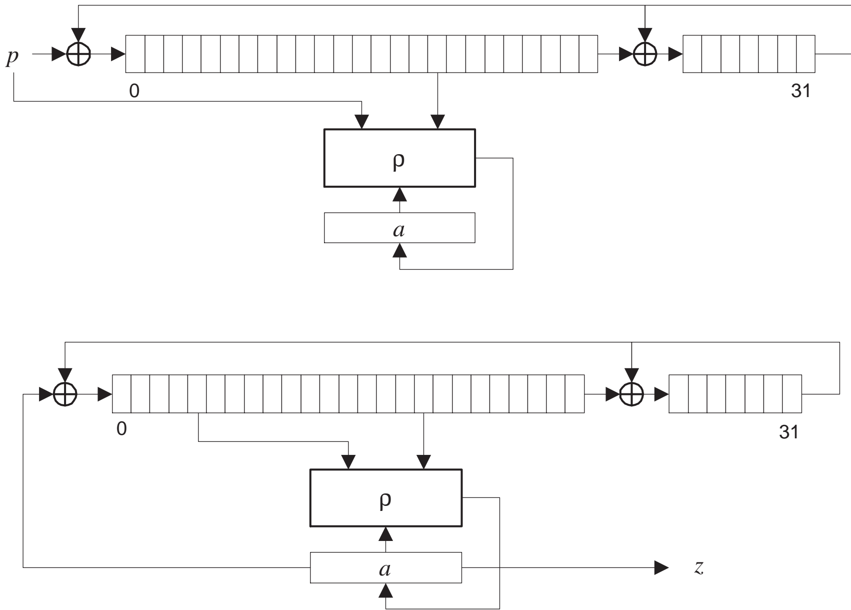

Fig. 2. Push (above) and Pull (below) modes of PANAMA.

3.1 The Panama hash function

The Panama hash function maps a message of arbitrary length M to a hash result of 256 bits. The Panama hash function is executed in two phases:

- **Padding** M is converted into a string M' with a length that is a multiple of 256 by appending a single 1 followed by a number d of 0-bits with $0 \leq d < 256$.
- **Iteration** The input sequence $M' = p^1 p^2 \ldots p^V$ is loaded into the Panama module according to Table 1.

After all input blocks have been loaded, an additional 32 blank Pull iterations are performed. Then the Hash result h is returned. The number of Push and Pull iterations to hash an V-block input sequence is $V + 33$.

Time step t	Mode	Input	Output
0	reset	–	–
$1, \ldots, V$	Push	p^t	–
$V + 1, \ldots, V + 32$	Pull	–	–
$V + 33$	Pull	–	h

Table 1. The sequence diagram of the iteration phase of the Panama hash function.

The design goal for the Panama hash function is that it should be *hermetic*. For the definition of this term we refer to [4]. In short, for a hermetic hash function, the following statements are true.

Assume we take as hash result the value of a subset of n bits of the (for Panama 256-bit) output:

- the expected workload of generating a collision is of the order of $2^{n/2}$ executions of the hash function,
- given an n-bit value, the expected workload of finding a message that hashes to that value is of the order of 2^n executions of the hash function,
- given a message and its n-bit hash result, the expected workload of finding a second message that hashes to the same value is of the order of 2^n executions of the hash function.

Moreover, it is infeasible to detect systematic correlations between any linear combination of input bits and any linear combination of bits of the hash result.

A hermetic hash function can be turned into a secure MAC by simply including a secret key in the message input. Its security is independent of the positions of the secret key bits in the message. It may be appended, pre-pended, inserted somewhere in the middle of the message or even be distributed over bits spread all over the message. Observe that this is not the case with hash functions of the MD4 family [8].

3.2 The PANAMA stream encryption scheme

The stream cipher is initialized by first loading the 256-bit key K, the 256-bit diversification parameter Q and performing 32 additional *blank* Pull iterations. During keystream generation an 8-word block z is delivered at the output for every iteration. In practice, the diversification parameter allows for frequent resynchronisation without the need to change the key.

