Implementation and Performance of the SIMON and SPECK Lightweight Block Ciphers on ASICs

Ray Beaulieu
Douglas Shors
Jason Smith
Stefan Treatman-Clark
Bryan Weeks
Louis Wingers

Crypto-Design Office
National Security Agency

Abstract. SIMON and SPECK are families of lightweight block ciphers proposed in June 2013 by the US National Security Agency. Here we discuss ASIC implementations of these algorithms, presenting in some detail how one implements the smallest bit-serial versions of the algorithms. We also give area and throughput results for a variety of implementations—bit serial, iterated, and partially and fully pipelined. To the best of our knowledge, each version of SIMON admits implementations with the smallest area of any comparable block cipher with a flexible key, and SPECK is close behind: at the 64-bit block/128-bit key size, for example, both can be realized in under 1000 GE. More surprisingly, however, since they were intended for use on constrained platforms, SIMON and SPECK allow for extremely high efficiency and high-throughput implementations; each version of SIMON, in particular, has the highest efficiency (throughput divided by area) of any comparably sized block cipher we've seen—lightweight or not.

Keywords: simon, speck, block cipher, lightweight, cryptography, internet of things, ASIC, hardware

Note. This paper was written in the Spring and Summer of 2014. It was submitted to a conference and was not accepted, and subsequently was not published. It is posted here, with a few minor edits, because it provides some more detail regarding the implementations we discuss in our original paper [1].

1 Introduction

Lightweight cryptography has been an active area of cryptographic research in recent years, and many lightweight block ciphers have been proposed. Since there is no single, obvious lightweight application for these designs to target, it's not surprising that different designs seek to optimize performance in different areas within the design space: high throughput in software *or* small area on ASICs or

FPGAs *or* small code size on microcontrollers *or*.... Unfortunately, good performance in one area often signals poor performance in others, and this is likely to cause problems when communication is required across a network consisting of many disparate devices. Such multi-platform applications are likely to be the rule rather than the exception for new applications of lightweight cryptography.

Simon and Speck are block cipher families that promise high performance across a range of platforms. In this paper, we focus on their ASIC performance. We provide details regarding our small-area implementations of these algorithms: the algorithms are capable of being implemented with extremely small footprints, where the cost of the flip-flops necessary to store state accounts for up to about 90% of the total area. To demonstrate one aspect of the flexibility of these algorithms, we also provide data showing that, if area is not tightly constrained, then it is possible to obtain high-throughput implementations with performance surpassing that of algorithms explicitly designed for this sort of application.

Here is a sample of our results: For applications requiring a small footprint, the 64-bit block/128-bit key versions of Simon and Speck can be implemented in 958 gate equivalents (GE) and 996 GE, respectively. These are smaller than the *80-bit-key* version of the leading hardware-oriented cipher PRESENT, which requires 1030 GE [3]. For low throughput applications, the 128-bit block/128-bit key versions of Simon and Speck can be implemented with 1234 and 1280 GE, respectively. Compare this with AES-128, with a minimal area of 2400 GE [8].

When high throughput is necessary (imagine a heterogeneous network, including both constrained and high-speed devices), Simon and Speck offer performance exceeding that of traditional algorithms such as AES. Helion offers a small AES core with a throughput of 650 Mbps and an area of 9500 GE, for an efficiency[1] of 68 [6]. With the same block and key size, Speck achieves 3.53 Gbps at 9662 GE, for an efficiency of 365, and Simon attains a throughput of 3.98 Gbps using 7279 GE, for an efficiency of 547. A fully pipelined (and fully key-agile) Simon 128/128 achieves an efficiency of 731, which is the highest we have seen for any 128-bit block cipher with a 128-bit key.[2]. Pipelined Speck 128/128 has an efficiency of 424, better than the highly efficient CLEFIA [10], whose (scaled) efficiency is 278.[3] High-throughput implementations of Simon and Speck are discussed further in Section 4.

2 ASIC Implementations

A brief review of the specifics of Simon and Speck is given in subsequent sections; complete details can be found in [1]. For the moment, we note that each

[1] Efficiencly is throughput in kbps divided by area in gate equivalents (GE); a GE is the area of the smallest `NAND` gate in the cell library under consideration.

[2] If key agility is not required (i.e., if it's OK to take multiple clock cyles to change keys), then efficiencies can be raised further, to 838 for Simon 128/128 and 605 for Speck 128/128. See Section 4.

[3] CLEFIA reaches 3.74 Gbps using 9330 GE, at 90nm; efficiency = 401 at 90nm, scaling to $\approx 401 \cdot 9/13 = 278$ at the 130nm feature size used in our work.

algorithm has ten variants, with block sizes ranging from 32 bits to 128 bits, and key sizes from 64 bits to 256 bits.

While a whole range of implementations are possible, we'll focus on the two extreme ends of the spectrum: (1) very small bit-serial implementations where area is tightly constrained and throughput is minimally important, and (2) high-throughput, fully pipelined implementations where efficiency is maximized, and area is not constrained. Many other implementations have been investigated (iterated, partially pipelined, etc.), and although we don't have the space to discuss them here, some of the results can be found in Tables 1 and 2.

3 Simon

Each of Simon and Speck has ten variants, parameterized by block and key size. Simon $2n/wn$ refers to the variant of Simon with block size $2n$ and wn bits of key, where n is one of 16, 24, 32, 48, or 64 and w is either 2, 3 or 4 (though not all fifteen possibilities are defined).

The smallest version of Simon, Simon 32/64, requires a total of $T = 32$ rounds; the largest, Simon 128/256, $T = 72$ rounds. Each instance of Simon produces ciphertext from plaintext by applying the following Feistel map for T rounds:

$$R_k(x, y) = (y \oplus (Sx \,\&\, S^8 x) \;\oplus\; S^2 x \;\oplus\; k,\; x),$$

where k is the round key, $\oplus$ denotes bitwise `XOR`, & is a bitwise `AND`, and S^j is a left circular shift by j bits, all on appropriately-sized words.

The Simon key schedules vary slightly depending on the number of key words w. For Simon $2n/wn$, round keys $k_0, \ldots, k_{w-1}$ are set equal to the original w words of key, and for $0 \leq i < T - w$ the round key k_{i+w} is given by

$$k_{i+w} = \begin{cases} c_i \oplus k_i \oplus S^{-3}(k_{i+1}) \oplus S^{-4}(k_{i+1}), & \text{if } w = 2, \\ c_i \oplus k_i \oplus S^{-3}(k_{i+2}) \oplus S^{-4}(k_{i+2}), & \text{if } w = 3, \\ c_i \oplus k_i \oplus k_{i+1} \oplus S^{-1}(k_{i+1}) \oplus S^{-3}(k_{i+3}) \oplus S^{-4}(k_{i+3}), & \text{if } w = 4. \end{cases}$$

Here, c_i is an n-bit round constant whose least significant bit is determined by a 5-bit version-dependent LFSR, with all other bits fixed through all rounds. Details can be found in [1].

