Abacus: A Candidate for SHA-3

Neil Sholer

WaveStrong, Inc.
neil@wavestrong.com

Version 1.0 - October 29th, 2008

Abstract. In this paper I present Abacus, a candidate for SHA-3. With a design heavily influenced by stream ciphers, Abacus is a byte-oriented cryptographic sponge that digests a message in one-byte blocks and outputs a hash of any byte length. Abacus uses three rolling arrays, a 4-byte MDS matrix, and an 8-bit S-box. Together, these three features provide excellent security. Becaus Abacus does not use the Merkle-Damgård construction, it is resistant to the attacks that plague the MD5/SHA family. Abacus's simple design and modest processing requirements make it well-suited for many platforms, from hardware to smartcards to software. Initial performance estimates are 25 cycles/byte in an optimized software implementation.

Table of Contents

1 Introduction

Over the last five years, a series of devastating breaks on MD4 [65], MD5 [45, 71], SHA-0 [8, 67], and to a lesser extent SHA-1 [9, 66], have catapulted hash functions into the limelight. Cryptographers all the way from Belgium to China, from academia to NIST, are asking themselves existential questions: How well do we understand hash functions? What makes a hash function strong? What properties should a hash function even have? [42, 61] Previously the undervalued workhorses of cryptography, hash functions have become cryptography's hottest—and most pressing—topic.

In November of 2007, NIST announced a public contest to design the next Secure Hash Standard, to be known as SHA-3 [32]. The candidate algorithm is required to be a drop-in replacement for SHA-2 [28], meaning that it takes the same inputs as SHA-2, and produces message digests of 224, 256, 384, and 512 bits. Not only is the candidate algorithm expected to be at least as secure as SHA-2, it should be faster. NIST further recommends that the candidate algorithm support randomized hashing, have tunable parameters to allow for speed/security tradeoffs, and be implementable on platforms ranging from resource constrained satellites to high-end 64-bit microprocessors.

In this paper I present Abacus, a new hash function submitted to NIST for consideration as SHA-3. Abacus was designed to resemble a stream cipher. While clearly descending from RC4 [64], Py [11], and SNOW [26], Abacus also incorporates design elements found in AES [24, 30], RadioGatún [6], and Mars [16].

Abacus hashes in streaming mode (i.e. one pass), digesting an arbitrary sized message in blocks of 8-bits. Similarly, the variable length hash output is generated one byte at a time. Abacus has a large internal state of 1056 bits that evolves as each input byte or output byte is processed. Before the hash output is generated, there is a series of blank rounds to increase the avalanche effect. This is a relatively new hash construction called a cryptographic sponge [5].

Simple and easy-to-understand, Abacus's pseudocode can fit on half a page. Abacus has a small instruction set consisting of exclusive or (XOR), addition, and table lookups. All operations are on bytes. While many hash functions gain their non-linearity by combining XOR with modular addition [27, 28, 62], Abacus uses an 8-bit-by-8-bit S-box to provide a known, provable level of non-linearity and local diffusion.

Like SNOW [26] and AES [24, 30], Abacus uses a maximum distance separable matrix (MDS) to achieve optimal diffusion across four bytes. Within one round, an input difference of one byte is guaranteed to alter four bytes widely spaced across the internal state. This property makes it difficult for an attacker to predict or control the internal state.

Abacus's internal state is comprised of a chaining variable, and three rolling arrays of lengths 5, 37, and 89 bytes. First introduced in Py [11], rolling arrays shift to the left by one position after each clock. Combined with four relatively-prime counters, the rolling arrays guarantee a minimum cycle length of 2^{45}, thereby making cyclical

attacks on the internal state infeasible and assuring the hash output is cycle-free for any practical length.

With its simple design, small instruction set, and modest resource requirements, Abacus is well-suited for smartcards, embedded hardware, 8-bit processors, and software running on 32- or 64-bit processors. Initial estimates show that Abacus can absorb data at about 25 cycles per byte, and can be represented in hardware using about 7,500 gates.

For a hash length of n bits, I claim that Abacus has preimage resistance of 2^n, second preimage resistance of 2^n, and collision resistance of $2^{n/2}$. Furthermore, Abacus is resistant to the local collisions [17], fixed points [25], multicollisions[40], length extension attacks [52], herding attacks [43], neutral bit attacks [8, 9, 70], and second preimage attacks [44, 52] that exploit the Merkle-Damgård construction.

I believe that Abacus meets and exceeds all the requirements for SHA-3.

1.1 Hash Function Basics

This section describes the basics of a generic hash function, and what properties one should have. For those readers who already know what a hash function is, feel free to skip ahead to section 2. Also, there is a glossary in appendix A.1 that defines many of the cryptographic and mathematical terms used throughout this paper.

Broadly speaking, a hash function is a chaotic mathematical function. Two very similar inputs will produce two very different outputs. Despite the appearance of randomness, however, a hash function is deterministic: the same input will always produce the same output.

Hash functions are useful to provide authentication and data integrity. Some of the most popular uses of a hash function are: creating a cryptographic checksum for a file; securely signing a digital document; storing a hashed version of a user's password so that the actual password remains secret; generating random numbers; and proving knowledge of a secret without revealing that secret.

A hash function can be thought of as a digital fingerprint. A hash function takes a message (or file) of nearly any length, digests it, and then outputs a fixed size hash (or fingerprint). In the same way that it is virtually impossible to find two people with the same fingerprint, it should be virtually impossible to find two messages with the same hash.

After a little bit of thought, it's clear that whenever the message length is longer than the hash length, there must be some cases where two different messages would have the same hash. This is called a collision. And in fact, the pigeonhole principle [2] guarantees that collisions will occur whenever the message length exceeds the hash length. Not to worry – as long as the hash function meets certain conditions [63], the odds of *finding* such a collision are astronomically small. These conditions are:

Hash Function Conditions - Informal

Preimage Resistance	It should be very hard to find any message that maps to a pre-determined hash value
Second Preimage Resistance	It should be very hard to find a second message that maps to the same hash value as a pre-determined first message
Collision Resistance	It should be very hard to find any two messages that map to the same hash value (regardless of what that hash value is)

Hash Function Conditions - Formal

Preimage Resistance	Given a hash value H, it is computationally infeasible to find a message M, such that Digest(M) = H
Second Preimage Resistance	Given a message M1, it is computationally infeasible to find a different message M2, such that Digest(M1) = Digest(M2)
Collision Resistance	It is computationally infeasible to find two different messages, M1 and M2, such that Digest(M1) = Digest(M2)

Informally, "computationally infeasible" means that it would take an unrealistic amount of time, space, money, or energy to accomplish a given task. Of course, different people (and different organizations) have different amounts of these resources. In 2006, Bart Preneel [61] estimated that a task of size 2^{74} would take a pedestrian hacker one year to complete. An intelligence agency, however, could complete a task of size 2^{100} in one year. (These estimates refer to second preimage attacks.) Keep in mind that Moore's Law predicts that computational speeds will double roughly every two years, so practically speaking, computationaly infeasibility is a moving target.

But formally speaking, it is possible to define computationally infeasible in a less subjective way. Assume that the hash output length is *n* bits. It should take $\mathbf{O}(2^n)$ steps to break the preimage resistance condition. The second preimage resistance condition should also take $\mathbf{O}(2^n)$ steps to break. And because of the birthday paradox [64], it is easier to break the collision resistance condition: it should take $\mathbf{O}(2^{n/2})$ steps.

Due to recent advance in the cryptanalysis of hash functions, some cryptographers are hypothesizing that the second preimage condition should be quite a bit lower than $\mathbf{O}(2^n)$, and perhaps depend on the message size [43, 44].

Please refer to [63] and [52] for a more thorough discussion of these conditions. For a nice discussion of computational complexity theory and Big Oh notation (asymptotic notation), refer to chapter 2 of [72].

2 Algorithm Specification

Throughout the rest of this paper, various symbols, functions, and terms will be used to represent the components of the Abacus hash function. They are defined below. Some readers may also want to refer to Appendix A.1 for a general list of math and crypto definitions.

Internal State

`ra`	Register A. The chaining variable byte.
`rb`	Register B. A rolling array of size 5 bytes.
`rc`	Register C. A rolling array of size 37 bytes.
`rd`	Register D. A rolling array of size 89 bytes.
`ctr1`	An 8-bit counter that ranges from 0 to 232
`ctr2`	An 8-bit counter that ranges from 0 to 238
`ctr3`	An 8-bit counter that ranges from 0 to 240
`ctr4`	An 8-bit counter that ranges from 0 to 250
`out`	An 8-bit output feedback variable. Data dependent.

Constants

`sbox`	An 8x8 substitution table. It is a permutation.

Functions

`mds4()`	A diffusion function that makes four bytes depend on one another. It uses a maximum distance separable matrix over **GF**(2^8).
`rotate()`	A function that rotates all of the rolling arrays to the left by one.

Algorithm Parameters

`MSG_LEN_BITS`	Length of the message to digest, in bits
`MSG_LEN_BYTES`	Length of the message to digest, in bytes. Use the ceiling of `(MSG_LEN_BITS/8)`.
`HASH_LEN_BITS`	Length of the hash output, in bits. It is always a multiple of 8.
`HASH_LEN_BYTES`	Length of the hash output, in bytes.
`NUM_ABSORB_CLOCKS`	Number of passes over each input message byte.
`NUM_SQUEEZE_CLOCKS`	Number of passes over each hash output byte.
`NUM_BLANK_ROUNDS`	Number of blank rounds after message is absorbed.

Terms

Absorb Phase	The input message is digested in one-byte chunks during the absorb phase.
Squeeze Phase	During the squeeze phase, the hash value is generated one byte at a time.
Blank Round	Blank rounds continue evolving the internal state by digesting a pre-determined number of "blank" bytes (i.e. null bytes).
Tap Position	A position in a rolling array that is "tapped", meaning that the value contained there gets read. A

	tap position affects the outcome of an S-box lookup.
Update Position	A position in a rolling array that is "updated", meaning that the value contained there gets XORed by another value and then sent through an S-box.
Round	During the absorb phase, one input message byte is processed per round. Similarly, one hash output byte is generated per round in the squeeze phase. A round consists of one or more clocks.
Clock	A single increment of the Abacus internal state. In each clock, one entry in each of the four registers is updated, the registers are rotated by one byte, and the counters are incremented by 1.
PRNG	A pseudo-random number generator. Unless otherwise specified, the PRNG is assumed to be cryptographically secure.
Related Message Attack	A method of cryptanalysis in which two similar messages are digested by two separate Abacus contexts that started off with identical internal states. The objective is to find correlations in Abacus's internal state and/or its hash output. Related message attacks were directly inspired by related key attacks [7].

2.1 Abacus in Plain English

Abacus is a cryptographic sponge [5]. It has separate phases for *absorbing* the message and for *squeezing* out the hash. Both phases are quite similar, but they have some important differences.

Before absorbing or squeezing any data, the Abacus **internal state must be initialized**. The four counters (`ctr1`, `ctr2`, `ctr3`, `ctr4`) are all set to zero. The output feedback variable, `out`, is also set to zero. To initialize the rolling array registers, fill them with the first 132 entries from the S-box. In particular, copy the first entry of the S-box into the `ra` register, copy the next five entries of the S-box into the `rb` register, copy the next 37 entries of the S-box into the `rc` register, and copy the next 89 bytes of the S-box into the `rd` register.

The **absorb phase** processes one byte of the input message per round. By default, each round has one clock, but there is an option to have multiple clocks per round. For the purposes of clarity the absorb phase will be described with just one clock.

At the beginning of the clock there is a **"first pass" of S-boxes**. The value at `ra[0]` is XORed with `rd[58]`, and the result passes through the S-box to become the new `ra[0]`. The value at `rb[0]` is XORed with `rc[24]`, and the result passes through the S-box to become the new `rb[0]`. The value at `rc[0]` is XORed with `rb[3]`, and the result passes through the S-box to become the new `rc[0]`. The value at

`rd[0]` is XORed with the current message byte, and the result passes through the S-box to become the new `rd[0]`.

After the first pass of S-boxes, the **four counters are combined with the registers**. The value at `ra[0]` is XORed with `ctr1`, the value at `rb[0]` is XORed with `ctr2`, the value at `rc[0]` is XORed with `ctr3`, and the value at `rd[0]` is XORed with `ctr4`.

Then, there is an **"MDS step"** in which the bytes from `ra[0]`, `rb[0]`, `rc[0]` and `rd[0]` affect each other through a linear transformation. Technically, the linear transformation is matrix multiplication in a finite field using a primitive polynomial. More simply, the MDS step can be thought of as a series of table lookups and XORs in which each input byte is invertibly combined with every other input byte. The MDS step produces four bytes as output, which are then assigned back into `ra[0]`, `rb[0]`, `rc[0]`, and `rd[0]`.

After the MDS Step, there is a **"second pass" of S-boxes**. `ra[0]` passes through the S-box to become the new `ra[0]`, `rb[0]` passes through the S-box to become the new `rb[0]`, `rc[0]` passes through the S-box to become the new `rc[0]`, and `rd[0]` passes through the S-box to become the new `rd[0]`.

Upon completing those steps, **increment the counters**. `ctr1` is incremented by 1, modulo 233. `ctr2` is incremented by 1, modulo 239. `ctr3` is incremented by 1, modulo 241. `ctr4` is incremented by 1, modulo 251. Finally, **rotate the rolling array registers** `rb`, `rc`, and `rd` to the left by one position. (Although `ra` is also a register, it is not rotated because it is only one byte long.)

In the six paragraphs above, a single clock of the absorb phase was described. If more than one clock is required per round, simply repeat all of the steps described above.

Between the absorb phase and the squeeze phase, there are several **blank rounds**. The exact number is customizable (i.e. tunable). To perform one blank round, simply absorb a null-byte (`0x00`). For example, if 135 blank rounds are required, absorb 135 consecutive null-bytes. (Please note: if `NUM_ABSORB_CLOCKS` is greater than one, then each blank round will also have that number of clocks.)

The **squeeze phase** reuses most of the functionality of the absorb phase. It is worth explaining the differences, however. The paragraphs below will assume there is only one clock per round of the squeeze phase.

Prior to the first S-box pass, the value at `rd[0]` is XORed with the output feedback variable, `out`. The other 3 registers (`ra[0]`, `rb[0]`, `rc[0]`) are XORed with the same values that they were XORed with in the absorb phase. Then, the next several steps are the same as in the absorb phase: there is a first pass of S-boxes, followed by XORing with the counters, followed by the MDS step, followed by a second pass of S-boxes which assigns modified values back into `ra[0]`, `rb[0]`, `rc[0]`, and `rd[0]`. But at this point, the squeeze phase adds several more steps.

The values at `ra[0]` and `rb[0]` are XORed together and then pass through an S-box, values at `rc[0]` and `rd[0]` are XORed together and then pass through an S-box, and the results of both S-box lookups are XORed together to produce the new value of `out`. This new value of `out` is copied into the current hash output byte. After completing these steps, finish the clock by incrementing the counters and rotating the rolling arrays.

Like in the absorb phase, it is straightforward to perform multiple clocks per round: simply repeat all of the steps of a single squeeze clock as many times as required. (Note: Since the index of the current hash output byte cannot increase until the round is over, the current hash output byte is effectively overwritten after each clock.)

2.2 Abacus Diagrams

Figure 1: Abacus's Registers

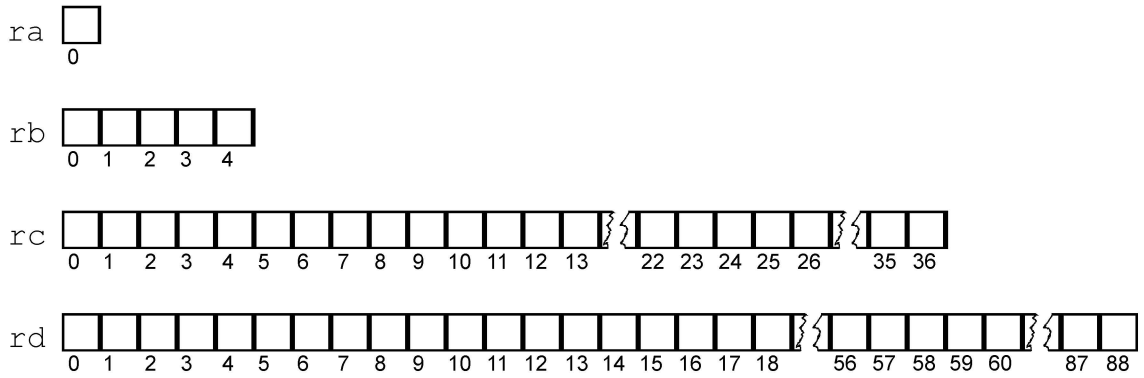
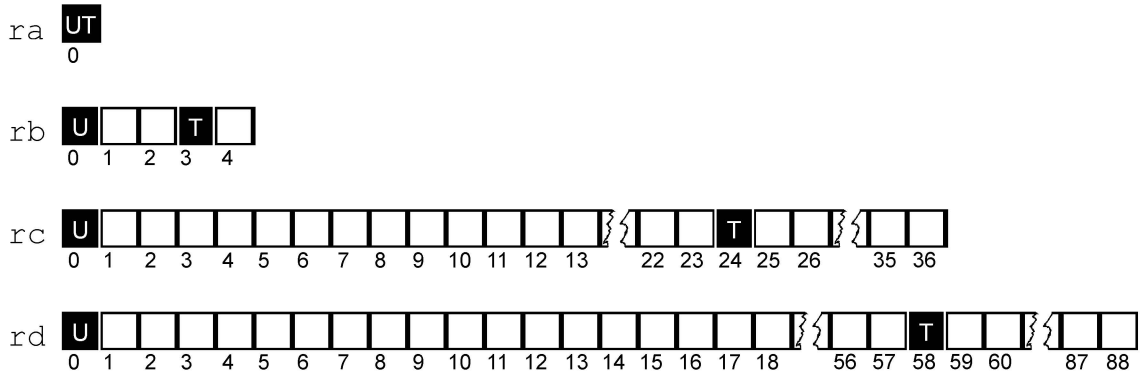


Figure 2: Abacus's Registers (Update and Tap Positions Shown)



In Figures 1 and 2, Abacus's registers are illustrated. The `ra` register is 1 byte long, the `rb` register is 5 bytes long, the `rc` register is 37 bytes long, and the `rd` register is 89 bytes long. Due to their lengths, `rc` and `rd` are drawn with jagged breaks to indicate that many bytes are not shown.

All four of the registers are updated at position 0. Since `ra` is only one byte long, it is tapped at position 0 as well. The `rb` register has a tap position at 3, the `rc` register has a tap position at 24, and the `rd` register has a tap position at 58. Because these registers are rolling arrays that rotate to the left, `rb`'s tap is 2 clocks away, `rc`'s tap is 13 clocks away, and `rd`'s tap is 31 clocks away.

Figure 3: Abacus's Absorb Phase

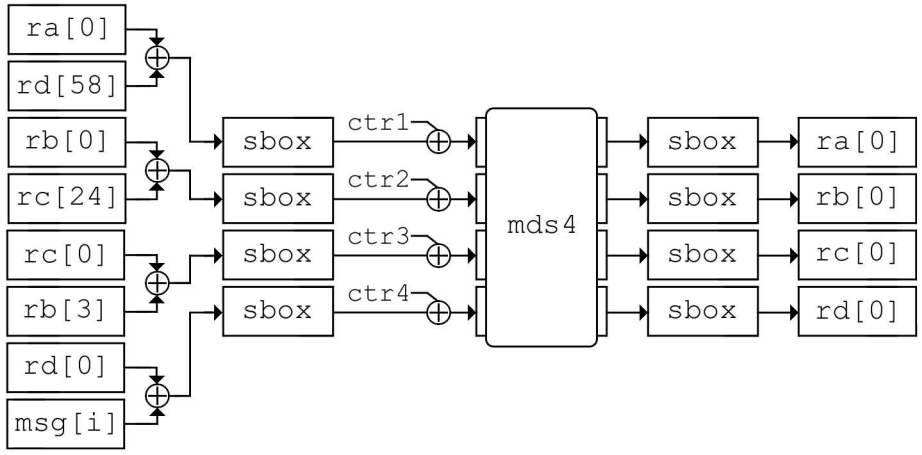
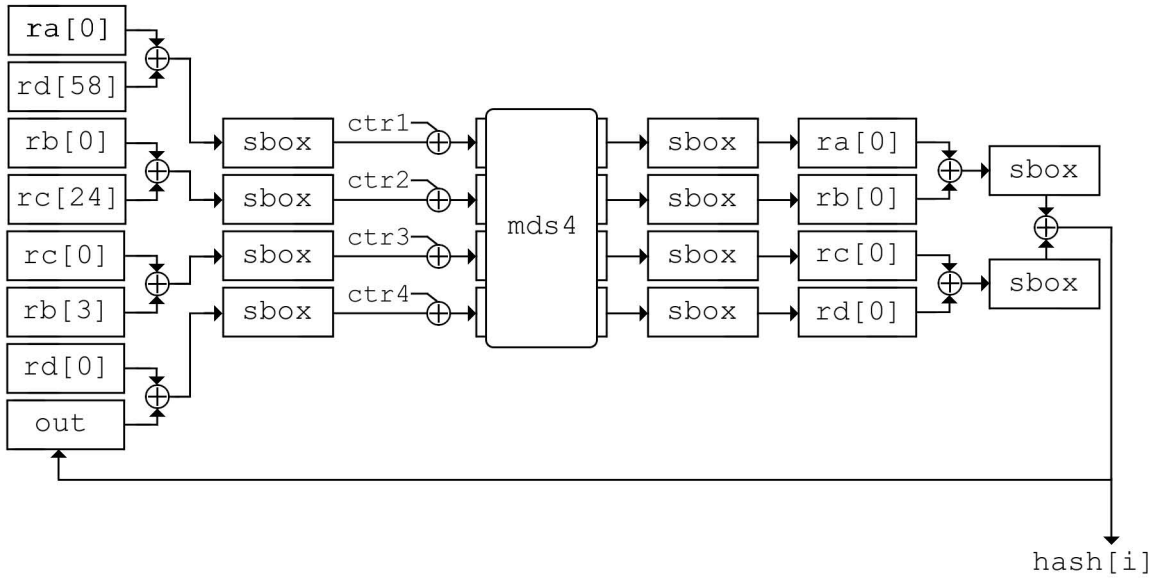


Figure 4: Abacus's Squeeze Phase



In Figure 3, one clock of Abacus's absorb phase is illustrated. `ra`, `rb`, `rc` and `rd` are the registers, `sbox` is the S-box, and `mds4` is the MDS matrix step. Also, `ctr1` to `ctr4` are the four counters. `msg[i]` is the current message block (one byte long).

