Customizable Sponge-Based Authenticated Encryption Using 16-bit S-boxes

Matthew Kelly[1], Alan Kaminsky[2], Marcin Łukowiak[1],
Michael Kurdziel[3], and Stanisław Radziszowski[2]

[1]Department of Computer Engineering, Rochester Institute of Technology
[2]Department of Computer Science, Rochester Institute of Technology
[3]Harris Corporation, Rochester, NY

September 2014

Abstract

Authenticated encryption (AE) is a symmetric key cryptographic scheme that aims to provide both confidentiality and data integrity. There are many AE algorithms in existence today. However, they are often far from ideal in terms of efficiency and ease of use. For this reason, there is ongoing effort to develop new AE algorithms that are secure, efficient, and easy to use.

The sponge construction is a relatively new cryptographic primitive that has gained popularity since the SHA-3 hashing competition was won by the sponge-based KECCAK algorithm. The duplex construction, which is closely related to the sponge, provides promising potential for secure and efficient authenticated encryption.

In this paper we introduce a novel authenticated encryption algorithm based on the duplex construction that is targeted for hardware implementation. We also provide explicit customization guidelines for users who desire unique authenticated encryption solutions within our security margin. Futhermore, our substitution step uses 16×16 AES-like S-boxes. We believe this to be the first algorithm in the literature to use such large bijective S-boxes.

1 Introduction

The overarching goal of symmetric key cryptography is to enable people to communicate privately over an insecure channel in the presence of adversaries. Two fundamental requirements for achieving this goal are encryption and authentication. Encryption provides *confidentiality* while authentication provides data *integrity* and assurance of message origin.

Many authenticated encryption algorithms are in existence today, but they are often unsatisfactory in terms of performance, security, or ease of use. Some algorithms require two passes per block of plaintext to encrypt and authenticate. This is generally undesirable because it often means a much slower algorithm. Other algorithms have been shown to be insecure or difficult to use properly. Many algorithms, such as the ones based on generic composition, require two keys. This should be avoided when possible because key management is a difficult problem.

Furthermore, a new authenticated encryption algorithm is needed that meets the stringent requirements of government and military applications. Such algorithms are not typically in the public

domain. The goal of this is partially to reduce or eliminate academic interest in cryptanalyzing the algorithm and publishing results. This stance is highly controversial. Still, the security of such algorithms depends entirely on the secrecy of the key and not on the secrecy of the algorithm. The assumption is still made that the enemy knows the details of the algorithm being used at any time [1].

For this reason, there is a need for a customizable authenticated encryption algorithm. This algorithm should remain secure as long as customizations are made within certain guidelines. The result is an algorithm which can be made unique on a per-user or per-application basis without the effort of cryptanalyzing every specific instantiation. We present such an algorithm here.

2 Sponge Construction

The sponge construction is a relatively new cryptographic primitive that has gained popularity since KECCAK won the Secure Hash Algorithm (SHA-3) competition in 2013 [2][3]. Essentially, it provides a way to generalize hash functions (which normally have outputs of fixed length) to functions with arbitrary length output. This generalization allows cryptographic sponges to be used for applications other than hashing.

Sponges are based on the iteration of an underlying function f. This function can either be a general *transformation* or a *permutation*. A transformation need not be bijective; that is, it may not be invertible. A permutation is bijective and thus invertible by definition. The security proofs are different for transformations and permutations, and there are advantages and disadvantages for each choice of a function type [4].

2.1 Sponge Parameters

The output Z of the parameterized sponge construction is given as

$$Z = \mathbf{sponge}[f, \mathbf{pad}, r](M, \ell),$$

where $\mathbf{pad}$ is a padding function for the input, r is the *rate* of absorption, M is the message (or other input) data, and ℓ is the desired output length. The sponge construction is stateless; there is no information stored between calls to it.

Figure 1 shows the sponge construction. It is split into two distinct phases: the *absorbing* phase and the *squeezing* phase. Inputs (e.g. message and/or key material) are absorbed in the first phase and the output (e.g. a MAC or keystream) is squeezed out in the second phase.

The state of the sponge construction is split into two contiguous portions: the *outer state*, which is accessible externally, and the *inner state*, which is hidden. The size of the outer state is given by the *rate* r and the size of the inner state is specified by the *capacity* c. The size of the entire state is $b = r + c$. The speed of the construction partially relies on the rate, while the security is partially dependent on the capacity.

The padding function $\mathbf{pad}$ is first applied to M to make it a multiple of r. M is then absorbed r bits at a time. More concretely, absorption is the process of XORing r-bit blocks into the state while interleaving with applications of the underlying sponge function f. If the rate is increased then more bits are absorbed at a time and thus the construction runs faster. However, increasing the rate means that the capacity must decrease and so there is a clear trade-off between speed and security.

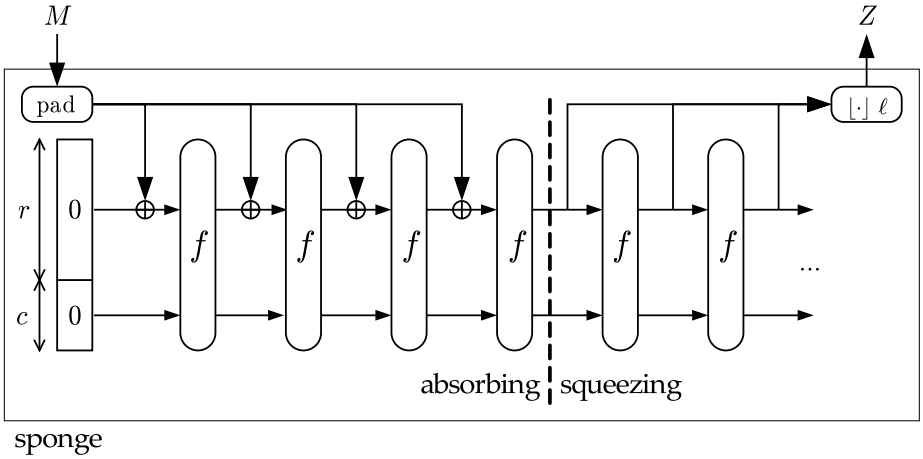

Figure 1: The sponge construction $\mathbf{sponge}[f, \mathbf{pad}, r]$ [4]

Squeezing consists of concatenating r bits at a time to an output bitstring Z that is truncated to ℓ bits. The sponge function f must be called once for each r bits of output after the first full block.