The sequence diagram of the PANAMA stream encryption scheme is given in Table 2.

Time step t	Mode	Input	Output
-34	reset	–	–
-33	Push	K	–
-32	Push	Q	–
$-31, \ldots, 0$	Pull	–	–
$1, \ldots$	Pull	–	z^t

Table 2. The sequence diagram of the PANAMA stream encryption scheme.

The design goal for the PANAMA stream encryption scheme is that it should be hermetic and *K-secure* (also defined in [4]). K-security implies among other things that, given part of the keystream outputs corresponding with a given key and for chosen values of Q, the most efficient way to gain knowledge on the key or on the complementary part of the keystream output, is exhaustive key search.

4 Discussion

4.1 The state updating transformation

γ is the simplest *nonlinear shift-invariant transformation.* Its propagation properties are described in detail in [4]. In short:

- The maximum correlation between its input and output diminishes exponentially with the Hamming Weight of the output selection vectors.
- The difference propagation probability diminishes exponentially with the Hamming Weight of the input difference vectors.

The transformation θ corresponds with the multiplication by a binary polynomial modulo $1 \oplus x^{17}$. It was selected from the invertible polynomials with Hamming weight 3 on the basis of its good diffusion properties. A single input difference gives rise to three output differences. For the vast majority of input difference vectors with a small (below 32) Hamming Weight, the Hamming Weight of the corresponding output vector is about three times higher.

The cyclic shift coefficients of π, described by the simple expression in (7), form an array of 17 different constants. The word permutation factor 7 is chosen to let every component of ρ depend on 9 state bits. For the chosen π parameters it has been verified that $\rho \circ \rho$ has propagation and correlation properties that are close to optimal with respect to the space of possible π parameters. On the average, a difference in a single bit diffuses to 6 bits after one iteration, 36 bits after two, 216 after three and all over the state after 4 iterations. Since γ, θ, π and σ are all invertible, the state updating transformation ρ is invertible.

σ includes the addition of a constant to a_0 to prevent symmetric properties. For the value of the constant 00000001_{hex} was chosen for its simplicity.

4.2 The hash function

The design of the Panama hash function differs from the currently popular MD4-derived designs such as MD5 [10], SHA-1 [7] and RIPEMD-160 [5] in three important ways:

- **Parallel iteration transformation:** the MD4-derived designs have an iteration function that consists in itself of a sequence of a large number of (simple) rounds, in Panama the iteration function consists of a single (more complicated) round with a parallel structure.
- **Large Chaining State:** the MD4-derived hash functions have a chaining variable that has the same size (16 or 20 bytes) as the hash result by design, the chaining variable of the Panama hash function comprising the internal state and buffer is over 1 Kbyte.
- **Presence of final transformation:** In the MD4-derived designs, the hash result is the final value of the chaining variable. For Panama the 32 blank Pull iterations form a final transformation mapping the final value of the chaining variable (state and buffer) to the hash result.

These differences are the consequence of a difference in design strategy. In the MD4-derived hash functions the iteration transformation is designed to be collision-resistant in itself. The iteration mechanism and the fact that the hash result is the value of the chaining variable after the last message block has been hashed, assure that the resulting hash function is collision-resistant. In the Panama hash function, it is the diffusion and nonlinearity realised by the successive application of the iteration function that is expected to prevent cryptographic weaknesses.

4.3 Collision resistance

In this section we explain the difficulties faced when trying to generate collisions for Panama.

The hash result is completely determined by the final value of the chaining variable: the state a^{V+1} and buffer contents b^{V+1}. The converse is however not true, and pairs of messages may be found with different values for the final

chaining variable both consistent with the same hash result. In this case, the collision actually results from the fact that the hash result is not equal to the final hashing state. We call this a *terminal* collision. Collisions in which two different messages give rise to equal chaining variable at a certain point are called *internal*. In hash functions without a final transformation, such as MD4 and its descendents, terminal collisions cannot occur.