In Figure 4, one clock of Abacus's squeeze phase is illustrated. `out` is the output feedback variable, and `hash[i]` is the current hash output (one byte long).

Not shown: After each clock, the counters are incremented by 1, and the registers are rotated to the left by one. Also, there is an option to have several clocks per round.

2.3 Internal State

Abacus has an internal state that consists of a chaining variable (called 'ra'), three rolling arrays called ('rb', 'rc', and 'rd'), and four counters. All of the datatypes are bytes. In the C programming language, an unsigned char is a byte. Depending on a given computer's CPU, it may be more efficient to implement these variables as 32- or 64-bit unsigned integers.

```
struct AbacusContext
{
    unsigned char ra;                      // 'ra' register
    unsigned char rb[5], rc[37], rd[89];  // 'rb', 'rc', 'rd' registers
    unsigned char ctr1, ctr2, ctr3, ctr4; // the four counters
}
```

2.4 Initialization

Prior to hashing the first block of any message, Abacus's internal state must be initialized in the following way:

- Set `ra[0]` to `sbox[0]`
- Set `rb[0 ... 4]` to `sbox[1 ... 5 ]`
- Set `rc[0 ... 36]` to `sbox[6 ... 42]`
- Set `rd[0 ... 88]` to `sbox[43 ... 131]`
- Set all counters (`ctr1`, `ctr2`, `ctr3`, `ctr4`) to zero
- Set the output feedback variable, `out`, to zero

2.5 Pseudocode: Absorb Phase

The pseudocode for the absorb phase is given below. The S-box is defined in section 2.7. The `mds4()` , `rotate()` , and `inc_ctrs()` functions are described in sections 2.8, 2.9, and 2.10 respectively. The `^` symbol denotes bitwise exclusive or.

```
For i = 0 to MSG_LEN_BYTES:
    For n = 0 to NUM_ABSORB_CLOCKS:

        ra     =  sbox[ra    ^ rd[58]]  ^  ctr1
        rb[0]  =  sbox[rb[0] ^ rc[24]]  ^  ctr2
        rc[0]  =  sbox[rc[0] ^ rb[ 3]]  ^  ctr3
        rd[0]  =  sbox[rd[0] ^ msg[i]]  ^  ctr4

        mds4(ra, rb[0], rc[0], rd[0])

        ra     =  sbox[ra   ]
        rb[0]  =  sbox[rb[0]]
        rc[0]  =  sbox[rc[0]]
        rd[0]  =  sbox[rd[0]]

        rotate()
        inc_ctrs()
```

2.6 Pseudocode: Squeeze Phase

The pseudocode for the squeeze phase is given below.

```
For i = 0 to HASH_LEN_BYTES:
    For n = 0 to NUM_SQUEEZE_CLOCKS:

        ra     =  sbox[ra    ^ rd[58]]  ^  ctr1
        rb[0]  =  sbox[rb[0] ^ rc[24]]  ^  ctr2
        rc[0]  =  sbox[rc[0] ^ rb[ 3]]  ^  ctr3
        rd[0]  =  sbox[rd[0] ^ out   ]  ^  ctr4

        mds4(ra, rb[0], rc[0], rd[0])

        ra     =  sbox[ra   ]
        rb[0]  =  sbox[rb[0]]
        rc[0]  =  sbox[rc[0]]
        rd[0]  =  sbox[rd[0]]

        out = sbox[ra ^ rb[0]]  ^  sbox[rc[0] ^ rd[0]]
        hash[i] = out

        rotate()
        inc_ctrs()
```

2.7 The S-box

The Abacus S-box, listed below, has 8 input bits and 8 output bits. The values are given in hexadecimal. The most significant bit occupies the leftmost position of the byte. The S-box is read from left to right and top to bottom. (For example, the entry at `02` is `f0`, and the entry at `7e` is `33`.)

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	e3	84	f0	d6	f9	f6	be	90	85	7d	28	43	12	c0	e1	b4
10	55	c7	8c	87	42	e0	d9	27	78	ec	cb	07	aa	95	c1	3f
20	b2	dc	26	a7	1f	df	f3	54	d2	e7	24	3e	32	d1	56	c6
30	35	73	f7	7b	62	29	52	80	a9	ba	ab	e9	02	53	6a	e4
40	67	a0	8e	fb	9a	79	4e	8d	e5	4a	41	af	5a	5c	a6	6b
50	16	5e	e8	3c	9c	5b	88	76	15	f4	60	bd	83	98	8f	c8
60	09	68	0d	18	65	45	04	ce	7a	f2	39	c5	9e	f1	17	ef
70	38	21	94	86	69	37	f5	ed	36	66	cf	3b	63	4b	33	b6
80	ff	bc	11	5d	b3	2b	d3	d0	3a	96	77	7c	1c	c2	fe	0a
90	c3	25	4d	fc	89	de	30	23	64	81	d5	ae	70	db	e6	7e
a0	b0	6f	0f	d7	bf	9b	c4	74	b7	57	4f	58	10	2d	a4	b9
b0	a2	ad	61	eb	ac	1a	a3	d8	2c	5f	91	2f	72	31	b1	82
c0	49	da	0c	ca	00	a1	b5	75	6e	47	6d	13	19	93	20	05
d0	01	9f	1d	44	8a	1e	50	34	fa	9d	a8	8b	0b	4c	a5	2e
e0	71	f8	40	cd	99	fd	51	59	0e	2a	3d	92	14	48	6c	ea
f0	46	22	cc	06	d4	97	e2	1b	dd	7f	bb	c9	b8	03	ee	08

2.8 The `mds4()` Function

The `mds4()` function takes four input bytes and modifies them. Specifically, all four input bytes affect one another through an invertible linear transform. Abacus uses the same finite field as AES (aka Rijndael), and very similar MDS matrix. For a thorough discussion of finite fields and MDS matrices, refer to the MixColumns step of the Advanced Encryption Standard (AES) [24, 30].

Here's a summary of how the MixColumns step works [68]: "Each column [i.e. byte] is treated as a polynomial over **GF**(2^8) and is then multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = x^3 + 3x^2 + 2x + 1$. The MixColumns step can also be viewed as a multiplication by a particular MDS matrix in Rijndael's finite field." (I modified the quote slightly so that the $c(x)$ polynomial is the same as Abacus's.)

Rijndael's finite field is: $x^8 + x^4 + x^3 + x + 1$
In hexadecimal, that finite field is: 0x1b

The MDS matrix is:

$$\begin{bmatrix} 1 & 1 & 3 & 2 \\ 2 & 1 & 1 & 3 \\ 3 & 2 & 1 & 1 \\ 1 & 3 & 2 & 1 \end{bmatrix}$$

Source code in the C programming language, shown below, demonstrates how to code the `mds4()` function. Although easy to read, this source code is not optimized. There are much more efficient ways to write this routine. Furthermore, the particular source code shown below is vulnerable to timing attacks because it uses conditional logic.

```
void mds4(unsigned char *r)// Takes an array of 4 bytes as input
{
    unsigned char a[4];    // The array 'a' is a copy of input array 'r'
    unsigned char b[4];    // 'a' multiplied by 2 in the finite field
    unsigned char c[4];    // 'a' multiplied by 3 in the finite field
    unsigned char i;       // A counter from 0 to 3
    unsigned char h;       // Temp byte; remembers if there's a high bit

    for(i = 0; i < 4; i++) {
        a[i] = r[i];
        h = r[i] & 0x80;    // High bit
        b[i] = r[i] << 1;   // Left shift by 1 is the same as mult by 2
        if(h == 0x80)       // But if there was a high bit, XOR by 0x1b
            b[i] ^= 0x1b;   // 0x1b is the finite field
        c[i] = b[i] ^ a[i]; // Mult by 3 is equal to (b[i] ^ a[i])
    }

    r[0] = a[0] ^ b[3] ^ c[2] ^ a[1];  // a[0] + 2*a[3] + 3*a[2] + a[1]
    r[1] = a[1] ^ b[0] ^ c[3] ^ a[2];  // a[1] + 2*a[0] + 3*a[3] + a[2]
    r[2] = a[2] ^ b[1] ^ c[0] ^ a[3];  // a[2] + 2*a[1] + 3*a[0] + a[3]
    r[3] = a[3] ^ b[2] ^ c[1] ^ a[0];  // a[3] + 2*a[2] + 3*a[1] + a[0]
}
```

(This source code originally appeared, in a different form, on Wikipedia [68]. Thanks to the original author.)

2.9 The `rotate()` Function

At the heart of any rolling array is the `rotate()` function [11]. Once per round, the array is rotated to the left by one, and the previous leftmost byte becomes the new rightmost byte. Another way of looking at the `rotate()` function is: the oldest byte drops off and reappears at the opposite end of the array as the newest byte. An example of rotating a rolling array with 6 elements is shown below.

```
19  72  31  96  52  40   (Round 0)
72  31  96  52  40  19   (Round 1)
31  96  52  40  19  72   (Round 2)
96  52  40  19  72  31   (Round 3)
                 (etc.)
```

Below, the `rotate()` function is demonstrated in C source code. Please keep in mind that this source code is only meant to be easy to read. Alternate implementations, such as those that use large arrays and increment the array pointer like a "caterpillar" [11], are much more efficient.

```
void rotate(AbacusContext* ctx)
{
    unsigned char tmp;   // Temp holding variable for wrap-around byte
    unsigned char i;     // Loop counter

    // Rotate the 'rb' rolling array
    tmp = ctx->rb[0];
    for (i = 1; i < 5; i++) { ctx->rb[i-1] = ctx->rb[i]; }
    ctx->rb[4] = tmp;

    // Rotate the 'rc' rolling array
    tmp = ctx->rc[0];
    for (i = 1; i < 37; i++) { ctx->rc[i-1] = ctx->rc[i]; }
    ctx->rc[36] = tmp;

    // Rotate the 'rd' rolling array
    tmp = ctx->rd[0];
    for (i = 1; i < 89; i++)  { ctx->rd[i-1] = ctx->rd[i]; }
    ctx->rd[88] = tmp;
}
```

2.10 The `inc_ctrs()` Function

During the `inc_ctrs()` function, all four of the counters are incremented by 1. This is modular addition, so if any given counter is larger than its maximum, it wraps around to zero.

The source code shown below is easy to understand but not very efficient. The `%` symbol denotes the modulo operator. In general, the modulo operator is slow because it is based on division. See section 8.4 for a more efficient implementation of the `inc_ctrs()` function.

```
void inc_ctrs(AbacusContext* ctx)
{
    ctx->ctr1 = (ctx->ctr1 + 1) % 233;
    ctx->ctr2 = (ctx->ctr2 + 1) % 239;
    ctx->ctr3 = (ctx->ctr3 + 1) % 241;
    ctx->ctr4 = (ctx->ctr4 + 1) % 251;
}
```

2.11 Tunable Parameters

Abacus has three tunable parameters. They are:

`NUM_BLANK_ROUNDS`. The number of blank rounds that occur after the absorb phase but before the squeeze phase. The recommended value is 135. Other logical values are 90 and 180.

NUM_ABSORB_CLOCKS. The number of passes that are done on each input byte. The recommended value is 1, but other reasonable values would be 2 and 4. In each pass, a single input byte affects 4 bytes of the internal state. So with 4 absorb passes, 16 bytes of internal state will be affected by one input byte. (But in this example, the number of distinct internal state bytes that are affected is lower than 16 because several bytes in the `fsm` will be altered more than once.)

NUM_SQUEEZE_CLOCKS. The number of passes that are done on each output byte. The recommended value is 1, but other reasonable values would be 2 and 4. For values greater than 1, the output function will drop `(NUM_SQUEEZE_CLOCKS-1)` bytes for every byte it keeps. This effectively truncates the output, making it lossy. And since the output feedback is done after each pass, setting `NUM_SQUEEZE_CLOCKS` to 2 or 4 requires the attacker has to guess one or more output feedback bytes.

There is no limit to how high these tunable parameters can be. `NUM_BLANK_ROUNDS` can be any non-negative integer value. `NUM_ABSORB_CLOCKS` and `NUM_SQUEEZE_CLOCKS` can be any integer value greater than 0. But because increasing any of these tunable parameters entails more work while computing the hash, common sense should be used. In general, I don't recommend setting `NUM_BLANK_ROUNDS` above 360, and I don't recommend setting `NUM_ABSORB_CLOCKS` or `NUM_SQUEEZE_CLOCKS` above 8.

For the purposes of analyzing a reduced-strength version of Abacus, `NUM_BLANK_ROUNDS` may be set to 0, 5, or 37.

2.12 Message Padding

If the message length, in bits, isn't a multiple of 8, simply append a series '0' bits at the end of the message until the message length *is* a multiple of 8.

The message length, hash length, and salt (i.e. randomized or arbitrary bit sequence), are all concatenated together to form a "train". Then, this train is prepended and appended to the overall message.

Listed below are the properties of each field in the train. The ‖ symbol denotes concatenation of bit sequences.

`MSG_LEN_BITS`	A 128-bit value. The most significant bit is the leftmost bit.
`HASH_LEN_BITS`	A 32-bit value. The most significant bit is the leftmost bit.
`SALT`	A variable length bit sequence that allows for randomized hashing. By default, the salt is assumed to be 0 bits, which is equivalent to not using a salt. The length of the salt, in bits, may be any non-negative integer that is divisible by 8.

The train can be represented graphically as:

```
( SALT ‖ HASH_LEN_BITS ‖ MSG_LEN_BITS )
```

When the message is prepended and appended with the train, it looks like:

```
( TRAIN ‖ MSG ‖ TRAIN )
```

But in some applications, the message length is not known until the entire message has been digested. For instance, SHA-2 [28], as well as the SHA-3 crypto API [57] both assume that the message length is unknown until the very last block has been digested. Abacus can accommodate this by setting `MSG_LEN_BITS` to 0 in the train that is prepended to the overall message. An example of this is shown below.

`NULL_128` is simply 128 '0' bits. The prepended train can be represented graphically as:

```
PTRAIN = ( SALT ‖ HASH_LEN_BITS ‖ NULL_128 )
```

And the appended train can be represented graphically as:

```
ATRAIN = ( SALT ‖ HASH_LEN_BITS ‖ MSG_LEN_BITS )
```

Now, the trains differ. Concatenate the "P-train" and "A-train" as shown below:

```
( PTRAIN ‖ MSG ‖ ATRAIN )
```

For the purposes of distinguishing between these two padding styles, let's call the first version, in which the `MSG_LEN_BITS` is known prior to digesting the message, **Alpha Padding**. The second version, in which `NULL_128` is used because the message length isn't known prior to digesting the message, will be called **Beta Padding**. I strongly recommend using Alpha Padding whenever possible.

A note on endian-ness: Abacus treats the `HASH_LEN_BITS` variable as a big-endian string of 32-bits. (The most significant bit is leftmost.) Similarly, Abacus treats the `MSG_LEN_BITS` field as a huge 128-bit big-endian string (the 128-bits are arranged from left to right, with the most significant bit on the far left.)

2.13 Randomized Hashing

Abacus supports randomized hashing [35], but does not require it. Instead of XORing the entire input message with a fixed length random string, as suggested in [59], Abacus appends *and* prepends a single random salt to the overall message. Refer to the salt in section 2.11.

3 Design Rationale

I designed Abacus to be:

Secure The hash function should be resistant to all of the attacks that have

crippled MD5 [45, 71] and SHA-0 [8, 9, 67] and reduced confidence in the Merkle-Damgård construction [25, 40, 43, 44, 70]. In particular, the hash function should provide 2^n bits of security against preimage attacks and second preimage attacks, and $2^{n/2}$ bits of security against collision attacks.

Fast When designing the hash function, choose components that are known to be fast and easy to optimize.

Simple Pseudocode for the hash function should fit on a notecard. Its diagrams should be easy to understand. As a direct extension of this simplicity, programmers will be able to implement the hash function easily and without mistakes.

Tunable At least one parameter should be tunable, in the sense that a user can alter it for a given message to achieve speed/security tradeoffs.

Small The memory requirements should be as small as possible (for the required security level), thereby allowing the hash function to perform well on resource constrained platforms.

3.1 High Level Rationale

To realize these goals, I made a number of decisions that affected Abacus's structure, behavior, and properties. These decisions are listed below.

Use Simple Byte-wise Operations. SHA-3 will need to run on many platforms. Like any industry standard, once SHA-3 has been adopted, people will be reluctant to stop using it – SHA-3 might be with us in 50 years. Therefore it is crucial for the hash function to use a data size and instruction set that is common to virtually any platform.

Avoid the Merkle-Damgård Construction. Cryptanalysis only gets better over time, not worse. Having sustained serious blows to its security in the last five years, the Merkle-Damgård construction is cleary in its death throes. A new approach to hashing is needed. The hash function should offer a compelling alternative to Merkle-Damgård.

Use an Expansion-Permutation Function. One of the main problems with all Merkle-Damgård hash functions is that the compression function is (by definition) lossy, which allows for two different message blocks to hash to the same intermediate hash value [52]. Instead, the hash function should use an expansion-permutation to ensure that two different message blocks can never cause the internal state to collide.

Use a Large Internal State. As recommended by Lucks [50], an internal state of $2n$ (where n is the hash output size) would be resistant to many kinds of second preimage attacks. A large internal state would also allow for a strong finalization function (or output transformation) that is a one-way compression of the internal state.

Use Block Cipher Components. Until recently, hash functions were designed in their own silo, oblivious to advances in block cipher cryptography and cryptanalysis. Block ciphers, being well-studied over the past 20 years, are much better understood than hash functions. When designing the hash function, keep block cipher cryptanalysis in mind at all times.

Use a Maximum Distance Separable Matrix. By using an MDS matrix, a single input byte difference affects four internal state bytes. AES uses this to great effect [24, 30]. This diffusion is optimal over a four-byte vector, thus increasing the avalanche effect and ensuring that local collisions and neutral bits are impossible. Furthermore, an attacker's control is limited because for every change he manages to predict or control, several others will occur without his control.

Use a Counter. Attacks such as multicollisions [40, 44] and fixed points [25] exploit Merkle-Damgård's static properties. The hash function should prevent such attacks by having one or more "moving parts" that evolve after each round, thereby ensuring that the internal state is dynamic. Futhermore, it seems wise to have at least one internal state variable that the attacker cannot control.

Guarantee a Minimum Cycle Length. Like a stream cipher, the hash function should be able to guarantee a minimum cycle length. This enables the hash function to produce long strings of hash outputs if necessary, and also to be resistant to attacks that rely on short or cyclical messages. As explained by Jenkins [39], the average expected cycle length can be much greater than the minimum cycle length. Also, use an invertible update function to prevent state collapse (which is when the internal state's entropy gradually deteriorates, thereby shortening the cycles and producing weird output correlations) [39].

Use an S-box for Non-linearity. Any cryptographic function requires strong non-linearity. Although mixing modular addition with XOR is fast and simple, it is a weak form of non-linearity that undermines the round functions of the MD5 [62] and SHA [27, 28] families. S-boxes are well-understood and heavily scrutinized by the cryptographic community. By using an S-box, the hash function will have known and provable levels of resistance to many of the most powerful cryptanalytic attacks. Furthermore, that S-box should be randomly generated in order to resist algebraic attacks [21]. New attacks will certainly be invented in the future, and a random S-box would be more likely to resist them than would an S-box that was optimized against just one or two kinds of attacks. With that in mind, the hash function's S-box should be given as many good cryptographic properties as possible.

Use Registers of Various Lengths. The hash function's internal state should have "short-term memory" and "long-term memory", meaning that changes caused by input message blocks will be "remembered" right away, as well as dozens of rounds later. The short-term memory allows for a rapid avalanche. On the other hand, the long-term memory provides redundancy in case something similar to a near-collision occurs: the input byte difference will reassert itself to make the internal states diverge again.

Have a Strong Forward and Backward Avalanche. If the avalanche was only strong going forward (as in a classical LFSR stream cipher), it would be much easier to launch attacks against the hash function's internal state, or even to "unhash" a message. Therefore it's important that the backward avalanche be as strong as the forward avalanche. The Mars block cipher was designed with this in mind [16].

Allow for Randomized Hashing. By forcing the hash function to be "online", randomized hashing prevents several annoying attacks, such as one-of-many attacks and precomputations. [10] Therefore, the hash function should support randomized hashing. This would also allow for elegant HMAC constructions and random bit generators that require a seed.

Append and Prepend Both the Message Length and Hash Length. Classic Merkle-Damgård strengthening appends the message's length to the end of the message. While this foils some length extension attacks, it's not good enough: messages with varying lengths should produce different intermediate hash values immediately. Also, the hash output length should be a parameter so that identical messages will hash to different values when the desired hash length differs. These ideas were directly inspired by the HAIFA framework [10].