2.2 Sponge Applications

The sponge construction can be used for many applications beyond hashing without modification. The applications listed in this section are of considerable relevance to the ultimate goal of efficient authenticated encryption. Note that the sponge construction is particularly attractive here (and in general with keyed modes) because to the sponge, there is no differentiation between the key, initialization vector (IV), and message data. All input data is treated exactly the same, and thus the design remains simple. This is in great contrast to traditional symmetric key cryptosystem design in which a key schedule is required.

2.3 MAC Generation

A Message Authentication Code (MAC) function is essentially a keyed hash function. In this case, $K||IV$ (the key concatenated with an initialization vector) is absorbed first and then the padded message is absorbed directly after. The output is squeezed r bits at a time until the desired length of the MAC is reached.

2.4 Bitstream Encryption

A MAC generation function built from a sponge construction is characterized by long absorbing phases (assuming a long message) and short squeezing phases. Bitstream encryption, i.e. using the sponge as a stream cipher, is characterized oppositely: absorbing is quick while squeezing is likely a much longer process. For this application, we simply absorb $K||IV$ and switch to the squeezing phase immediately. Squeezing continues until a keystream is no longer needed. The keystream is XORed with the message to produce the ciphertext.

2.5 Generic Security

The security of the sponge construction is based on the assumption that the underlying sponge function f is secure. That is, if f is indistinguishable from random then so should be the sponge construction it is instantiated within. Consequently, cryptographers designing a system based on the sponge construction need only be concerned with designing and cryptanalyzing a secure underlying function. The sponge construction, when used properly, is said to be secure against *generic attacks* – attacks which do not exploit any specific properties of the underlying sponge function. We call this the *generic security* of the construction [4].

The generic security of keyed constructions is higher than unkeyed. For our purposes we are interested only in the security of the keyed sponge construction where a permutation is used for f. Jovanovic et. al [5] proved in 2014 that the generic security level of keyed sponge constructions is lower bounded by

$$\min(2^{(r+c)/2}, 2^c, 2^{|K|}).$$

3 Duplex Construction

The duplex construction is highly related to the sponge construction. The main differences are that the duplex construction maintains state between calls and that there no longer exists a clear separation between the absorbing and squeezing phases. Absorbing and squeezing happen essentially at the same time, hence "duplexing". The duplex mode has several applications, with authenticated encryption being the one of obvious interest to us.

3.1 Duplex Parameters

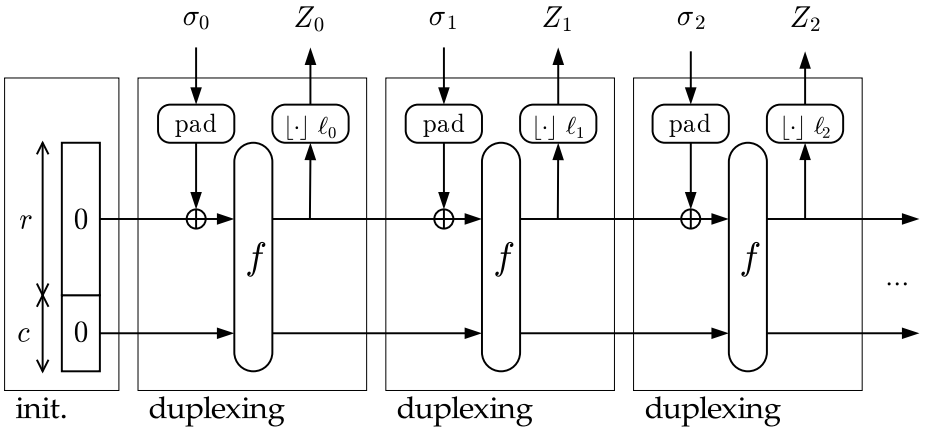

Figure 2: The duplex construction $\mathbf{duplex}[f, \mathbf{pad}, r]$ [4]

Parameters for the duplex construction are mostly the same as for the sponge construction. However, since the duplex construction maintains state, we build a *duplex object* D and make calls to it. The function which processes inputs and produces outputs is called $\mathbf{duplexing}$:

$$Z_i = D.\mathbf{duplexing}(\sigma_i, \ell_i)$$

Figure 2 shows the duplex construction. The i-th input is denoted σ_i and the i-th output is denoted Z_i, which is truncated to ℓ_i bits. Inputs are absorbed and processed at the same time that outputs are squeezed. For a duplex object it is possible to have an empty input or to not request an output. A *blank call* is a call to $\mathbf{duplexing}$ for which no input is provided ($|\sigma_i| = 0$). A *mute call* is a call for which no output is requested ($\ell_i = 0$).

3.2 Duplex for Authenticated Encryption

Authenticated encryption is easily achieved using the duplex construction. Figure 3 shows such a use case. First, we construct a duplex object D. Then we absorb K (or optionally $K||IV$) using one or more mute calls to $D.\mathbf{duplexing}$. More that one mute call may be required if the length of the key exceeds the rate r. We denote a header input to D as A; these arbitrary length inputs are authenticated but not encrypted. We denote a body input to D as B; these arbitrary length inputs are both encrypted and authenticated. A inputs are absorbed using one or more mute calls to $D.\mathbf{duplexing}$. B inputs are absorbed in a similar fashion and then the keystream Z is XORed with B to produce the ciphertext C. The tag T is produced using a blank call to $D.\mathbf{duplexing}$ after all header and body inputs have been processed.