4 Pipelined Simon

Although designed primarily for use on constrained platforms, Simon and Speck also support extremely high throughput implementations. And because high-throughput pipelined versions are very easy to understand and describe (all the control logic, in particular, disappears), we lead off by discussing pipelined implementations of Simon.

There are two sorts of pipelining we consider, a *key-agile* and a *non-key-agile* type. In both cases the round function is fully unrolled, so a T-round algorithm with a $2n$-bit state requires $T+1$ $2n$-bit registers to buffer the input and hold all the intermediate values of the state, along with T copies of the round function. For the key-agile version, we also fully unroll the key expansion, so—for an algorithm using w n-bit words of key, i.e., with a key size of wn bits—T wn-bit registers are required to hold the intermediate states of the key schedule. Also required are $T-1$ instantiations of the key schedule logic. In the non-key-agile version, we implement one copy of the key schedule logic, together with enough enable flip-flops to hold all the round keys.

For both versions, after the pipeline is full, we produce one encryption per clock cycle. In the key-agile case, the key can be different with each encryption, after an initial latency (here $T+1$ cycles) necessary to fill the pipeline. In this case we can exclusively use the cheaper D flip-flops, as each register gets input from only one place.

In the non-key-agile case, larger enable flip-flops are needed to store the round keys, but despite this we save considerably on area because we only require one copy of the key schedule logic. By loading round keys in through the bottom of the stack of key storage registers and letting them propagate up, we minimize muxing, at the expense of raising the latency to $2T+1$ cycles ($T+1$ to load the key and generate round keys, and $T+1$ to fill the data pipeline, with plaintext loading happening in parallel with the last round key load), and this overhead will be imposed whenever the key is changed. This, of course, is not the only way of doing things: at the cost of additional muxing, the latency could be lowered to $T+1$ cycles.

We also considered pipelined versions of the algorithms which computed two or more rounds per clock cycle. Here the depth of the pipeline is reduced, with a consequent area reduction. At the same time, however, the maximum clock speed is lowered. For some versions of Simon, two-rounds-per-clock implementations are the most efficient.

A straightforward key-agile pipelined Simon 128/128, for example, at one round per clock cycle, is easy to model. It requires $2 \cdot 128 \cdot 69 - 64 = 17600$ D flip-flops, $68 \cdot 64 = 4352$ `NAND`s, and $3 \cdot 64 \cdot 68 + 2 \cdot 64 \cdot 67 = 21504$ `XOR/XNOR`s. There is no further control, and the constants in the key schedule, which are fixed for each round, are absorbed into `XOR`s and `XNOR`s. In the cell library we used, `XOR/XNOR`s are 2 GE, and D flip flops are 4.25 GE, so the area is $17600 \cdot 4.25 + 4352 \cdot 1.00 + 21504 \cdot 2.00 = 122160$ GE, which is exactly what we achieved for an actual VHDL implementation, when there was no clock constraint. Note that extremely high clock speeds can be achieved for such an implementation, because the critical path involves just a few `XOR`s and an `AND`.

In the general case, we require $(w+2)(T+1)n - w(w-1)n/2$ D flip-flops, nT `NAND`s, and $3nT + (2 + [w=4])(T-w+1)n$ `XOR/XNOR`s.

A basic non-key-agile, pipelined Simon 128/128, one round per clock, uses $128 \cdot 69 = 8832$ D flip-flops for the data path, $64 \cdot 68 + 128$ enable flip-flops (6 GE each) to store the key and the round keys, $68 \cdot 64 = 4352$ `NAND`s, $3 \cdot 64 \cdot 68 + 2 \cdot 64 = 13184$ `XOR/XNOR`s, and 100 or so GE of control logic. This totals to $8832 \cdot 4.25 + 4480 \cdot 6.00 + 13184 \cdot 2.00 + 100 = 90884$ GE.

In the general case we require $2n(T+1)$ D flip-flops, $n(T+w)$ enable flip-flops, nT `NAND`s, and $3nT + (2 + [w = 4])n$ `XOR/XNOR`s.

If decryption functionality is necessary, it can easily by supplied in the non-key-agile setting using just the encryption hardware, with just a few hundred additional GE of control (but with $3T+1$ cycle latency). The resulting circuit will load key into key registers, step forward T times, reverse the order of the words, and then step T times—effectively backing up the key schedule—to generate and store the decrypt round keys. The ciphertext words to be decrypted are loaded in reverse order, and plaintext is read out in reverse order.

The highest efficiency we obtained for Simon 128/128 was 838 for a pipelined non-key-agile implementation. We have not implemented the joint encrypt/decrypt version, but as we've noted, this will add only a few hundred gates to the 104790, resulting in an efficiency exceeding 830. The highest efficiency reported in the literature (that we could find) is 401 for CLEFIA encrypt+decrypt [10], which scales to approximately $401 \cdot 9/13 \approx 278$ at the 130nm feature size we used. The highest reported efficiency for AES encrypt+decrypt is 174 at 90nm (= 121 at 130nm). Simon's efficiency is 3 times that that of CLEFIA and nearly 7 times that of AES.

Comparing Simon 128/256 to the stream cipher SALSA20, for example, yields an even starker contrast. The highest efficiency we've seen reported for SALSA20 is 35.9 (669 Mbps at 18626 GE) [2]; a pipelined non-key-agile implementation of Simon 128/256 encrypt+decrypt can achieve 87.2 Gbps at 110875 GE, for an efficiency of 786—a 20-fold improvement.

5 Serialized Simon

In this section we discuss our smallest, bit-serial implementations of Simon, which we described in VHDL using generics to allow selection of the block size, key size, and amount of serialization. We also discuss how we modeled our implementations prior to using VHDL aid in the search for further optimizations. The modeling described was extremely helpful in developing our VHDL code, as we had very clear expectations of how much area should be required for an optimal implementation, and we were ultimately able to match these values quite closely.

See the Appendix (in particular, Table 1), where we report areas for small, serialized implementations of the algorithms.

