Design and Cryptanalysis of a Customizable Authenticated Encryption Algorithm

by

Matthew Joseph Kelly

The thesis "Design and Cryptanalysis of a Customizable Authenticated Encryption Algorithm" by Matthew Joseph Kelly has been examined and approved by the following Examination Committee:

Alan Kaminsky

Professor, Department of Computer Science

Primary Advisor

Marcin Łukowiak

Associate Professor, Department of Computer Engineering

Primary Advisor

Michael Kurdziel

Harris Corporation

Reza Azarderakhsh

Assistant Professor, Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Design and Cryptanalysis of a Customizable Authenticated Encryption Algorithm

I, Matthew Joseph Kelly, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

_____

Matthew Joseph Kelly

_____

Date

Dedication

To my parents, Donna and Martin.

Acknowledgments

Abstract

Design and Cryptanalysis of a Customizable Authenticated Encryption Algorithm

Matthew Joseph Kelly

Supervising Professors: Alan Kaminsky & Marcin Łukowiak

It is common knowledge that encryption is a useful tool for providing confidentiality. Authentication, however, is often overlooked. Authentication provides data integrity; it helps ensure that any tampering with or corruption of data is detected. It also provides assurance of message origin. Authenticated encryption (AE) algorithms provide both confidentiality and integrity / authenticity by processing plaintext and producing both ciphertext and a Message Authentication Code (MAC). It has been shown too many times throughout history that encryption without authentication is generally insecure. This has recently culminated in a push for new authenticated encryption algorithms.

There are several authenticated encryption algorithms in existence already. However, these algorithms are often difficult to use correctly in practice. This is a significant problem because misusing AE constructions can result in reduced security in many cases. Furthermore, many existing algorithms have numerous undesirable features. For example, these algorithms often require two passes of the underlying cryptographic primitive to yield the ciphertext and MAC. This results in a longer runtime. It is clear that new easy-to-use, single-pass, and highly secure AE constructions are needed. Additionally, a new AE algorithm is needed that meets stringent requirements for use in the military and government sectors.

This thesis explores the design and cryptanalysis of a novel, easily customizable AE algorithm based on the duplex construction. Emphasis is placed on designing a secure pseudorandom permutation (PRP) for use within the construction. A survey of state of the art cryptanalysis methods is performed and the resistance of our algorithm against such methods is considered. The end result is an algorithm that is believed to be highly secure and that should remain secure if customizations are made within the provided guidelines.

Contents

Chapter 1

Introduction

The overarching goal of cryptography is to enable people to communicate privately over an insecure channel in the presence of adversaries. This notion may be abstracted further; for example, the sending and receiving party may be the same person writing to and reading from a storage medium over time. Two requirements for achieving this goal are encryption and authentication. Encryption provides *confidentiality* while authentication provides data *integrity* and assurance of message origin.

Encryption without authentication has been shown to be insecure in practical instances several times in recent history. For example, Wireless Equivalence Privacy (WEP), an algorithm for "securing" communications over wireless networks, is badly broken due (in part) to a lack of proper authentication [1]. In fact, it is so badly broken that people with no cryptographic expertise can use existing tools on standard consumer machines to break into networks that use 104-bit WEP in less than 60 seconds [2].

Another example of a flaw in prevalent systems due to a lack of proper authentication came in 2002. An attack was found on the CBC mode of encryption employed by protocols such as SSL and IPSEC. The attack could have been avoided had an authenticated encryption scheme been used [3].

1.1   Motivation

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 required 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 [4].

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. First, it is important to understand how authenticated encryption has been evolving over the years.

1.2 Overview of Authenticated Encryption

1.2.1 AE Through Generic Composition

The naïve approach to authenticated encryption is generic composition. In this construction, one simply computes the ciphertext under a given key and computes the MAC under a different key. These operations can be done in any order or at the same time. There are thus three generic composition types: *Encrypt-and-MAC*, *MAC-then-Encrypt*, and *Encrypt-then-MAC*. The names here are straightforward and describe exactly the order of operations. In Encrypt-and-MAC, the encryption and authentication operations are performed in parallel and the ciphertext and MAC are concatenated together at the end. MAC-then-Encrypt computes the MAC, appends it to the plaintext, and then encrypts the result. Encrypt-then-MAC computes the ciphertext first and then computes the MAC over the ciphertext.