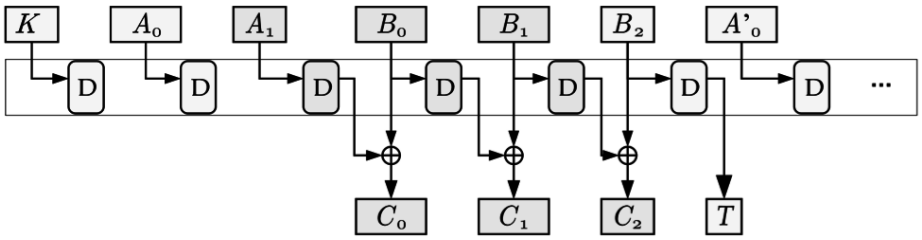

Figure 3: The duplex construction as used for authenticated encryption [6]

For example, the body B consists of three blocks of size r and so it requires three calls to be completely absorbed. An intermediate tag is requested after the first header and body pair is processed and before the next header begins. This is a very typical use case for e.g. network traffic. The tag can be of arbitrary length.

Clearly, the ciphertext and tags produced at any point depend on all of the previous inputs to D. Intermediate tags can be produced if desired, since blank calls can be made at any time. In summary, using the duplex construction for authenticated encryption provides the following advantages:

1. Easy to use
2. Single key required
3. Single-pass for encryption and authentication
4. Support for intermediate tags
5. Support for Additional Authenticated Data (AAD, or headers)
6. Secure against generic attacks
7. Ability to trade off speed and security by adjusting r

We note that the duplex construction may require *domain separation*, a generic mechanism for eliminating output ambiguity. For example, domain separation may simply consist of appending a *frame bit* that alternates in value to the end of every input in order to demarcate different data types.

3.3 Security

A reduction is used to prove the security of the duplex construction. Any calls made to the duplex construction can be reduced to calls to the keyed sponge construction. Therefore the security of the duplex depends on the security of the corresponding sponge, which can be shown to be secure against generic attacks. For a rigorous proof, we refer to [7].

4 Algorithm Specification

Our authenticated encryption algorithm is based on a simplified duplex construction. Padding and domain separation are assumed to be done at some higher level in the overall system if needed. For this reason, it is sufficient to specify only the duplex parameters and the sponge function f.

4.1 Duplex Parameters

We allow two key sizes: 128 bits and 256 bits. Our construction uses a 512-bit internal state, so we have $b = 512$. The rate r is 128 bits for both key lengths, which means that the capacity c is 384. Keeping the rate at a constant 128 bits for both instantiations means that switching between key lengths is a trivial task.

The capacity $c = 384$ provides sufficient security against generic attacks for both 128- and 256-bit keys. As explained in Section 2, we know from [5] that the generic security level is

$$\min(2^{(r+c)/2}, 2^c, 2^{|K|}).$$

For a 128-bit key, the generic security level is 2^{128}. For a 256-bit key, the generic security level is 2^{256}.

4.2 Permutation f

Our underlying sponge function f is a permutation and thus is invertible. For compactness, we specify only the forward permutation here as the inverse is not required for practical purposes.

The permutation consists of a number of rounds. Each round can be represented as the composition of several subfunctions or *steps*: a substitution, a bitwise permutation, a mixing layer, and the addition of a round constant.

4.2.1 Substitution Step

The substitution step is a bricklayer permutation that uses 32 identical, bijective 16×16 S-boxes. This step is the main source of confusion within the permutation. Furthermore, it is the only non-linear step, as is typical with many substitution-based symmetric key algorithms [8].

To the best of our knowledge this is the first cryptosytem to use such large S-boxes. We believe that, at the time of writing, the largest S-boxes used in the literature are the 8×8 bijective S-boxes used by the Advanced Encryption Standard (AES) [9][10].

Our S-box is an AES-inspired design taken directly from Wood's thesis on the subject [11]. The primary reason for using this particular class of 16-bit S-boxes is that they are efficiently implementable in hardware. Rather than being based on a random mapping, they are based on multiplicative inversion in a finite field followed by an affine transformation. This allows us to implement a circuit which performs the field operations rather than use the corresponding (and prohibitively large) look-up table.

This S-box is based on multiplicative inversion in $\mathrm{GF}(2^{16})/\langle p(x)\rangle$ where

$$p(x) = x^{16} + x^5 + x^3 + x + 1.$$

We represent an input to the S-box (and inverse S-box) as a 16-bit column vector

$$x = \begin{pmatrix} x_{15} & x_{14} & \dots & x_1 & x_0 \end{pmatrix}^{\mathrm{T}},$$

where x_{15} is the MSB. Using this notation, the forward S-box function is given as

$$\mathbf{S}(x) = \begin{pmatrix} 0&0&1&0&0&0&0&1&0&0&1&1&1&1&1&0 \\ 1&1&0&0&0&0&0&1&0&1&1&0&1&0&1&0 \\ 1&1&0&0&1&0&1&1&0&1&0&1&0&0&1&1 \\ 1&1&1&0&0&0&1&0&0&1&1&0&0&0&0&0 \\ 1&1&0&0&0&1&1&0&0&1&1&1&1&0&1&1 \\ 0&1&0&0&0&0&1&1&0&1&1&1&1&1&0&1 \\ 0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0 \\ 1&0&1&1&1&0&1&1&0&0&0&1&0&1&1&1 \\ 0&1&0&0&0&0&0&0&1&0&0&1&1&1&0&1 \\ 1&0&1&1&0&0&0&1&0&0&1&0&1&0&0&0 \\ 1&0&1&0&0&1&1&1&0&0&1&1&0&1&0&0 \\ 1&0&1&1&1&0&1&1&1&1&0&1&1&0&0&1 \\ 1&0&1&0&0&1&0&1&1&0&0&1&0&0&0&1 \\ 0&1&0&0&0&1&1&1&1&0&0&0&0&0&0&1 \\ 1&0&0&0&1&1&0&1&0&1&1&1&1&0&0&0 \\ 1&1&0&1&0&1&1&0&1&0&0&1&1&0&0&0 \end{pmatrix} \begin{pmatrix} x_{15} \\ x_{14} \\ x_{13} \\ x_{12} \\ x_{11} \\ x_{10} \\ x_9 \\ x_8 \\ x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix}^{-1} \oplus \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

A hardware implementation for this particular S-box requires just 1238 XOR gates and 144 AND gates.