The designs were not taken through place and route, therefore the estimates do not include factors like clock tree distribution, power distribution, and buffering of large fanout signals. However, given the small size of the designs these factors should be minimal.

5.1 Serializing the round function

Serialization refers to the ability to build a circuit that computes a fraction of a round in a clock cycle. This allows for reduced-area implementations, but at a cost to throughput, as many cycles may be needed to complete an encryption.

Serial implementations are parameterized by the number of bits b updated per clock cycle. For an *unserialized* or *iterated* implementation, b is equal to the word size n for SIMON and $2n$ for SPECK. SIMON and SPECK admit serial implementations for any integral value of b, but these implementations are particularly efficient if b is a divisor of the word size n, and those are the implementations we consider in this paper.[4]

Figure 1 shows (a version of) the most aggressive level of serialization for SIMON $2n$, where one bit is updated per clock cycle, i.e., $b = 1$. We use a total of $2n$ clock cycles to load the plaintext, and then nT cycles to accomplish the encryption. The $b = 1$ case is a good place to start, as understanding this case allows one to understand serialization for $b > 1$. (For example, a $b = 2$ serialization can be obtained simply by imagining how one double steps the diagram of Figure 1.)

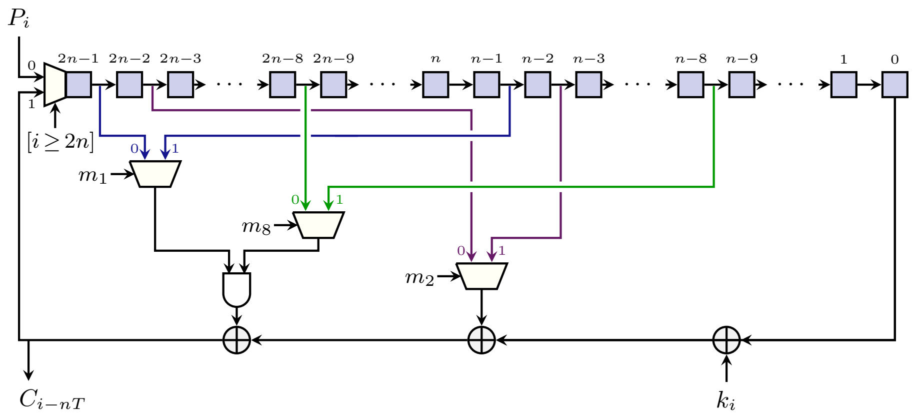

Fig. 1. SIMON round function serialization, one bit at a time. The clock steps from $i = 0$ to $(T + 2)n$. P_i denotes the i^{th} bit of plaintext, loaded at time i. k_i represents the round key bit required at time i. The control bits m_k into the MUXes are given by counter-dependent values $m_k = m_k(i) = [(i \bmod n) \geq k]$ (where $[a] = 1$ if a, else 0). Ciphertext bits are output during the final $2n$ cycles of encryption.

We note that while 1-bit serialization is the easiest case to understand and think about, in many cases one may prefer 2, 3, or even 4-bit serialization for an area-constrained application, which would double, triple, or quadruple the throughput, at the cost of only a handful of additional gates. See Table 1.

It's apparent from the diagram that very little combinational logic is necessary to implement SIMON: The entire "cryptographic portion" of Figure 1 consists of a single AND gate and three XOR gates. Three MUXes are necessary as the bits that feed into these gates are sometimes on the left half of the picture, and sometimes on the right, depending on the clock. In the cell library we used,

[4] For b a nondivisor of n, a slightly different approach, involving a swap of the high byte of the left and right words, is needed to keep the amount of MUXing under control.

`AND`s are 1.25 GE, `XOR`s are 2, and `MUX`es are 2.25. The combinational logic here requires just $3 \cdot 2.25 + 3 \cdot 2 + 1.25 = 14$ GE. We will push forward this running gate count to get an estimate of the size of the entire Simon circuit, but we'll limit this model to only the simplest of logic gates. (An actual VHDL implementation will make use of a full cell library and likely realize a reduction in this area.)

The vast majority of the circuit area in Figure 1 is devoted to holding the state. For this we need $2n$ flip-flops. One of these is a scan flip-flop, which can take input from two places, and the others are smaller D flip-flops. Scan flip-flops are used initially to load plaintext and subsequently to receive values updated by the combinational logic, while D flip-flops update their contents in exactly the same way at each tick of the clock. Scan flip-flops in our cell library cost 6.25 GE and D flip-flops cost 4.25 GE, so the total area for the flip-flops is $6.25 + (2n - 1) \cdot 4.25 = \frac{17n}{2} + 2$ GE.

5.2 Serializing the key schedule

The key schedule diagram is also quite simple, The combinational logic required for everything except the constant add amounts to 2 `MUX`es and 3 `XOR`s for the 2- and 3-word key schedules, and an additional 2 `MUX`es and 2 `XOR`s for the 4-word key schedules. The diagram for the special case of the 2-word key schedule is shown in Figure 2. If w denotes the number of words in the key schedule, then, a total of $10.5 + 8.5 \cdot [w = 4]$ GE[5] are required for the key schedule combinational logic. Again, the major cost is for the flip-flops: one scan flip-flop for loading purposes, and an additional $nw - 1$ D flip-flops. This area is $\frac{17n}{4} w + 2$ GE.

Loading the entire key requires nw clock cycles, but after just n cycles, the first word of key is available for use. To keep things in sync with the plaintext loading, we will wait until $t = 2n$ to start reading out key bits and performing encryption steps.

5.3 Serialized control

Some further logic is required for the control and for the round constant appearing in the key schedule. Specifically: (1) five flip-flops and a few `XOR`s for the 5-bit linear feedback shift register (LFSR) which generates the sequence of low bits used in the round constant (the LFSR doubles as state required to count steps of the cipher), (2) two `XOR`s and an `AND` in order to accomplish the constant add in the key schedule, (3) a 1-up counter large enough so that, together with the LFSR, we can generate enough state to count to $2n + nT$ (loading and encrypting). This requires a $\lceil \log_2(n(T+2)/31) \rceil$-bit counter. (4) Finally, control for the `MUX`es needs to be computed. This control and the circuitry to step the counter amounts to about a dozen `AND`s and half a dozen `XOR`s. A rough estimate,

[5] Here $[x] = 1$ if x, else 0.