There is much ongoing debate about which generic composition method is best and each method has pros and cons in terms of security and performance. As a simple example, consider the performance of each method at a high level of abstraction. In Encrypt-and-MAC, the encryption (decryption) and authentication can happen in parallel. If using MAC-then-Encrypt, decryption must be performed before the MAC can be verified. The MAC can be checked before decryption in the case of Encrypt-then-MAC, thus saving the performance hit of decryption for messages in which authentication fails [5].

1.2.2 AE Modes of Operation

There are many existing block cipher modes of operation that provide authenticated encryption. The main difference between a generic composition and a mode of operation for authenticated encryption is that a generic composition always requires two keys, while a mode of operation generally requires only one. This is a huge benefit because key management is a difficult task, so it is desirable to limit the number of keys required for a cryptographic algorithm.

Patent-Encumbered Modes

The first notable modes of operation for AE were created by Jutla, an IBM researcher, in the year 2000. The two modes are called Integrity Aware Cipher Block Chaining (IACBC) and Integrity Aware Parallelizable Mode (IAPM) [6]. IACBC, as the name suggests, was inspired by the CBC mode of operation. This mode is highly serial in nature. IAPM was inspired by the Electronic Codebook (ECB) mode, which is highly parallelizable but insecure when used directly. Jutla's modes are historically noteworthy in that they provide authenticated encryption in a single pass with minimal overhead; however, they come with many disadvantages. For example, they do not support additional authenticated data (e.g. headers that are only authenticated, not encrypted). An even bigger problem is that they are highly patent encumbered. On top of all this, they require two keys and the underlying block cipher decryption mode. As a result of the lack of popularity, there has not been much cryptanalysis performed on IACBC or IAPM.

In 2001, Rogaway et al. [7] introduced a new AE mode called Offset Codebook (OCB) mode. This mode is based on IAPM, but with many added features. OCB requires only a single key and provides support for additional authenticated data. Furthermore, it is highly parallelizable and also a single-pass mode. The main disadvantages of OCB are that it also requires the block cipher decryption mode and that it is also patent encumbered.

Patent-Free Modes

Counter with CBC-MAC (CCM) mode was created by Whiting et al. in 2003 as a response to OCB's patented status [8]. CCM mode is patent-free but has many disadvantages. Like all of the patent-free modes that will be discussed here, it requires two passes under a single key. It does not, however, require the decryption mode of the underlying block cipher since it is built on Counter mode. CCM mode does not support block sizes other than 128 bits and does not support stream processing. This is a significant disadvantage because stream processing is often exactly what a AE construction is used for (e.g. streaming network traffic). Rogaway and Wagner present a detailed analysis of all of the disadvantages of CCM mode in [9].

Carter-Wegman + Counter (CWC) mode was introduced by Kohno et al. in 2004 as an attempt to improve upon the deficiencies of CCM mode [10]. Like CCM mode, it is a single-key mode that requires two passes. Unlike CCM, the second pass does not consist of any invocations of the underlying block cipher. It instead uses what is called a Carter-Wegman (CW) universal hash function. This CW hash function operates over a prime field and thus requires integer multipliers which consume a large amount of area when implemented in hardware.

McGrew and Viega created Galois/Counter Mode (GCM) in 2004 in an effort to improve upon the efficiency of CWC mode [11]. Instead of operating in a prime field, GCM performs multiplication in a binary Galois field. As a result, it consumes much less area in hardware. Like all other patent-free modes described here, it is single-key and two-pass. It should be noted that some people may argue that GCM is single-pass because only the first pass calls the underlying block cipher. GCM has seen widespread adoption by the cryptographic community. It is arguably the most used mode of operation for AE in existence today. Indeed, GCM is a main reason for the advent of Intel's carry-less multiplication instruction (PCLMULQDQ) in their 2010 line of processors [12].

