We'll Encrypt As Feistel As Possible!
A Comparative Timing Analysis of Traditional Block Ciphers, Public Key Systems, and NSA's SIMON & SPECK

Gradeigh D. Clark, Marlon Orellana, Rutgers University
gradeigh.clark@rutgers.edu, marlono1224@gmail.com

The National Security Agency (NSA) has recently released two new block ciphers: SIMON and SPECK. They are, in the NSA's words, meant to be "lightweight'" block ciphers that can encrypt and decrypt very quickly. SIMON is a hardware optimized cipher algorithm and SPECK is software optimized. Quick encryption and decryption would make them ideal for various mobile and Internet of Things applications, which is the NSA's intent by releasing them. Herein, we perform a timing analysis of SIMON and SPECK and compare them to traditional block ciphers (AES and SERPENT) as well as public key encryption systems (RSA and El Gamal). Results indicate that SIMON and SPECK are indeed extremely quick at encrypting and decrypting in comparison to AES, SERPENT, RSA, and (especially) ElGamal.

Contents

This work is for Rutgers Course 16:332:559 - **Advanced Information and Network Security**, with Dr. Wade Trappe.
The template used herein is a modification of the **ACM Small Standard Format**, available in full at http://www.acm.org/publications/latex_style/v2-acmsmall.zip.

May 2014.

1. INTRODUCTION

Lightweight cryptographic protocols are needed to address security concerns in pervasive computing technologies [Beaulieu et al. 2013]. A specific goal of this is to create an algorithm that can be both software and hardware independent, which current standarized block ciphers are not [Beaulieu et al. 2013].

To that end, the NSA introduced SIMON and SPECK in 2013 to address the needs for lightweight block ciphers. Herein, we examine how quickly they run in comparison to AES, SERPENT, RSA, and ElGamal and judge their applicability for lightweight security concerns.

All experiments were run on an Intel i7 processor computer running CentOS operating at 2.67 GHz.

2. SYMMETRIC KEY ENCRYPTION

Symmetric key encryption are so named because there is only one key method used to encrypt and decrypt a message. As such, sender and receiver tend to require the same key when trying to communicate with each other. Before the era of public key cryptography, this was hard to manage – the main point of failure in modern algorithms would be giving away the key over an unsecured connection. Today, that problem is solved with public key cryptography; it is not uncommon for systems to encrypt messages with symmetric cryptography (which provides better security at faster speeds) and then have the ciphertext and key transmitted with public key cryptography.

Block ciphers are a type of symmetric cipher where the message is broken into separate blocks and is encrypted on a block by block basis. Each modern block cipher obeys a Feistel round structure, wherein encryption and decryption is broken down further in smaller and easier-to-manage blocks (they key itself is even broken down into subkeys). Often, the exact reversal of the algorithm steps allows for decryption of the encrypted ciphertext.

2.1. AES - Advanced Encryption Standard

The AES algorithm [NIST 2000] was issued by the National Institute of Standard and Technology in 2000 after a protracted competition to replace the aging DES cipher. AES uses three different key sizes of 128, 192, and 256 bits to encrypt and decrypt block sizes of 128 bits (though technically can be larger). Out of many, Rijndael [Daemen and Rijmen 2002] was chosen to serve as the AES algorithm.

The steps are a big protracted to get into. For each key size, Rijndeal algorithm uses a different number of rounds. The key size of 128 bits uses 10 rounds. The key size of 192 bits uses 12 rounds. The key size of 256 bits uses 14 rounds. Where each round contain of numerous processing steps and each contain four different stages:

(1) SubBytes: This is a 16x16 lookup table to find a replacement byte.
(2) ShiftRows: This scrambles the byte order inside the 128-bit block.
(3) MixColumns: This mixes the columns after the rows; now the ciphertext is almost totally dependent on the original plaintext.
(4) AddRoundKey: Adds the round key to the output of the previous step.

2.1.1. Experimental Design. For this, we'll just vary the three key sizes and see what effects it has on the run time. Our examination of messages did not seem to yield any interesting results beyond common noise. Increasing the message size would just result in a linear increase of the run time, depending on the cipher mode – the result is trivial. As such, we will just focus on looking at the runtime of the encryption step for different key sizes.

2.1.2. Results. We ran the encryption algorithm at 1000 iterations for each key size. The results are seen in the figure below.

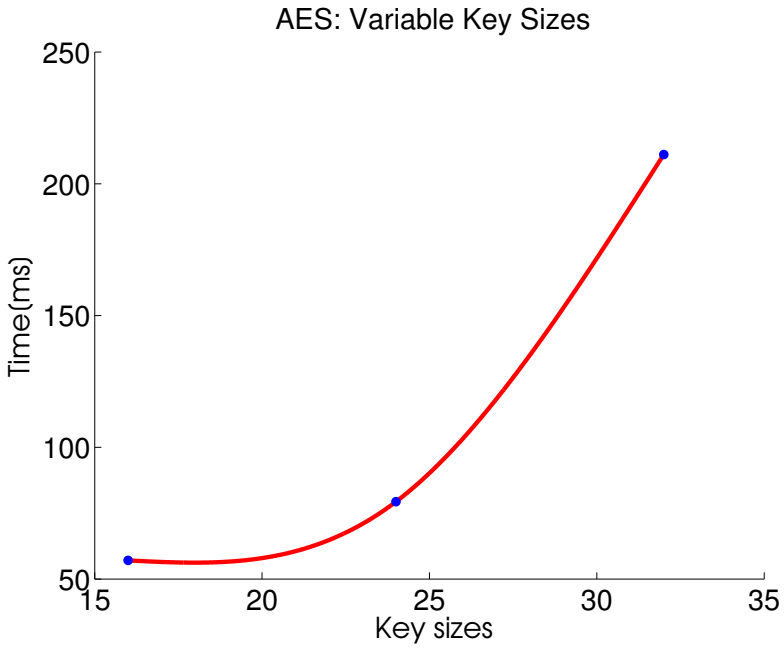

Fig. 1: Plot of the encryption time as a function of key size with a spline fit through the data. The standard sizes are K = 16, 24,and 32 bytes. The mean time it took in each plot is T = 57ms, 79ms, and 211ms for those key sizes, respectively.