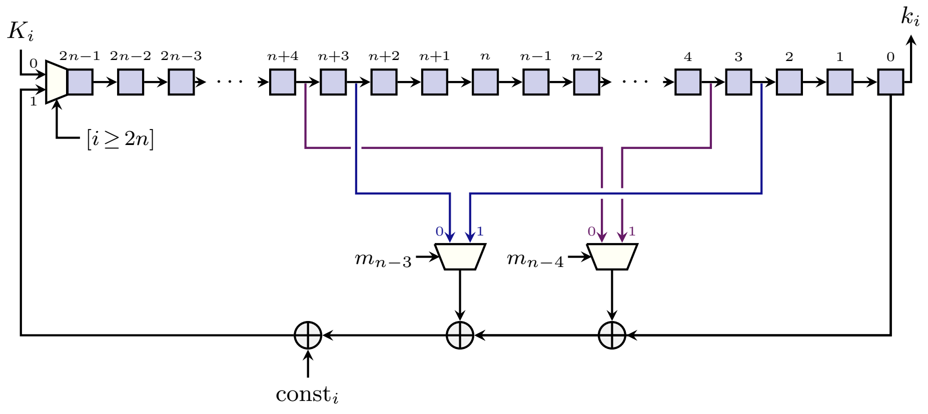

Fig. 2. Simon (two-word) key schedule serialization, one bit at a time; other sizes are similar. The clock steps from $i = 0$ to $(T + 2)n$. const$_i$ represents the constant bit required at time i, K_i the key bit to be loaded at time i, and k_i the round key bit produced at time i. The control bits m_k into the `MUX`es are given by $m_k = m_k(i) = [(i \bmod n) \geq k]$ (where $[a] = 1$ if a, else 0).

ignoring unavoidable (but minor) case-dependent details, is that all this requires $55 + \lceil \log_2(n(T + 2)/31) \rceil$ GE.

Our estimate for the total area, then, for the $b = 1$ serialized implementation of Simon $2n/wn$ is about

$$\frac{17n}{4} \cdot (w + 2) + \frac{17}{2} \cdot [w = 4] + \frac{25}{4} \cdot \lceil \log_2(n(T + 2)/31) \rceil + 83.5$$

gate equivalents. The throughput is $\frac{2n}{n(T+2)}$ bits per clock cycle, $\frac{200n}{n(T+2)}$ kbps for a 100 kHz clock. See Table 1 for actual areas of these implementations.

This formula agrees with our actual results for the $b = 1$ serialized versions of Simon to within about 15 GE in every case. It turns out that for this aggressive level of serialization, 91% to 96% of the area required goes toward flip-flops to hold the state, key, 5-bit LFSR, and 1-up counter.

We'd like to point out an additional subtlety with implementing the LFSR that determines the round constant. Depending upon the block size and the level of serialization, there are two natural ways to do this. The simplest way is to step the register once per round, exactly as in the algorithm description. This is conceptually simple, and takes advantage of the very lightweight stepping rule given for the counter, but does require the use of enable flip-flops so that values can be kept fixed for the duration of a round. In some instances, this yields the lowest-area implementation.

Another approach, loosely illustrated in Figure 3, is to step the register once with each clock cycle, rather than stepping it only once per completed round. This requires the use of a suitable power of the original linear update map, chosen so that after stepping during each clock cycle, we've effectively carried out just

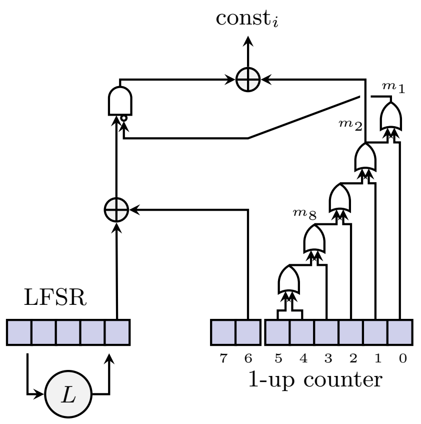

Fig. 3. Simon 128/128, one bit at a time: control. Other sizes are similar. The 6-bit counter needed to track bits within a round is extended by 2 bits so that enough state is present to count up to the requisite $(68+2)\cdot 64$ steps. Additional control bits needed for the `MUX`es in the key schedule (m_{n-3} and m_{n-4}) are not shown. Also not shown are control values signaling the end of loading and the beginning of cipher output.

one step of the original map. In the special case of our running example, Simon 128/128 serialized at $b = 1$ bit per clock cycle, the linear map that accomplishes this is $L = U^{16}$, where U is the linear map that does the 5-bit LFSR stepping. (This works since $L^{64} = U^{16\cdot 64 \bmod 31} = U$.) To produce the bits needed in the correct order for the bit-serial version, we output $((0,0,0,0,1)\,L^i\,(0,0,0,0,1)^t \oplus i_6)\,\overline{m}_1) \oplus m_2$, where i is an 8-bit counter, i_j is its jth bit, and $\overline{c}$ denotes the complement of the bit c. $m_k = [(i \bmod 64) \geq k]$, which equals the `OR` of the set $\{i_5, i_4, \cdots, i_\ell\}$ if $k = 2^\ell$. Implementing this particular choice of L requires a fair amount of combinational logic, and so for this case the simple method of stepping the LFSR only once per round probably produces the smaller implementation. In general, however, it's worth considering both possibilities and allowing the hardware simulator to determine which is optimal.

6 Speck

Although Speck was optimized for software performance, in keeping with our design goals, hardware performance was not neglected. For area critical applications, Speck has ASIC implementations which are nearly as small as those of Simon and smaller than those of the existing hardware-optimized (flexible key) lightweight block ciphers.

The Speck family of block ciphers supports ten variants, using the same combinations of block, and key size as Simon. The key dependent Speck round function is defined by

$$R_k(x,y) = ((S^{-\alpha}(x) + y) \oplus k, S^{\beta}(y) \oplus (S^{-\alpha}(x) + y) \oplus k),$$

where k is the round key, $\oplus$ denotes bitwise XOR, $+$ means addition modulo 2^n and S^j and S^{-j} represent left and right circular shifts, respectively, by j bits. $\alpha = 7$ for SPECK 32/64, and $\alpha = 8$ for all other versions. In each case $\beta = \alpha - 5$.

The round function is applied $T = 22$ times for the smallest variant, SPECK 32/64, all the way up to $T = 34$ times for the largest variant, SPECK 128/256.

The SPECK key schedule makes use of the SPECK round function in a straightforward manner and is described Section 6.3.