EAX mode was introduced in 2004 by Bellare et al. as another alternative to CCM mode [13]. Like CCM mode, it is not parallelizable. Like all other modes listed here, it is single-key and two-pass. EAX mode is derived from a generic composition called EAX2 by the authors. EAX2 mode leaves the Pseudorandom Function (PRF) used for MAC computation and the underlying block cipher mode used for encryption unspecified. EAX2 uses two keys. EAX is a specific instantiation of EAX2 that uses only a single key. Like GCM, EAX mode has also seen widespread adoption.

1.2.3 Other Existing AE Constructions

Stream Cipher Based

There are two notable stream ciphers that provide authenticated encryption: Helix [14] and Phelix [15], both from Whiting et al. Phelix is the successor of Helix and was submitted to the recent eSTREAM competition which was a competition aimed at yielding better stream ciphers. Phelix, though very fast, is not currently a viable AE candidate because of its vulnerability to differential-linear attacks as exposed by Wu and Preneel [16]. The security flaw relates to lack of resistance to

misuse of the stream cipher. Still, new stream cipher based AE constructions are probable in the future.

Duplex Constructions

The duplex construction is based on a sponge construction and has promising applications to authenticated encryption. This construction has many extremely desirable features. It is single-key, single-pass, and supports additional authenticated data and intermediate MACs. For these reasons, the duplex construction forms a foundation for this thesis; see Chapter 3 for an in-depth treatment.

At the time of writing there is a new AE competition going on called CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness). Of the 57 first-round candidates, there are nine duplex-based submissions that are not withdrawn: Artemia, Ascon, ICEPOLE, Ketje, Keyak, NORX, PRIMATEs, STRIBOB, and π-Cipher [17]. It is far too early to tell which of these, if any, will be found to be secure and efficient enough for practical use. Furthermore, none of these submissions are readily customizable at an algorithmic level. Therefore they are not suitable for our purposes.

1.3   Our Contributions

The main contribution of this thesis is a novel, customizable authenticated encryption algorithm based on the duplex construction. This construction is suitable for hardware (e.g. FPGA) implementation. An itemized list of specific contributions is listed here:

1. Authenticated encryption algorithm specification

    (a) Includes customization suggestions and guidelines

2. Software model of the algorithm (C/C++)

    (a) Includes various tests and test data generators

    (b) Includes $\text{GF}(2^{16})$ library for algorithm verification

3. Tool for finding suitable bitwise permutations for algorithm customization (Python)

4. Survey of relevant cryptanalysis techniques and cryptanalysis results

5. Statistical test results

6. Tool for ensuring acceptable distribution of *P-values* (Python)

The rest of this thesis is organized as follows. In Chapter 2 we discuss some mathematical concepts that are central to this thesis. In addition, we explain the notation used for the remainder of this document. Chapter 3 explains in detail the sponge and duplex constructions as well as their resistance to generic attacks. The duplex construction forms the basis for the algorithm designed in this thesis. In Chapter 4 we provide a full specification for the AE algorithm as well as providing guidelines for customizations that can be made without reducing our security margin. Cryptanalysis of our construction is considered in Chapter 5, where we provide a survey of potentially relevant attacks. A variety of statistical tests and their results for our algorithm are presented in Chapter 6. Finally, we conclude in Chapter 7 and provide some ideas for future work.

Chapter 2

Mathematical Foundation and Notation

There are several mathematical concepts central to this research that we will briefly explain here. The notation used in the rest of this thesis is also explained here for clarity.

2.1 Algebraic Structures

2.1.1 Groups

An algebraic *group* is a set of elements G together with a binary operation $*$ (e.g. multiplication or addition) that satisfies the following properties:

1. *Associativity*. The operation is associative, i.e. $(a * b) * c = a * (b * c)$ for all $a, b, c \in G$.

2. *Closure*. G is closed under $*$; that is, $a * b \in G$ for all $a, b \in G$.

3. *Identity*. There is an element $e \in G$ (called the *identity*) such that $a * e = e * a = a$ for all $a \in G$.