Use Blank Rounds. The hash function should ensure that the internal state is decorrelated prior to generating the hash output. Blank rounds are used in RadioGatún [6] for this reason. A similar concept, but applied to stream ciphers, is discussed in [53]. In that paper, Mironov recommends that RC4 will produce much more secure output by dropping *n* of the first output bytes (effectively the same as using *n* blank rounds).

3.2 Low Level Rationale

`NUM_BLANK_ROUNDS`. I have conducted billions of related message (i.e. related key) attacks to find out the exact point at which two similar message produce decorrelated internal states. This happens at 90 blank rounds (assuming one clock per round). Out of curiosity, I tested the output for its decorrelation, and discovered that the output is *very nearly* decorrelated after just 5 blank rounds, and completely decorrelated after 37 blank rounds (again assuming one clock per round). Furthermore, my initial cryptanalysis shows that linear cryptanalysis is infeasible after 98 blank rounds (sections 5.1 and 5.2). But by default, I recommend 135 blank rounds in order to provide an extra buffer of security.

`NUM_ABSORB_CLOCKS`. The purpose of increasing `NUM_ABSORB_CLOCKS` is to allow a single input byte difference to affect more than 4 internal state bytes and to pass through more than 5 active S-boxes. After two absorb clocks, 8 internal state bytes are affected. (But only 7 are distinct; the value at `ra[0]` will also be modified on the second absorb clock.) I believe that Abacus's absorb phase is strong enough after just one clock: 5 active S-boxes in single round is quite a bit, and the blank rounds ensure at least one full avalanche of the internal state. Also, keep in mind that increasing the `NUM_ABSORB_CLOCKS` to 2 or 4 will make Abacus noticeably slower (by approximately

2x and 4x, respectively).

`NUM_SQUEEZE_CLOCKS.` Setting `NUM_SQUEEZE_CLOCKS` to 2 or 4 is a simple and efficient way to increase the security of Abacus. Multiple squeeze clocks come at very little additional cost because the extra work is only done during the squeeze phase (i.e. 28 to 64 rounds once the message is digested). For a medium or long size message, this extra work is negligible compared to the cost of absorbing the entire message. Because 135 blank rounds gives both the internal state and the output sufficient time to become decorrelated, I believe that a `NUM_SQUEEZE_CLOCKS` value of 1 is adequate

Counter Ranges. The ranges of the four counters (i.e. 233, 239, 241, and 251) were chosen because those are the four largest prime numbers less than 256. By being both prime and close to 255, the counters' ranges ensure a maximal period. In particular, this period is $2^{31.6}$.

Register Lengths. The four registers `ra`, `rb`, `rc`, and `rd` have different lengths because each register has a different function. `ra` is the chaining variable, and as such, must provide feedback from clock to clock. `rb` acts as short-term memory, ensuring that an input byte difference reasserts itself just a few rounds after first appearing. `rc` acts as medium-term memory, allowing an input byte difference from a few dozen rounds ago to reassert itself. And finally, `rd` functions as long-term memory, guaranteeing that an input byte difference from several dozen rounds ago is able to reassert itself. In combination, the short-, medium-, and long-term memory makes the round function of Abacus very resistant to chosen-message attacks: the attacker has to know and control internal state differences from multiple different rounds that span a large amount of message blocks. Also, similar to the counter ranges, the regsiters' lengths were chosen to be prime numbers to allow the period to be maximal. Their period is 16465, and combined with the counter ranges, that's about 2^{45}.

Tap and Update Positions. When a register *r* is updated, the value at *r*[0] is sent through the S-box in combination with another value. Hence, the value at r[0] has an effect on the round's outcome. Tap positions also have an effect on the round's outcome, although the value at a given tap position is not modified during that round. The tap positions of the various registers were chosen so that they form a "comb" with widely spaced teeth. Also, the positions are prime numbers because it seems wise to avoid any simple recurrence patterns.

An ordered list of all the tap and update positions is: { **1**, 2, **5**, 13, 31, **37**, **89** } (The entries in bold are update positions.) `ra` provides the joint tap/update position that's 1 clock away; `rb` provides the tap position that's 2 clocks away and the update position that's 5 clocks away; `rc` provides the tap position that's 13 clocks away and the update position that's 37 clocks away; and `rd` provides the tap position that's 31 clocks away and the update position that's 89 clocks away. All together, a single input byte difference will affect the round function a total of 7 times.

Notice that the tap positions are "front-heavy", meaning that they tap an input byte difference closer to when it first appears. This is to avoid attacks analogous to

neutral bits [8] or local collisions [17], in which an input byte difference would have no effect on the internal state for many rounds. By being tapped close to when the input difference first appears, the idea is that the input difference will cause considerable changes right away, thereby starting a strong avalanche.

S-box Properties. The Abacus S-box was randomly generated. It has several good cryptographic properties. Some of these properties are quite easy to find in a random S-box (such as nonlinear order of 7, which is virtually guaranteed), while others don't seem to occur in nature (such as Max LAT bias of 28). To generate such a rare S-box, I used a special algorithm. The straightforward approach, which is to generate a complete S-box and then test it for the desired properties, is exceedingly slow and ineffective. My method was to insert one entry at a time, testing for the desired properties after each insertion. Reject the new entry if it violates any of the desired properties; otherwise, keep that entry and proceed to the next entry. (Some tricks are necessary to make certain tests yield meaningful results when the table is incomplete, such as for Max DDT entry and Max LAT bias.) I was able to find the Abacus S-box in about 6 hours. The specifics of the S-box generation algorithm are outside the scope of this paper, but I would be happy to provide the algorithm and source code upon request.

In particular, the Abacus S-box was randomly generated to have the following properties:

- The Maximum Difference Distribution Table (DDT) entry is 6
- The Maximum Linear Approximation Table (LAT) bias is 28
- The Nonlinear Order is 7
- The Shortest Cycle is 12 (Hence, the S-box is a derangement)
- The Distance to Strict Avalanche Criteria (DSAC) is 8
- The Distance to Maximum Order Strict Avalanche Criteria (DMOSAC) is 16
- The Distance to Bit Independence Criteria (DBIC) is 18
- The Distance to Maximum Order Bit Independence Criteria (DMOBIC) is 30

The various S-box properties are defined in appendix A.2. Sections 5.1 through 5.4 discuss how some of these properties make Abacus resistant to various kinds of cryptanalysis. If any of these properties proves to be detrimental to Abacus's security, I will gladly generate another S-box without the offending property/properties.

4 Security: Hash Function Attacks

In the aftermath of a five year onslaught against all commonly-used hash functions, it seems wise to survey the damage. Clearly the Merkle-Damgård construction has passed the end of its useful life. In fact, every descendent of MD4, including MD5, SHA-0, SHA-1, and probably even SHA-2 [70], cannot provide the security they promise. Any new hash function will have to take all of their weaknesses, as well as Merkle-Damgård's weaknesses, into account.

Abacus was specifically designed to be resistant to second preimage attacks, fixed points, multicollisions, herding attacks, neutral bit attacks, length extension attacks, local collisions, and related key (related message) attacks.

4.1 Second Preimage Attacks

Recently all manner of second preimage attacks have been making headlines. This section is dedicated to generic second preimage attacks. (Refer to sections 4.2 through 4.9 for more specific kinds of attacks, such as fixed points and multicollisions).

As discussed in section 1.1, a second preimage attack is when an attacker is given a first message, and tries to find a second message, where both messages hash to the same value [63]. A more formal way of expressing this is: Given a message M1, it should be hard to find a different message M2, such that Digest(M1) = Digest(M2).

The precise definition of "hard" is a moving target, and seems to get lower after each successful new attack is published. The classical requirement, though, is that it should take about $\mathbf{O}(2^n)$ steps to find a second preimage.

Very long messages make it much easier to find second preimages [52] on Merkle-Damgård hash functions that don't use length padding. (Length padding is the when the total message length is appended to the message to "finalize" it.) Let's consider an example: suppose there is an attacker who is trying to produce second preimage collisions on a 192-bit hash function which lacks length padding. Now, suppose that attacker generates a long message, M, of length 2^{62} blocks and then hashes it, storing all of the 2^{62} intermediate hash values on some massive, searchable disk drive. Each intermediate hash represents a slightly longer version of M, until finally the full 2^{62} blocks are digested. As each intermediate value is generated, the attacker searches through his database for a match satisfying the condition: Digest(M_{CurLen}) = Digest ($M_{SomeEarlierLen}$). There is a 1 in 2^{130} chance that such a match will occur. Hence, the true difficulty of finding a second preimage drops from 2^{192} to 2^{130}.

Length padding foils this attack, because CurLen differs from SomeEarlierLen. This will force the intermediate hash values to un-collide. Since Abacus uses length padding, it is resistant to very-long-message attacks.

But even without the length padding, Abacus would be resistant to the very-long-message second preimage attack because Abacus's internal state is secret. Abacus does not have an intermediate hash value in the classical sense. Instead, Abacus has a final hash value that is derived from its secret internal state, and calculated through a separate function only after all message bytes have been digested. If intermediate hash values don't exist for Abacus, the attacker certainly can't store them.

4.2 Fixed Points

In a hash function, fixed points occur when the intermediate hash value does not change after digesting a given message block. Essentially, the hash value hashes onto *itself*. Put formally: a fixed point occurs for some intermediate hash value, H, when: H = Digest(H, CurMsgBlock)

Fixed points are fundamental problem for any Merkle-Damgård hash function. In [25], Dean showed that if fixed points can be easily found, the security of second preimage resistance breaks down, especially when the message being digested is very long. Unfortunately, fixed points are easy to find for a Merkle-Damgård hash function. This attack is particularly worrisome because it effectively bypasses length padding, and can extend a message by one block.

Along these lines, the authors of [44] show how to use fixed points to create a powerful new second preimage attack called expandable messages. Starting from an arbitrary initialization vector, expandable messages are groups of messages of varying lengths whose hash values collide just prior to entering the finalization function (i.e. just before the the message's length is appended). These expandable messages can be quite long, and can be used to generate second preimages for a lot less than $\mathbf{O}(2^n)$ work.

In order for a fixed point to exist, the hash function must have no moving parts between each application of the compression function. It has to be a static compression function: given the same input, it must always produce the same output.

But Abacus does have moving parts. Thanks to the counter and the rolling arrays, Abacus can be in 2^{45} different states. After absorbing a single input message byte, Abacus enters a new state – or several sequential new states if `NUM_ABSORB_CLOCKS` is greater than one. In the unlikely event of a given input byte having no effect on the internal state at a given time, Abacus's arrays and counter will immediately increment, thereby entering a new state and rendering fixed points impossible.

4.3 Multicollisions

Joux's multicollisions [40] were a watershed moment in the history of hash functions. He shocked the cryptographic community with two revelations: 1) that once a single collision has been found, it's pretty easy to find many others, and therefore, 2) that using two independent hash functions in parallel to digest a given message and then concatenating the hash output of the two functions is only as secure as the stronger of the two hash functions.

A multicollision is a set of distinct messages of the same length that all hash to the same value. In effect, the multicollision technique generates many second preimages with the same hash value, requiring just slightly more than $\mathbf{O}(2^{n/2})$. The size of a multicollision set is always a power of 2, because multicollisions are built exponentially (i.e. 2×2×2× ... ×2). Essentially, multicollisions are obtained by chaining together several birthday attacks. These attacks must be done in order (i.e. from first to last).

Begin with an initialization vector, H_0, for the hash value, and then launch a birthday attack to find two distinct messges, M_1 and N_1, that map the initialization vector to the same hash value, H_1. Now, use H_1 as the new initialization vector and launch another birthday attack to find two more distinct messages M_2 and N_2 that map H_1 to the same hash value, H_2. Repeat many times.

Symbolically, the multicollision technique looks like:

$$H_1 = \text{Digest}(H_0, M_1) = \text{Digest}(H_0, N_1)$$
$$H_2 = \text{Digest}(H_1, M_2) = \text{Digest}(H_1, N_2)$$
$$H_3 = \text{Digest}(H_2, M_3) = \text{Digest}(H_2, N_3)$$
$$\ldots \qquad \ldots$$
$$H_k = \text{Digest}(H_{k-1}, M_k) = \text{Digest}(H_{k-1}, N_k)$$

As each collision is found, the number of possible combinations doubles until you have a very large set of multicollisions. For instance, in the above example:

$$\text{Digest}(H_0, M_1 \| M_2 \| M_3) = \text{Digest}(H_0, M_1 \| N_2 \| N_3) = \text{Digest}(H_0, N_1 \| N_2 \| M_3)$$

Just using the message blocks M1 through M3, and N1 through N3, there are 8 possible ways to concatenate the message blocks to form an overall message. Each of those 8 overall messages are guaranteed to have the same final hash value. (Remember the 1,2,3 order must be maintained.) Length padding can't stop these attacks because all of the message blocks are the same size, and the overall message is the same length. It's feasible to have multicollisions of size 256, 65536, 4294967296, or even many terabytes.

Multicollisions exploit several properties of iterated hash functions built upon compression functions. First, the birthday attack is $\mathbf{O}(2^{n/2})$, and therefore the fastest kind of generic hash function attack (it can also be done using very little memory). When k birthday attacks are done sequentially, as shown above, the total work required is only $\mathbf{O}(k2^{n/2})$ – meaning that it's very cheap to find exponentially more multicollisions. Second, the compression function calculates an intermediate hash value after each message block, so any collisions within a single block can immediately be exploited.

Elaborating on Joux's [40] and Dean's [25] work, Schneier and Kelsey showed that it is possible to find expandable messages—and hence second preimages—surprisingly easily by using multicollisions, even when it is hard to find fixed points [44].

Since multicollisions rely on cheap, fast birthday attacks, a simple defense is to use a larger intermediate hash value. In [50], Lucks proposes a "wide pipe" strategy in which the intermediate hash value is twice as large as the desired hash output size. For instance, if the algorithm's advertised hash length is 256 bits, then the intermediate hash value should be 512 bits or more. Abacus does this, because its internal state is 1056 bits and therefore more than twice the size of the largest required hash length of 512 bits. Hence, Abacus is resistant to multicollisions.

Additionally, Abacus is resistant to multicollisions because the internal state is

secret, and not publically knowable after successive message blocks have been digested. Only at the end of the entire message, when the squeeze phase begins, is the hash output calculated. In order to build a multicollision, it's necessary to have several intermediate hash values all chained together. Since those intermediate hash values don't exist, a multicollision can't be performed.

And finally, remember that multicollisions rely on the notion of two distinct input blocks *M* and *N* which hash to the same intermediate hash value. Essentially, a collision must occur within the compression function. The only analogous attack on Abacus would be to make the internal state collide after digesting a single message block. But during its absorb phase, Abacus uses an expansion-permutation function, not a lossy compression function. So if two distinct input blocks *M* and *N* are fed into the absorb phase, they are guaranteed to change at least two internal state bytes. Hence, collisions on the internal state can never be caused by a single input block difference.

4.4 Herding Attacks

Invented by Kelsey and Kohno [43], the herding attack is used to break commitment schemes. It's a kind of preimage attack, but also a new attack in its own right. The attacker claims to have a message of great future value (i.e. which country will win the World Cup in 2022). Along with this claim, the attacker publishes a "target" hash value, which is supposedly the hash of his prophetic message. Then, right after the 2022 Wolrd Cup is over, the attacker publishes his message containing the winning country (along with a mysterious suffix), and, *voila* – it hashes to the hash value he published years ago.

Of course it's a scam. By carefully choosing an appropriate hash value, the attacker is able to launch an effective preimage attack on it in much less time. The target hash value is found by cleverly using the multicollisions of 4.3 to build a collision tree (aka "diamond structure") out of many random intermediate hash values (or chaining variables). One benefit of the herding attack is that it's relatively practical: that mysterious suffix only needs to be a few dozen blocks long. But due to the large size of the collision tree, herding attacks have large time and space requirements.

Kelsey and Kohno recommend that any new hash function should be resistant to herding attacks (also called "chosen target forced prefix" attacks) [43].

Because Abacus is resistant to multicollisions (see section 4.3), this eliminates the possbility of building the multicollision trees (or diamond structures) that the herding attack requires. Furthermore, Abacus's moving parts (i.e. the counter and rolling arrays) would defeat the iterative/recursive nature of the herding attack collision trees, becuase those trees assume static relationships between intermediate values and input messages. And if those arguments aren't enough, Abacus is also resistant to herding attacks because it does not use intermediate hash values which collision trees are built from.

4.5 Length Extension Attacks

A length extension attack is when an attacker can modify an existing message by appending a suffix, and still be able to predict the final hash value. There are two related styles of length extension attacks. The ‖ operator denotes concatenation.

Style 1. Informally: if two different messages of the same length happen to collide into the same hash value, then an attacker can append a single arbitrary string onto both messages and the hash value will still be equal.

More formally: if Digest(Msg1) = Digest(Msg2) and Len(Msg1) = Len(Msg2) for two distinct messages Msg1 and Msg2, then Digest(Msg1‖Sufx) = Digest(Msg2 ‖Sufx)

Abacus is resistant to style 1 length extension attacks. The internal state is secret, and so, unlike in a classical Merkle-Damgård hash function, Abacus's hash output is not computed after each block. No publicly readable intermediate hash value gets passed to the next compression function call. Instead, each 8-bit message block evolves a portion of the internal state by changing several bytes. The attacker can feed selected bytes into the absorb phase, but he can't see what's going on inside the internal state. It's a black box model. In order to perform a style 1 length extension attack, the attacker would need to find two distinct messages of the same length that caused a total collision of the internal state. Even if the attacker was able to do this, there would be no way for him to know until the output was produced. The difficulty of causing a collision on the internal state is approximately 2^{528} for a birthday attack, or 2^{1056} for a second preimage attack. Both of those numbers are at least as large as the 2^{512}, the maximum recommended size of the hash's output. Therefore style 1 length extension attacks are not effective against Abacus.

Style 2. In a style 2 length extension attack, an attacker can append a suffix onto an original message and compute the hash for it, without needing to know the original message's contents [52]. All the attacker needs is the length of the original message, plus its final hash value. This is a big threat to some message authentication schemes. Any Merkle-Damgård hash function is susceptible to this undesirable property.

More formally: Assume there is an original message, Msg, which the attacker does not know. Given the Len(Msg), and the Digest(Msg), the attacker can calculate Digest(Msg‖Sufx) for a suitable suffix.

Style 2 length extension attacks don't apply to Abacus. In order to conduct a style 2 length extension attack, the attacker would need to recover the entire internal state. (This is analogous to obtaining the final hash value in a style 2 length extension attack on a traditional Merkle-Damgård construction). The only way for the attacker to recover the internal state is to derive some internal state data from the final hash value. In reality, the attacker gains no value from having the final hash value, because the final hash value is a non-linear, one-way function of the internal state. But assume the worst case scenario: that the attacker is able to recover 512 internal state bits from the 512 bit hash value. Well, there are still 544 more unknown internal state bits. (Aside: Having 512 bits of the internal state is not useful to the attacker, because the blank rounds ensure the internal

state will totally decorrelate before the output hash is generated.) The only way to recover those other 544 bits would be to guess them, which is harder than a brute force second image attack of 2^{512}. Hence, Abacus is resistant to style 2 length extension attacks.

4.6 Related Message Attacks

Abacus began its life as a direct adaptation of RC4 [64], vaguely similar to RC4 Hash [18]. But after reading about the punishing attacks on RC4 [33, 53], I specifically designed Abacus to resist these vulnerabilities. This is relevant because Abacus's absorb phase is a lot like the key setup phase of a generic stream cipher. In a stream cipher, a string of secret bytes stirs the internal state, after which, the internal state is able to generate a very long string of random-looking output. So, when a hash function is based on a stream cipher, the key becomes the message, and the output keystream becomes the hash.

RC4 has a weak key setup routine. It leaks information about the key because RC4's internal state is highly correlated with the key.

With the purpose of seeing why those correlations existed, I wrote a suite of tests to analzye RC4's key setup routine. Some strikingly anomalies caught my eye. In particular, after the 256 key setup rounds, the internal states of two different RC4 contexts often look *way* too similar (i.e. between 60% and 98% the same) when two similar (related) keys are used. This is especially true with long keys.

Then I observed how related keys could cause RC4's internal state to "diverge" when an input difference occured, but then "converge" (or collide) just a few bytes later. This was caused by statistical flukes, but also by lucky arrangements of differences between the two keys.

Essentially, the randomly incrementing counter, **j**, was a single point of failure. One input difference would affect two entries in the internal state (S-box), and also knock **j** off track, making it diverge. But a second, well-placed input difference could bounce **j** back on track, making it converge. Then, for the next dozen (or two hundred) rounds until that tiny S-box difference is randomly "hit", the two RC4 contexts would have virtually identical internal states.

There are two solutions to this very serious problem. The first, as suggested by Mironov [53], is to drop between 768 and 3072 of the first output bytes – essentially using a large number of blank rounds to decorrelate the internal state. By default, Abacus uses 135 blank rounds, which guarantees a full avalanche on `rd`, and more than 3 full avalanches on `rc`. This is sufficient to make Abacus's internal state fully decorrelated.

The second solution, which Abacus also uses, is simple: magnify and scatter the input differences. It's crucial to guarantee that a difference in a single input byte changes several bytes in the internal state. Preferably, these "changed-bytes" should be widely

spaced so that if one is cancelled out by a second input byte difference, there's a fallback changed-byte nearby to force the internal states to diverge again. Ideally, there would be yet another fallback changed-byte very far away, acting as long-term memory and making the internal state evolve more chaotically.