6.1 Pipelined SPECK

As with SIMON, we implemented key-agile and non-key-agile pipelined versions of SPECK. (Refer to Section 4 for a discussion of what this means.) For SIMON, we estimated areas for these implementations, and we could do the same for SPECK. However, this becomes a bit more complicated, and in particular depends on area-vs.-throughput considerations that affect the choice of adder. Because of space constraints we have omitted those estimates here.

We note that the most efficient pipelined versions of SPECK always compute one round per clock cycle. See Table 2. Also observe in the table that in passing from the key-agile to the non-key-agile version, a slightly larger improvement is realized than was the case for SIMON; this is because SPECK has a heavier key schedule than SIMON, so more is to be gained by removing the multiple instantiations of it needed for the key-agile pipeline.

6.2 Serializing the round function

The Feistel structure of SIMON facilitated our bit-serial implementation, as an entire word remained unchanged through the course of a round. SPECK, on the other hand, is not a proper Feistel-stepping block cipher. As noted in [1], though, we can think of SPECK as "the composition of two Feistel-like maps," and it is easiest to consider these one at a time, and then put them together.

Specifically, we view $R_k(x, y)$ as the composition of the following two maps:

$$f_1(x, y) = (S^{\alpha}(y), (S^{-\alpha}(x) + y) \oplus k);$$
$$f_2(x, y) = (y, S^{-5}(x) \oplus y).$$

For the $b = 1$ serialization, we process these two steps in n cycles each, and so a single round is executed in the same number of cycles as the block size. This is twice the number of cycles SIMON required. This is perhaps not surprising, since the number of rounds required for each version of SPECK is quite close to half the number of rounds required for the similarly-sized version of SIMON.

The task of simultaneously conceptualizing the 1-bit serializations of both SIMON and SPECK is made a little easier if the diagrams for the two logics are as similar as possible. In order to create a diagram for SPECK that closely mimics Figure 1, we choose to reverse the usual order of the pair of state registers (the words x and y appearing in R_k, f_1 and f_2), viewing y as the bits in positions $2n - 1, 2n - 2, \ldots, n$, and x as the bits in positions $n - 1, n - 2, \ldots, 0$.

Given this notational convention, the plaintext (x, y) is shifted into our $2n$-bit state beginning with the least significant bit of x, and continuing through the most significant bit of y. Before sketching out the entire circuit, we first diagram our separate, fairly simple, implementations of f_1 and f_2.

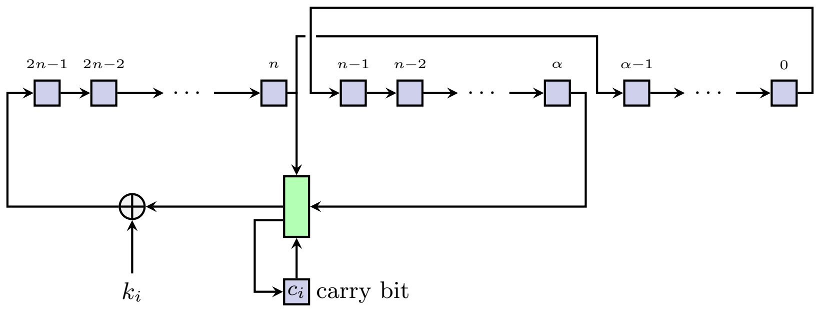

Fig. 4. Serialized, one-bit-per-clock implementation of the partial SPECK round function $f_1 : (x, y) \mapsto (S^{\alpha}(y), (S^{-\alpha}(x)+y) \oplus k)$. The green rectangle represents a full adder, which takes as input two bits to be added, together with a carry bit, and returns the sum bit and the next carry bit.

Figure 4 shows a circuit for computing f_1. Note the word stored in positions $n - 1$, $n - 2$, …, 0 is slowly overwritten by a left-shifted version of the word originally stored in positions $2n - 1$, $2n - 2$, …, n. The result of the modular addition and the `XOR`ed key is accumulated in position $2n - 1$.

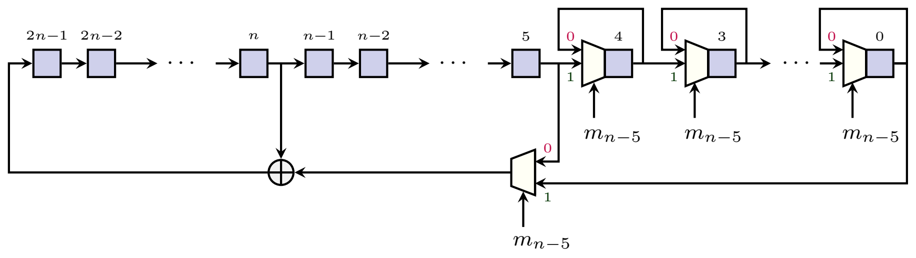

Fig. 5. Serialized, one-bit-per-clock implementation of the partial SPECK round function $f_2 : (x, y) \mapsto (y, S^{-5}(x) \oplus y)$. The 5 rightmost enable flip flops allow the contents to be "frozen" for the first $n - 5$ of n steps; here $m_k = m_k(i) = [i \geq k \pmod n]$.

Figure 5 illustrates the computation of the second function f_2, which is further split into two parts. In the first part, the least significant 5 bits of the state are kept fixed and the `MUX` selects the value of bit 5 as input to the `XOR`. During the final 5 clock cycles, the state resumes its normal left-to-right motion, and the `MUX` selects bit 0 as input to the `XOR`. Note that this portion of the algorithm becomes more difficult conceptually when multiple bits are processed per cycle, and additional `MUX`ing is required to prevent the least significant 5 bits of the state from being overwritten until they are needed at the end of the round.

The notional hardware that combines loading ($2n$ cycles suffice to load the plaintext and enough key to start encrypting and stepping the key schedule) and both computation steps is illustrated in Figure 6. The input to bit position $2n-1$ is interpreted as ciphertext during the final $2n$ cycles of encryption.