4. *Inverses*. For each element $a \in G$ there exists an element $a^{-1} \in G$ (called the *inverse* of a) such that $a * a^{-1} = a^{-1} * a = e$.

A group may also have the property that $a * b = b * a$ for all $a, b \in G$. In other words, it is commutative. We call these groups *abelian* groups. The most common example of a group is probably $\mathbb{Z}$, the set of integers under addition, where the identity $e = 0$.

2.1.2 Rings

An algebraic *ring* is a set of elements R together with two binary operations $\cdot$ (called multiplication, with $a \cdot b$ denoted ab for brevity) and $+$ (called addition) that satisfies the following properties:

1. R is an abelian group under addition; its identity is called 0.

2. *Associativity*. Multiplication and addition are both associative.

3. *Distributivity*. $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$; that is, multiplication distributes over addition.

A ring is called abelian if multiplication also commutes over it. The most common example of a ring is also $\mathbb{Z}$, but now with multiplication as well as addition.

2.1.3 Fields

An algebraic *field* is a set of elements $\mathbb{F}$ together with the operations $\cdot$ and $+$ such that:

1. $\mathbb{F}$ is an abelian ring.

2. $\mathbb{F}$ is an abelian group under multiplication; its identity is called 1.

Groups, rings, and fields (all of which are *algebraic structures*) may be of finite or infinite cardinality. The cardinality of an algebraic structure G is called its *order* and is denoted $|G|$.

We use an exponential representation as shorthand for applying an operation $*$ to the same element. For example, $a * a * a = a^3$. For some structure G, the *order* of an element $a \in G$ is the smallest integer k such that $a^k = e$. For structures of finite order, it is a well-known result (Lagrange's theorem) that the order of an element divides the order of the structure [18].

2.1.4 Galois Fields

Finite fields are also commonly referred to as Galois fields (GFs). It is another well-known result that all Galois fields are of prime power order. A Galois field of order p^k is denoted GF(p^k) or $\mathbb{F}_{p^k}$, where p is prime. p is called the *characteristic* of the field and it is the smallest positive integer such that $a^p = 0$. Fields with $p = 2$ are typically of the most interest to cryptographers, and they are often called binary Galois fields. k is called the *degree* of the field.

When the degree of a field is greater than one, its elements can be represented as polynomials belonging to the set of equivalence classes modulo an irreducible polynomial $f(x)$. The degree of this polynomial, denoted $\deg(f(x))$, is equal to the degree of the field. We denote a GF of order p^k along with its irreducible polynomial as GF(p^k)/$\langle f(x) \rangle$. Elements in the field are represented using a polynomial basis in the indeterminant x with the form

$$a = \alpha_{k-1}x^{k-1} + \alpha_{k-2}x^{k-2} + \ldots + \alpha_1 x + \alpha_0,$$

where $\alpha_i \in \mathbb{Z}_p$. For example, in a binary Galois field we have $\alpha_i \in \mathbb{Z}_2$. Since operations in the GF are done modulo $f(x)$, any element $a \in$ GF(p^k) must satisfy $\deg(a) < k$.

For a binary Galois field we often represent elements in binary or hex format, as this is how they will be implemented in practice. For example, consider the element $x^{15} + x^3 + x^2 + 1$ in GF(2^{16}). This element can be represented in hex as 0x800d, where the most-significant bit (MSB) corresponds to x^{15}.

Addition in finite fields is done by performing element-wise addition over $\mathbb{Z}_p$. In a binary Galois field, this is equivalent to the bitwise XOR operation. Multiplication is performed by doing polynomial multiplication as usual and then reducing modulo $f(x)$ if needed. There are methods of optimizing this operation in both software and hardware.

Interpreting binary strings as elements of a binary Galois field is only useful for certain operations. Other times, we interpret portions of our state as arbitrary bitstrings for which certain other operations are defined.

2.2 Bitstrings

Bitstrings are strings of elements in $\mathbb{Z}_2$; in other words, they are binary strings. We say $s \in \mathbb{Z}_2^n$ if s is a bitstring of length n. $\mathbb{Z}_2^*$ indicates bitstrings of arbitrary but finite length and $\mathbb{Z}_2^\infty$ denotes

bitstrings of infinite length.