Abacus has tap positions at 1, 2, 13, and 31 clocks away, and update positions (which also affect the S-box lookups) at 5, 37, and 89 clocks away. Therefore, Abacus has seven different, widely-spaced bytes that provide short-term, medium-term, and long-term memory for a single input byte difference.

Because Abacus uses blank rounds, and guarantees that a single input byte difference affects at least two internal state bytes, it is resistant to the related key attacks that haunt RC4. Furthermore, Abacus is resistant to related message attacks in general.

4.7 Carry Bits

Because the MD5 [62] and SHA [27, 28] families gain their non-linearity from combining modular addition with bitwise operations, they are susceptible to attacks on the carry bits. (Binary addition works by using carry bits. When two numbers, X and Y, are added together to form Z, a carry bit scrolls from right to left as the bits are added. If the bits at X_i and Y_i are both 1, Z_i is set to 0, and the carry bit is set to 1 and carried to the left. This is similar to carrying the 1 when adding decimal numbers by hand.)

If not for carry bits, SHA and MD5 would consist soley of bitwise operations, and in particular, a large number of XORs. A variant like this would be extremely easy to break: XORs are commutative, so toggling a bit in the input would toggle a bit in the output [64]. But the carry bits are non-linear with respect to bitwise operations, and hence, provide a weak kind of non-linearity. By only moving from right to left, carry bits don't allow an input bit difference to propagate to the right. Even worse, carry bits are very "local": an input difference usually only affects nearby bits. All successful attacks on MD5 [45, 71] and SHA [8, 9, 66, 67] exploit the weaknesses inherent in using carry bits for non-linearity.

Abacus does not rely on combining XORs and modular addition. Instead, Abacus uses a highly non-linear S-box in which all output bits depend heavily on all input bits. With its non-linearity being provided by a strong S-box, Abacus is immune to attacks that are based on carry bits.

4.8 Local Collisions

First introduced by Joux in [17], local collisions exploit weaknesses in the round function of a hash function to create collisions after just a few rounds. Targeting specific steps of a particular algorithm, local collisions are very specialized—almost surgical—attacks. The most successful attacks on the SHA [8, 9, 66, 67] family are all this way. After creating a local collision, the attacker tries to follow a differential path through many (ideally all)

rounds of the hash function. Then the attacker is able to find two different messages that hash to the same value ("full collisions"), or nearly the same value ("near-collisions"). These messages are one block long, and it is assumed that the hashing started from the same initialization vector.

Neutral bit attacks, first introduced by Biham and Chen in [8], and brought to a stunning climax in [9], are a powerful means of creating local collisions. Neutral bits are bits in the input message that have no effect on the intermediate hash value for a number of rounds. The more neutral bits an attacker can find, the more messages he can try, and the more control he has over the hash value. This translates into higher odds of finding a collision. For example, Biham and Chen used neutral bits to bypass the first 22 rounds of SHA-0, thereby reducing the complexity of finding a collision from 2^{80} to 2^{56}.

Abacus is immue to neutral bits because a single input difference is guaranteed to immediately change four bytes of internal state. Hence, it's not possible for an input difference to have no effect on the state for several rounds after it was first encountered.

Local collisions attack the compression function. By definition, a compression function must map a large set to a smaller set, so collisions are guaranteed to exist [2, 52]. But since Abacus has an expansion-permutation function, a difference of a single bit in the 8-bit input block is guaranteed to change four separate bytes in the internal state. Assuming the local collision attack starts from the same initialization vector (i.e. internal state), there is never a case where two different input blocks map to the same byte. Hence, Abacus is immune to local collisions.

Let's consider a near-collision attack on Abacus. We start at round 0, when two Abacus contexts, *ab1* and *ab2*, are identical. The attacker feeds a byte into *ab1*, and different byte into *ab2*. In the absorb function, that input difference passes through the S-box, and then the `mds4()` step. Because the `mds4()` has a branch of 5, four bytes of the internal state are guaranteed to differ. Hence, *ab1* and *ab2* differ by at least 4 bits. Therefore, it's impossible for a single input byte difference to cause an internal state difference of less than 4 bits. Now, advance to round 1 by incrementing Abacus's counters and rotating the rolling arrays. The chaining variable at `ra[0]`, which was changed by the input byte difference, will also influence this current round. Observe that *ab1* and *ab2* do not differ in the values stored at `rb[0]`, `rc[0]`, or `rd[0]`.

At this point, the attacker has two options: feed identical bytes into *ab1* and *ab2*, or feed different bytes into *ab1* and *ab2*. If the attacker feeds identical bytes into *ab1* and *ab2*, then the only difference going into the `mds4()` step is the value at `ra[0]`. Due to the branch property of `mds4()`, all four of `mds4()`'s output bytes must change. The value at `ra[0]` only holds one byte, so the previous round's change is overwritten. Therefore, *ab1* and *ab2* would differ by 7 bytes instead of 8.

On the other hand, if the attacker fed different bytes into *ab1* and *ab2*, then two difference bytes go into the `mds4()` step, and the branch property guarantees that at least 3 output bytes differ. (But as mentioned in the previous paragraph, ra[0] can only store one byte, so only 2 out of those 3 changed-bytes will always be maintained.) No

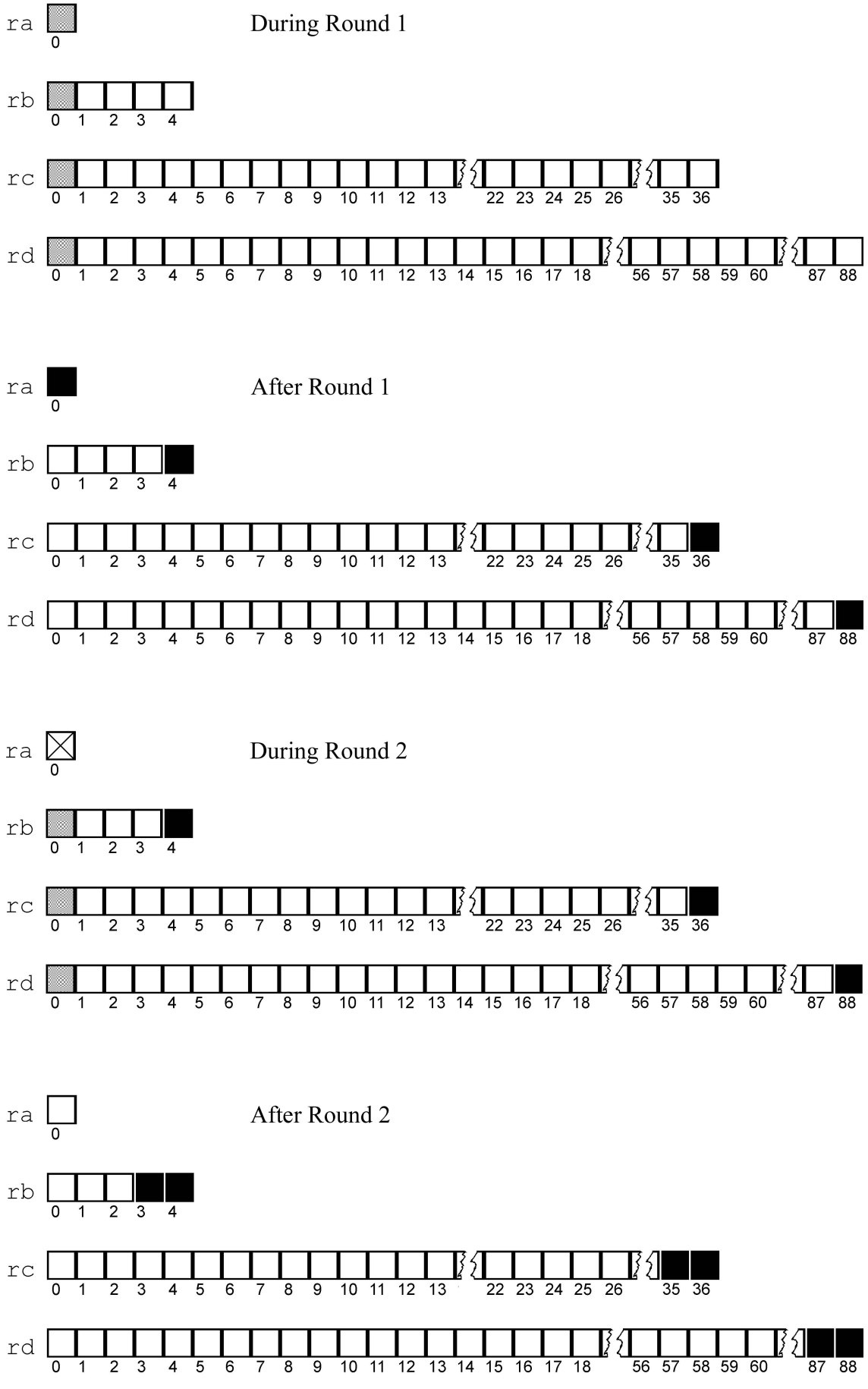
matter what choice the attacker makes, 2 more bytes are guaranteed to change at the end of round 1.

Hence, after two rounds, a single input byte difference is guaranteed to change 6 or more bytes of the internal state (and by implication, at least 6 bits). The expansion-permutation property of Abacus prevents near-collisions.

(The argument above assumes that `NUM_ABSORB_CLOCKS` is set to 1. If `NUM_ABSORB_CLOCKS` is greater than 1, then each input byte causes more than 6 bytes of internal state to differ after two rounds.)

Shown below, in Figure 5, is a depiction of how a single input byte difference affects multiple bytes in the registers over the course of two rounds. During round 1, four bytes of internal state are changed, and after round 1 is finished, those bytes get rotated to the other end of their arrays. During round 2, the attacker feeds in another input byte difference, thereby canceling out the change in `ra[0]`. (Note that in practice, any one of the four update positions would be cancelled out at random – the attacker wouldn't be able to know or control which one got cancelled. Also, the MDS's branch property is pretty strong, so there's a good chance 4 internal state bytes would still change, even if the attacker did pass in a second input byte difference.) Finally, at the end of round 2, the arrays are rotated again.

Figure 5: Minimum Changed-bytes After 2 Rounds



But since near-collisions are the lynchpin of many real-world attacks, let's consider a broader type of near-collision attack. Again, suppose we have two Abacus contexts, *ab1* and *ab2*. Assume that after digesting two different messages, not necessarily of the same length, that *ab1* and *ab2* differ by just a few bits. (Disregard length padding for this example.) We know that after 90 blank rounds, Abacus's 1056-bit internal state is decorrelated. Then after 135 blank rounds, the internal state is certainly decorrelated. So even if *ab1* and *ab2* differed by just 1 bit prior to the blank rounds, they will differ by about 512 bits once the blank rounds are finished. Hence, this type of near-collision does not affect Abacus, either.

4.9 Multi-block Collisions

Sometimes it's not necessary to create a full collision in one block – a near-collision can be sufficient. Then, by causing several near-collisions in a row, an attacker causes a full collision. Joux et al. used this technique to find a full collision on SHA-0 in just 4 message blocks [9]. (It took 80,000 hours of computer time.) Since several consecutive message blocks are required, these collisions are known as multi-block collisions.

As shown in section 4.7, Abacus is resistant to near-collisions. Hence, a multi-block collision attack which relies on several near-collisions is impossible.

Now, let's consider a broader type of multi-block attack on Abacus. Near-collisions are not necessary, and there can be several dozen multi-blocks. Suppose we have two Abacus contexts, *ab1* and *ab2*, that start out with the same internal state. In round 0, the attacker feeds two different bytes into the absorb function. Due to the `mds4()` step, which has a branch of 5, that single input byte difference is guaranteed to change four bytes of the internal state. So now *ab1* and *ab2* differ by four bytes. (There are four "changed-bytes".) Furthermore, those changed-bytes are widely distributed across Abacus's internal state: they are at `ra[0]`, `rb[0]`, `rc[0]`, and `rd[0]`. Due to the tap positions at `rb[3]`, `rc[24]` and `rd[58]`, we know that the changed-bytes will affect the absorb function 1, 2, 5, 13, 31, 37, and 89 clocks later.

So, for the next 89 rounds, the attacker will have to do three things: 1) control all internal state changes so that *ab1* and *ab2* don't change any further, 2) neutralize the influence of the changed-bytes when they are tapped 1, 2, 13, and 31 rounds later, and 3) cancel out the changed-bytes that originally occupied `ra[0]`, `rb[0]`, `rc[0]` and `rd[0]` when they are updated 1, 5, 37, and 89 rounds later. It's impossible to do all three things at once. Even if the attacker is able to cancel out two or three bytes at a time, the branch property guarantees that he changes two or three more in the process. This is a losing battle. The attacker is trying to do surgery with a shovel.

4.10 Precomputations and One-of-Many Attacks

Not even an ideal hash function is resistant to precomputation attacks [10, 52]. In this scenario, the attacker calculates and stores a huge number of message/hash value

pairings. Then, when the attacker wants to launch a preimage attack on some arbitrary hash value *H*, he first searches his database of precomputed pairings to see if *H* occurs as any of the hash values. If *H* does occur, then the attacker simply looks up the message that corresponds *H* and he has found a preimage. Similarly, if the attacker wants to find a second preimage for some arbitrary message, *M1*, he searches his database for another message, *M2*, such that *M1*'s hash value matches *M2*'s hash value.

Both of these attacks are clearly related. And not surprisingly, they are equally effective at reducing the difficulty of finding a preimage or second preimage. In particular: by precomputing 2^k message/hash value pairings, the attacker reduces the complexity of finding a preimage or second preimage to 2^{n-k}, where *n* in the hash functions output size in bits.

Another similar attack is the one-of-many attack, also called the multiple targets attack [10, 52]. In this attack, the attacker has multiple targets in mind. If the attacker has a group of 2^k messages and he wants to find a second preimage on one of those messages (but doesn't care which one), then the his odds are 2^{n-k}. Ideal hash functions are also vulnerable to this attack.

The only known solution to these attacks is to start the hash from an initialization vector (IV) that always changes. For instance, the IV could be a timestamp or a random bitstring (salt). Because when the IV is an unknown, the hash value becomes a function of the IV *and* the message. Using a hash function in this manner ensures that the hash value is dependent on who did the hashing and when it was done. Abacus supports randomized hashing, and therefore allows for this solution.

5 Security: Block Cipher Attacks

Over the past two decades, cryptographers have made tremendous advances in the field of cryptanalysis. The earliest, most significant attacks are differential and linear cryptanalysis, both of which came out in the early 1990's. Since then, a number of other attacks have been discovered—some very broad, such as algebraic attacks—and some rather specialized, such as interpolation attacks.

Because block ciphers are so prominent, all of these attacks were developed for—and used almost entirely against—block ciphers. Applying these attacks to a hash function is not very straightforward, due to the significant differences between hash functions and block ciphers. It's important to discuss these attacks, though. And it's important to take them seriously. After all, a hash function is like a block cipher turned on its head: a block cipher's key is analogous to a hash function's message.

5.1 Differential Cryptanalysis

Perhaps the most famous and seminal of all attacks, differential cryptanalysis was invented by Biham and Shamir [12]. The premise is simple: the attacker chooses two

plaintexts with a known and fixed difference between them, and sends both plaintexts through the cipher to obtain two ciphertext outputs. He then compares the two ciphertexts to find their difference, and keeps track of the result. After doing this for many plaintexts, subtle non-random patterns will emerge and the attacker will be able to gradually recover bits of the key.

Differential cryptanalysis is a chosen plaintext attack, meaning that the attacker must be able to encrypt plaintexts of his choosing. It is also a probabilistic attack, meaning it is based on statistics and probability: the more plaintexts the attacker controls, the higher his certainty that a given key bit is correct. Eventually he'll be 100% certain of all key bits.

To succeed at differential cryptanalysis, the attacker must be familiar with the exact workings of the cipher he's attacking. He'll usually start by building statistical models of the cipher's non-linear function(s), such as the S-box. In particular, he's looking for input differences that cause certain output differences to occur more often than a perfectly random function would do. An input difference / output difference pair that occurs too often is called a characteristic. Such characteristics are extremely useful to an attacker because they allow him to trace a differential "trail" from round to round throughout the cipher by observing non-random behavior in the cipher's output. Each round multiplies the complexity of following that trail. Here's a somewhat simplified example: if a characteristic has a 1/20 chance of occurring, then after two consecutive rounds it has a 1/400 chance of occuring. After 5 rounds, a 1 in 3200000 chance. With odds like this, it becomes clear why differential attacks often take (far) more than 2^{40} chosen plaintexts to succeed. (In a real-world cipher, an input difference doesn't always go through one S-box per round, so the math isn't quite as simple.)

Most block ciphers need to be concerned about differential cryptanalysis because very few real-world S-boxes have a perfectly flat distribution of characteristics. Defending against differential cryptanalysis is pretty straight forward [37]: use an S-box with strong non-linearity, and ensure that each input difference passes through many S-boxes as possible. (Whenever an input difference passes through an S-box, it's referred to as an "active S-box".) Indeed, after a certain number of rounds, the chosen plaintexts required to mount a differential attack will be larger than the cipher's advertised security. A 2^{205} attack is pretty useless if the key is only 2^{192} bits: the attacker is better off guessing the key outright. A cipher is said to be immune to differential cryptanalysis when it can prove that the best possible attack requires too many plaintexts.

The Abacus S-box has a Max DDT entry of 6, which is the best possible for a random S-box. Therefore, the highest-likelihood characteristic has a $2^{-5.415}$ chance of occuring. ($6/256 = 2^{-5.415}$). Abacus's internal state is 1056 bits, so 196 active S-boxes are required before the internal state is immune to differential cryptanalysis. (1056 / 5.415 = 195.19). For a 512-bit hash output, 95 active S-boxes are required. If we can show that Abacus has at least 196 active S-boxes, then Abacus will be immune to differential cryptanalysis on its internal state as well as its output.

Let's assume an attacker hashes two messages, *M* and *N*. He uses two different Abacus contexts, *ab1* and *ab2*, to hash the messages. Prior to hashing *M* and *N*, the

internal states of *ab1* and *ab2* are identical. For the purposes of this argument, assume Round 1 is when *M* and *N* differ for the first time.

Round 1. M_i differs from N_i. Since the contents of *ab1* and *ab2* are identical going into round 1, the only difference during the absorb phase is msg[i]. So, msg[i] passes through one active S-box and then enters the `mds4()` step. Since the MDS matrix has a branch of 5, a single input difference is guaranteed to affect all four output bytes. Those four changed-bytes then pass through a second series of S-boxes, all four of which become active. Thus, at the end of the first round, there are 5 guaranteed active S-boxes.

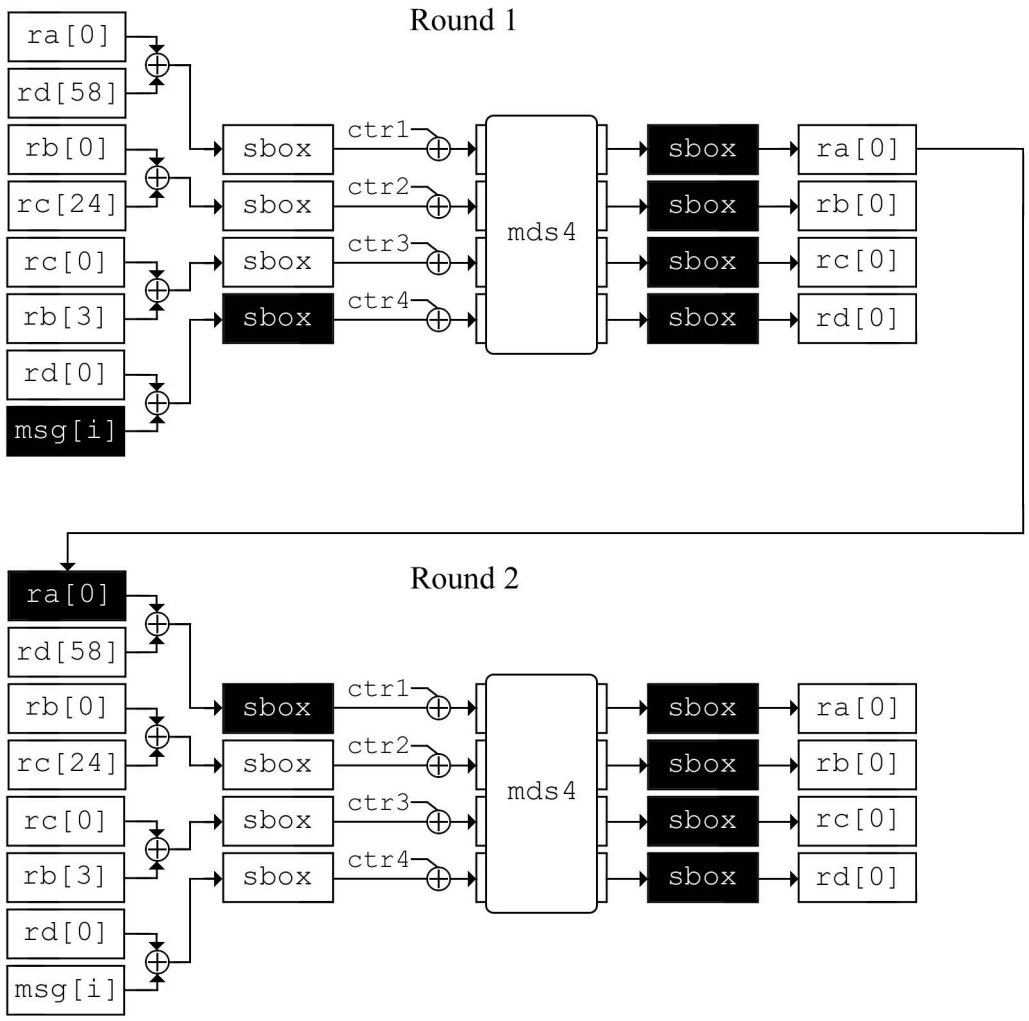
Round 2. After the internal state is clocked, the changed-byte at `rb[0]` enters `rb[4]`, the changed-byte at `rc[0]` enters `rc[36]`, and the changed-byte at `rd[0]` enters `rd[88]`. Therefore, the only changed-byte from round 1 that exerts an influence in round 2 is `ra[0]`. Similar to round 1, the value at `ra[0]` passes through one active S-box, enters the `mds4()` step, changes all four `mds4()` output bytes, and finally, the four changed-bytes enter four S-boxes and make them active. Hence, 5 active S-boxes are also produced in round 2.