Generating an internal collision implies two different messages that give rise to a bitwise difference pattern in the hashing state *and* buffer that *dissolves*, i.e., that ends up being all-zero for some iteration. Realising this, while theoretically possible, is assumed to be infeasible because of the large diffusion and distributed nonlinearity of ρ combined with the fact that every message block affects the hashing state for a large number of iterations. Difference patterns in the buffer induce a so-called *differential characteristic* or equivalently *differential trail* [4] in the hashing state.

A possible strategy for finding internal collisions would be to look for an input difference pattern that can give rise to a differential trail that ends in a zero difference somewhere in the computation. Other strategies are

- finding fixed points of the Push mode or more general, fixed sequences of the Push mode,
- meet-in-the-middle attacks exploiting the invertibility of the iteration function.

These attacks have been considered in the design strategy and in the choice of the state and buffer updating transformations.

Because of the linear feedback in the buffer, a difference pattern in a single message block gives rise to an infinite difference propagation in the buffer. This is illustrated in the right-hand side of Fig. 3. Only difference patterns in the input sequence that meet a particular condition give rise to a finite difference propagation in the buffer.

The simplest difference pattern that gives rise to a finite difference propagation in the buffer is illustrated in the left-hand side of Fig. 3. All other message differences that give rise to a finite difference propagation in the buffer are superpositions (linear combinations) of shifted (in time and space) instances of this difference pattern.

Even the simplest difference pattern affects the state a during 5 different iterations, spread over a range of 32 iterations. Fig. 4 illustrates the difference propagation in σ resulting from a similar difference pattern in p that gives rise to a finite difference propagation and that has $p^{1'} = d_0, d_1, \ldots, d_7$. Four of the five difference vectors are in general different. For all other message difference patterns, the number of iterations with a non-zero difference pattern and the number of distinct difference vectors in σ are both larger.

If the difference pattern in the hashing state and/or buffer is not equal to zero after the loading of the last input block, the diffusion and nonlinearity of the transformation formed by the 32 final Pull iterations are assumed to make the controlled generation of terminal collisions infeasible.

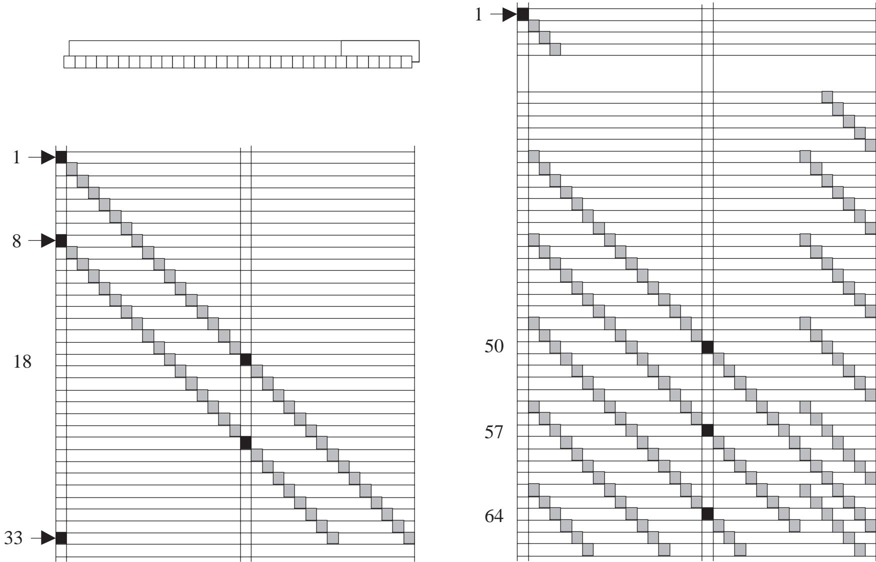

Fig. 3. Difference propagation in the buffer of PANAMA.