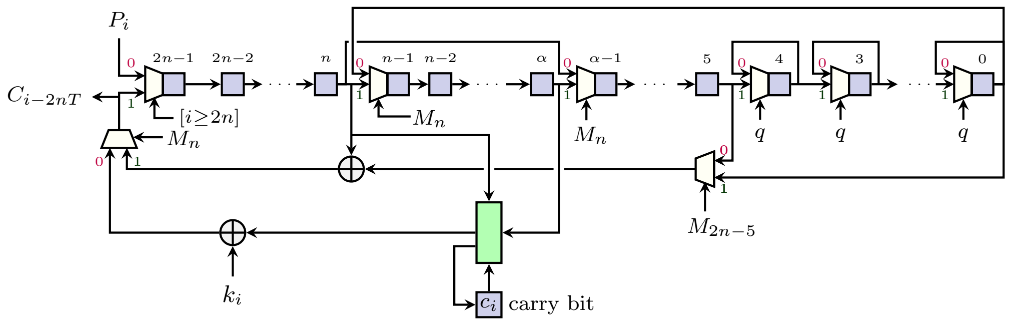

Fig. 6. One-bit-at-a-time serialized Speck round function. This figure combines the previous two, so that n steps of f_1 can be computed, followed by n steps of f_2. The clock steps from $i=0$ to $(2T+2)n$. The control bits are defined in terms of $M_k = M_k(i) = [i \geq k \pmod{2n}]$. Here $q = M_{2n-5}$ `OR` $\overline{M_n}$, so that the contents of the right five enable flip flops are frozen in steps $n, n+1, \ldots, 2n-6$ during each round of $2n$ steps.

6.3 Serializing the key schedule

Unlike Simon, the Speck key schedule reuses the round function, but with a counter value in place of the round keys. Since there may be more than two words of key, one or more of the words stored in the wn key flip-flops will remain unchanged at the end of the round. Specifically, we break the key schedule map into the composition $g_2 \circ g_1$, where g_1 and g_2 are defined differently depending upon the number of words of key w.

w	$\mathbf{g_1}$	$\mathbf{g_2}$
2	$(S^{\alpha}(b),\, (S^{-\alpha}(a)+b) \oplus \texttt{ctr})$	$(b,\, b \oplus S^{-5}(a))$
3	$(a,\, S^{\alpha}(c),\, (S^{-\alpha}(b)+c) \oplus \texttt{ctr})$	$(c,\, a,\, c \oplus S^{-5}(b))$
4	$(a,\, b,\, S^{\alpha}(d),\, (S^{-\alpha}(c)+d) \oplus \texttt{ctr})$	$(d,\, a,\, b,\, d \oplus S^{-5}(c))$

In cases where an entire word of key state is held constant, we prefer to rotate the word in place rather than introducing n additional large enable flip-flops. This likely uses more power, but saves area.

6.4 Serialized control

The primary role of the control block in Speck is to determine when encryption has completed. Since Speck makes use of the round number in the key schedule, an integer counter is used to track the current round rather than a potentially more efficient solution like a shift register. The size of this counter is tailored to the number of rounds used for a particular combination of block and key size and is therefore 5 or 6 bits wide.

Serialized implementations must also track progress within a round and determine when a round has been completed. In this implementation this is also done with an integer counter.

As with Simon, we can estimate the area required for these 1-bit serialized versions of Speck. The round function requires roughly $8 \cdot \frac{25}{4} + (2n - 8) \cdot \frac{17}{4} + 2 + \frac{23}{4} + 2 \cdot \frac{9}{4}$ GE, the key schedule about $(8 + w - 2) \cdot \frac{25}{4} + (wn - (8 + w - 2)) \cdot \frac{17}{4} + 2 + \frac{23}{4} + 2 \cdot \frac{9}{4}$ GE, and the control about $C \cdot \frac{25}{4} + (2C - 2) + (2C - 2)2$ GE, where $C = \lceil \log_2(2n(T + 2)) \rceil$ is the size of the counter. In fact this yields a very slight overestimate of the results we get (less than 10 GE in all but one case), presumably due to optimizations we haven't accounted for here.

References

1. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The Simon and Speck Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404 (2013). `http://eprint.iacr.org/`.
2. Bernstein, D. Snuffle 2005: the Salsa20 encryption function. `cr.yp.to/snuffle.html`.
3. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007, LNCS vol. 4727, pp. 450–466. Springer, Heidelberg (2007).
4. Cannière, C.D., Dunkelman, O., Knežević, M.: KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: CHES 2009, LNCS vol. 5747, pp. 272–288. Springer, Heidelberg (2009).
5. Gong, Z., Nikova, S., Law, Y.W..: KLEIN: A New Family of Lightweight Block Ciphers. In: Juels, A., Paar, C. (eds.) RFID. Security and Privacy, LNCS vol. 7055, pp. 1–18 (2012).
6. Helion Technology Ltd.: Standard AES cores. Available at `http://www.heliontech.com/aes\_std.htm`.
7. Knudsen, L., Leander, G., Poschmann, A., Robshaw, M.J.B.: PRINTcipher: A Block Cipher for IC-printing. In: Mangard, S., Standaert, F. (eds.) CHES 2010, LNCS vol. 6225, pp. 16–32, Springer, Heidelberg (2010).

8. Moradi, A., Poschmann, A., Ling, S., Parr, C., Wang, H.: Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In: Patterson, K.G. (ed.) Advances in Cryptology — EUROCRYPT 2011, LNCS vol. 6632, pp. 69–88, Springer, Heidelberg (2011).
9. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo: An Ultra-Lightweight Blockcipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011, LNCS vol. 6917, pp. 342–357. Springer, Heidelberg (2011).
10. Sugawara, T., Homma, N., Aoki, T., Satoh, A.: High-performance ASIC Implementations of the 128-bit Block Cipher CLEFIA. In: Circuits and Systems, ISCAS 2008, pp. 2925–2928.
11. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: TWINE: A Lightweight, Versatile Block Cipher. In: Leander, G., Standaert, F. (eds.) ECRYPT Workshop on Lightweight Cryptography, pp. 146–169. November 2011. Available at `http://www.uclouvain.be/crypto/ecrypt\_lc11/static/post\_proceedings.pdf`.
12. Wu W., Zhang, L: LBLOCK: A Lightweight Block Cipher. In: ACNS 2011, LNCS vol. 6715, pp. 327–344, Springer, Heidelberg (2011).
13. Yap, H., Khoo, K., Poschmann, A., Henricksen, M.: EPCBC — A Block Cipher Suitable for Electronic Product Code Encryption. In: Lin, D., Tsudik, G., Wang, X. (eds.) CANS 2011, LNCS vol. 7092, pp. 76–97. Springer, Heidelberg (2011).

A ASIC Performance

Table 1 shows results for a variety of small ASIC implementations for the Simon and Speck *encryption* algorithms (since one would typical use encrypt-only modes in lightweight cryptography, i.e., ones which never invoke the decryption algorithm). The VHDL code in both cases was synthesized in Synopsys Design Compiler version 2012.06, and a commercial $0.13\,\mu m^2$ CMOS library was used as a synthesis target. Area results were converted to gate equivalents by dividing the reported area by the area of the smallest drive-strength 2-input `NAND` gate.