Hence, any two-round differential trail has 10 active S-boxes. After 40 blank rounds (i.e. 20 two-round pairs), there are approximately 200 active S-boxes. As discussed above, 196 active S-boxes are required to make Abacus resistant to differential cryptanalysis. Therefore, 135 blank rounds gives us about 670 active S-boxes – more than sufficient to make Abacus immune to differential cryptanalysis.

(Please note that this analysis of Abacus's active S-boxes assumes that `NUM_ABSORB_CLOCKS` is 1. If `NUM_ABSORB_CLOCKS` is 2 or more, there would simply be more active S-boxes per round, thus making Abacus resistant to differential cryptanalysis after fewer blank rounds.)

A diagram showing a two-round differential trail and its active S-boxes is shown below.

Figure 6: Active S-boxes After 2 Rounds



In Figure 6, a two-round differential trail within Abacus is illustrated. In round 1, the first input difference occurs in `msg[i]`. This difference passes through one S-box before entering the `mds4`. Coming out of the `mds4`, the input difference is guaranteed to change all four output bytes. Then, these four changed-bytes proceed to pass through a second series of S-boxes. At the end of round 1 there are 5 active S-boxes.

Round 2 is very similar. The chaining variable, `ra[0]`, contains a changed-byte from round 1. `ra[0]` passes through one S-box. During the `mds4` step, the changed-byte affects all four of the output bytes, and then the four changed-bytes pass through their four S-boxes. Thus, during round 2, there are an additional 5 active S-boxes. By combining the active S-boxes from round 1 and round 2, there are a total of 10 active S-boxes after two consecutive rounds.

5.2 Linear Cryptanalysis

Inspired by Biham and Shamir's work on differential cryptanalysis, Matsui invented linear cryptanalysis [51]. Like differential cryptanalysis, linear cryptanalysis is a probabilistic chosen-key attack that requires a deep understanding of the target cipher's inner workings. Linear cryptanalysis also uses the concept analyzing the S-box for high-likelihood characteristics (or biases) and then following their trail through the cipher. And just like in differential cryptanalysis, a cipher defends itself against linear cryptanalysis by using a strong S-box and proving that a single input difference passes through enough S-boxes to foil a chosen plaintext attack [37]. The main difference between these two types of cryptanalysis is that linear cryptanalysis finds its high-likelihood characteristics in a very different manner.

Linear cryptanalysis seeks to build linear approximations to any of the non-linear behavior in the cipher. Nearly always, this means building a linear approximation of the S-box. But instead of analyzing output differences that are caused by particular input differences, the attacker seeks to bypass the S-box's non-linearity altogether by finding an affine (or linear) transformation that approximates the S-box's behavior. (An affine transformation is a kind of Boolean function that multiplies a bit-string by a constant, and returns a single bit.) For an *N*-bit S-box, the attacker tries all possible pairings of *N*-bit constants, *aff1* and *aff2*, and adds up the number of times when input $\cdot$ *aff1* = sbox [input] $\cdot$ *aff2* holds true.

Now, in a perfect S-box, any affine transformation would hold true exactly 50% of the time, meaning that no information is leaked. Real-world S-boxes, though, contain slight biases. Any bias is exploitable. If a certain affine transformation holds true even 51% of the time, (or say 48.5% of the time) that is sufficient to launch a linear cryptanalytic attack. The smallest possible bias for a *N*x*N* algebraic S-box such as AES's is 50% ± 12.5%. For a random S-box, the smallest possible bias is 50% ± 21.9%. Clearly, then, linear cryptanalysis is a concern for virtually any cryptosystem.

With its Max LAT bias of 28, which is the best possible for a random S-box, the Abacus S-box's most biased entry occurs with odds of $2^{-2.193}$. ($28/(256/2) = 2^{-2.193}$) The number of active S-boxes required in order for the internal state to be resistant to linear cryptanalysis is 1056 / 2.193 = 481.6. Similarly, it takes 512 / 2.193 = 233.5 active S-boxes for a 512-bit hash output to be resistant to linear cryptanalysis. By showing that a single input difference has at least 482 active S-boxes, we can say that linear cryptanalysis is ineffective against Abacus.

As shown in section 5.1, Abacus has 10 active S-boxes for any two-round differential trail. Therefore, after 49 two-round pairs (or 98 blank rounds), Abacus is resistant to linear cryptanalysis. (482 / 10 = 48.2.) The recommended value of 135 blank rounds, which produces about 670 active S-boxes, is more than adequate to make Abacus immune to linear cryptanalysis.

(Again, please note that increasing the `NUM_ABSORB_CLOCKS` will increase the number of active S-boxes, thereby making Abacus resistant to linear cryptanalysis in fewer blank

rounds.)

5.3 Algebraic Attacks

Algebraic attacks, first introduced in 2002 by Courtois and Pieprzyk [22], are a contentious topic. Many cryptographers ridicule these attacks, or, at the very least, seriously doubt that they will ever be practical. But in 2008, Courtois, Bard, and Wagner used an algebraic attack to break the KeeLoq cipher with just 2^{28} plaintexts [20]. KeeLoq is a proprietary block cipher widely-used in the automotive industry. It's what allows a wireless key to unlock car doors and disable alarms. Back in 1995, KeeLoq's designer apparently sold it for $10 million. Now it's not worth 10 dollars. Not only is the attack of Courtois, Bard, and Wagner fast and practical, it would enable the attacker to literally drive off with your car. The days of comfortably ignoring algebraic attacks are over. It seems likely that in the next 10 years, algebraic attacks will become *the* predominant attack.

To launch an algebraic attack, the attacker builds a (somewhat) small set of simple equations that completely defines the cryptosystem. Generally, there are a few thousand equations and a few thousand variables. These equations are algebraic, and in particular, quadratic multivariate polynomials. At the core of any algebraic attack is the set of equations that defines the S-box by mapping its input bits to its ouput bits. Simple, algebraic S-boxes are much easier to define, using far fewer equations and terms. Once the S-box is modeled, it's mechanical to define the whole cryptosystem: just combine the equations that define the S-box into a system of sparse quadratic equations. From there, the attacker solves the equations by using a few known plaintexts. Now it's possible for him to express all ciphertext bits as a function of plaintext bits and encrypt or decrypt as he pleases.

Unlike linear and differential cryptanalysis, which typically require almost absurd amounts of chosen plaintexts, an algebraic attack only requires a few chosen plaintexts. Even more impressively, the complexity of an algebraic attack grows "nearly polynomially" [23] with the number of rounds instead of exponentially.

Because the S-box is usually the only non-linear part of any cipher, the number of equations required to model it profoundly affects the complexity of an algebraic attack. S-boxes like AES's [24, 30], which are based on the simple inverse function over a Galois field, and Serpent's [1], which are a tiny 4-bits, are easy to define using these sparse quadratic equations. Random S-boxes, on the other hand, are notoriously hard to define using quadratic equations [23]. In fact, the amount of work required to define the equations for a random S-box is *double-exponential* in relation to the S-box's size. By contrast, defining the equations for the AES S-box only requires an amount of work that is *exponential* in relation to the S-box size.

Abacus is resistant to algebraic attacks because it has an 8-bit random S-box.

5.4 Higher Order Differentials

Introduced by Knudsen in [46], higher order differentials are useful for attacking ciphers with S-boxes of low nonlinear order. A cipher that used weak nonlinear transforms (i.e. alternating XOR with addition) instead of an S-box would also be vulnerable to higher order differentials. The attack proceeds by representing the entire cipher as a system of Boolean polynomials of low degree. Then, the attack differentiates the Boolean polynomials using higher and higher order differentials. Eventually, the Boolean polynomial of the cipher breaks down and becomes zero. This is similar to recursively differentiating a polynomial in calculus: after (n+1) differentiations, an n-th order polynomial becomes zero.

In practice, this attack is only useful if the nonlinear order of the overall block cipher is lower than the claimed security. For instance, if a block cipher claims to have 128 bits of security but can be represented as a Boolean polynomial of degree 97, then that block cipher is weaker than the expected (or claimed) 128 bits ─ and therefore susceptible to a higher order differential attack.

The Abacus S-box has a nonlinear order of 7, which is maximal. Every time a byte passes through the S-box, it increases the overall hash function's nonlinear order. After a given byte has passed through the S-box once, the degree of its Boolean polynomial is 7. After two passes, it's 49. After three passes, it's 343. And after four passes, the degree is 2401. Because Abacus's internal state is 1056 bits, any Boolean polynomial of degree 1056 or higher would be adequate to make Abacus immune to higher order differential attacks. This only requires that the same byte passes through four S-boxes. As show in section 5.1, a single input byte difference passes through 10 active S-boxes after 2 rounds. Some of these lookups are done in parallel, but the number of *serial* S-box lookups for a single byte difference is exactly 4 (two passes per round × 2 rounds). Hence, higher order differential attacks don't affect Abacus after 2 consecutive rounds.

5.5 Interpolation Attacks

As described in [38], the interpolation attack is similar (on a high level) to both algebraic attacks and higher order differential attacks. Interpolation attacks are particularly useful for attacking ciphers that use an S-box based on algebraic or rational functions. Such functions include quadratic equations and inverses over a Galois field.

In a nutshell, the interpolation attack represents a cryptosystem's input and output as polynomial or rational expressions of one another. At first, those expressions have a lot of unknown coefficients. The attack collects several known input and output pairs like in algebra: to solve for N unknowns, find N or more data points. Then, by using a standard Lagrange interpolation, the attack gradually solves for each unknown by using those input and output pairs. The most alarming thing about the interpolation attack attack is that it doesn't require very many known input / ouput pairs. All the attacker needs is N such pairs. Because once he does have N pairs, the expressions that map the

input to the output are solved for, and the attacker will be able to encrypt or decrypt at will. Bizarrely, this attack does not recover the key: it works around it.

The best defense against an interpolation attack is to make the number of unknown coefficients, N, as large as possible. At a certain point, it is infeasible for the attacker to find more than N pairs – for instance if N is several billion. When N has surpassed that value, interpolation attacks no longer work.

With its random, non-algebraic S-box, Abacus is not susceptible to interpolation attacks.

5.6 Slide Attacks

Wagner and Biryukov's slide attacks [13, 14] exploit highly repetitive and cyclical round functions. The attack is particularly devastating against ciphers that use the same subkey in every round. Even if a small set of subkeys is used in a cyclical way, the slide attack can break the cipher faster than brute force. In essence, a slide attack will attack a cipher's key schedule. The lesson here is to use separate keys in each round, or at the very least, use the keys in a complex or irregular order.

Because block ciphers tend to use the same permutation function in each round [64], the only difference from round to round is the subkey. But if the cipher had other variables (not necessarily secret) that changed several times, each change would help to thwart a slide attack. For instance, the XTEA cipher [56] was designed to be resistant to slide attacks in two ways: 1) it uses a non-secret counter that increments after each round, and 2) the key schedule is irregular due to pseudo-randomly choosing 1 of 4 subkeys for each round.

In order for a block cipher to be vulnerable to a slide attack, it usually only has 2 to 8 subkeys that are used in a simple, repetitive order such as: k1, k2, k3, k4, k1, k2, k3, k4. The slide attack creates models of the cipher's round functions – one model per subkey. Through using a small number of models in a fixed order, a slide attack exploits a clever type of birthday attack to recover key information in less than expected time.

To adapt the slide attack to hash functions, treat the hash function's input message like a block cipher's subkeys. Now, identify any variables in the round function. In the case of Abacus, there are seven main variables: the counters `ctr1` through `ctr4` , and the positions of `rb`, `rc`, and `rd`. Between these four variables, Abacus can be in 2^{45} possible configurations. A slide attack would have to build separate round function models for each of those 2^{45} configurations – a daunting task that would render the slide attack useless. Furthermore, the attacker never has total control over the subkeys, because for every byte he feeds into the absorb function, several bytes from the rolling arrays are also combined with it. Between the enormous number of models required, and the hard-to-control subkey schedule, Abacus is resistant to slide attacks.

5.7 Related Key Attacks

Of all the block cipher attacks discussed so far, related key attacks are perhaps the most relevant to a hash function. With that in mind, Abacus was designed to be extremely resistant to related key attacks. See section 4.6 for a discussion of Abacus's resistance to related key attacks (in this context, the same thing as related message attacks).

6 Security: Side-Channel Attacks

The above mentioned attacks are all rather theoretical, in the sense that they apply to the algorithm in an abstract way, and can be done asynchronously and remotely. They're offline attacks, ivory tower attacks.

But there's a grittier, more streetwise kind of cryptanalysis called side-channel attacks [47, 48, 64]. Side-channel attacks exploit electrical, accoustical, and temporal side-effects of the cryptosystem as it operates in real-time. Arguably, such attacks are redundant (i.e. if the attacker is close enough to monitor your CPU's cacheing, then he can probably obtain your crypto key in other, faster ways), or even unfair (because algorithms are designed to be strong on paper, divorced from any physical implementation) [64]. Nonetheless, side-channel attacks are a reality and need to be taken seriously. The same algorithm, implemented in two different ways, can have vastly different security in the real world. (And as such, side-channel attacks are also called "implementation attacks".)

6.1 Timing Attacks

Certain computational operations take longer than others. Timing attacks, introduced in [47], monitor a piece of cryptographic software or hardware for variations in timing. By carefully analyzing those timing fluctuations, and through a modest understanding of how a given algorithm works, an attacker might be able to obtain crypto key data. For instance, a subroutine with lots of if-statements that act upon the intermediate data would be quite vulnerable. This is because the exact values of the data would cause different if-blocks to be entered or not entered, and the resultant timing fluctuations would correlate with the data that was inputted into the subroutine.

Constant-time operations (such as XOR, and in some systems, ADD), are not susceptible to timing attacks. Table lookups, especially on small tables, should also be safe. In general, any subroutine or module should always execute with the same number of clock instructions, and in the same amount of time. Operations that are notorious for non-constant time execution, and therefore vulnerable to timing attacks, are: conditional logic (ifs, while loops, for loops), branching (jumps, breaks, gotos), variable bitwise rotation, division, multiplication, and large memory access calls.

The core of Abacus's absorb and squeeze phases is a series of table lookups (the

S-boxes) and XORs (when bytes are fed into the S-box, and during the MDS step). Neither of these are susceptible to timing attacks.

The counters, however, use modular addition, which is non-constant time on some systems. Thankfully a simple solution exists: just feed the counter into a lookup table that emulates modular addition over the desired field size. The lookup table would simply take a value *c*, and return (*c*+1) mod *fSize*, where *fSize* is the field size (i.e. 251).

At first glance, the rolling arrays might also seem problematic. The most efficient way to implement them is to use large arrays that increment their pointer after each clock. When several hundred (or several thousand) clocks have elapsed, the contents from the end of the array get copied to the beginning of the array, and the pointer is reset. This inherently requires conditional logic.

But because the position of the rolling arrays is neither secret nor data dependent, it doesn't really matter if the attacker can deduce those positions. Likewise for the counters: they are not secret or data dependent.

6.2 Power Analysis

As with timing attacks, different computational operations can consume different amounts of power (electricity). Attacks that exploit these fluctuations in power are called power analysis attacks [48]. All computational operations take electricity. Dig down deep enough, and it's always circuits and gates and electrical current. Since all software runs on microprocessors, and since a microprocessor is just a complex integrated circuit, any algorithm that's implemented in software ultimately consumes power. And in the case of special purpose hardware, the circuits and gates are even more apparent. By eavesdropping on power consumption, an attacker can deduce information about the algorithm's internal state. He simply looks for spikes and valleys (sometimes very small ones) in power usage that are directly caused by the secret input data and it goes through different execution paths.

Conveniently, nearly all of the same precautions that make an algorithm resistant to timing attacks will also help to make it resistant to power analysis attacks. Avoid conditional statements, branching, multiplication, division, large table lookups, and variable rotation. As discussed in section 6.1, Abacus uses XORs and table lookups, which are (somewhat) resistant to power analysis attacks.

The question is, exactly how resistant? This seems to be an open question. Differential power analysis [48] is quite powerful, and can detect even minute fluctuations in power – perhaps even down to a single bit of a single XOR. In the end, the best ways of protecting against power analysis attacks seem to be unrelated to the algorithm. They are: adding noise to the signal to confuse the attacker; adding dummy registers that balance power consumption by making it near-constant; and most obviously, shielding the emitted power so that it is harder to monitor [69].

6.3 A Final Word on Side-Channel Attacks

At this point, power analysis attacks and timing attacks are still rather new. It is not exactly clear how to design around them. By the classification system proposed to categorize AES candidates [73], Abacus is pretty resistant to side-channel attacks because it only uses XOR and table lookups for its sensitive operations. But according to some pessimistic research [19], virtually every algorithm is vulnerable to power analysis attacks. It could be a moot point: I'm of the mind that before side-channel attacks could compromise a real-world cryptosystem, rubber hose cryptanalysis (i.e. seduction and torture) would get the job done first.

7 Security Claims

Abacus has a secret internal state. Even if two different messages did cause a collision on the internal state, the attacker would have no indication of this until the hash output was generated. But a collision on the hash output does not imply a collision on the internal state: it's quite possible that the internal state differs but randomly happened to generate the same output. Hence, the attacker can never be sure when he caused a collision on the internal state, or merely on the output.

Before any hash output is generated, Abacus has many blank rounds which ensure that at least one full avalanche occurs. Hence, two messages that differ by just 1 bit would result in internal states that differ by about 512 bits.

As discussed in section 3.1, Abacus's avalanche effect is virtually as strong going backward as it is going forward. So even if the internal state was somehow obtained, the attacker would have great difficulty recovering any useful preimage information from it. The strong backward avalanche ensures that any slight difference in his guesses would cause the internal state to immediately diverge and start towards an avalanche.

Abacus is resistant to both differential and linear cryptanalysis, as discussed in sections 5.1 and 5.2. Therefore, any chosen plaintext attack would not yield meaningful correlations in the internal state. And as argued in section 4.6, Abacus is resistant to related message attacks, thereby ensuring strong collision resistance and preimage resistance.

Finally, even if there were some kind of meaningful correlation in the internal state after performing the blank rounds, statistical tests show that Abacus's hash output is *very nearly* decorrelated after just 5 blank rounds, and fully decorrelated after 37. Hence, the hash output is capable of masking defects or correlations (however unlikely they may be) in the internal state.

With all of the above arguments in mind, I claim: For a hash output size of n bits, Abacus has **preimage resistance** of 2^n, **second preimage resistance** of 2^n, and **collision**

resistance of $2^{n/2}$. (These claims do not apply if the attacker is using one-of-many attacks or precomputations, because an ideal hash function is not resistant to such attacks.) Furthermore, I claim that Abacus is resistant to fixed points, multicollisions, herding attacks, length extension attacks, local collisions, and related message (related key) attacks.

8 Performance

As the SHA-3 contest progresses, NIST would behoove itself to encourage experts to optimize the various candidate algorithms. Speaking for myself, I am not an expert in hardware design, smartcard programming, or software optimization. A talented engineer in any of those fields could almost certainly beat my performance estimates. For anyone who tries, I thank you in advance.

Please note that in this document, the words "clock" and "clocks" refer to the operation of advancing Abacus's internal state via rotating the rolling arrays and incrementing the counters. I will use the terms "clock cycle" or "clock cycles" to refer to timing estimates for hardware and software.

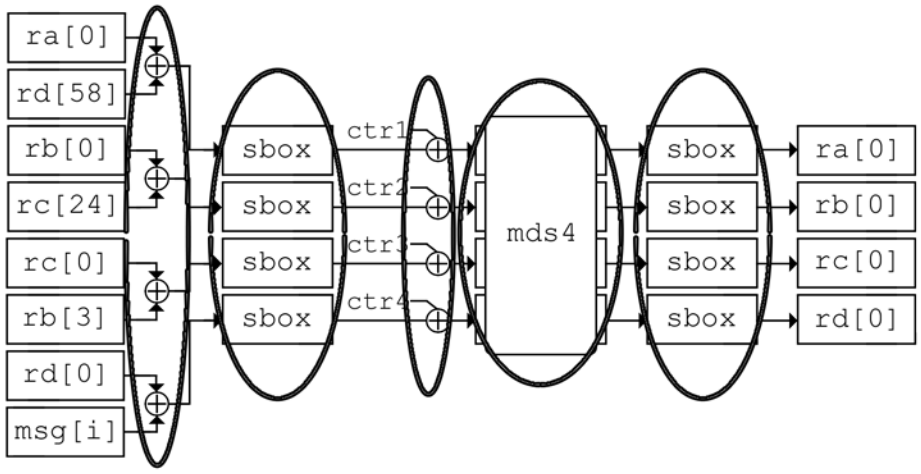
8.1 Parallelism

Many of the steps in Abacus can be done in parallel. In particular, the four XORs at the beginning of the absorb and squeeze phases can be done in parallel. Then, the first pass of S-boxes can be done in parallel. After those S-boxes, the XORing by the counters can be done in parallel. The MDS step itself can be done in parallel (meaning that the `mds4()` function's four output bytes are calculated simultaneously instead of sequentially). The second pass of S-boxes is also parallelizable. And when clocking the internal state, the `rotate()` function can rotate all three rolling arrays in parallel, while simultaneously all four counters can be incremented in parallel. An implementation that did all of these operations in parallel would be at least four times faster than one that did these operations serially.