$t = 1$	-	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	-	-	-	-	-	-	-	-
$t = 8$	-	d_2	d_3	d_4	d_5	d_6	d_7	d_0	d_1	-	-	-	-	-	-	-	-
$t = 18$	-	-	-	-	-	-	-	-	-	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
$t = 25$	-	-	-	-	-	-	-	-	-	d_2	d_3	d_4	d_5	d_6	d_7	d_0	d_1
$t = 33$	-	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	-	-	-	-	-	-	-	-

Fig. 4. Difference patterns in σ.

In internal collisions the difference pattern in the state and buffer is zero before the final hashing state has been reached. Clearly, this restricts the difference pattern in the input sequence to superpositions of instances of the simple difference pattern given in the left-hand side of Fig. 3. These input difference patterns give rise to non-zero difference vectors in σ over a span of at least 32 iterations. We believe that controlling the differential trails in the hashing state over these large numbers of iterations to obtain a collision is too hard to be a threat to the security of the hash function.

One can avoid the need for this type of control over large numbers of iterations by eliminating differences in the state immediately after they have been created. As can be seen in Figure 4, this must however be repeated for 5 different instances, with the same set of input differences.

4.4 The stream encryption scheme

For every Pull iteration 16 words of the buffer are injected into the state and 8 state words are given at the output. In the short term, the number of buffer bits that are injected into the state is twice as large as the number of bits that are given at the output. It can be checked that this is the case for any number of iterations smaller than 12. The feedback from the state to the buffer causes the buffer contents to be renewed every 32 iterations. These factors cause the correlations between output bits and linear combinations of state and buffer bits to be too small to be of practical use for cryptanalysis.

Resynchronization attacks [4] should be made infeasible by the 32 blank Pull iterations after loading the initialization blocks. Because of the feedback from the state to the buffer in the Pull mode, the state and almost all buffer stages depend in a complicated way on these blocks at the end of the initialization phase. We expect that there are no exploitable 32-round differential trails.

5 Implementation aspects

PANAMA's heavy reliance on bitwise-logical operations on 32-bit words make it well suited to implementation on 8, 16, or 32-bit processors, except that its use of 32-bit rotations does somewhat favor 32-bit architectures.

Accesses to buffer b and operations between buffer b and state a can all be performed 64-bits at a time on processors that support this word size. Since a is not an integer multiple of 64-bits a dummy 32-bit word should be pre-pended to a_0 for properly aligning $\{a_1, a_2\}$ through $\{a_{15}, a_{16}\}$ to 64-bit boundaries. Buffer b should also be 64-bit aligned.

So, θ and γ can be performed efficiently with word sizes up to 32-bits, while π especially favors 32-bit architectures. σ and λ can both be performed efficiently with word sizes up to 64-bits. In the following analysis we concentrate on PANAMA's performance on 32-bit processors.

5.1 Theoretical performance limits

To determine the maximum theoretical performance of Panama in its hash and stream cipher modes on a suitably parallel processor architecture we seek to identify the software *critical path* through the algorithm. This is the closed path through the algorithm's computational flowgraph that has the largest *weighted* instruction count, the weighting being the number of cycles of latency associated with each type of instruction.

For instance, on most processors the result of a simple operation like an addition or XOR can be used in the subsequent cycle - these instructions are said to have a one cycle latency. On modern high performance processors it is also common for shifts and rotates to be single-cycle instructions. However reading from memory takes several cycles. Even when the data is in the CPU's local cache it commonly suffers a two or three cycle latency on modern deeply pipelined processors.

We start by examining the software critical path through ρ. For each of γ, π, θ, and σ, all seventeen 32-bit words a_0 to a_{16} can in principle be updated in parallel. The software critical path through these four routines is a total of 7 cycles (3 cycles for γ, 1 cycle for π, 2 cycles for θ, and 1 cycle for ρ), all corresponding to single-cycle instructions.