4.2.2 Bitwise Permutation Step

Bitwise permutations are easily implementable in hardware via a simple rerouting of wires. Compared to a permutation on the words of the state, a bitwise permutation intuitively provides much better diffusion. The bitwise permutation step is the main source of long-range (i.e. across the entire state) diffusion in the algorithm.

The bitwise permutation also helps maximize the minimum number of active S-boxes by being subject to certain constraints. We use a permutation that satisfies the following properties:

1. All outputs of a given S-box go to 16 different mixers
2. The permutation is a *derangement*; it has no fixed points
3. High order; it does not repeat within the number of rounds
4. No low order bits; the order of any bit equals the order of the overall permutation

5. Easily definable by some affine function

There is obviously no cryptographic significance to how "easy" it is to express a bitwise permutation. This is merely to cut down on the search space and to avoid having to provide a table with 512 entries to express the permutation. The *order* of a bitwise permutation is the number of times it must be applied before it ends up in its original orientation. A particular bit in a bitwise permutation also has the notion of order and a bit of low order is one that returns to its original position before the entire permutation begins to cycle. If a bit has order zero, it is unaffected by the bitwise permutation and is called a *fixed point*.

We chose the following permutation to use for our algorithm since it is the first function to satisfy all properties:

$$\pi(x) = 31x + 15 \bmod 512$$

This bitwise permutation has order 32. For a complete listing of all bitwise permutations that satisfy our requirements, refer to Appendix A.

4.2.3 Mix Step

The purpose of the mix step is to provide local diffusion (i.e. across two words) and increase the linear and differential branch numbers of a round from two to three. We use a mixer based on multiplication by a 2×2 matrix in $\mathrm{GF}(2^{16})$ modulo the irreducible polynomial

$$p(x) = x^{16} + x^5 + x^3 + x^2 + 1.$$

The mixer takes two words A and B as input and produces outputs A' and B' as follows:

$$\begin{pmatrix} A' \\ B' \end{pmatrix} = \begin{pmatrix} 1 & x \\ x & x+1 \end{pmatrix} \begin{pmatrix} A \\ B \end{pmatrix}$$

The MSB of each word is taken as the leftmost bit and is represented by x^{15}.

The forward mixer is efficiently implementable in hardware. Figure 4 shows how this matrix multiplication is implemented. The $x*$ operation is a multiplication by x in $\mathrm{GF}(2^{16})$. Its implementation, which is shown in Figure 5, is very simple.

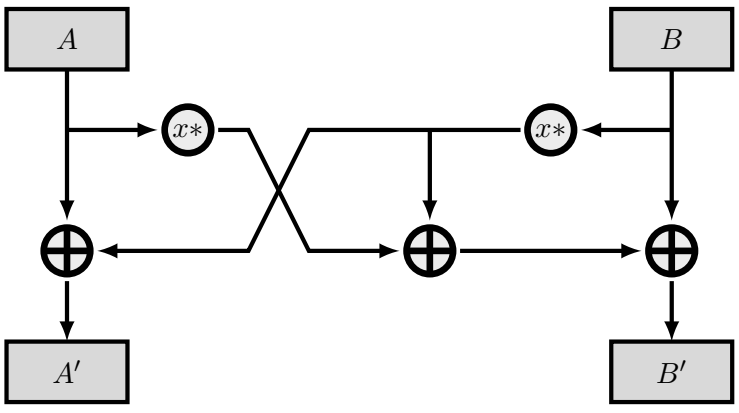

Figure 4: Hardware implementation of the forward mixer function.

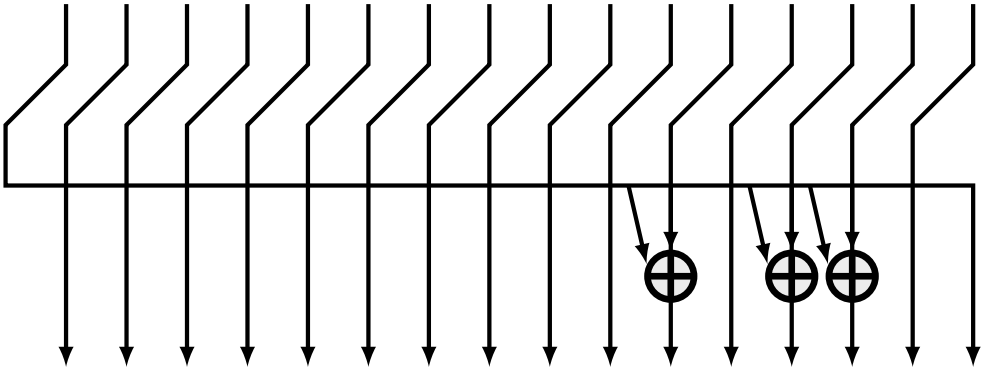

Figure 5: Hardware implementation of the $x*$ function. The leftmost bit is the MSB.