This parallelism would be most beneficial in hardware, where the circuits could have truly parallel execution paths. In software, some parallelism could also be realized, because modern compilers are pretty good at executing two constant-time instructions in a single clock single.

Below, a diagram illustrates which steps can be performed in parallel during the absorb phase. The squeeze phase (not shown) would be similar, but with two additional parallel steps: 1) XORing `ra[0]` with `rb[0]`, and `rc[0]` with `rd[0]`, prior to the third S-box pass, and 2) the third S-box pass itself.

Figure 7: Parallelization in Abacus



8.2 Efficient MDS Implementation

In section 2.8, a simple method for computing the MDS step is shown. That method is elegant because it requires only a few bytes of memory, and the instructions clearly show how the finite field multiplication is performed. Hardware implementations would most likely use a similar solution: combine XOR, bit shifting, and some conditional logic to emulate multiplication by 2 and 3 in the finite field.

But when memory is not a concern, there is a faster way to implement finite field multiplication. Precompute two lookup tables, each of size 256 bytes. One table stores all of the values for multiplication by 2, and the other table stores all of the values for multiplication by 3. These tables can be called "ff_mult boxes". Below, example code in C is given.

```
unsigned int  i;
unsigned char hibit, m2;
unsigned char ff_mult_box2[256];
unsigned char ff_mult_box3[256];

// FF MULT by 2
for (i = 0; i < 256; i++)
{
    hibit = i & 0x80;
    m2 = i << 1;          // LSH by 1 is equiv. to mult by 2
    if (hibit != 0) { m2 ^= 0x1b; }   // 0x1b is the FF
    ff_mult_box2[i] = m2;
}

// FF MULT by 3
```

```
for (i = 0; i < 256; i ++)
    ff_mult_box3[i] = ff_mult_box2[i] ^ i;
```

Abbreviated versions of these tables are shown below. It can be argued that building a lookup table for multiplication by 3 is not strictly necessary, because multiplication by 3 can be done on-the-fly by XORing `ff_mult_box2[`*x*`]` with *x* itself. While that is certainly true, a second lookup table for `ff_mult_box3` does save one XOR per lookup, and hence 4 XORs per MDS step.

```
unsigned char ff_mult2_box[256] = {
    0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e,
    0x10, 0x12, 0x14, 0x16, 0x18, 0x1a, 0x1c, 0x1e,
    ...   ...   ...   ...   ...   ...   ...   ...

    0xfb, 0xf9, 0xff, 0xfd, 0xf3, 0xf1, 0xf7, 0xf5,
    0xeb, 0xe9, 0xef, 0xed, 0xe3, 0xe1, 0xe7, 0xe5
};


unsigned char ff_mult3_box[256] = {
    0x00, 0x03, 0x06, 0x05, 0x0c, 0x0f, 0x0a, 0x09,
    0x18, 0x1b, 0x1e, 0x1d, 0x14, 0x17, 0x12, 0x11,
    ...   ...   ...   ...   ...   ...   ...   ...
    0x0b, 0x08, 0x0d, 0x0e, 0x07, 0x04, 0x01, 0x02,
    0x13, 0x10, 0x15, 0x16, 0x1f, 0x1c, 0x19, 0x1a
};
```

Finally, the fastest option is also the most expensive (in terms of memory). Build four 8x32 bit lookup tables. Each table effectively multiplies the same input byte by an entire column of the MDS matrix (refer to section 2.8 for that matrix.)

```
// Declare the four 8x32 lookup tables:
unsigned int bmult1231[256];
unsigned int bmult1123[256];
unsigned int bmult3112[256];
unsigned int bmult2311[256];

// Build each 8x32 table from one column from the MDS matrix
for (i = 0; i < 256; i ++) {
    bmult1231[i] = i | (ff_mult_box2[i]<<8) |
                   (ff_mult_box3[i]<<16) | (i<<24);
    bmult1123[i] = i | (i<<8) | (ff_mult_box2[i]<<16) |
                   (ff_mult_box3[i]<<24);
    bmult3112[i] = (ff_mult_box3[i]) | (i<<8) |
                   (i<<16) | (ff_mult_box2[i]<<24);
    bmult2311[i] = (ff_mult_box2[i]) | (ff_mult_box3[i]<<8) |
                   (i<<16) | (i<<24);
}
```

8.3 Efficient Rolling Array Implementation

Rolling arrays can be implemented in three main ways. Depending on both the platform and the objectives, one of these three may be preferable to the others.

In-place rotation. Simple and inuitive, in-place rotation works well for memory constrained environments. The rolling arrays can be implemented in-place by using a FOR loop, which rotates the array to the left by one. The only overhead memory required for this is two bytes: a temporary variable, and a loop counter. An example of in-place rotation over an array of size 37 is shown below, using C pseudocode.

```
tmp = rc[0];
for (i = 1; i < 37; i++) { rc[i-1] = rc[i]; }
rc[36] = tmp;
```

Caterpillar arrays. Caterpillar arrays are very fast in software because they take advantage of the abundant RAM that modern computers have. First described in [11], caterpillar arrays are several times larger than a rolling array's "logical size". (i.e. The `rc` rolling array has a logical size of 37, but its caterpillar array might be 400 bytes.) Store the caterpillar array's offset in a pointer. After each clock, assign the entry from caterpillar_array[0] to caterpillar_array[*n*], where *n* is the array's logical size. Then, just increment the caterpillar array's pointer by 1. Eventually, when the pointer reaches the end of its allocated space, reset it to zero and copy the whole array back down to the start of the caterpillar array. Below, an example is given in C pseudocode.

```
rc[37] = rc[0];
rc++;
if (rc == end_of_array_rc)
{
      rc = start_of_array_rc;
      for (int j = 0; j < 37; j++)
            rc[j] = end_of_array_rc[j];
}
```

In the above example, the rolling array has a logical size of 37. There `start_of_array_rc` pointer keeps track of where the allocated array actually begins. Over the course of many clocks, the `rc` pointer can stray hundreds or thousands of positions away `start_of_array_rc`. The `end_of_array_rc` pointer keeps track of where the allocated memory runs out. Note that in the above example, `end_of_array_rc` is actually 37 places *earlier* than the true end of the buffer, to allow for efficient copying. For example, if the `rc` caterpillar array is 400 bytes, then `end_of_array` would be defined as `start_of_array_rc + (400-37)`.

The benefits of this implementation are twofold: 1) the rotating can be done is just two simple operations of assigning a value and incrementing a pointer, and 2) the values in the rolling array can be referred to with direct addressing (i.e. `rc[24]`) instead of variable addressing (i.e. `rc[(j+24) % 37]`). For these two reasons, caterpillar arrays are by far the fastest implementation in software.

The fastest implementation I wrote uses caterpillar arrays of the following sizes:

rb[30], rc[300], rd[400]. It is not clear why these sizes yield faster results than other sizes.

Variable addressing. Like in-place rotation, varible addressing is an in-place operation. However, variable addressing is faster because it does not require FOR loops. Instead, a few extra variables are needed, as well as some modular additions.

Variable addressing uses index counters to iterate over the array. For example, consider `rc` , which has 37 bytes. One index counter would keep track of the update position, and another index counter would keep track of the tap position. To initialize these counters, do the following:

```
rc_upd_idx = 0;
rc_tap_idx = 24;
```

Then, after each clock, increment these index counters in the following way:

```
rc_upd_idx = (rc_upd_idx + 1) % 37;
rc_tap_idx = (rc_tap_idx + 1) % 37;
```

Each rolling array would require its own set of index counters. In the body of the absorb or squeeze routine, the code would look something like this:

```
ra             = sbox[ra             ^ rd[rd_tap_idx]]
rb[rb_upd_idx] = sbox[rb[rb_upd_idx] ^ rc[rc_tap_idx]]
rc[rc_upd_idx] = sbox[rc[rc_upd_idx] ^ rb[rb_tap_idx]]
rd[rd_upd_idx] = sbox[rd[rd_upd_idx] ^ msg[i]]
```

Alternatively, each rolling array could use just one index counter: the update index. Then, calculate the tap index on-the-fly, in relation to the update index. For example: `(rd_upd_idx + 58) % 89` would be equivalent to `rd_tap_idx`.

8.4 Efficient Counter Implementation

Modular addition is slow on many platforms. And since Abacus uses prime numbers that are not a power of 2, the modulo operation is even slower. Luckily, a much more efficient method of implementing the counters does exist.

For each of the modulus values (233, 239, 241, 251), build a "mod box", which is a lookup table of size *k*. Each mod box takes an input, *x*, and returns (*x* + 1) mod *k*. Essentially, this is a speeed/memory trade-off. By using more memory in the form of lookup tables, the counter increments can be sped up greatly. This approach also prevents timing attacks and power analysis attacks, because table lookups are a constant time operation. An example mod box where *k* = 7 is shown in C pseudocode below.

```
unsigned char modbox7[7] = {1, 2, 3, 4, 5, 6, 0};
```

This technique works well in software because RAM is cheap. But on a memory constrained device, such as a smart card, these mod boxes would not be very desirable: it might be a better compromise to use conditional logic to bypass the modulo operator altogether. C pseudocode illustrates this idea below.

```
ctr1++;    ctr2++;    ctr3++;    ctr4++;
if (ctr1 == 233) {ctr1 = 0;}
if (ctr2 == 239) {ctr2 = 0;}
if (ctr3 == 249) {ctr3 = 0;}
if (ctr4 == 251) {ctr4 = 0;}
```

While optimizing Abacus, I found that the conditional logic technique, show just above, is nearly as fast as using the mod boxes.

8.5 Memory Requirements

At a bare minimum, Abacus requires 136 bytes of memory to store its internal state (132 bytes for the registers, 4 bytes for the counters.) This assumes that the S-box can be stored as a constant table, or hard-wired

But if the S-box must also be in RAM, then Abacus requires 392 bytes of memory (136 bytes for the internal state + 256 bytes for the S-box).

The MDS step can be sped up by storing the finite field multiplication results in one or two tables of sizes 256 bytes. This would cost an additional 256 or 512 bytes of memory. And the fastest option, which is building the four 8x32 lookup tables, would cost 4096 bytes of memory.

For efficient counter implementations, lookup tables (i.e. mod boxes) can be used for each modulus size. Each table would require k bytes of memory, where k is the modulus. The cost to store all four mod boxes is 233 + 239 + 241 + 251 = 964 bytes.

Depending on how the rolling arrays are handled, the memory requirements can vary substantially. In-place rotation costs an additional two bytes. Variable addressing costs 4 or 8 bytes. Caterpillar arrays, however, can cost up to several thousand bytes of memory.

Table 1: Isolated Memory Costs for Abacus

Feature	Bytes
Internal State	36
Loop counters, temp variables	6
The S-box	256
One 8x8 MDS Lookup Table	256
Two 8x8 MDSLookup Tables	512
Four 8x32 MDS Lookup Tables	4096

Caterpillar Arrays	500-4000
Mod Boxes for Counters	964

Table 2: Combined Memory Costs for Abacus

Features	Bytes
Internal State, Loop ctrs & temp vars	142
Internal State, S-box, Loop ctrs & temp vars	398
Internal State, S-box, Loop ctrs & temp vars, One 8x8 MDS Lookup table	648
Internal State, S-box, Loop ctrs & temp vars, Two 8x8 MDS Lookup tables	904
Internal State, Loop ctrs & temp vars, S-box, Caterpillar arrays	898-4398
Internal State, S-box, Mod boxes for counters	1356
Internal State, S-box, Four 8x32 MDS Lookup tables	4488
Internal State, S-box, Caterpillar Arrays, Four 8x32 MDS Lookup Tables	5538

8.6 NIST Reference Platform

As specified in the SHA-3 federal register notice [32], the NIST Reference Platform is a 2.4 GHz Intel Core 2 Duo processor, with 2 GB of RAM, running 32- and 64-bit versions of Windows Vista Ultimate.

I conducted my tests on two different personal computers (i.e. "machines"). The first machine, which I'll refer to as "NIST Reference Platform", is a 2.1 GHz Intel Core 2 Duo T8100 processor with 4 GB of DDR2 RAM, running 64-bit Windows Vista Home Premium, SP1. Due to the processor speed difference (i.e. 2.1 GHz vs 2.4 GHz), I scaled up all of my results by a factor of 14.28. (I did not scale the calculations based on RAM or Windows version because it's doubtful that either would have an effect on the hash function's performance.)

And then, to get a 32-bit perspective, I ran the same tests on a 1.79 GHz Pentium 4, with 768 MB of RAM, running Windows XP, SP 2. (Throughout this section I'll refer to this machine simply as "Pentium 4").

Table 3: Hashing Speed of Abacus on a Personal Computer

Optimized 32-bit code. Machine / Compiler	Mbits / Sec	Est. Clock Cycles/Byte
Pentium 4 / Visual C++ 6.0	191	75.0
Pentium 4 / Visual Studio .NET 2003	200	71.6
NIST Reference Platform / Visual C++ 6.0	431	44.5
NIST Reference Platform / Visual Studio .NET 2003	523	36.7
Optimized 64-bit code. Machine / Compiler	**Mbits / Sec**	**Est. Clock Cycles/Byte**
Pentium 4 / Visual Studio .NET 2003	95	150.7
NIST Reference Platform / Visual Studio .NET 2003	282	68.0

I obtained these results by running an optimized 32-bit version of Abacus (compiled with Visual C++ 6.0) on each machine, and then by running an optimized 64-bit version of Abacus (compiled with Visual Studio 2003 .NET) on each machine. The test itself was simple: make Abacus absorb a 512-bit string (i.e. 64 message 8-bit blocks) 390625 times (that comes out to 200 megabits of data). During these tests, keep track of how many milliseconds were consumed, and divide that by 1000 to obtain the number of seconds. Then, divide 200 by the number of seconds to obtain a megabits-per-second score. I performed this test 10 times and took the best out of ten.

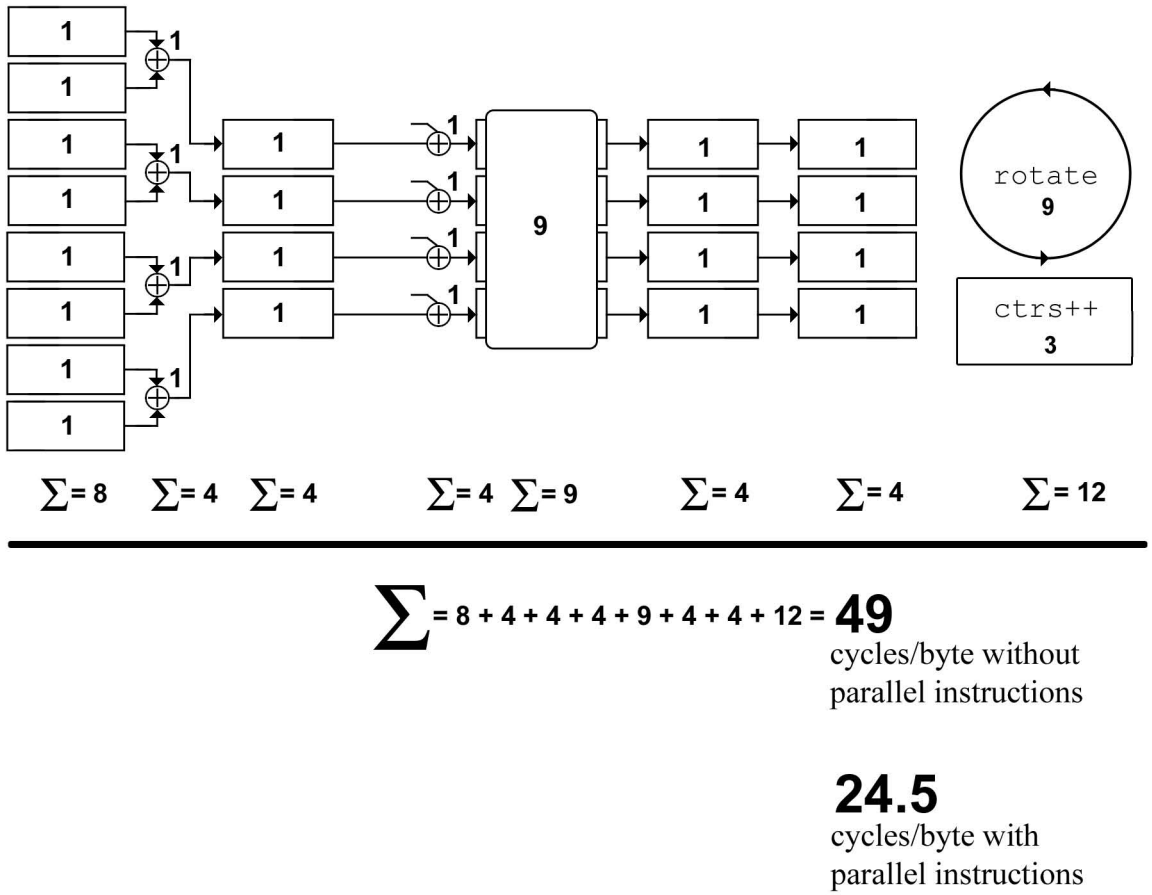
45.9 clocks per byte does not sound very impressive, but please keep in mind I am not an expert at optimizing C-code. Also keep in mind that the best-known timings for a given algorithm often take months of fine tuning by several experts. For instance, Gladman's optimized version of SHA-256 runs at 20.4, 23.3, and 41.0 cycles/byte, depending on the platform [34]. In another paper by Hilewitz et al., the authors mention that the best published score for SHA-256 is 12 cycles/byte [36]. Which of these numbers is correct? All of them? None of them? The answer seems to be that a) when speed really matters, somebody will code the algorithm in Assembly, and b) an algorithm's speed is heavily dependent on both the compiler and the machine. With that in mind, I believe that Abacus can run at about 25 cycles/byte.

If that number seems optimistic, consider the case of Rijndael (which became AES). The authors' optimized code, which shipped on the AES CD-ROM, ran at about 950 cycles/block for a 128-bit block. Shortly thereafter, Gladman (the same man who optimized SHA-2), released an optimized version of Rijndael that ran at 363 cycles/block for a 128-bit block [24]. That's a 2.6x speedup!

Let's consider a low-level analysis of each operation in Abacus's absorb phase. For each message byte that gets absorbed, there are 8 XORs, 8 S-box lookups, 4 counter increments, 12 register assignments, an MDS matrix multiplication, and three rolling array rotations. By using the four 8x32 tables to precompute the MDS matrix multiplication, the MDS step can be done in 9 cycles. Then, by using caterpillar arrays, the rolling arrays can be rotated in 9 cycles. If all of these operations were done serially, that would only take 49 cycles. (My best results were noticeably better than that, at 36.7 cycles.) But modern microprocessors and compilers are quite good at combining several instructions into a single clock cycle, especially constant-time instructions. All of Abacus's steps can be achieved with constant-time instructions. Therefore, I'm certain that a talented software optimizer could get Abacus running in 25 cycles/byte or less.

Below, I show a diagram of this clock cycle analysis.

Figure 8: Clock Cycle Analysis



In Figure 8, a clock cycle analysis of Abacus's absorb function is performed. To simplify analysis, assume each round has one clock (i.e. `NUM_ABSORB_CLOCKS` = 1). The operations have been grouped together for ease of understanding. Although no platform is specified, these estimates would be most accurate in software running on a personal computer. The `mds4` step, the `rotate` step, and the `ctrs++` step are highly variable, and, depending on implementation goals and constraints, may take more cycles. In this analysis, I've assumed the following: 1) the `mds4` step uses 8x32 lookup tables and combines the result, 2) the `rotate` function uses caterpillar arrays and constant addressing, and 3) the counter increment is done by using mod box lookup tables.

Please note that this is a worst-case estimate of overall clock cycles per byte. In many software implementations, the constant-time instructions can be parallelized so that two are done per clock cycle.

Furthermore, it may be possible to combine serial instructions into one clock cycle. For instance, the counter XORs may be combined with the output of the first S-box lookup. Similarly, the output of the second S-box pass may be directly assigned to caterpillar_array[*n*], thereby saving one instruction.

For the rest of the tables in this section, I assume that Abacus can absorb a byte of message data at 25 cycles/byte. Because the squeeze phase is so similar (it has 6 more instructions, all of them constant-time), I assume that it is about 15% slower, and hence, the squeeze function can run at 29 cycles/byte.

Table 4: Setup Time for Abacus on a Personal Computer

Prepended Data	Hash Size	Est. Clock Cycles
(None)	224	152
(None)	256	152
(None)	384	152
(None)	512	152
Hash Length	224	252
Hash Length	256	252
Hash Length	384	252
Hash Length	512	252
Hash Length, Message Length	224	652
Hash Length, Message Length	256	652
Hash Length, Message Length	384	652
Hash Length, Message Length	512	652
Hash Length, Message Length, Salt	224	1452
Hash Length, Message Length, Salt	256	1452
Hash Length, Message Length, Salt	384	1452
Hash Length, Message Length, Salt	512	1452

To generate the estimates in table 4, I made the following assumptions:

- It takes 152 clock cycles to setup the internal state (132 cycles to copy the S-box entries into the various registers, 4 cycles to reset the counters to zero, 1 cycle to reset the output feedback byte to zero, and 15 clocks of overhead)
- The "Hash Length" field is 32-bits (4 bytes) long
- The "Message Length" field is 128-bits (16 bytes) long
- The"Salt" is 256-bits (32 bytes) long
- An optimized implementation of Abacus can absorb a message at the rate of 25 cycles/byte.