Many RISC processors have enough registers to hold state a in its entirety, so that when encrypting or hashing a long block of data we need not keep accessing any of a_0 through a_{16} from memory. However, buffer b is too large to be register based and is most efficiently implemented as a fixed circular-buffer in memory, with moving pointers used to create the appearance of a shift-register.

Notably, since the accesses to b are not data-dependent (i.e. not table-look-ups) all address calculations can be done well in advance and do not contribute to the software critical path. Also, since the stages which are read are several stages delayed from those that are written, the read-data can in principle be fetched one or more update cycle ahead of time, from which it becomes clear that updating the buffer is not on the software critical path.

We now explore the number of 32-bit instructions needed for each iteration of Push and Pull. If state a is fully held in registers and the rotation amounts in π are all hard-coded, then ρ entails a total of 16 reads (from buffer stages 4 and 16 for Pull, or input p and buffer stage 16 for Push) and 17×7 logical operations (actually one less than this because the zero-rotation need not be performed).

Updating buffer b involves 16 reads (buffer stages 24 and 31), 16 XOR operations, and 16 writes (buffer stages 0 and 25), plus a number of operations to update the pointers to the accessed stages in order to simulate a shift register. As a minimum these involve an increment and mask operation per pointer. Reading from stage 31 and writing to stage 0 takes only one pointer, likewise for stages 24 and 25 because of the way the circular-buffer is used to emulate a shift-register. Consequently there are four pointers to be updated for a Pull iteration, and three for a Push. The masking operation implements circular arithmetic on the pointers - a convenience arising from the buffer size being a power of two.

For each Pull iteration, applying the cipher output to the data stream involves 8 reads from the plaintext buffer, 8 XOR operations, and 8 writes to the ciphertext buffer, plus at least one additional instruction to update a pointer to these buffers. Each iteration ciphers 32 bytes of data.

In the case of Push we have already accounted for reading the input p under our discussion of ρ, so all that is left is updating the pointer to the input data.

Thus, ignoring for the moment the few extra instructions necessary for maintaining the loop, we have a workload of 189 instructions for each iteration of Push and 215 instructions for each Pull. This is equivalent to about 5.9 instructions per byte hashed or 6.7 instructions per byte enciphered.

An estimate of how many fully pipelined execution units the algorithm might usefully exploit can be obtained by dividing the total number of operations per iteration by the number of cycles in the critical path. This number ideally should be no less than the number of parallel execution paths in the target processors so that no CPU resources are left idle.

For PANAMA this works out to 189 or 215 instructions per iteration divided by 7 cycles per iteration, from which we estimate that the hashing and encryption modes might reasonably exploit processors with up to 27 and 31 parallel 32-bit datapaths respectively.

5.2 Benchmarked performance

The generous amount of parallelism in PANAMA lends itself naturally to efficient implementation on a processor capable of a high degree of instruction-level parallelism. To demonstrate the impressive throughputs achievable in such cases a highly optimized implementation was developed for the Philips TriMedia TM-1000 processor. The TriMedia processor is a Very Long Instruction Word (VLIW) CPU containing five 32-bit pipelined execution units sharing a common set of 128 32-bit registers. All five execution units can perform arithmetic and logical operations, but loads, stores, and shifts are each supported by only two of them. The two execution units that support shifts are distinct from the two that support loads and stores. Given an appropriate instruction mix the processor can issue up to five instructions per clock cycle.

The operations in PANAMA can be efficiently expressed in C-code except for the bitwise rotations. The optimized implementation was written completely in C-code except for resorting to a library call to access the processor's native 32-bit rotate instruction.