All of the data in Table 1 shows our results assuming a clock speed of 100 kHz. (We note that this clock constraint is very easily met by almost any conceivable implementation of the algorithms.) This data shows (1) the smallest-area implementations we were able to achieve for various levels of serialization b, b dividing the word size; (2) the smallest areas for iterated, one-round-per-clock, implementations (which update n bits for Simon and $2n$ bits for Speck, where n is the word size); and results for slightly larger, but higher-efficiency, implementations. In the latter case, we allowed up to 6 rounds per clock cycle for Simon and 3 for Speck, and chose the most efficent implementation we obtained.

Table 1: Small-area hardware performance: serialized and one-round-per-clock ASIC implementations of SIMON and SPECK. Area is given in GE, and throughput is at a clock speed of 100 kHz.

		SIMON			**SPECK**		
block/ key size	**bits/ cycle**	**area (GE)**	**tput (kbps)**	**eff**	**area (GE)**	**tput (kbps)**	**eff**
32/64	1	523	5.6	0.011	580	4.2	0.007
	2	535	11.1	0.021	642	8.3	0.013
	4	566	22.2	0.039	708	16.7	0.026
	8	627	44.4	0.071	822	33.3	0.041
	16/32	722	88.9	0.123	850	123.1	0.145
	$4n/3n$	1456	355.6	0.244	1643	355.6	0.217
48/72	1	631	5.1	0.008	693	4.3	0.006
	2	639	10.3	0.016	752	8.5	0.011
	3	648	15.4	0.024	777	12.8	0.016
	4	662	20.5	0.031	821	17.0	0.021
	6	683	30.8	0.045	848	25.5	0.030
	8	714	41.0	0.057	963	34.0	0.035
	12	765	61.5	0.080	1040	51.1	0.049
	24/48	918	123.1	0.134	1152	192.0	0.167
	$6n/3n$	2356	685.7	0.291	2279	533.3	0.234
48/96	1	739	5.0	0.007	794	4.0	0.005
	2	750	10.0	0.013	857	8.0	0.009
	3	763	15.0	0.020	884	12.0	0.014
	4	781	20.0	0.026	932	16.0	0.017
	6	804	30.0	0.037	961	24.0	0.025
	8	839	40.0	0.048	1081	32.0	0.030
	12	898	60.0	0.067	1167	48.0	0.041
	24/48	1062	120.0	0.113	1254	177.8	0.142
	$6n/4n$	2788	685.7	0.246	2867	685.7	0.239
64/96	1	809	4.4	0.005	860	3.6	0.004
	2	815	8.9	0.011	918	7.3	0.008
	4	838	17.8	0.021	984	14.5	0.015
	8	891	35.6	0.040	1095	29.1	0.027
	16	1004	71.1	0.071	1338	58.2	0.044
	32/64	1213	142.2	0.117	1522	220.7	0.145
	$6n/3n$	3124	800.0	0.256	2991	640.0	0.214

Continued on next page

block/ key size	**bits/ cycle**	**area (GE)**	**tput (kbps)**	**eff**	**area (GE)**	**tput (kbps)**	**eff**
64/128	1	958	4.2	0.004	996	3.4	0.003
	2	968	8.3	0.009	1058	6.9	0.007
	4	1000	16.7	0.017	1127	13.8	0.012
	8	1057	33.3	0.032	1247	27.6	0.022
	16	1185	66.7	0.056	1506	55.2	0.037
	32/64	1415	133.3	0.094	1658	206.5	0.125
	$5n/3n$	3290	640.0	0.195	3120	640.0	0.205
96/96	1	955	3.7	0.004	1012	3.4	0.003
	2	965	7.4	0.008	1067	6.9	0.006
	3	971	11.1	0.011	1089	10.3	0.009
	4	984	14.8	0.015	1134	13.8	0.012
	6	1007	22.2	0.022	1157	20.7	0.018
	8	1037	29.6	0.029	1267	27.6	0.022
	12	1088	44.4	0.041	1328	41.4	0.031
	16	1151	59.3	0.052	1514	55.2	0.036
	24	1263	88.9	0.070	1673	82.8	0.049
	48/96	1580	177.8	0.113	2058	320.0	0.155
	$6n/3n$	4372	960.0	0.220	4229	872.7	0.206
96/144	1	1160	3.5	0.003	1217	3.3	0.003
	2	1169	7.0	0.006	1269	6.6	0.005
	3	1175	10.5	0.009	1297	9.8	0.008
	4	1189	14.0	0.012	1345	13.1	0.010
	6	1211	21.0	0.017	1371	19.7	0.014
	8	1242	28.1	0.023	1485	26.2	0.018
	12	1292	42.1	0.033	1558	39.3	0.025
	16	1354	56.1	0.041	1751	52.5	0.030
	24	1467	84.2	0.057	1928	78.7	0.041
	48/96	1784	168.4	0.094	2262	300.0	0.133
	$6n/3n$	4660	960.0	0.206	4481	872.7	0.195
128/128	1	1234	2.9	0.002	1280	3.0	0.002
	2	1242	5.7	0.005	1338	6.1	0.005
	4	1263	11.4	0.009	1396	12.1	0.009
	8	1317	22.9	0.017	1488	24.2	0.016
	16	1430	45.7	0.032	1711	48.5	0.028
	32	1665	91.4	0.055	2179	97.0	0.045
	64/128	2090	182.9	0.088	2727	376.5	0.138
	$6n/3n$	5820	984.6	0.169	5527	1066.7	0.193

Continued on next page

block/ key size	bits/ cycle	area (GE)	tput (kbps)	eff	area (GE)	tput (kbps)	eff
128/192	1	1508	2.8	0.002	1566	2.9	0.002
	2	1514	5.6	0.004	1627	5.8	0.004
	4	1536	11.1	0.007	1687	11.6	0.007
	8	1587	22.2	0.014	1797	23.2	0.013
	16	1700	44.4	0.026	2038	46.4	0.023
	32	1937	88.9	0.046	2536	92.8	0.037
	64/128	2369	177.8	0.075	3012	355.6	0.118
	$6n/3n$	6204	984.6	0.159	5859	1066.7	0.182
128/256	1	1782	2.6	0.001	1840	2.8	0.002
	2	1792	5.3	0.003	1901	5.6	0.003
	4	1823	10.5	0.006	1967	11.1	0.006
	8	1883	21.1	0.011	2087	22.2	0.011
	16	2010	42.1	0.021	2341	44.4	0.019
	32	2272	84.2	0.037	2872	88.9	0.031
	64/128	2756	168.4	0.061	3284	336.8	0.103
	$6n/3n$	7356	984.6	0.134	6480	984.6	0.152

Table 2 shows our results for several high-throughput versions of Simon and Speck. We present data for a small iterated version with reasonably high efficiency, a mid-sized partially unrolled version, our most efficient key-agile and non-key-agile pipelined implementation. As area increases, we see corresponding improvements in efficiency. But the reader should note that even the smallest have quite high efficiencies, as compared to many existing algorithms.