AES in this analysis was set to a constant message length of 432 bytes and the initialization vector was set to 16 bytes in order to produce a 16 byte block cipher. . The factor that affected the run time of this algorithm would be the three different key sizes of 16, 24, and 32 byteS. The spline fit makes it appear as if the run time is quadratic or at least polynomial; this likely isn't the case. Although we averaged over 1000 iterations, in the end the key size only really affects the number of rounds. So we're looking at the speed difference between 10, 12, and 14 rounds which has this kind of appearance. In total, the slowest time was 211ms for the largest key.

It wouldn't be surprising to assume that the runtime has some strange nature with respect to the key size. The rounds are implementing some convoluted functions, which can result in polynomial behavior.

2.2. SERPENT - Second Place

Coming in second place to Rijndeal in the AES contest was SERPENT [Biham et al. 1998], designed by Ross Anderson, Eli Biham and Lars Knudsen. SERPENT uses a symmetric key block cipher with 128 bits of plaintext and a key size varying from lengths of 64 to 256 bits.

SERPENT encrypts the 128 bit plaintext to a 128 bit ciphertext in 32 rounds. The cipher is a substitution-permutation network that consists of initial permutations with each of the 32 rounds having a key mixing operation, passing through the S-boxes and a linear transformation. At the last round, the linear transformation is replaced

by a additional key mixing operation and a final permutation. The initial and final permutations are used to shorten an optimized implementation of the cipher.

2.2.1. Experimental Design. Similar to AES, the only real variable worth analyzing here is the key size. The message size has only a purportedly linear effect on the output since messages longer than 128 bits will just get split up. So we will fix the message size to be constant and look at the effects of different key lengths on run time.

2.2.2. Results. We ran the algorithm for 1000 iterations for each key and averaged the results together. The results are seen in the figure below.

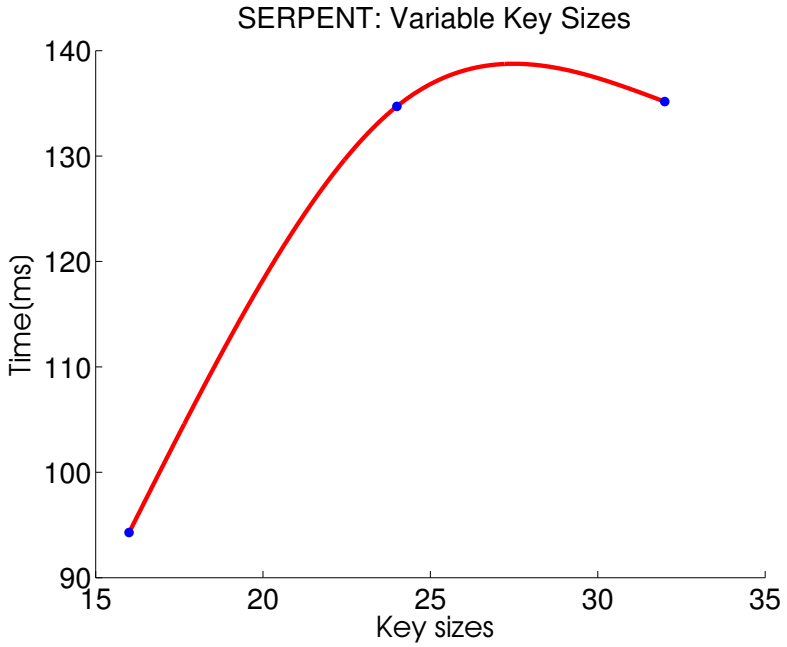

Fig. 2: Plot of the encryption time as a function of key size with a spline fit through the data. The standard sizes are K = 16, 24, and 32.. The mean time it took in each plot is T = 94.2ms, 134.7ms, and 135.16ms for those key sizes, respectively.

Please take note that the fit through the data is a spline; a deliberate overfit to exaggerate the curve behavior.

For the first two cases, SERPENT is out performed by AES. Indeed, we structured our algorithm to operate in CBC mode; SERPENT has speed issues from this. SERPENT benefits from parallelized operations like one would see in hardware implementations. What is interesting to note is that the speed has effectively levelled off at the higher key sizes; this is because the number of rounds are the same for all keys. At the higher levels, SERPENT's increasing keys will not affect the performance! Combine that with the lookup table that it uses and it starts to outperform AES at higher keys (recall that they have more rounds at higher keys and a more complicated round structure than SERPENT). The results are as expected. At higher key sizes, SERPENT's speed wouldn't appreciably change at all; if AES ever needed higher keys it would continue its quadratic/polynomial climb upwards.

3. PUBLIC KEY ENCRYPTION

Public key encryption algorithms differ from symmetric key ones in that there are two keys: one public key for encryption and one private key for decryption. Typically, if Alice wants to send a message to Bob she will encrypt a plaintext with Bob's public key and leave it in the open for all to see. Since only Bob has the private key, he can decrypt it at his leisure without fear of anyone learning what Alice was trying to tell him.