4.2.4 Add Round Constant Step

This step consists of adding a 512-bit value to the state using bitwise XOR in order to disrupt symmetry and prevent slide attacks. The round constant RC_i for round i is given by the formula

$$RC_i = \mathbf{SHA3\text{-}512}(\mathbf{ASCII}(i)),$$

where $\mathbf{ASCII}(i)$ is a function that provides the one or two byte ASCII representation of i and $\mathbf{SHA3\text{-}512}$ is the SHA-3 hash function that outputs a 512-bit message digest.

4.3 Number of Rounds

This algorithm uses 10 rounds for a 128-bit key and 16 rounds for a 256-bit key. The number of rounds is determined, as is typical with block ciphers and permutations, by calculating the number needed for resistance to linear and differential cryptanalysis and adding some buffer to increase the security margin. For a more in-depth treatment, refer to Section 5.

4.4 Customization

While a specific instantiation is specified here, our algorithm is highly customizable within our security margin. This could be useful in the case that different users want unique, proprietary algorithms. We list several possible customizations here.

4.4.1 State Initialization

In the given specification, the inner state (like the outer state) is initialized to zero. This is not a requirement; indeed, the inner state could be initialized to any 384-bit value. Each user could generate their own unique value to set during the initialization phase. This happens before the first mute calls that absorb the key.

4.4.2 S-boxes

The AES-inspired S-box used here is efficiently implementable in hardware. There are certainly many other cryptographically secure 16-bit S-boxes, but randomly generated ones may not be suitable for hardware implementation due to size constraints. This is an area for further research. Still,

several other AES-like 16-bit S-boxes are presented in [11]. Any new S-box introduced into the algorithm shall be analyzed to determine its linear and differential characteristics and the number of rounds should be adjusted accordingly if necessary. This analysis can easily be performed using the tools mentioned in Section 5.

4.4.3 Bitwise Permutations

The bitwise permutation provided in this algorithm specification is one of many that satisfies the constraints we impose. We provide all suitable bitwise permutations in Appendix A. Users may select any of these without need for further cryptanalysis.

4.4.4 Mixers

Our mixer is based on a specific 2×2 matrix multiplication in $\mathrm{GF}(2^{16})$ modulo a specific irreducible polynomial $p(x)$. Many matrices are expected to satisfy the constraints that we impose. These constraints are:

1. The 2×2 matrix should be invertible in $\mathrm{GF}(2^{16})/\langle p(x) \rangle$
2. The matrix should have differential and linear branch number equal to three (the maximum possible)

Like the addition of a new S-box, any new matrix introduced to the algorithm should be analyzed to ensure it meets these constraints. This analysis can also easily be performed using the tools mentioned in Section 5.

4.4.5 Round Constants

The round constants presented here are based on SHA-3 hash values. However, they could be any values that satisfy the following constraints. Round constants should be:

1. Unique for each round; to prevent against slide attacks
2. Random, pseudorandom, or highly asymmetric; to reduce symmetry in the state

The round constants are not expected to have any cryptographic significance outside of this. Different users can generate their own unique set of round constants without difficulty.

5 Preliminary Cryptanalysis

The duplex construction has been shown to be secure against generic attacks by the KECCAK team [4]. Therefore it is sufficient for us to assess only the security of the underlying sponge permutation f.

We provide an overview of our preliminary cryptanalysis here. Emphasis is placed on the most general and prevalent forms of attacks. Resistance against these techniques should result in resistance against many other less general techniques. Our aim is to provide intuitive explanations of prevalent methods and simply explain why our permutation should be resistant. Further cryptanalysis, as with all cryptosystems, is always welcome for future work.

5.1 Differential Cryptanalysis

Differential cryptanalysis was publicly introduced by Biham and Shamir in 1991 in their landmark paper on the subject [12]. Since then, it has been applied with varying degrees of success to a great number of cryptosystems. As such, it is a fundamental requirement of symmetric key cryptosystem design to prove resistance to differential cryptanalysis.

5.1.1 Overview

The goal of differential cryptanalysis is to exploit non-random behavior of a system (in our case, a permutation) with regard to the propagation of differences. A *difference*, denoted ΔX, is the bitwise XOR (for our case) of two bitstrings. For example,

$$\Delta X = X' \oplus X''$$

is the difference between bitstrings X' and X''. For differential cryptanalysis, a difference ΔX is fed through a system and a resulting difference ΔY is obtained. The pair of these two related differences is called a *differential* and is denoted $(\Delta X, \Delta Y)$.

Differentials occur with some associated probability. For an ideal system the probability of a given differential is $1/2^n$ where n is the length of the bitstrings involved. A system is said to exhibit non-random behavior if the magnitude of the probability p_D for some differential $(\Delta X, \Delta Y)$ is much greater than the ideal value. This information could be used to mount an attack on the system [13].

To launch an effective attack, a cryptanalyst first has to focus on the S-boxes. The S-boxes are analyzed to determine their maximum differential probabilities.

A *differential trail* or *characteristic* is the propagation of non-zero differentials throughout a system (i.e. across rounds). A *differentially active* S-box is an S-box that has a non-zero difference at its input during an attack; it is part of a differential trail.

Proving resistance against differential cryptanalysis for a substitution-permutation network requires computing a lower bound on the minimum number of active S-boxes across some number of rounds. The more active S-boxes there are, the less likely an attack is to succeed since exponentially more chosen plaintexts will be needed for additional each active S-box. The *branch number* of an operation is of particular importance here. It can be simply defined as the minimum number of active S-boxes across just two rounds of a system (e.g. our permutation). The technique of maximizing the branch number of a round is known as the *wide trail design strategy*. It is the main design strategy behind AES, which has a round branch number of five [14][9].

The number of plaintext/ciphertext pairs required to mount a successful differential attack should exceed the number required for a brute force attack. As the differential probability reduces across rounds, more pairs are required for a successful attack. We loosely refer to the number of pairs required as the *complexity* of the differential attack. The number of rounds is increased until this complexity exceeds that of a brute force method.