In every case, the highest efficiency is to be had by fully pipelining, but the resulting circuits are quite large, at least by "lightweight" standards, ranging up into hundreds of thousands of gates.

We note that we have much more data, representing many different areas, clock speeds, and efficiencies. Table 2 shows a representative sample.

The columns in Table 2 show the **area** in gate equivalents, the maximal clock speed (**clk**) in MHz and the throughput (**tput**) in Mbps at that clock speed, the efficiency (**eff**) in kbps/GE, rounds computed per clock cycle (**rpc**), the number of pipeline stages (**stg**) (1 for an iterated implementation, 2 for a version that can hold two full blocks and keys at a time, and the total number of rounds for a fully pipelined implementation), and the average number of clock cycles between cipher outputs (**cyc**), including any cycles necessary for loading. The implementations shown, in order of increasing efficiency, are iterated, one-round per clock (**it**); partially pipelined (**pp**); key-agile, fully pipelined (**kap**); and non-key-agile, fully pipelined (**nkap**).

Table 2: High-throughput ASIC performance for Simon and Speck, 0.13 μm^2 CMOS library.

algorithm	area	tput	eff	clk	type	rpc	stg	cyc
Simon 32/64	902	589	653	625	it	1	1	34
	2952	2079	704	715	pp	1	3	11
	15918	16933	**1064**	529	kap	2	16	1
	12914	21691	**1680**	678	nkap	1	32	1
Speck 32/64	970	368	379	299	it	1	1	26
	4413	2567	582	481	pp	1	4	6
	24089	16737	**695**	523	kap	1	22	1
	14044	16277	**1159**	509	nkap	1	22	1
Simon 48/72	1067	843	790	667	it	1	1	38
	4531	3693	815	769	pp	1	4	10
	33224	41121	**1238**	857	kap	1	36	1
	21202	32618	**1538**	680	nkap	1	36	1
Speck 48/72	1555	706	454	368	it	1	1	25
	5625	3192	568	399	pp	1	4	6
	27025	19101	**707**	398	kap	1	22	1
	17571	18670	**1063**	389	nkap	1	22	1
Simon 48/96	1344	790	587	625	it	1	1	38
	8793	5647	642	588	pp	2	4	5
	26249	25418	**968**	530	kap	2	18	1
	21477	32426	**1510**	676	nkap	1	36	1
Speck 48/96	1929	764	396	429	it	1	1	27
	7036	3711	527	464	pp	1	4	6
	29318	18384	**627**	383	kap	1	23	1
	23865	22947	**962**	478	nkap	1	23	1
Simon 64/96	1417	970	685	667	it	1	1	44
	5762	4156	721	714	pp	1	4	11
	32426	33870	**1045**	529	kap	2	21	1
	32685	43782	**1340**	684	nkap	1	42	1
Speck 64/96	1887	678	359	307	it	1	1	29
	6668	3078	462	433	pp	1	3	9
	42869	23872	**557**	373	kap	1	26	1
	28113	23625	**840**	369	nkap	1	26	1
Simon 64/128	1751	870	497	625	it	1	1	46
	5518	3048	552	714	pp	1	3	15
	44322	34243	**773**	535	kap	2	22	1
	35187	44208	**1256**	691	nkap	1	44	1
Speck 64/128	2014	634	315	307	it	1	1	31
	4974	1916	385	419	pp	1	2	14
	48056	23908	**498**	374	kap	1	27	1
	28949	23214	**802**	363	nkap	1	27	1
Simon 96/96	1780	1127	633	634	it	1	1	54
	7692	5275	686	769	pp	1	4	14
	55488	50794	**915**	529	kap	2	26	1
	59801	65355	**1093**	681	nkap	1	52	1

Continued on next page

algorithm	**area**	**tput**	**eff**	**clk**	**type**	**rpc**	**stg**	**cyc**
Speck 96/96	2678	964	360	301	it	1	1	30
	7484	3160	422	329	pp	1	3	10
	62211	31673	**509**	330	kap	1	28	1
	55456	39393	**710**	410	nkap	1	28	1
Simon 96/144	1982	1084	547	632	it	1	1	56
	23640	14310	605	596	pp	2	7	4
	98065	80341	**819**	837	kap	1	54	1
	62432	65538	**1050**	683	nkap	1	54	1
Speck 96/144	3294	1045	317	348	it	1	1	32
	8746	3391	388	353	pp	1	3	10
	69484	31641	**455**	330	kap	1	29	1
	51254	35608	**695**	371	nkap	1	29	1
Simon 128/128	2342	1145	489	626	it	1	1	70
	7279	3980	547	715	pp	1	3	23
	146287	106961	**731**	836	kap	1	68	1
	104790	87798	**838**	686	nkap	1	68	1
Speck 128/128	3290	880	268	234	it	1	1	34
	9662	3531	365	303	pp	1	3	11
	98003	41531	**424**	324	kap	1	32	1
	81153	49117	**605**	384	nkap	1	32	1
Simon 128/192	2774	1202	433	667	it	1	1	71
	17069	8033	471	565	pp	2	4	9
	166772	106943	**641**	835	kap	1	69	1
	111037	95104	**857**	743	nkap	1	69	1
Speck 128/192	4343	1159	267	326	it	1	1	36
	7937	2439	307	324	pp	1	2	17
	110960	42384	**382**	331	kap	1	33	1
	71877	41304	**575**	323	nkap	1	33	1
Simon 128/256	3419	1081	316	625	it	1	1	74
	42368	14830	350	579	pp	2	8	5
	233204	100078	**429**	782	kap	1	72	1
	110875	87193	**786**	681	nkap	1	72	1
Speck 128/256	5159	1287	249	382	it	1	1	38
	18742	5618	300	307	pp	1	5	7
	123074	42356	**344**	331	kap	1	34	1
	70280	39483	**562**	308	nkap	1	34	1