2.2.1 Operations

There are several relevant operations we define for bitstrings:

- $\lfloor s \rfloor_\ell$ indicates the truncation of a bitstring s to ℓ bits

- $s||t$ indicates the concatenation of bitstrings s and t, with the resulting length being $|s| + |t|$

- $s \lll r$ indicates the logical left rotation of bitstring s by r bits

- $s \ggg r$ indicates the logical right rotation of bitstring s by r bits

- $s \oplus t$ indicates the exclusive-or (XOR, or bitwise addition modulo 2) of bitstrings s and t, where $|s| = |t|$ is assumed

- $\mathbf{HW}(s)$ indicates the Hamming weight of bitstring s

- $\mathbf{HD}(s, t)$ indicates the Hamming distance between bitstrings s and t, where $|s| = |t|$ is assumed

The *Hamming weight* of a string is defined as the number of nonzero elements it contains. The *Hamming distance* between two strings is the number of elements that differ between them. Note that for elements of $\mathbb{Z}_2^n$,

$$\mathbf{HD}(s, t) = \mathbf{HW}(s \oplus t).$$

For this work, a bitstring of length 16 is called a *word*. The state of our cryptosystem, as we will see, is an element of $\mathbb{Z}_2^{512}$ that may be considered at times as the concatenation of 32 contiguous words. Where it matters, the MSB will be specified for a given operation.

2.3 Shannon's Information Theory

It is widely recognized that Claude Shannon laid the foundation for modern symmetric key cryptography in the late 1940s through two seminal papers on information theory and secrecy systems. His major relevant contributions include the notions of entropy, confusion, and diffusion [19][20].

2.3.1 Entropy

Entropy, as defined by Shannon, is the measure of the amount of information in some message M. It is denoted as $H(M)$ and is quantitatively defined as

$$H(M) = \log_2 n,$$

where n is the number of possible *meanings* of the message. For example, the entropy of a message containing only information about what month it is is $\log_2(12) \approx 3.58$ since there are 12 months. Another way to look at this is that exactly $\lceil 3.58 \rceil = 4$ bits are required to minimally encode in binary what month it is.

Entropy also measures *uncertainty*. The uncertainty of a message M is the number of plaintext bits needed to be recovered from ciphertext in order to learn the plaintext. For example, if we know that the plaintext is either "HEADS" or "TAILS" when provided with the ciphertext "ASDFQJWE", then the uncertainty is unity. We need only learn one bit (if chosen correctly) to learn the plaintext.

We often discuss the entropy of cryptosystems with respect to their key space. For example, a symmetric key system with a 128-bit key should have an entropy of 128 bits.

2.3.2 Transformations

A *transformation* is simply a function

$$t \colon X \to Y,$$

where X is the domain and Y is the codomain. Some transformations on a space are *entropy-preserving*; that is, if an input x has entropy $H(x)$ before the transformation, it has the exact same entropy (and thus uncertainty) after the transformation. An obvious requirement for entropy-preserving transformations is that they are bijective. We call a bijective transformation in which the domain is equal to the codomain a *permutation*. This notion is of central importance to the remainder of this thesis.

2.3.3 Confusion and Diffusion

Also of central importance to this thesis are Shannon's qualitative notions of confusion and diffusion. *Confusion* is a generic method used to obscure the relationship between the plaintext and ciphertext. For example, substitution provides confusion. *Diffusion* is a generic method that aims to dissipate the redundancy of plaintext throughout the ciphertext. For example, bitwise permutations provide diffusion. In theory, extremely large-scale substitutions are sufficient to create a secure symmetric key system. However, in practical block cipher systems both confusion and diffusion are required for efficiency reasons [21].

Chapter 3

Sponge and Duplex Constructions

Our algorithm is based on the duplex construction, a highly flexible new cryptographic primitive derived from the sponge construction with promising applications to authenticated encryption. We describe both the sponge and duplex constructions in this chapter and provide details about resistance to generic attacks.