Table 5: Cost of Generating a Single Message Digest on a Personal Computer

Message Length	Hash Size	Estd. Clock Cycles
8 bit	224 bit	4212
512 bit	224 bit	5787

1024 bit	224 bit	7387
65536 bit	224 bit	208987
8 bit	256 bit	4382
512 bit	256 bit	5903
1024 bit	256 bit	7503
65536 bit	256 bit	209103
8 bit	384 bit	4792
512 bit	384 bit	6367
1024 bit	384 bit	7967
65536 bit	384 bit	209567
8 bit	512 bit	5256
512 bit	512 bit	6831
1024 bit	512 bit	8431
65536 bit	512 bit	210031

To generate the estimates in table 5, I made the following assumptions:

- An optimized implementation of Abacus can absorb a message at the rate of 25 cycles/byte.
- There are 135 blank rounds between the absorb and the squeeze phase
- A blank round costs the same as absorbing a message byte (i.e. 25 cycles/blank round)
- An optimized implementation of Abacus can squeeze out a hash byte at the rate of 29 cycles/byte.

8.7 8-bit Processor Estimates

Although Abacus can function on an 8-bit smartcard, there are some drawbacks. First, there is the issue of the rolling arrays. The efficient implementation discussed above, which requires several hundred (or several thousand) extra bytes for the large, is not very desirable on a smartcard. This is because smartcards tend to have extremely low amounts of RAM or ROM: they prize memory even more than they prize speed. To conserve memory, Abacus would most likely be implemented with the rolling arrays "in place". In such an implementation, three FOR loops would have to iterate over the three rolling arrays once per clock to rotate them.

Alternatively, the rolling arrays would be emulated using variable addressing: the addressing of `rb`, `rc`, and `rd` would not use constant lookups (i.e. `rc[24]`), but instead would use variable lookups (i.e. `rc[rc_upd_idx]` and `rc[rc_tap_idx]`). After each clock, the `_upd_idx` and `_tap_idx` would be incremented for the three rolling arrays. This would be noticeably faster than having to rotate the entire array after each clock.

The straightforward way of incrementing the four counters is not the fastest. The modulo operator would be cost-prohibitive on a smartcard (and possibly not even available). Instead, use the conditional-increment, as described in section 8.4. (i.e. `ctr++;  if (ctr == modulus) {ctr = 0;}`)

The MDS step also presents a problem. There are two options: 1) compute all of the finite field multiplications on-the-fly (using bitshifts and conditional XORs), or 2) hardwire the finite field multiplication into two lookup tables. For the purpose of this analysis, I assume that option 2 was chosen. Furthermore, I assume that all instructions are done serially (i.e. that parallelization is not possible).

Cycles-per-Byte Estimates on a Smartcard

8 XORs +
4 counter increments (at 2 cycles a piece) +
8 S-box lookups +
12 register reads +
6 index increments to emulate the rotating arrays (at 2 cycles a piece) +
the MDS step (which has 12 XORs and 8 lookups)
_____
= 68 cycles/byte to absorb

The squeeze phase has 6 more instructions (2 S-box lookups, 3 XORs, and one assignment), so I estimate that the squeeze phase would take 74 cycles/byte.

Admittedly, these estimates are somewhat crude. It's possible that Abacus would be faster (or slower) in an actual smartcard implementation. I'd also guess that an expert could beat my estimates by at least 15%.

Table 6: Setup Time for Abacus on a Smartcard

Prepended Data	Hash Size	Est. Clock Cycles
(None)	224	152
(None)	256	152
(None)	384	152
(None)	512	152
Hash Length	224	424
Hash Length	256	424
Hash Length	384	424
Hash Length	512	424
Hash Length, Message Length	224	1512
Hash Length, Message Length	256	1512
Hash Length, Message Length	384	1512
Hash Length, Message Length	512	1512
Hash Length, Message Length, Salt	224	3688

Hash Length, Message Length, Salt	256	3688
Hash Length, Message Length, Salt	384	3688
Hash Length, Message Length, Salt	512	3688

To generate the estimates in table 6, I made the following assumptions:

- It takes 152 clock cycles to setup the internal state (132 cycles to copy the S-box entries into the various registers, 4 cycles to reset the counters to zero, 1 cycle to reset the output feedback byte to zero, and 15 clocks of overhead)
- The "Hash Length" field is 32-bits (4 bytes) long
- The "Message Length" field is 128-bits (16 bytes) long
- The"Salt" is 256-bits (32 bytes) long
- An optimized implementation of Abacus can absorb a message at the rate of 68 cycles/byte on a smartcard.

Table 7: Cost of Generating a Single Message Digest on a Smartcard

Message Length	Hash Size	Estd. Clock Cycles
8 bit	224 bit	11320
512 bit	224 bit	15604
1024 bit	224 bit	19956
65536 bit	224 bit	568308
8 bit	256 bit	11616
512 bit	256 bit	15900
1024 bit	256 bit	20252
65536 bit	256 bit	568604
8 bit	384 bit	12800
512 bit	384 bit	17104
1024 bit	384 bit	21436
65536 bit	384 bit	569788
8 bit	512 bit	13984
512 bit	512 bit	18268
1024 bit	512 bit	22620
65536 bit	512 bit	570972

To generate the estimates in table 7, I made the following assumptions:

- An optimized implementation of Abacus can absorb a message at the rate of 68 cycles/byte on a smartcard.
- There are 135 blank rounds between the absorb and the squeeze phase

- A blank round costs the same as absorbing a message byte (i.e. 68 cycles/blank round)
- An optimized implementation of Abacus can squeeze out a hash byte at the rate of 74 cycles/byte on a smartcard.

8.8 Hardware Estimates

Most of what Abacus does can be implemented very efficiently in hardware. Since fixed bitwise and bytewise permutations are essentially free in hardware, the rolling arrays can be rotated at almost no cost. Furthermore, bytewise XOR is a very cheap operation (perhaps 20-30 gates), and bytewise addition is also fairly low (80-100 gates). It is known also [24] that MDS matrix multiplication and finite fields can be implemented fairly efficiently in hardware. The most expensive component of Abacus, by far, is the S-box. It might take between 1000 and 3100 gates.

Because the absorb and squeeze round functions process data one byte at a time, Abacus has a very small footprint in hardware. There are not dozens of complex, heterogeneous rounds. There is just one round. Furthermore, the similarity between squeeze and absorb rounds suggests that much of the logic could be shared and re-used between the absorb module and the squeeze module.

In [15], the authors mention that AES can be implemented in 3,600 gates, and in [3] the authors mention that RC4 can be implemented in 12,951 gates. Since Abacus is midway between those two in terms of complexity, I'd estimate Abacus's gate count to be 7,500.

9 Additional Uses for Abacus

Hash functions, in their barest form, compute message digests that look random. In the real world, however, hash functions are often used in combination with other data and protocols to provide solutions for common, security-critical tasks.

9.1 Hash Message Authentication Codes (HMAC)

Abacus may be used as a keyed hash message authentication code (HMAC) [31]. The purpose of an HMAC is to provide data integrity and authentication of a message by computing a hash that only a valid user would be able to generate. Essentially, the HMAC accepts a key while hashing a message. This ensures that the hash output value will be dependent upon a key that is only known to the people who are valid users/authenticators.

The HMAC construction is only secure as long as the underlying hash function is secure. Because Abacus is resistant to the attacks that exploit Merkle-Damgård (i.e. message extension, multicollisions, fixed points), it seems certain that Abacus would provide a stronger HMAC than MD5 or SHA-1.

In the following formula for the HMAC construction, *m* is the message, *k* is the key, H is the underlying hash function, ipad and opad are strings of constant bytes, the ‖ symbol denotes concatenation, and the ⊕ symbol denotes XOR:

$$HMAC_k(m) = H\big((k \oplus opad) \parallel H((k \oplus ipad) \parallel m) \big)$$

Abacus can be used in the exact method shown above because it is a drop-in replacement for the SHA-2 family. Bearing in mind that HMAC assumes the message block size is larger than the hash size, I recommend using "pseudo-blocks" of 512-bits for a 224- or 256-bit hash sizes, and using pseudo-blocks of 1024-bits for 384- and 512-bit hash sizes. (A pseudo-block is simply a string of 8-bit message blocks treated as one logical unit.)

Alternatively, Abacus can simplify the HMAC process. By accepting a randomization string (salt), Abacus incorporates the core concept behind HMAC: keying a hash function. Simply use the salt as a key, and digest the message in one pass (without having to call the hash function twice as shown in the HMAC formula two paragraphs above). In Abacus, the opad and ipad strings are not necessary because the internal state is initialized to the S-box, which is a complex and non-linear array of bytes. It may be useful to increase the number of blank rounds to 180 so that the key has even more time to spread entropy throughout the internal state prior to generating the hash output.

9.2 Deterministic Random Bit Generators (DRBG)

Being designed like a stream cipher, Abacus is particularly well-suited to generate long strings of pseudo-random bits. As described in NIST SP 800-90, deterministic random bit generators (DRBGs) built from non-invertible (one-way) hash functions can be used to generate random bit sequences [58]. In sections 4, 5, 6 and 7, I've shown that Abacus is secure and one-way, and therefore appropriate for use as a DRBG. And because Abacus can be used as a drop-in replacement for the SHA-2 family, it can easily be substituted for SHA-224, SHA-256, SHA-384, and SHA-512 in the Hash_DRBG construct described in chapter 10 of SP 800-90 [58].

Abacus has several advantages when being used as a DRBG. In particular, Abacus accepts randomization strings (salts) that can contain the entropy data, nonce, and personalization strings that are required to initialize the Hash_DRBG construct [58]. Abacus also has its own counters, and in combination with the rolling arrays, they guarantee a minimum period of 2^{45}. If Abacus is chosen for SHA-3, NIST can simplify the requirements of the Hash_DRBG construct.

Due to Abacus's large internal state, seed lengths of 888 or higher are probably

appropriate (888 is the size recommended for SHA-384 and SHA-512).

9.3 Digital Signatures (DSS)

As described in FIPS 186-2 [29], the Digital Signature Standards (DSS) requires a hash function that is both one-way and collision resistant. In chapter 4 of FIPS 186-2, the hash function is required to generate hash lengths of 160, 224, and 256 bits. By being a drop-in replacement for the SHA-2 family, Abacus can easily replace SHA-2 in the Digital Signature Algorithm.

Abacus's ability to generate variable length hashes would also be helpful for the DSS. If it becomes necessary to expand the hash length of DSS from 256 bits to 512 bits, or even an arbitrary length, Abacus could easily accommodate that. Whatever the hash size may be, I recommened using pseudo-blocks (as described in section 9.1) that are larger than the hash size. In particular, a hash size of 224- or 256-bits would require a pseudo block of 512-bits, and a hash size of 384- or 512-bits would require a pseudo-block of 1024 bits.

9.4 Pseudo-Random Functions (PRF)

A pseudo-random function (PRF) is one that produces random looking output, no matter how similar the inputs are. In the same way that a random oracle's "answers" are assigned at random to any given "question" [4], a PRF's output must also hide correlations between input questions and make recovery of the input infeasible. Abacus meets these requirements with its preimage resistance and second preimage resistance. Furthermore, Abacus uses blank rounds for the explicit purpose of providing strong decorrelation of the internal state.

10 Advantages and Disadvantages

Disadvantages.

1) The Abacus **S-box is random**, and therefore impossible to simplify. In hardware, it requires more gates, and on a smartcard, it requires more bytes of memory. These are certainly disadvantages for some uses. However, the random S-box is a source of strength against algebraic attacks and unknown future attacks.

2) A further drawback of Abacus is that it **uses blank rounds**. These blank rounds effectively tack on a constant amount of work to finalize the hash value. For short messages (i.e. 1 to 150 bytes), the cost of executing the blank rounds will be larger than absorbing the message itself.

3) Unlike the typical Merkle-Damgård hash function, Abacus uses a more robust form of

message padding which embeds the hash length, the message length, and a random salt into the beginning and end of the overall message. All together, this **padding adds at least 40 bytes** of information to the message, thus increasing the cost of hashing for small messages.

Advantages.

1) By design, Abacus is **platform-agnostic**. It doesn't rely on tricks or optimizations that only exist on one brand of microprocessor or smartcard. It doesn't favor big-endian or little-endian architecture. Instead, Abacus has a flexible and simple set of instructions that will be efficient and easy to implement on any conceivable platform.

2) With its heavy use of **parallelism**, Abacus will perform well in hardware and also on future microprocessors. Speed increases of 2x to 4x are quite possible.

3) Abacus has **three tunable parameters** than can be increased or decreased at will to make speed/security trade-offs. For classified government applications, Abacus could use 180 blank rounds and 2 absorb clocks per round.

4) Abacus's byte-at-a-time structure makes the hash function **easy to analyze**. The effect of a single message byte can be easily observed, which makes cryptanalysis clean and simple. Furthermore, this byte-at-a-time design gives Abacus a small hardware footprint and makes Abacus's performance much easier to optimize.

5) Additionally, Abacus was designed with a **wide margin of security**. Its internal state is more than twice the size of the maximum hash size currently required by NIST. After 98 blank rounds, Abacus is resistant to all known cryptanalytic attacks, but by default it has 135 blank rounds to provide a security buffer.

6) Abacus produces a **variable length hash**. This flexibility makes Abacus a great choice for SHA-3, because over time, computers will get faster and larger hash sizes will be necessary for security. Abacus is fully capable of expanding its hash size to accommodate current and future needs.

11 Conclusions

Abacus is a new cryptographic primitive. With its novel combination of rolling arrays, counters, an MDS matrix, and an S-box, Abacus resembles a stream cipher more than it does a classical hash function.

Initial cryptanalysis efforts find no significant weaknesses in Abacus. In fact, Abacus is resistant to many serious attacks, such as fixed points, multicollisions, second preimage attacks, herding attacks, and length extension attacks.

Over the next several months, new and interesting attacks on Abacus will

probably emerge. I invite—and encourage—anyone to try to break Abacus.

11.1 Future Directions

Abacus lends itself well to alternate designs.

It would be interesting to design a nibble-wise version of Abacus that operates on 4-bit nibbles. For instance, the S-box would be 4x4, the rolling arrays would contain nibbles, and all operations would be on nibbles. Due to time constraints, I was not able to conduct any such tests. There would be, however, definite value in cryptanalyzing a reduced wordsize version of Abacus.

On the other side of the spectrum, it is easy to imagine a version of Abacus that operates on 10-bit words, or even 12-bit words. The S-boxes would be much larger, but also much stronger. The counters would be prime numbers close to 1024, or even 4096. And of course the average expected cycle lengths would grow exponentially.

In extremely memory constrained environments, such as an RFID tag, it could be useful to design a version of Abacus with reduced register sizes, perhaps 19 and 41 bytes for `rc` and `rd`. An 8-bit S-box built from recursive 4x4 S-boxes, such as in Crypton [49], would also cut down the memory and gates required. These changes would reduce the minimum cycle length and the expected security, but could enable the hash function to run on platforms that otherwise wouldn't accomodate one.

Personal computers, on the other hand, don't really need to worry about memory constraints. `rc` and `rd` could be expanded to, say, 73 and 191 bytes, respectively. This would increase the minimum cycle length to 2^{48}, and the average expected cycle length to 2^{2190} . Between the internal state being much larger, and the tap positions spanning a greater amount of history, the hash function would be significantly more secure.

Finally, for ultra-secure applications, Abacus could use secret, application-specific S-boxes, like in the GOST block cipher [64]. It's possible to imagine an Abacus variant with four different and indepedent S-boxes, two of which are algebraically generated, and two of which are randomly generated (but endowed with many good cryptographic properties). And during the absorb and squeeze phases, add a third or fourth series of S-boxes to increase the number of active S-boxes. Just thinking about it makes me want to start coding ;-)

12 Acknowledgements

Abacus would not exist if not for the groundbreaking work of many cryptographers who came before me. I'd like to thank all of them. In particular, the following algorithms and concepts were instrumental:

AES	MDS Matrix and byte-oriented design
HAIFA	Embedding the message length, hash length, and salt in the message
Mars	Forward and Backward mixing should be equally strong
Py (Roo)	Rolling arrays
RadioGatún	Cryptographic sponge with blank rounds
RC4	Invertibly absorbing the key into a large, byte-oriented internal state
RC4 Hash	Ways of using RC4 as a hash function
SNOW	Using a special purpose finite-state machine and shift registers

References

[1] R. Anderson, E. Biham and L. Knudsen. *Serpent: A Proposal for the Advanced Encryption Standard.* AES Submission, 1998. Available online at: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.7154*

[2] J. A. Anderson. *Discrete Mathematics with Combinatorics (2nd Edition)*. Prentice Hall, 2003.

[3] L. Batina, J. Lano, N. Mentens, S. B. Ors, B. Preneel, and I. Verbauwhede. Energy, performance, area versus security trade-offs for stream ciphers. 2004. Available online at: *http://www.cosic.esat.kuleuven.be/publications/article-497.pdf*

[4] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conferences on Computer and Communications Security*, pages 62–73. ACM Press, Fairfax, 1993.

[5] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. Sponge Functions. *Ecrypt Hash Workshop 2007*. Available online at: *http://sponge.noekeon.org/SpongeFunctions.pdf*

[6] G. Bertoni, J. Daemen, G. Van Assche and M. Peeters. *RadioGatún, a belt-and-mill hash function.* NIST's Second Cryptographic Hash Workshop, 2006. Available online at: *http://csrc.nist.gov/pki/HashWorkshop/2006/Papers/VANASSCHE_RadioGatun_0720.pdf*

[7] E. Biham. New Types of Cryptanalytic Attacks Using Related Keys. In *Journal of Cryptology*, Vol.7, No.4, pages 229–246. Springer-Verlag, 1994.

[8] E. Biham and R. Chen. Near-Collision of SHA-0. In *Proceedings of CRYPT 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 290–305. Springer-Verlag, 2004.

[9] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet and W. Jalby. Collisions of SHA-0 and Reduced SHA-1. In *Proceedings of Eurocrypt 2005*, volume 3494 of *Lecture Notes in Computer Science* pages 36–57. Springer-Verlag, 2005.

[10] E. Biham and O. Dunkelman. *A Framework for Iterative Hash Functions – HAIFA*, NIST's Second Cryptographic Hash Workshop, 2006. Available online at: *http://csrc.nist.gov/pki/HashWorkshop/2006/Papers/DUNKELMAN_NIST3.pdf*

[11] E. Biham and J. Seberry. Py (Roo): A Fast and Secure Stream Cipher using Rolling Arrays. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/023, 2005.

[12] E. Biham and A. Shamir. *Differential Cryptanalysis and the Data Encryption Standard.* Springer-Verlag. New York, 1993.

[13] A. Biryukov and D. Wagner. Slide Attacks. In *Fast Software Encryption – 6th International Workshop, FSE'99*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer-Verlag, 199

[14] A. Biryukov and D. Wagner. Advanced Slide Attacks. In *Advances in Cryptology – EUROCRYPT2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 589–606. Springer-Verlag, 2000.

[15] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450-466. Springer-Verlag, 2007.

[16] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Julta, S. M. Matyas Jr., L. O'Connor, M. Peyravian, D. Safford, N. Zunic. *Mars - a Candidate Cipher for AES*. In Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST), 1998.

[17] F. Chabaud and A. Joux. Differential Collisions in SHA-0. In *Advances in Cryptology–Crypto'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer-Verlag, 1998.

[18] D. Chang, K.C. Gupta and M. Nandi. RC4-Hash: A New Hash Function based on RC4. In *Progress in Cryptology - Indocrypt 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 80–94. Springer-Verlag, 2006.

[19] S. Chari, C. Jutla, J. R. Rao and P. Rohatgi. *A Cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards*. Second Advanced Encryption Standard Candidate Conference, pages 133–147, 1999.

[20] N. T. Courtois, G. V. Bard and D. Wagner. *Algebraic and Slide Attacks on KeeLoq*. 2008. Available online at: *citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.73.8273*

[21] N. Courtois, A. Klimov, J. Patarin and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Proceedings Of Eurocrypt 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer-Verlag, 2000.

[22] N. Courtois and J. Pieprzyk. *Cryptanalysis of block ciphers with overdefined systems of equations*. Cryptology ePrint Archive, Report 2002/044, 2002. Available online at: *http://eprint.iacr.org/2002/044.*

[23] N. T. Courtois. General principles of Algebraic Attacks and New Design Criteria for Cipher Components. In *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 474-491. Springer-Verlag, 2004.

[24] J. Daemen and V. Rijmen. *AES Proposal: Rijndael*. Document version 2, March 9th, 1999.

[25] R. D. Dean. *Formal Aspects of Mobile Code Security.* Ph.D. dissertation, Princeton University, 1999.

[26] P. Ekdahl and T. Johansson. A New Version of the Stream Cipher SNOW. In *Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography,*

volume 2595 of *Lecture Notes in Computer Science*, pages 47–61. Springer-Verlag, 2002.

[27] *Federal Information Processing Standard (FIPS 180-1), Secure Hash Standard*. National Institute of Standards and Technology - NIST, 1995.

[28] *Federal Information Processing Standard (FIPS 180-2), Secure Hash Standard*. National Institute of Standards and Technology - NIST, 2002.

[29] *Federal Information Processing Standard (FIPS 186-2) – Digital Signature Standard (DSS).* National Institute of Standards and Technology - NIST, 2000.

[30] *Federal Information Processing Standard (FIPS 197) – Advanced Encryption Standard (AES).* National Institute of Standards and Technology - NIST, 2001.

[31] *Federal Information Processing Standard (FIPS 198) – The Keyed-Hash Message Authentication Code (HMAC).* National Institute of Standards and Technology - NIST, 2002.