Since the parallelism present in the PANAMA algorithm is vastly more than is available in the TriMedia CPU we would hope to be able to completely saturate the processor, i.e. have very few vacant instruction slots. However, there may be other resource constraints that prevent this. For instance, π calls for the intensive use of rotate instructions, of which the TriMedia CPU can only issue two per cycle. Filling the other three instruction slots on each cycle requires overlapping the execution of π with that of γ and/or θ. For the benchmarked implementation each of these routines was expressed as fully unrolled in-line code, leaving the

TriMedia C-compiler to recognize and exploit the allowable overlap between γ, π, and θ.

The loop for iterating Pull mode compiled into 234 TriMedia assembly instructions. The difference between this and the 215 instructions previously counted comes from the instructions for maintaining the loop and from some additional overhead involved in the pointer updates associated with making the circular buffer appear as an LFSR. The scheduled code was tightly packed by the compiler into 47 instruction-issue cycles, i.e. a sustained 4.98 instructions per cycle were scheduled for issue, out of a theoretical maximum of 5. This is an unusually high utilization of the TriMedia CPU even compared to its efficiency on media-processing tasks for which it was designed.

Compiled code for Push showed comparable overhead and code density.

The optimized C-code was benchmarked on a 100 MHz TriMedia processor by encrypting or hashing a 128 Kbyte data buffer. We choose this buffer size as several times larger than the on-chip data cache so as to make the reported performance be representative of the sustainable encryption or hashing performance to external memory, in this case comprised of synchronous DRAM. At the level of performance achieved by PANAMA external memory bandwidth can become a significant factor in the overall performance. For the encryption benchmark the data buffer was encrypted in-place so as to minimize the performance loss arising from memory accesses. No off-chip cache was present (the Trimedia chip does not actually support off-chip cache).

An encryption throughput of 4.7 bits per cycle was achieved, equivalent to 1.7 cycles per byte, or 470 Mbps on a 100 MHz processor. This includes all loop-overhead and cycles lost to cache misses, memory accesses, etc. This is uncommonly fast among stream ciphers. For comparison, two other acknowledged fast software ciphers – RC4 [12], and SEAL [11], are reported as capable of 10.6 cycles per byte and 3.5 cycles per byte, or 75 Mbps and 230 Mbps respectively when benchmarked under these same conditions [3]. PANAMA is also slightly faster on this processor than the variants of WAKE described in [3].

Notably, PANAMA achieves its speed advantage not by having the lowest work-factor among these ciphers. For instance SEAL has a work factor of 4.25 instructions per byte on the TriMedia processor compared to PANAMA's 6.7. Rather, PANAMA's speed advantage comes from the substantial degree of parallelism present in the algorithm, an attribute that can be well exploited by a VLIW processor such as the TriMedia. Accordingly it should be noted that PANAMA's advantage may be diminished when running on processors having less instruction-level parallelism than the CPU reported here.

On the TriMedia processor PANAMA achieves a hashing throughput of 5.1 bits per cycle, equivalent to 1.6 cycles per byte, or 510 Mbps on a 100 MHz device. We are not aware of published performance figures for implementations of other currently popular hash functions on the TriMedia processor against which to directly compare PANAMA's speed. Still, a simple comparison shows that the per-byte workload of PANAMA is similar to that of MD4 [9], the fastest member of the family of hash functions to which MD5, SHA-1 and RIPEMD-160 all

belong. Benchmarks for these popular hashes have been published for the Intel Pentium processor in [2] from which we can make comparisons to Panama.

Performance of an optimized C-code implementation of Panama on a 200 MHz Pentium Pro (using a library function for rotate) was measured at 198 Mbps for ciphering and 214 Mbps for hashing, i.e. a throughput of 0.99 bits per cycle for ciphering and 1.07 bits per cycle for hashing. This compares to hashing speeds reported in [2] for SHA-1 and RIPEMD-160 of 0.24 bits per cycle and 0.21 bits per cycle respectively for optimized C-code[1]. [2] also reports optimized assembly-code versions of SHA-1 and RIPEMD-160 as achieving 0.54 bits per cycle and 0.44 bits per cycle respectively. It is currently unknown what further speed improvement could be achieved for Panama by assembly coding it, but even without such improvement it shows about a 2× speed advantage over assembly coded versions of these other hashes.