5.1.2 Algorithm Resistance

To determine the resistance of our algorithm to differential cryptanalysis, we first have to determine the maximum differential probability of our S-box. We determined this value to be $p_{D,max} = 2^{-14}$

using an S-box evaluation program called `Eval16BitSbox` [15] written with Kaminsky's Parallel Java 2 library [16].

Next, it is necessary to determine the branch number of a round. For this, we only need to analyze the mixer. We purposefully designed a mixer with differential branch number equal to three, meaning that minimally three S-boxes will be differentially active between two rounds. This is in fact the maximum achievable branch number for a transformation defined by multiplication by a 2×2 matrix. To verify this, we used a SAT solver called CryptoMiniSat [17]. This SAT solver takes as input a Boolean equation in conjunctive normal form (CNF) and determines if it is *satisfiable*; that is, if it can ever produce an output of '1' for any set of input values. Our CNFs were generated using Kaminsky's `SatProblem` Java class [15].

The CNFs generated are unsatisfiable if and only if the mixer has differential branch number equal to three since it answers the following question: *is it possible to have a difference in only one input and only one output?* Through SAT solver analysis we determined that this is not possible for our mixer; that is, if there is a non-zero difference in only one input, there must be a non-zero difference in each output. In the event that there is a difference in both inputs, there may be a difference in only one output. This still leads to a differential branch number of three since two S-boxes must have been active in the previous round to lead to those two input differences. The probability of a difference in either output is $p_{D,out} = 2^{-15}$.

With all of this information, it is possible to calculate the number of rounds needed for resistance to differential attacks. The worst-case probability of successfully propagating a difference over two rounds is given by

$$(p_{D,max})^{\mathcal{B}_D} \cdot p_{D,out},$$

where $\mathcal{B}_D = 3$ is the differential branch number. From this we constructed Table 1, which shows the worst-case differential probabilities for higher numbers of rounds.

Rounds	Worse Case Differential Probability
2	2^{-57}
4	2^{-114}
6	2^{-171}
8	2^{-228}
10	2^{-285}
12	2^{-342}
14	2^{-399}
16	2^{-456}

Table 1: Worst case differential probabilities over increasing rounds

Therefore the complexity of a differential attack exceeds the complexity of a brute force search of a 128-bit keyspace at six rounds. To increase our security margin significantly, we require 10 rounds for a 128-bit key. For a 256-bit key, 16 rounds are required to achieve a similar security margin.

5.2 Linear Cryptanalysis

Linear Cryptanalysis was first introduced by Matsui in 1993 in his landmark paper, [18]. As with differential cryptanalysis, it is a fundamental requirement of symmetric key cryptosystem design to

prove resistance against linear attacks.

5.2.1 Overview

Linear cryptanalysis is surprisingly similar to differential cryptanalysis in many ways. However, for linear cryptanalysis we are concerned with estimating the behavior of a system using linear expressions rather than highly probable differential characteristics. As with differential cryptanalysis, the first step is to analyze the S-boxes involved in the substitution-permutation network. An S-box, by definition, should be highly nonlinear to provide sufficient confusion. However, it is possible to uncover linear approximations of S-box outputs that occur with high (or low) probability. We can represent our S-box as a vectorial Boolean function

$$S\colon \mathbb{Z}_2^{16} \to \mathbb{Z}_2^{16}$$

in which the input X and output Y are represented as row vectors, e.g.

$$X = \begin{pmatrix} X_1 & X_2 & \dots & X_{15} & X_{16} \end{pmatrix},$$

where $X_i \in \mathbb{Z}_2$. Favoring typical convention found in the literature, X_1 is the MSB [13]. This notation allows us to easily represent linear approximations in the form

$$\left(\bigoplus_{i=1}^{16} X_i \right) = \left(\bigoplus_{i=1}^{16} Y_i \right).$$

The ideal *linear probability* p that such an approximation holds true is exactly equal to 1/2. We are concerned with deviations from this ideal probability, known as the *linear bias*, ϵ. Clearly, $\epsilon = p - 1/2$. We found the maximum linear bias of our particular S-box to be $\epsilon_{L,max} = 2^{-8}$ using the same program as previously described.

The *linear branch number* $\mathcal{B}_L$ is the minimum number of linearly active S-boxes across two rounds of our permutation. As with differential cryptanalysis, it depends solely on our mixer.

5.2.2 Algorithm Resistance

Recall that we have verified via SAT solver analysis that the differential branch number of our mixer is three. In [9], Daemen and Rijmen prove the following result: the linear branch number of a linear transformation specified by multiplication by a matrix M is equal to the differential branch number of the linear transformation specified by the transpose of that matrix. Therefore, a sufficient condition for the differential and linear branch numbers to be equal is that the matrix is symmetric. Our matrix is symmetric, and therefore we know $\mathcal{B}_D = \mathcal{B}_L$ without the need for further analysis.

The final step to prove the resistance of our algorithm against linear cryptanalysis involves determining the linear bias of two complete rounds of our permutation. To combine linear biases, we use Matsui's Piling-Up Lemma from [18]:

$$\epsilon = 2^{n-1} \prod_{i=1}^{n} \epsilon_i,$$

where $n = 3$ is the number of linearly active S-boxes across two rounds and $\epsilon_i = \epsilon_{L,max} = 2^{-8}$ is the worst case linear bias of those S-boxes. Note that we need not consider the mixer since, being a

linear function, it must have a maximum linear probability $p_{mix} = 1$, corresponding to a maximum bias of $\epsilon_{mix} = 1/2$. This gets cancelled out. Also from Matsui's paper, we know that the number of plaintext/ciphertext pairs (again referred to loosely as the *complexity*) needed to exploit the overall bias ϵ is approximately ϵ^{-2}. Using this information, we constructed Table 2.