[32] *Federal Register*, Vol. 72, No. 212. USA, November, 2007.

[33] S. Fluhrer, I. Mantin and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. *Eighth Annual Workshop on Selected Areas in Cryptography.* 2001.

[34] B. Gladman. *SHA1, SHA2, HMAC, and Key Derivation in C.* Available online at: *http://fp.gladman.plus.com/cryptography_technology/sha/index.htm*, accessed October 25th, 2008.

[35] S. Halevi and H. Krawczyk. *Update on Randomized Hashing.* 2006. Available online at: http://csrc.nist.gov/groups/ST/hash/documents/HALEVI_UpdateonRandomizedHashing0824.pdf

[36] Y. Hilewitz, Y. L. Yin, R. B. Lee. Accelerating the Whirlpool Hash Function Using Parallel Table Lookup and Fast Cyclical Permutation. In *Fast Software Encryption*, volume 5086 of *Lecture Notes in Computer Science*, pages 173-188. Springer-Verlag, 2008.

[37] H. M. Heys. *A Tutorial on Linear and Differential Cryptanalysis.* Memorial University of Newfoundland, Canada, 2002. Available online at: *www.engr.mun.ca/ ~howard/PAPERS/ldc_tutorial.ps*

[38] T. Jakobsen and L. R. Knudsen. The Interpolation Attack on Block Ciphers. In *Fast Software Encryption, FSE'97*, volume 1267 of *Lecture Notes in Computer Science*, pages 28–40. Springer-Verlag, 1997.

[39] R. Jenkins. *Isaac and RC4.* 1996. Availabe online at: *http://burtleburtle.net/bob/rand/isaac.html*

[40] A. Joux. Multicollisions in Iterated Hash Functions. In *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer-Verlag, 2004.

[41] L. Keliher. Substitution-Permutation Network Cryptosystems Using Key-Dependent S-Boxes. M.S. Thesis, Queen's University, Kingston, Canada, 1997.

[42] J. Kelsey. *Some Current Thinking on Hash Functions Within NIST*. 2005. Available online at: *www.ecrypt.eu.org/stvl/hfw/Kelsey.ppt*

[43] J. Kelsey and T. Kohno. Herding hash functions and the nostradamus attack. In *Advances in Cryptology - Eurocrypt'2006*, volume 4004 of *Lecture Notes in Computer Science*, pages

183–200. Springer-Verlag, 2006.

[44] J. Kelsey and B. Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2*n* Work. In *Advances in Cryptology-Eurocrypt'2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer-Verlag, 2005

[45] V. Klima. *Tunnels in hash functions: MD5 collisions within a minute.* Cryptology ePrint Archive, Report 2006/105. 2006. Available online at: *http://eprint.iacr.org/2006/105*

[46] L. R. Knudsen. Truncated and Higher Order Differentials. In *Fast Software Encryption – Second International Workshop*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer-Verlag, 1995.

[47] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[48] P. Kocher, J. Jae and B. Jun. Differential Power Analysis. In *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[49] C. H. Lim. A Revised Version of CRYPTON - CRYPTON Version 1.0. *Proceedings of the 1999 Fast Software Encryption Workshop*, 1999.

[50] S. Lucks. *Design Principles for Iterated Hash Functions*, IACR ePrint Archive, 2004. Available online at: *http://eprint.iacr.org/2004/253.pdf*

[51] M. Matsui. Linear Cryptanalysis Method for DES Cipher. In *Advances in Cryptology – EUROCRYPT' 93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer-Verlag, 1994.

[52] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, USA. 1999.

[53] I. Mironov. (Not so) random shuffles of RC4. In *Advances in Cryptology—CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 304–319. Springer-Verlag, 2002.

[54] S. Mister and C. Adams. Practical S-Box Design. *Workshop on Selected Areas in Cryptography (SAC '96) Workshop Record*, Queens University, 1996, pages 61–76.

[55] S. Murphy and M. Robshaw. Essential algebraic structures within the AES, in *Proceedings of Crypto '02*, volume 2442 of *Lecture Notes in Computer Science*, pages 17–38. Springer-Verlag, 2002.

[56] R. Needham and D. Wheeler. *Tea extensions.* 1997. Available online at: *http://www.cix.co.uk/~klockstone/xtea.pdf*

[57] NIST. *ANSI C Cryptographic API Profile for SHA-3 Candidate Algorithm Submissions.* 2008. Available online at: *http://csrc.nist.gov/groups/ST/hash/documents/SHA3-C-API.pdf*

[58] NIST. *NIST Special Publication 800-90: Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised) (SP 800-90).* March, 2007. Available online at: *http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf*

[59] NIST. *Draft NIST Special Publication 800-106: Randomized Hashing for Digital Signatures (SP 800-106).* July, 2008. Available online at: *http://csrc.nist.gov/publications/drafts/800-*

106/2nd-Draft_SP800-106_July2008.pdf

[60] L. J. O'Connor. On the distribution of characteristics in bijective mappings. In *Advances in Cryptology, EUROCRYPT 93*, volume 765 of *Lecture Notes in Computer Science*, pages 360–370, Springer-Verlag, 1994.

[61] B. Preneel. *Hash Functions: Past, Present and Future.* 2005. Available online at: *http://homes.esat.kuleuven.be/~preneel/preneel_asia05.pdf*

[62] R. Rivest. *The MD5 message-digest algorithm*. Request for Comments (RFC) 1321, Internet Activities Board, Internet Privacy Task Force, 1992.

[63] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Fast Software Encryption - FSE'2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer-Verlag, 2004.

[64] B. Schneier. *Applied Cryptography second edition*. John Wiley & Sons, Inc., New York, 1996.

[65] X. Wang, X. Lai, D. Feng, H. Chen and X. Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In *Advances in Cryptology-Eurocrypt'2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 1–18, Springer-Verlag, 2005.

[66] X. Wang, Y. L. Yin and H. Yu. Finding Collisions in the Full SHA-1. In *Advances in Cryptology-Crypto'2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36, Springer-Verlag, 2005.

[67] X. Wang, H. Yu and Y. L. Yin. Efficient Collision Search Attacks on SHA-0. In *Advances in Cryptology-Crypto'2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 1–16, Springer-Verlag, 2005.

[68] Wikipedia article. *Rijndael mix columns*. Available online at: *http://en.wikipedia.org/wiki/Rijndael_mix_columns*, accessed October 20th, 2008.

[69] Wikipedia article. *Side-channel attack*. Available online at: *http://en.wikipedia.org/wiki/Side_channel_attack*, accessed October 20th, 2008.

[70] H. Yoshida and A. Biryukov. *Analysis of a SHA-256 Variant*. In Proceedings of SAC 2005.

[71] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology-Eurocrypt'2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35, Springer-Verlag, 2005.

[72] M. A. Weiss. *Data Structures & Algorithm Analysis in Java.* Addison Wesley Longman, 1999.

[73] J. Daemen and V. Rijmen. *Resistance Against Implementation Attacks. A Comparative Study of the AES Proposals*. Second Advanced Encryption Standard Candidate Conference, 1999.

Appendix

A.1 Math and Crypto Definitions

Below is a list of mathematical and cryptographic terms. These definitions are meant for the lay-reader. For a more rigorous treatment, refer to [52] and [64].

Avalanche Effect	A basic measure of a function's randomness. The avalanche effect is when a change in a single input bit will cause half of the output bits to change (on average).
Avalanches	A cryptosystem that undergoes a complete avalanche effect *x* times is said to have *x* avalanches.
Bit	A number that can take two values: 0 or 1.
Bitwise Operation	A mathematical operation that is performed on bits.
Block	A fixed-size array of bits. 64-bit, 128-bit, 512-bit, and 1024-bit blocks are common. Most hash functions, especially those that use the Merkle-Damgård construction, digest a message one block at a time. Abacus uses an 8-bit block.
Block Cipher	An encryption paradigm in which a single key is used to encrypt and decrypt data. The encryption is done on blocks of 64-bits or larger. Every bit of the ciphertext should depend on every bit of the plaintext and every bit of the key.
Boolean Function	A function that takes a bit-array of a particular size and returns a single bit.
Branch Number	In a linear transformation, the branch number (or just "branch") expresses how many outputs are guaranteed to change if a certain number of inputs is changed. A branch number is optimal when it equals $n + 1$, where n is the size of the array.
Byte	A byte is 8 bits. A number that can take 256 values, ranging from 0 to 255. All operations on bytes can be thought of as taking place in the Galois Field, **GF** (2^8).
Chaining Variable	A chaining variable that provides feedback from round to round *within* a hash function, or provides feedback from block to block *between* two consecutive calls to a hash function.
Cipher	A code. An algorithm. A method of secret writing. Also, abbreviation for "block cipher" or "stream cipher".
Ciphertext	Encrypted data.
Clock	Advancing a cryptosystem by one position. Usually, clocking entails shifting a register or array to the left by one position, and possibly incrementing a counter. Abacus does both. Sometimes "clock" is synonymous with "round". In Abacus, a round has one or more clocks.
Cryptanalysis	The science of breaking ciphers.
Cryptographic Sponge	A kind of hash function that has a large (usually secret) internal state, uses a permuatation function instead of a compression function, and tends to output a variable length hash value. Cryptographic sponges "asborb" a message and "squeeze" out a hash.
Cryptography	The science of making and breaking ciphers.

Cryptology The science of making ciphers.

Cryptosystem A general term for any kind of cryptographic system. A cryptosystem can refer to a block cipher, a hash function, or a stream cipher, or a protocol involving one of those.

Diffusion Spreading the influence of bits or bytes, so that they affect many other bits or bytes. Diffusion is what allows cryptosystems to achieve the avalanche effect, and also to be resistant to various kinds of cryptanalysis.

Digest To hash a message. To take a large value and make it smaller. Also, "digest" is an abbreviation of message digest.

Exclusive Or Often abbreviated as "XOR". A bitwise operation that is extremely common in cryptography because it's fast, invertible, and simple. For two bits, *j* and *k*, the exclusive or operation returns 1 when *j* and *k* are different from each other, and returns 0 if *j* and *k* are the same. Denoted as ^

Finite Field A mathematical concept in which all operations (such as addition, subtraction, multiplication, and division) take place in a set with a finite number of elements. If the result of an operation would be larger than the size of the set, or negative, then that result "wraps-around". In cryptography, finite fields of 2^8 and 2^{32} are very common.

Galois Field Denoted **GF**. A Galois field is another name for a finite field.

Hamming Distance Abbreviated as HD. The number of elements that differ between two equal-length strings. Just as with Hamming Weight, that string can be any datatype. The HD between [5, 6, 7, 8, 9] and [5, 6, 3, 8, 4] is 2. The HD between the bit-string [01110000] and [01110100] is 1, since 1 bit differs.

Hamming Weight Abbreviated as HW. The number of non-zero elements in a string. That string can be bits, bytes, nibbles, 32-bit words, or any other datatype. For example, the string of 8 bits represented as 00100011 has a HW of 3. A string of 5 bytes represented as 0xA9 0x4E 0x00 0x05 0x7D has a HW of 4, because 4 bytes are non-zero.

Hash "Hash" can be used as a verb, "to hash", meaning to digest something by using a hash function. "Hash" is also used as a noun to refer to the output of a hash function.

Hash Function A mathematical function that takes a message of practically any size and returns a fixed size output. Typically, the output looks random in relation to the input.

Initialization Vector Often abbreviated as IV. Before a message is digested, the hash function is initialized to a certain starting point. In a classic Merkle-Damgård hash function, the chaining variable is simply assigned to a fixed *n*-bit value. Similarly, Abacus initializes its internal state prior to hashing.

Intermediate Hash Value A chaining variable that is passed between two consecutive calls to a compression function. The Merkle-Damgård construction requires an intermediate hash value to be updated as each message block is digested. Abacus does not use an intermediate hash value in the strict definition of the term.

Internal State Data, usually secret, that is stored inside a cryptosystem and updated after each

	round. Instead of using an intermediate hash value, Abacus uses an internal state.
Iterate	To repeat. "Iterate" is often used in reference to rounds.
Key	The key that allows a user to encrypt and decrypt data using a cipher.
Linear	A function is linear when it preserves relationships of addition and multiplication. In other words, multiplicative and additive relations between the input elements are also present in the output elements.
Linear Transformation	A transformation (i.e. simple function) that acts on arrays of bits or bytes. Linear transformations can be used to ensure good diffusion in a block cipher or hash function. But because of their linearity, linear transforms don't provide any security by themselves.
Maximum Distance Separable	Commonly abbreviated as MDS. Maximum distance separable is a theory that's used to describe functions with strong diffusion properties.
MDS	See "Maximum Distance Separable".
MDS Matrix	A matrix that represents an MDS code.
Merkle-Damgård	A common, well-understood, and elegant framework for building hash functions. The Merkle-Damgård construction is built around a compression function that takes a large block and generates a smaller hash output. This hash output is passed from block to block as a chaining variable.
Message	The information that is being digested by a hash function. A message can be practically any length. In real life, a message tends to be a file or a short string of text.
Message Digest	Synonym for "hash function".
mod 2^n	Modular addition in a finite field of size 2^n.
Modular Addition	Addition that takes place in a finite field. If the result of the addition is larger than the finite field size, then the result wraps back around starting from zero. The number 259 in a finite field of size 256 would wrap back around and become 3.
Non-linear	A function is non-linear when it does not preserve relationships of addition and multiplication. Certain non-linear functions are extremely chaotic, and tiny input differences can result in massive, unpredictable output differences.
Plaintext	Raw, unencrypted data.
Pseudocode	A series of instructions that looks similar to a computer program, but is meant for human reading. Normally, pseudocode is higher-level and less formal than a computer program would be.
Pseudo-random	Since cryptosystems and mathematical functions are deterministic, they can never be truly random. Instead, their output is called "pseudo-random". Strong pseudo-random functions are indistinguishable from a random function even after enormous amounts of scrutiny.
Random	Unpredictable, chaotic. In order to destroy correlations between input and

	output bits, every cryptosystem strives to produce random-looking output.
Round	Most hash functions and ciphers are built from small, relatively simple functions called rounds. By repeating (or "iterating") a round, a cipher becomes stronger and more random-looking.
Stream Cipher	A kind of cipher that takes a key, and then produces a very long sequence of random-looking bits (or bytes). Stream ciphers are usually much faster than block ciphers or hash functions, and are built using different design principles.
Subkey	It's common for a block cipher to have several subkeys, which are derived from the (main) key.
S-box	A substitution table that takes one number and returns another. S-boxes are usually permutations, and tend to have equal sized inputs and outputs (i.e. 8-bits each). At the core of many block ciphers and stream ciphers there is an S-box. They provide non-linearity to a cryptosystem.
Word	Depending on the context, a word is either 16, 32, or 64 bits. Operations on words take place in **GF**(2^{16}), **GF**(2^{32}), or **GF**(2^{64}), respectively.
XOR	Exclusive Or.

A.2 S-box Properties

As listed in section 3.2, the Abacus S-box has several properties that are well-understood within the cryptographic community. The definition of each property is given below.

Maximum Difference Distribution Table (DDT) entry. This value is directly related to differential cryptanalysis. To find the maximum DDT entry for an 8-bit S-box, build a two-dimensional table with 256 entries going across and 256 going down. The columns represent input constants (ICs), and the rows represent output constants (OCs). Iterate over all 65,536 entries in the table and tally up the number of times that $\text{sbox}[x \oplus \text{IC}] \oplus \text{sbox}[x] = \text{OC}$ for each value x. The maximum number of times this occurs is the Max DDT entry for the S-box. This number should be as low as possible. 4 is the lowest possible for a mathematical S-box, and 6 is the lowest possible for a random S-box [60]. Also, see [12] for the authors' pioneering work on differential distributions.

Maximum Linear Approximation Table (LAT) bias. This value is directly related to linear cryptanalysis. To find the maximum LAT bias, perform an algorithm similar to the one used for finding the Max DDT entry. While iterating over the 65,536 entries, try to match the following equation for each value x: $(\text{HammingDistance}(x \,\&\, \text{IC}, \text{sbox}[x] \,\&\, \text{OC}) \,\%\, 2) = 0$, where the & symbol denotes bitwise AND, the % symbol denotes modulo, and HammingDistance() returns the number of bits that differ between two bytes. Tally up the number of times that equation holds true for a given IC/OC pair, and subtract the result from 128. Take the absolute value of that. Now, this is the bias for a particular aMask/bMask pair. The Max LAT bias for the overall table is the largest of such biases. 16 is the lowest possible for a mathematical S-box, and 28 is the lowest possible for a random S-box [60]. Also, see [51] for his brilliant initial work on linear

cryptanalysis and the piling-up lemma.

Nonlinear Order. The nonlinear order of an S-box is the minimum nonlinear order over all combinations of affine functions of the input. In other words, the higher the nonlinear order is, the farther the S-box is from being an affine function (or set of affine functions). For an S-box of size *n* bits, the maximal nonlinear order is *n*-1. Nearly every random 8-bit S-box has a nonlinear order of 7. See [46] for more details.

Shortest Cycle. All permutations have cycles. A cycle occurs when *x* is recursively fed into the S-box and eventually lands back on *x*. For instance, if sbox[19] = 19, then that is a cycle of length one (also called a fixed point). If sbox[sbox[254]] = 254, then that is a cycle of length 2. An S-box with no fixed points is called a derangement.

Distance to Strict Avalanche Criteria (DSAC). Strict avalanche criteria requires that if any given input bit is flipped, then any given output bit flips exactly 50% of the time. In practice, no 8-bit S-box can meet the strict avalanche criteria, so instead, the "distance" to strict avalanche criteria is measured. The lower this number is, the better the S-box.

Distance to Maximum Order Strict Avalanche Criteria (DMOSAC). This property is similar to DSAC, only even more grueling. Maximum order strict avalanche criteria requires that if any combination of input bits is flipped, then any given output bit flips exactly 50% of the time. Again, it's not possible for an 8-bit S-box to meet MOSAC, so the distance is measured. The lower this number is, the better the S-box.

Distance to Bit Independence Criteria (DBIC). Bit Independence Criteria requires that if any given input bit is flipped, any two output bits will flip independently of each other. Like with DSAC and DMOSAC, it's not possible for an 8-bit S-box to meet the bit independence criteria, so instead, the distance is measured. Again, the lower this distance, the stronger the S-box.

Distance to Maximum Order Bit Independence Criteria (DMOBIC). This property is similar to DBIC, except instead of flipping one input bit, any number of input bits is flipped. Then, the output is measured to see if any two output bits flip independently of each other or are correlated. Because an 8-bit S-box can't fulfill MOBIC perfectly, the distance to MOBIC is measured. The lower this distance, the stronger the S-box.

See [41] for a lucid discussion on cycles, DSAC, DMOSAC, DBIC, and DMOBIC. In [54], the authors discuss how to build S-boxes with various desirable properties.

A.3 Test Vectors

The following test vectors are in hexadecimal (i.e. 0xFF = 255). The hash length is 256-bits in all of these tests, and the message that generated the hash is shown on the first line of each cluster.

```
# ShortMsgKAT_256.txt
# Algorithm Name: Abacus
# Principal Submitter: Neil Sholer

Len = 0
Msg = 00
MD = 8740006A59E57CE233E5445C3DD8B5D17ED6C8DBEB76DD32358BC5ABFF819C62

Len = 1
Msg = 00
MD = 52BE81BDA27A5660205DA2ECA85CEED2D5F1BCAC65646FBD92B50EFE0A773A62

Len = 2
Msg = C0
MD = C3609D465E172BFC28091BEC8F5E642674CB6EADB6169D31955F739BE1DCE9C9

Len = 3
Msg = C0
MD = 7CFE5C00A6FC927605AE986A5DB1B3E4A1D4534BC131EFE8F16D4CC720D322D5

Len = 4
Msg = 80
MD = C053C97F15C8BBD005B60FA24480F4418CA26EEDAC227ED7183BFB02F239C46B

Len = 5
Msg = 48
MD = 20CF1FF2D1470D1A2280322EE3441021E568213758312915A8EEC2E4074DC247

Len = 6
Msg = 50
MD = 5D104285C466C496F4C15B94E542FB787E5F4DA88FF15898662942BF1A3337B3

Len = 7
Msg = 98
MD = 7A464D6279BD0EFCA749E018D692BF9C03978615B6E4FF2844223A72186D6685

Len = 8
Msg = CC
MD = CD70E14C2E2FFDB554C718A66AE69C01B9A8FF11C2EF1C60A2E35294294D4A98

Len = 9
Msg = 9800
MD = DF968D6D5F8DE36B13261A40FB5A0F109E58F29DE35D722D4C6ACB56BB61B60A

Len = 10
Msg = 9D40
MD = 89E6F62165D9EDB7197DE7FEB6C5AF2C6E82508AA59409C7334EB99293EED868

Len = 11
Msg = AA80
```

```
MD = B552212F62BD1F186910801FF51C025D84661AA476A50BF6378BFEB1FC50A400

Len = 12
Msg = 9830
MD = 4D0351611FDB8988FA2B57A6DF6715F23DD5D769765A74D460E244440CBCF971

Len = 13
Msg = 5030
MD = 11697613BA899ED7DBA921833B793F96E94F2723DD7B2205FE95EB90DE4F8FF7

Len = 14
Msg = 4D24
MD = 07A0BF419E04CA7BB5E53DD445798AD1AAFB9E617020E90C95D6537D8E67460A

Len = 15
Msg = CBDE
MD = 5176B26C98108F99B47BBCE1C9A45545CCFAF237A20765D5D1358218402468B2

Len = 16
Msg = 41FB
MD = B5648D0E638911215CD9AE5DA9C4DBF1F698957844BD7ECBD2CA4A226CD42DAA
```