Public key cryptosystems derive their security from the mathematical difficulty of certain problems that, without some kind of special or a-priori knowledge about terms used in them, are intractable to solve. These types of problems are structured in such a way that brute force solutions are not feasible if the keys exceed a particular size. Some examples of these problems might be prime number factorization [Rivest et al. 1978] or finding the logarithm of a base raised to a power modulo another number. [ElGamal 1985] We will look at both of these from a surface level.

3.1. RSA – Security from Factorization

The RSA algorithm [Rivest et al. 1978] was introduced in was introduced in 1977 and named for its principal authors: Ronald Rivest, Adi Shamir, and Len Adleman. It is a public key cryptosystem that derives security from the difficulty of integer factorization. The algorithm steps below are given verbatim by Trappe et. al [Trappe and Washington 2006]:

(1) Bob chooses secret primes p and q and computes $n = pq$.
(2) Bob computes $\phi(n) = (p-1)(q-1)$.
(3) Bob chooses e with $\gcd(e, \phi(n)) = 1$.
(4) Bob computes d with $de \equiv 1 \mod n$.
(5) Bob makes n and e public, and keeps p, q, d secret.
(6) Alice encrypts m as $c \equiv m^e \mod n$ and sends c to Bob.
(7) Bob decrypts by computing $m \equiv c^d \mod n$.

There are two naive ways to attack RSA:

— Take the ciphertext, c, and try to compute $m \equiv c^{1/e} \mod n$.
— Take the encryption exponent, n, and try to factor it such that $\phi(n)$ is known. With that, find d such that $de \equiv 1 \mod \phi(n)$.

Neither of which are are straightforward to do. The first one is difficult because it would require case-by-case searches for values of m until something is produced. Knowing that m can be on the same order as n means that an exhaustive search would take quite a long time – so it is easy to conclude that finding the root of a number modulo another is hard. The second one is difficult because, if factoring is hard (it is), then there is no fast way to find d. So the security of RSA is ensured by this.

3.1.1. Experimental Design. The important variables in the encryption steps are: the key size of the primes (N), the message size (M), and the key e. There are three different experiments to try:

(1) **Effect of N.** Fix the message, m, and the encryption exponent, e, to be a constant size. Vary the size of the prime numbers p and q and see how that affects the encryption time. We are only measuring the time it takes to do encryption, which is the modular exponentiation component – one executed line. Since we are measuring time with System.nanoTime(), there can be resolution problems. To combat this, we average the results over 2000 iterations of the same encryption command.

(2) **Effect of M.** Fix the key size, N, and the encryption exponent, e, to a constant size. Vary the size of the message m, M, and see what effect that has on encryption time. It is equally useful to look at this over all of the key sizes listed above. The planned messages size should be less than or equal to the size of the smallest key – 128 bits. This leaves the planned message sizes as: 2, 4, 8, 16, 32, 64, 128. For this, we wil keep the encryption exponent sufficiently large (2^16+1) such that resolution doesn't become too much of a problem. This will be averaged over 2000 iterations.
(3) **Effect of e.** Fix the message, m, the key size, N, to a constant size. Vary the size of the encryption exponent. e is typically selected to be a Fermat prime ($2^k + 1$). This will be averaged over 2000 iterations.

3.1.2. Results. For the variable N experiment, we used key sizes of 2^k bits. The message was a randomly generated number of size $M = 128$ bits and the encryption exponent, e, was fixed at $2^{16} + 1$. The results are seen below; this is averaged over 2000 iterations.

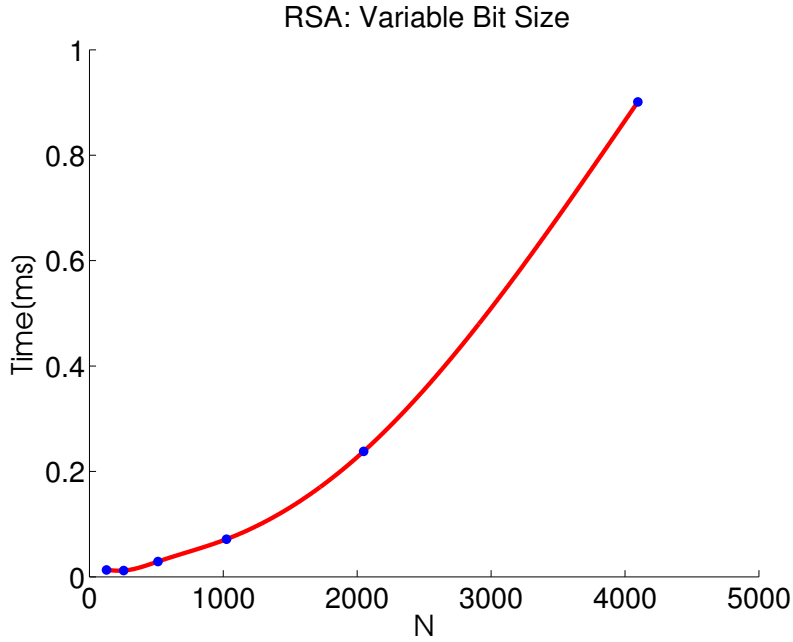

Fig. 3: Spline fit of the encryption time as a function of bit size for the primes with a spline fit through the data. The standard sizes are N = (128, 256, 512, 1024,2048, 4096) bits. The mean time it took in each plot is T = (0.0130,0.0119,0.0290,0.07140.237 and 0.901) ms, respectively.