Since the Pentium Pro can in principle issue two arithmetic or logical instructions per cycle compared to five for the TriMedia chip one may wonder why the throughput per cycle of Panama on the Pentium Pro is barely one fifth that achieved on the TM-1000. In part the reason is that Panama's large state cannot be maintained in the small register set of the 'x86 architecture, with the result that code for the Pentium Pro requires massively more load and store instructions than are required for the TriMedia, or for that matter other RISC processors with a generous complement of registers. Since both SHA-1 and RIPEMD-160 are substantially unhampered by the limited register set of the 'x86 architecture, we would expect Panama's advantage over them to be all the greater on processors not having this limitation.

In considering Panama's suitability to an application it should be borne in mind that the performance figures reported are for large block sizes. When hashing small blocks, or encrypting with frequent key changes or resynchronization, the overhead of the accompanying 32 blank Pull iterations may significantly impact the performance. A key change or resynchronization takes about as long as encrypting 1000 bytes. Similarly, each hashed block has a fixed overhead equivalent to hashing about 1000 bytes.

6 Conclusions

We have presented a new cryptographic module capable of cryptographic hashing and stream encryption suited for applications where large amounts of data must be protected. It has been shown that the inherent parallelism allows extremely fast software implementations on VLIW processors.

[1] [2] reports performance on a 90 MHz Pentium, while here we report performance on a 200 MHz Pentium Pro. By converting all results into the normalized measure of bits per cycle we attempt to provide a uniform basis for comparison, however the reader is cautioned that no allowance has been made for the architectural differences between the Pentium and Pentium Pro.

References

1. E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems," *Journal of Cryptology*, Vol. 4, No. 1, 1991, pp. 3–72.
2. A. Bosselaers, R. Govaerts, J. Vandewalle, "Fast Hashing on the Pentium", *Advances in Cryptology – Proceedings Crypto'96 LNCS 1109*, N. Koblitz, Ed., Springer-Verlag, 1996, pp. 298–312.
3. C.S.K. Clapp, "Optimizing a fast stream cipher for VLIW, SIMD, and superscalar processors," *Fast Software Encryption, LNCS 1267*, E. Biham, Ed., Springer-Verlag, 1997, pp. 273–287.
4. J. Daemen, "Cipher and hash function design strategies based on linear and differential cryptanalysis," *Doctoral Dissertation*, March 1995, K.U.Leuven.
5. H. Dobbertin, A. Bosselaers, B. Preneel, "RIPEMD-160: A Strengthened version of RIPEMD," *Fast Software Encryption, LNCS 1039*, D. Gollmann, Ed., Springer-Verlag, 1996, pp. 71–82.
6. FIPS 180, *Secure Hash Standard*, Federal Information Processing Standard (FIPS), Publication 180, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., May 1993.
7. FIPS 180-1, *Secure Hash Standard*, Federal Information Processing Standard (FIPS), Publication 180-1, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., April 1995.
8. B. Preneel and P.C. van Oorschot, "On the Security of Two MAC Algorithms", *Advances in Cryptology – Proceedings Eurocrypt'96 LNCS 1070*, U.M. Maurer, Ed., Springer-Verlag, 1996, pp. 19–32.
9. R.L. Rivest, *The MD4 message-digest algorithm*, Request for comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force, April 1992.
10. R.L. Rivest, *The MD5 message-digest algorithm*, Request for comments (RFC) 1321, Internet Activities Board, Internet Privacy Task Force, April 1992.
11. P. Rogaway and D. Coppersmith, "A Software-Optimized Encryption Algorithm," *Fast Software Encryption, LNCS 809*, R. Anderson, Ed., Springer-Verlag, 1994, pp. 56–63.
12. B. Schneier, *Applied Cryptography, Second Edition*, John Wiley & Sons, 1996, pp. 397–398.