Rounds	Worst Case Linear Bias	PT/CT Pairs Required
2	2^{-22}	2^{44}
4	2^{-44}	2^{88}
6	2^{-66}	2^{132}
8	2^{-88}	2^{176}
10	2^{-110}	2^{220}
12	2^{-132}	2^{264}
14	2^{-154}	2^{308}
16	2^{-176}	2^{352}

Table 2: Worst case linear biases and linear attack complexities over increasing rounds

Therefore the complexity of a linear attack exceeds the complexity of a brute force search of a 128-bit keyspace at six rounds. To increase our security margin significantly, we require 10 rounds for a 128-bit key. For a 256-bit key, 16 rounds are required to achieve a similar security margin.

5.3 Algebraic Attacks

Differential and linear cryptanalysis take a probabilistic approach to estimating the behavior of a system. In contrast, algebraic attacks take a deterministic approach in that they aim to find mathematical models of a system that hold with unity probability. For example, in 2001 Ferguson et al. [19] introduced an elegant and complete algebraic representation of AES. The ability to create such a simple mathematical representation of the cipher initially raised alarm throughout the cryptographic community. However, the security of AES seems to be uncompromised since we believe it is far too difficult to solve such an algebraic system - the algebraic complexity of the AES S-box is too high.

Until there is reason to believe otherwise, it seems that these algebraic attacks would be highly ineffective against our permutation. Even if there were a practical algebraic attack demonstrated on AES, which all literature indicates as highly implausible right now, the much larger size of our S-box and therefore the much higher algebraic complexity (see [11]) leads us to conjecture that our permutation would still be resistant.

6 Conclusions

In this paper we presented a customizable authenticated encryption algorithm based on the duplex construction that is targeted for hardware implementation. We believe this algorithm to be highly secure against known attacks. In particular, we provided proof of resistance against linear and differential attacks as well as solid reasoning for resistance against algebraic attacks.

6.1 Future Work

There are two primary areas for further work relating to the algorithm presented here. As with all cryptosystems, further cryptanalysis is always appreciated. In particular, it would be interesting to determine the best possible linear and differential trails across several rounds.

The other area of work is the hardware implementation of the algorithm described here. In particular, quantitative results relating to the resource usage for an FPGA implementation are of great interest. We are confident that our permutation is designed in such a way that a relatively small amount of resources will be required.

References

[1] M. T. Kurdziel and J. Fitton, "Baseline Requirements for Government and Military Encryption Algorithms," in *MILCOM 2002. Proceedings*, vol. 2, pp. 1491–1497, IEEE, 2002.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "The KECCAK reference," *NIST SHA-3 Submission Document*, January 2011. `http://http://keccak.noekeon.org/Keccak-reference-3.0.pdf`.

[3] S.-j. Chang, R. Perlner, W. E. Burr, M. S. Turan, J. M. Kelsey, S. Paul, and L. E. Bassham, "Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition." NIST Internal Report 7896, November 2012. `http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf`.

[4] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Cryptographic Sponge Functions," 2011. `http://http://sponge.noekeon.org/CSF-0.1.pdf`.

[5] P. Jovanovic, A. Luykx, and B. Mennink, "Beyond $2^{c/2}$ Security in Sponge-Based Authenticated Encryption Modes." Cryptology ePrint Archive, Report 2014/373, 2014. `http://eprint.iacr.org/`.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Duplexing the sponge: single-pass authenticated encryption and other applications," August 2010. `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/presentations/DAEMEN_SpongeDuplexSantaBarbaraSlides.pdf`.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Duplexing the sponge: single-pass authenticated encryption and other applications," in *Selected Areas in Cryptography*, pp. 320–337, Springer, 2012.

[8] D. Stinson, *Cryptography: Theory and Practice, Second Edition*. CRC/C&H, 3rd ed., 2006.

[9] J. Daemen and V. Rijmen, *The Design of Rijndael: AES-The Advanced Encryption Standard*. Springer, 2002.

[10] NIST, "Specification for the Advanced Encryption Standard (AES)." Federal Information Processing Standards Publication 197, November 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[11] C. A. Wood, "Large Substitution Boxes with Efficient Combinational Implementations," Master's thesis, Rochester Institute of Technology, 2013.

[12] E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," *Journal of CRYPTOLOGY*, vol. 4, no. 1, pp. 3–72, 1991.

[13] H. M. Heys, "A Tutorial on Linear and Differential Cryptanalysis," *Cryptologia*, vol. 26, no. 3, pp. 189–221, 2002.

[14] J. Daemen and V. Rijmen, "The Wide Trail Design Strategy," in *Cryptography and Coding*, pp. 222–238, Springer, 2001.

[15] A. Kaminsky, "Block Cipher Analysis." `http://cs.rit.edu/~ark/parallelcrypto/blockcipheranalysis/`, 2014.

[16] A. Kaminsky, "Parallel Java 2 Library." `http://cs.rit.edu/~ark/pj2.shtml`, 2014.

[17] M. Soos, "CryptoMiniSat." `http://msoos.org/cryptominisat2/`, 2014.

[18] M. Matsui, "Linear Cryptanalysis Method for DES Cipher," in *Advances in Cryptology—EUROCRYPT'93*, pp. 386–397, Springer, 1994.

[19] N. Ferguson, R. Schroeppel, and D. Whiting, "A Simple Algebraic Representation of Rijndael," in *Selected Areas in Cryptography*, pp. 103–111, Springer, 2001.

A Bitwise Permutations

The following bitwise permutations defined by the affine function

$$\pi(x) = \alpha x + \beta$$

satisfy all required properties listed in Section 4. The order of all bitwise permutations listed here is 32.