The figure indicates that there is a polynomial relationship between the size of the primes and the time it takes to execute the algorithm. Generally, this is true. The larger the prime number the longer it takes to perform the modular exponentiation – we rely on Karatsuba Multiplication for things like this, which in the best case performs in the area of $O(N^2)$ so the result here is not unexpected. Note that the time it takes is different from AES because we were encrypting a large message there but a short one here.

For the variable M experiment, we fixed the size of the primes at $N = 8192$ bits so we could get messages large enough to encrypt such that the clock resolution didn't affect our performance. We averaged these results over 2000 iterations.

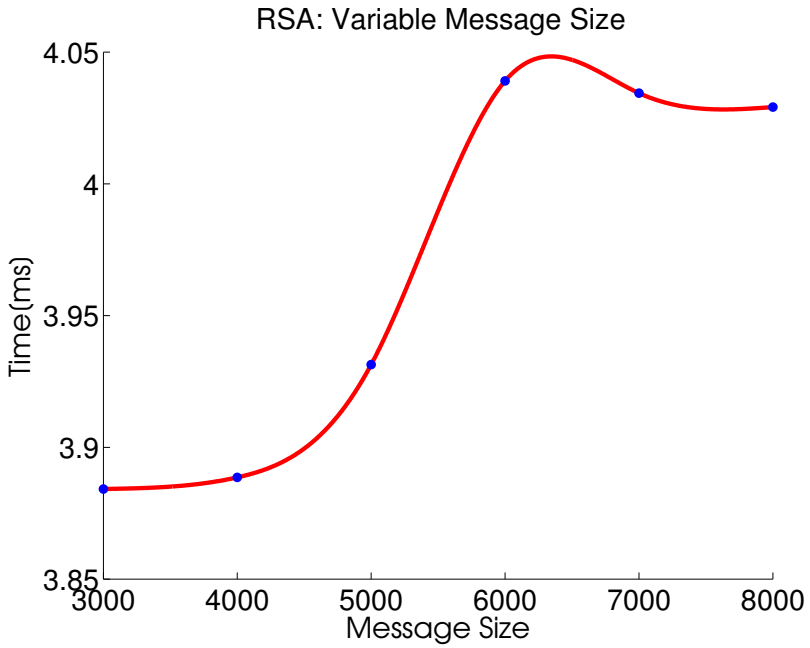

Fig. 4: Spline fit of the encryption time as a function of message size with a spline fit through the data. The standard sizes are M = (3000,4000,5000,6000,7000,8000) bits. The mean time it took in each plot is T = (3.88, 3.89, 3.931, 4.03, 4.034, and 4.029) ms respectively.

From the figure, it can be assumed the message size has a negligible effect on the performance. This is not unexpected; the prime numbers and the modulo is where most of our calculation time is wasted. The message size may as well be stated to have no effect on the outcome given that the difference between the smallest and largest point timewise was around 0.2 ms with a difference bit-wise of 4000 bits! With that said, the message size clearly affects nothing.

The results of the variable e experiment can be seen in the figure.

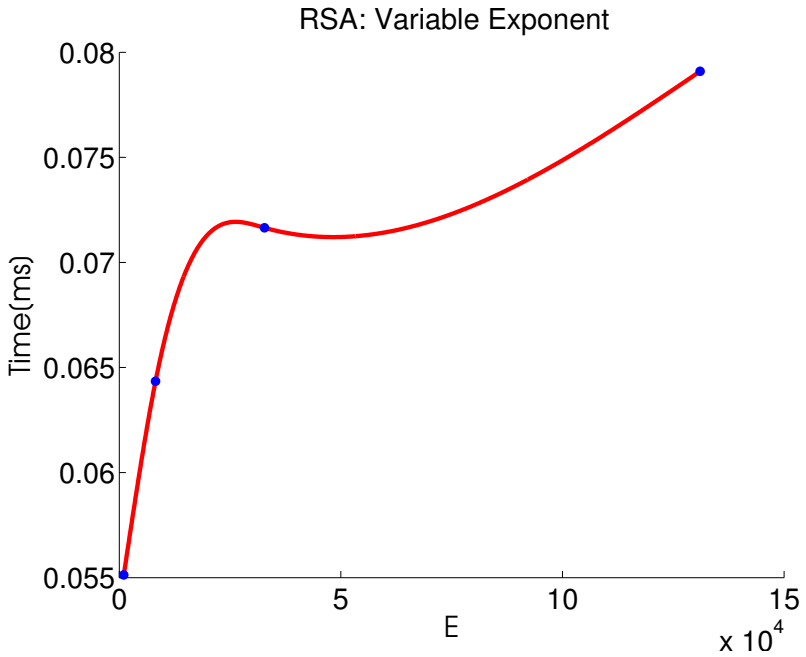

Fig. 5: Spline fit of the encryption time as a function of encryption exponent with a spline fit through the data. The standard sizes are e = (1025,8193,32769,131073) bits. The mean time it took in each plot is T = (0.0551, 0.0643, 0.0716, 0.0790) ms respectively.

Varying the exponent changes the time it takes to perform modular exponentiation at the encryption step. However, even naive multiplication by squaring wouldn't take too long – the power of 2 is 16 or larger and that in and of itself wouldn't be many iterations of a for loop. The effect is extremely negligible – a few tenths of a millisecond. The relationship appears it would be logarithmic, which would make sense under the doubling assumption.

3.2. El Gamal – Security from Logarithms

The El Gamal algorithm [ElGamal 1985] was introduced in 1985 by its namesake, Taher Elgamal. It is a public key cryptosystem that derives its security from the fact that it is difficult to find the logarithm of a number modulo n. Structurally, it obtains much of its formalism from the Diffie-Hellman Key Exchange protocol [Diffie and Hellman 1976]. The algorithm steps are given verbatim from Trappe et. al [Trappe and Washington 2006]:

(1) Bob selects a large prime p and a primitive root α.
(2) Bob selects a secret integer a and computes $\beta \equiv \alpha^a \mod p$.
(3) Bob makes (p, α, β) public and hides the value of a.
(4) Alice selects a secret random integer k and computes $r \equiv \alpha^k \mod p$.
(5) Alice then chooses a message m such that it satisfies $0 \leq m < p$ and computes $t \equiv \beta^k m$.
(6) Alice sends the pair (r, t) to Bob.
(7) Bob obtains the original message by doing $tr^{-a} \equiv m \mod p$.

The naive way to break El Gamal would be to try:

$$a \equiv log_{\alpha}(\beta) \mod p$$

Where the only algorithm we obtain from this is a brute force one, which is non-polynomial in execution time. Although discrete logarithms haven't been proven to be hard [Diffie and Hellman 1976], it has been assumed to be such in the security community since there hasn't been any easy way to break them.

3.2.1. Timing Analysis. The encryption step is:

$$t \equiv \beta^k m \mod p.$$

where $\beta \equiv \alpha^a \mod p$. This is again, in the worst case, modular exponentiation with a multiplication term tacked on for the message. The runtime should be similar to RSA.

3.2.2. Experimental Design

(1) **Effect of β.** Fix the message, m, and the encryption exponent, k, to be a constant size. Although we could vary the size of α and a and measure the effects individually, this does not seem to yield anything of note. β is a public key term that is computed prior to the encryption; it should affect the time insofar as it would take longer to get the modulus value t depending on its size. Since the easiest way to change β size is through the secret integer, we will vary β by varying a with a constant α. This will be averaged over 2000 iterations.
(2) **Effect of M.** Fix β and k to a constant value . Vary the size of the message m, M, and see what effect that has on encryption time. We'll average this over 2000 iterations.
(3) **Effect of k.** Fix the message, m, and β to be constant. Vary the size of the encryption exponent, k. This will be averaged over 2000 iterations.
(4) **Effect of N.** Fix everyhing that doesn't depend on the size of the prime number, p (which we refer to as N for the size) and see how the time is affected. This will be averaged over 2000 iterations.

3.3. Results

The results for varying secret integer a (to vary β) is seen in the figure.

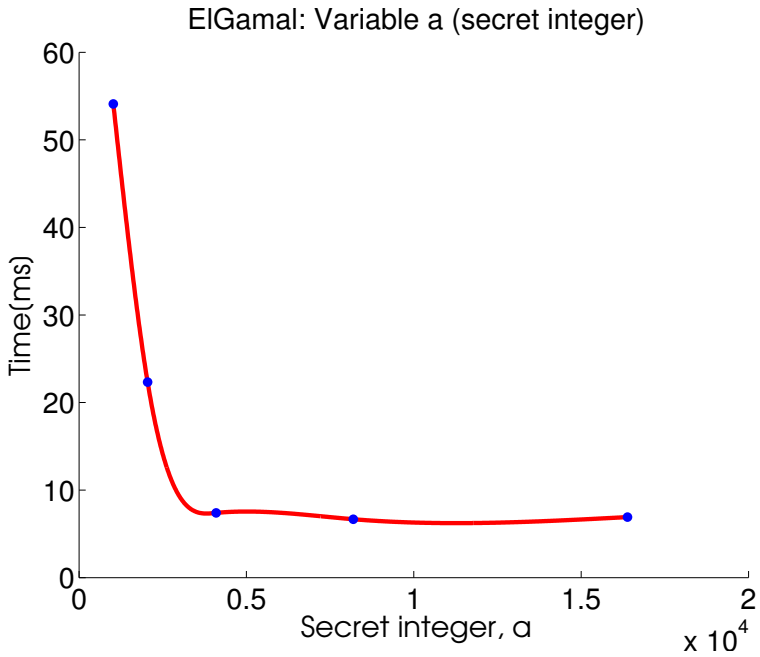

Fig. 6: Spline fit of the encryption time as a function of the secret integer with a spline fit through the data. The standard sizes are a = (1024,2048,4096,8192,16384) bits. The mean time it took in each plot is T = (54.10, 22.33, 7.39, 6.67, 6.91) ms respectively.

Interestingly, having a large secret integer caused a roll off in encryption time from 54ms to a low of 6.67ms. There isn't any logic to govern this; the message size was kept sufficiently large as well as the prime number value (each 1024 bits). It's possible the solution here is embedded in the fact that there might have been resolution problems at those bitlevels. A different reason could be that the beta value was reduced modulo p at such high exponent levels (the base level was set at a constant 3 with a 1024 bit prime, which could explain a lot).