α	**31**	**33**	**95**	**97**	**159**	**161**	**223**	**225**	**287**	**289**	**351**	**353**	**415**	**417**	**479**	**481**
Corresponding β values	15	16	15	16	15	16	15	16	15	16	15	16	15	16	15	16
	31	48	31	48	31	48	31	48	31	48	31	48	31	48	31	48
	47	80	47	80	47	80	47	80	47	80	47	80	47	80	47	80
	63	112	63	112	63	112	63	112	63	112	63	112	63	112	63	112
	79	144	79	144	79	144	79	144	79	144	79	144	79	144	79	144
	95	176	95	176	95	176	95	176	95	176	95	176	95	176	95	176
	111	208	111	208	111	208	111	208	111	208	111	208	111	208	111	208
	127	240	127	240	127	240	127	240	127	240	127	240	127	240	127	240
	143	272	143	272	143	272	143	272	143	272	143	272	143	272	143	272
	159	304	159	304	159	304	159	304	159	304	159	304	159	304	159	304
	175	336	175	336	175	336	175	336	175	336	175	336	175	336	175	336
	191	368	191	368	191	368	191	368	191	368	191	368	191	368	191	368
	207	400	207	400	207	400	207	400	207	400	207	400	207	400	207	400
	223	432	223	432	223	432	223	432	223	432	223	432	223	432	223	432
	239	464	239	464	239	464	239	464	239	464	239	464	239	464	239	464
	255	496	255	496	255	496	255	496	255	496	255	496	255	496	255	496
	271		271		271		271		271		271		271		271	
	287		287		287		287		287		287		287		287	
	303		303		303		303		303		303		303		303	
	319		319		319		319		319		319		319		319	
	335		335		335		335		335		335		335		335	
	351		351		351		351		351		351		351		351	
	367		367		367		367		367		367		367		367	
	383		383		383		383		383		383		383		383	
	399		399		399		399		399		399		399		399	
	415		415		415		415		415		415		415		415	
	431		431		431		431		431		431		431		431	
	447		447		447		447		447		447		447		447	
	463		463		463		463		463		463		463		463	
	479		479		479		479		479		479		479		479	
	495		495		495		495		495		495		495		495	
	511		511		511		511		511		511		511		511	

Table 3: Bitwise permutations satisfying all desired properties

B Round Constants

Constant	Hex Value
RC_1	00197a4f5f1ff8c356a78f6921b5a6bfbf71df8dbd313fbc5095a55de756bfa1 ea72406950051492 94f2a2e419ae251fe2f7dbb67c3bb647c2ac1be05eec7ef9
RC_2	ac3b6998ac9c5e2c7ee8330010a7b0f87ac9dee7ea547d4d8cd00ab7ad1bd5f5 7f80af2ba711a9eb137b4e83b503d24cd7665399a48734d47fff324fb74551e2
RC_3	ce4fd4068e56eb07a6e79d007aed4bc8257e10827c74ee422d82a29b2ce8cb07 9fead81d9df0513bb577f3b6c47843b17c964e7ff8f4198f32027533eaf5bcc1
RC_4	5058cb975975ceff027d1326488912e199b79b916ad90a3fe2fd01508cd7d7c0 1bc8aaa4d21a8473fb15f3b151ab9e44172e9ccb70a5ea04495af3ec03b5153e
RC_5	84da272d13a44f0898ee4ea53334c255d894cc54d357c55466d760debde482a2 44c128df641e80673a8bc34a1620d880b7965e549f313ddccfd506b073413b87
RC_6	bb93aaa23b38ea96c9346ef91e184982bf50e91033f4354ecb20d3c7390c2b41 862e8825ec3d0fee0a6f978881f90728c6748e4aed8b732350075d6c2bdd8e4b
RC_7	fe32f3eba76626dedf36622bfdc5ccd33db2f3e0dd7c3c128298ea78c1cc7fee 1a140edb8e57cd5824c7f4b817c0fc94e70da5b9399faaf9a848a46ad30679e9
RC_8	952ba02486b818febc0ec98559df27c79357838f011b1e5bc11f2cfb6fc0573e 545978c2bc5b390f44907f8da0dfd68206fe4521f86ba6c879ec1e69caed9533
RC_9	b41e6bb4ed20294016399c268da6bf88c89e2dc118a361b3560ee8daed973a8f 9778df40e308c1206fa42f97f3fd3f63d2b4b3b57eb5bcbec6ad64d46216b692
RC_{10}	6954a418cecc43633bd526c2499dfc16b832f58b216b9a8b226a6a0b7918d364 a7939004339de0ba08e2b547e64dc5622e24b0c4f8f415d9e0a84cb94b6c5f3f
RC_{11}	2e4b9ad37091e3e5a218c5e57b33ed3470ba4f31fbcf16424684fdd5cde38e88 9eae3f018b37af58c24ccc8af57abc2c6911408dd20ef6435e4494a3e6599a06
RC_{12}	aa42aca73bd7f8a17e987f281422b266e44f0de1615d2d393c620c8c5a2c80b4 f06178c8455bf98179603f2f1bcb30b2559f282c799e40533b0665f97a2a706a
RC_{13}	969c39ae2dc16834310344c0579d0ffdfde01772dbf9a4cab984953c395d7791 1510f39e5f37295e3611a1d46101460daf731ddbdab1ec1bbc512edc44680d8d
RC_{14}	8a1e6ce31f0b526d884b584aa1a5ae4294fcf85fd2e525f959ed1a54233359c7 c5fece6d24775e7d4a9ad97c2632a3be5b331a8f580f557b269e7b65123a5992
RC_{15}	9bd64a932f09672def04b6a94753a3e4087a1c3895078dc70927fcd774888dfd 400b95fd1c6a0b2a91a1ba44eea09f5163dba4dfa9da7b8eb97d791cab566437
RC_{16}	48401f65c2d2d9e71fe47bd80b28d834eee8fff3be9aa4608cba33e6fedce0b1 693c80cdc36db7f504e4abea23ccc6729a030f5b3e035fb59c2c788215cf84a8

Table 4: Round constants for up to 16 rounds