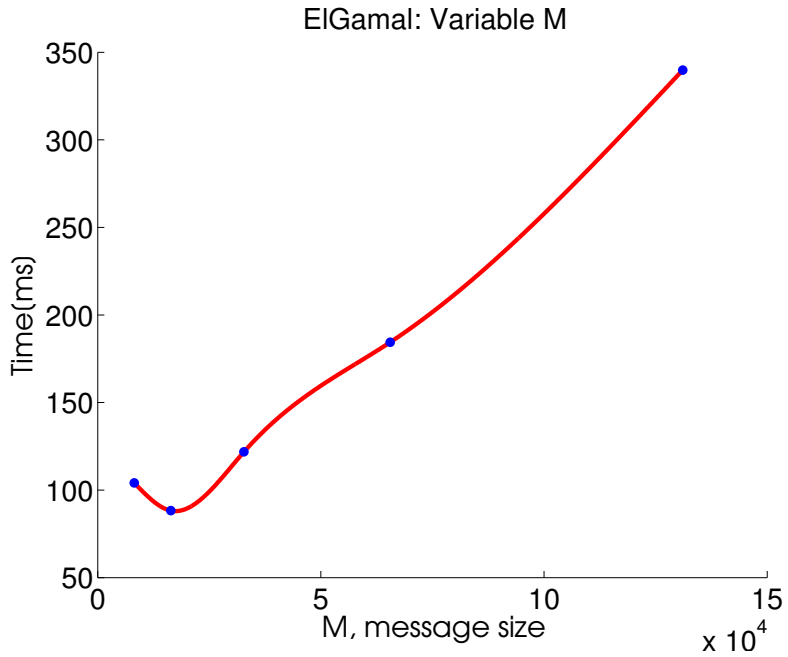

Fig. 7: Spline fit of the encryption time as a function of the message size,M, with a spline fit through the data. The standard sizes are M = (8192,16384,32768,65536,131072) bits. The mean time it took in each plot is T = (104.12, 88.24, 121.82, 184.42, 339.80) ms respectively.

The time growth is effectively linear with the size of the message. This doesn't come as any great surprise; we avoided this experiment in the symmetric cryptography section for jut this reason. The high bit levels of the messages were necessary to avoid resolution problems with the timing measurements. It can be easily seen that larger messages take longer.

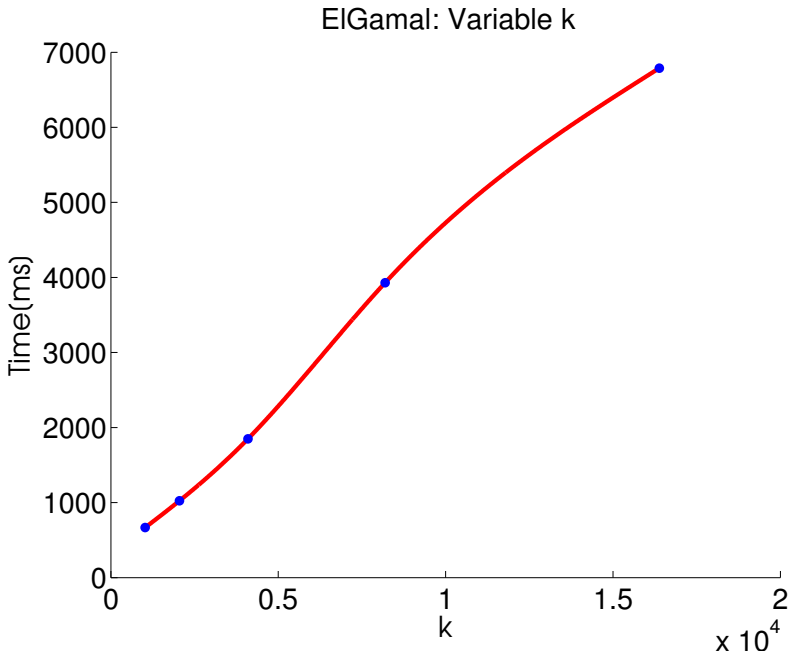

Fig. 8: Spline fit of the encryption time as a function of the encryption exponent, k, with a spline fit through the data. The standard sizes are k = (1024,2048,4096,8192,16384) bits. The mean time it took in each plot is T = (667.34, 1023.6, 1849.7, 3929.3, and 6788.8) ms respectively.

The time appears to vary linearly with the encryption exponent, k. The speed is quite of a steep rise; the size of the primes are only 128 bits, so it surprising how strong a linear effect the k size has on the encryption time. A ten multiple of the bit size corresponds nearly to a 10 multiple of the encryption time.

The results of the N size experiment can be seen in the figure.

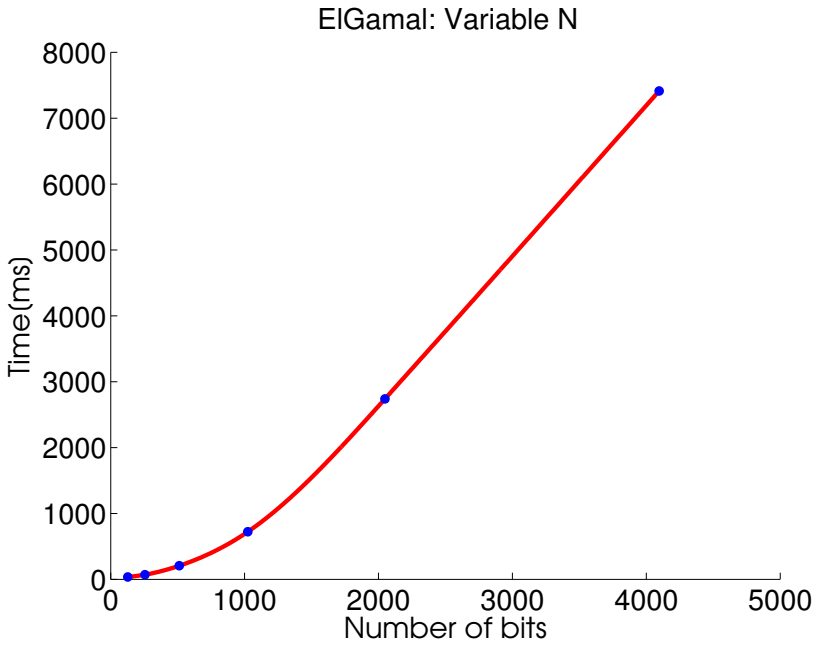

Fig. 9: Spline fit of the encryption time as a function of the bit size of the primes, N, with a spline fit through the data. The standard sizes are N = (128,256,512,1024,2048, 4096) bits. The mean time it took in each plot is T = (34.73, 70.85, 205.94, 722.01, 2738.3, 7411.9) ms respectively.

Varying the size of the prime, p, appears to give a linear or polynomial relationship. This is the largest contributor to the algorithm runtime; large keys slows the algorithm down considerably – El Gamal can be seen to be considerably slower than RSA in this regard. RSA can still manageably iterate over 4096 bit key sizes – however, El Gamal takes considerably longer to do so. Calculating the pairs needed to make these takes so long (and you CANNOT reuse these parameters or else risk being attacked). This is one reason why RSA is more widely used.

4. SIMON & SPECK

NSA created SIMON and SPECK to address computational constraints in pervasive computing technologies. Block ciphers present usefulness as cryptographic primitives – they are a building block from which true security can be obtained. AES, by the NSA's argument, is insufficient for pervasive computing problems due to issues with hardware requirements [Beaulieu et al. 2013].

These algorithms aren't especially different from traditional block ciphers. Encryption is performed over many rounds with generated subkeys and decryption involves passing the message through a reversal of the algorithm and its subkeys. One particular difference is the lack of S-boxes seen in SIMON & SPECK; these hamper hardware serialization [Beaulieu et al. 2013]. Implementation novelty lies therefore in round functions that can provide decent cryptographic properties while not beng computationally expensive.

4.1. SPECK

SPECK is the lightweight cipher meant for software applications. SPECK differs from high-speed AES algorithms in its implementation – it is extremely straightforward to perform. There are only really two parts. The first is the key expansion, iterated over $T-2$ subkeys with the l array AND k_0 being populated with initial values:

$$l_{i+m-1} = (k_i + S^{-\alpha} l_i) \oplus i$$

$$k_{i+1} = S^{\beta} k_i \oplus l_{i+m-1}$$

where S refers to a circular shift; positive exponent is a left shift and a negative exponent is a right shift.

The encryption step is just iterated over $T-1$ subkeys:

$$x = (S^{-\alpha} x + y) \oplus k[i]$$

$$y = S^{-\beta} y \oplus x$$

Decryption is the same in reverse order.

4.1.1. Experimental Design. Not much to analyze here! We can vary the word sizes according to the NSA's specifications. The key size is the same as the message size so not much to do here. We'll do 2000 iterations of each of SPECK's sizes: 32/64, 48/96, 64/128, and 128/256. The goal will be to randomly generate new messages, key expansions, etc. at each point and just time it. Doing a constant message size doesn't seem to mesh here; we change the message size to be processed at the same size of the words at each stage so it only needs to be run once or avoid padding problems. The key size is always fixed to a multiple of the block size so not much value in examining it.

4.1.2. Results. The results for SPECK are shown in the figure.

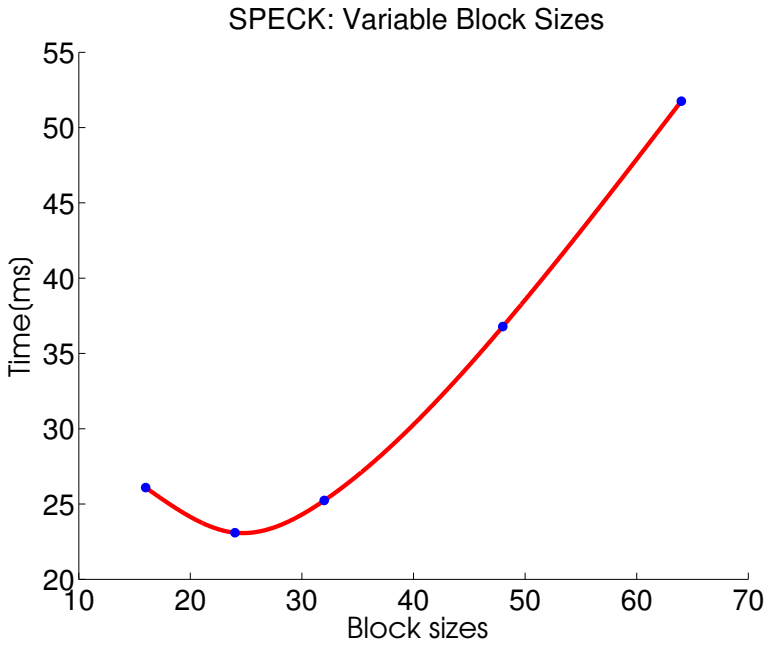

Fig. 10: Plot of the encryption time as a function of key size with a spline fit through the data. The standard sizes are K = (32,48,64, and 128). The mean time it took in each plot is T = (26.09, 23.09, 25.23, 36.78, 51.75) ms for those key sizes, respectively.

It seems like the behavior of the curve is mostly linear. There are likely just resolution issues in timing with low block sizes such that the 16 bit data point appears higher than it normally should. This is actually a big difference from AES, which has clear polynomial behavior in its curve. It is faster than SERPENT as well at all levels, never breaking 50ms even at message sizes that are equal to the size of the block at every stage. This is owed to the simplicity of the operation and the fact that the keys are generated prior to the algorithm actually encrypting; encryption is only three lines of pseudocode.

4.2. SIMON

SIMON is the other member of NSA's new block ciphers. SIMON is meant to be implemented on Hardware. Simon has a constant, z, defined to be the is the cryptographic separation between different version of SIMON having the same block size by defining 5 five different sequences which are $z_0 z_4$. T Following the pseudo code from the whitepaper, the first thing to do to encrypt is to generate a key expansion. The steps are:

$$x_0 = x$$

$$x = y \oplus [(Sx)\&(S^8 x)] \oplus S^2 x \oplus k[i]$$

$$y = x_0$$

4.2.1. Experimental Design. This is exactly the same as SPECK. We will vary the block size and measure the runtime. For this, we will see what happens if the message size is kept constant since it wasn't examined in SPECK.

4.2.2. Results. The results of SIMON can be seen in the figure.

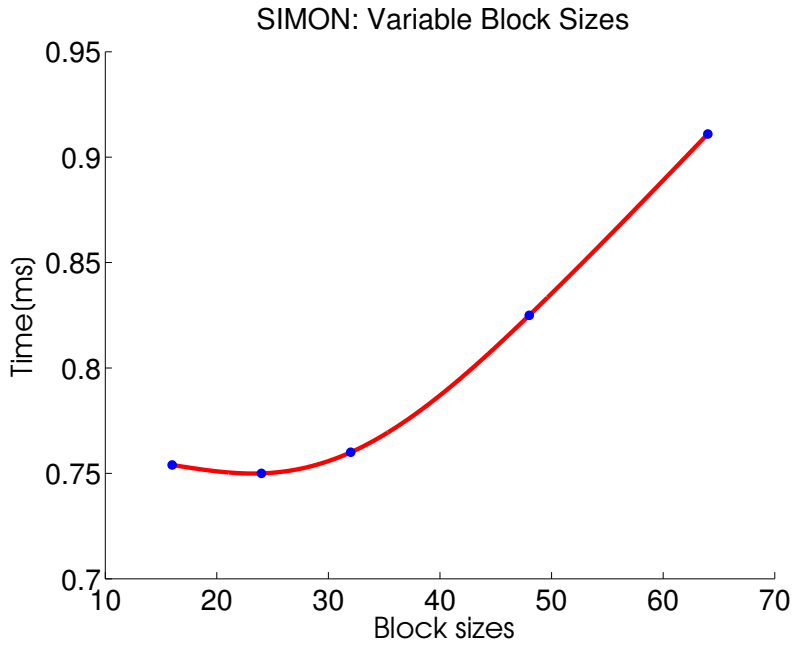

Fig. 11: Plot of the encryption time as a function of key size with a spline fit through the data. The standard sizes are K = (32,48,64, and 128). The mean time it took in each plot is T = (0.754, 0.750, 0.760, 0.825, 0.911) ms for those key sizes, respectively.

Like SPECK, the runtime is effectively linear in the key size. At a constant block size of 16 bits, the time is significantly different from that of SPECK by large factors. However, this can be attributed to some differences in implementation – the SPECK implementation works using Java's BigIntegers and the SIMON one has been implemented to work on constant byte arrays (but doesn't scale well to larger bit sizes). The shape of the curves and the order of growth is about the same, so there's no noticeable difference between the two algorithms. However, they certainly are faster than AES and SERPENT.

5. CONCLUSIONS

SIMON and SPECK are incredibly fast; they're faster than every other algorithm presented here! However, that is owed to their intentionally simplistic designs. These are meant to be used on both big data and quick data processing opportunities wherein reduced security is accepted to improve latency. They exist to appease two opposing sides: developers who want flexibility and cryptographers who want security. AES and SERPENT implementations don't scale well to the level that SIMON and SPECK can; indeed, AES requires hardware that would severly reduce its performance abilities in comparison [Beaulieu et al. 2013]. SIMON and SPECK are fast precisely because they have non-complicated round functions, generally perform a small number of operations, and have simplistic key-generation schedules. On the other hand, if absolute

security is needed (even for a lightweight application), then it can't be fully recommended that these be used, especially since detailed cryptanalysis for them isn't fully available yet.

ACKNOWLEDGMENT

Many acknowledgements go to the authors of the various algorithms presented herein. Without their hard work, this survey would not be possible. Liberal use of their knowledge and effort has been used, with a dash of rewriting or further explanation to try to open the algorithms to a broader audience.

REFERENCES

Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. 2013. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404. (2013). http://eprint.iacr.org/.

Eli Biham, Ross Anderson, and Lars Knudsen. 1998. Serpent: A new block cipher proposal. In *Fast Software Encryption*. Springer, 222–238.

Joan Daemen and Vincent Rijmen. 2002. *The design of Rijndael: AES-the advanced encryption standard*. Springer.

Whitfield Diffie and Martin E Hellman. 1976. New directions in cryptography. *Information Theory, IEEE Transactions on* 22, 6 (1976), 644–654.

Taher ElGamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*. Springer, 10–18.

NIST. 2000. Announcing the ADVANCED ENCRYPTION STANDARD (AES). (2000).

Ronald L Rivest, Adi Shamir, and Len Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

Wade Trappe and Lawrence C Washington. 2006. *Introduction to cryptography with coding theory*. Pearson Education India.