3.1 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 [22][23]. 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. We present a few here but there are numerous more possibilities. The sponge construction is stateless; there is no information stored between calls to it.

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 versus permutations, and there are advantages and disadvantages for each choice of a function type [24].

3.1.1 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.

Figure 3.1 shows the sponge construction. It is split into two distinct phases: the *absorbing* phase and the *squeezing* phase. This is where the term "sponge" comes from. 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
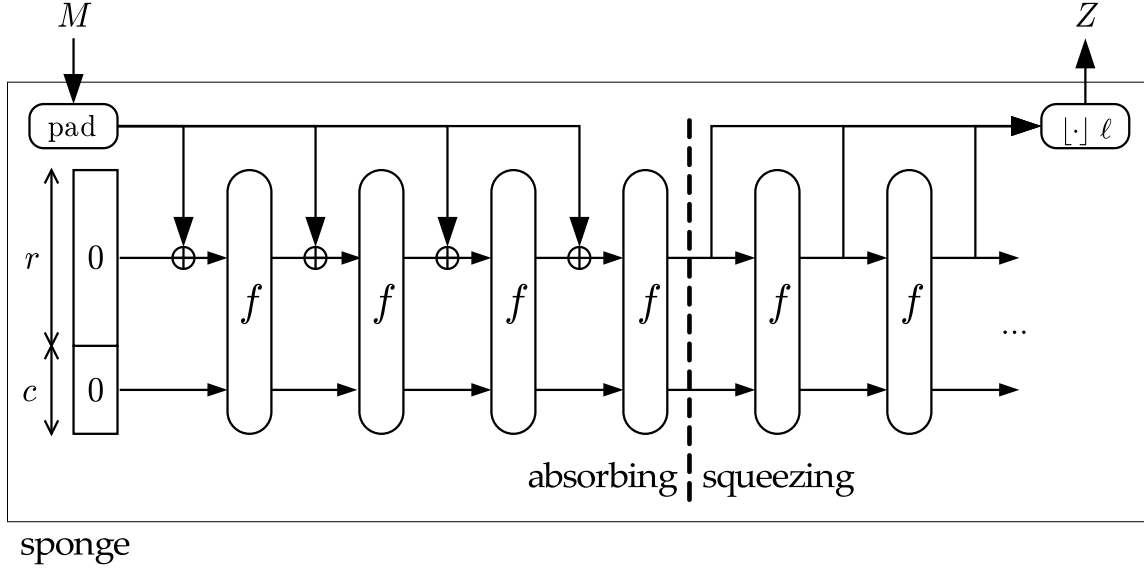
Figure 3.1: The sponge construction $\mathbf{sponge}[f, \mathbf{pad}, r]$ [24]

$b = r + c$. The speed of the construction partially relies on the rate, while the security is partially dependent on the capacity (see Section 3.1.3).

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. 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.

Simplified Sponge

A padding function is required for the classical sponge construction in order to ensure that inputs can be reformed into bitstrings with length equal to a multiple of r. Two basic requirements of a padding function are that it be reversible and that it never produce identical outputs for different inputs. We omit the lower level details of good padding functions here for brevity and refer the interested reader to [24] instead.

It is also useful, as we will see, to consider a sponge construction in which no padding function $\mathbf{pad}$ is required. The parameterized interface for such a construction becomes

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

Alternatively, we can say that $\mathbf{pad}$ exists but it is trivially given as

$$\mathbf{pad}(M) = M.$$

This lets us remain aligned with the original definition. In either case, we assume that any necessary padding is performed at some higher level in the overall system. This simplification allows us to focus more design effort towards the underlying sponge function f and ignore the details of padding.

3.1.2 Applications

Hashing

The sponge construction was originally envisioned as a generalization of hash functions to functions with arbitrary length output. Using the sponge construction for hashing is straightforward. We denote a hash function by

$$\mathbf{H} \colon \mathbb{Z}_2^* \to \mathbb{Z}_2^\ell,$$

where ℓ is the desired length of the message digest. In this case, the message M is padded and absorbed as usual. After absorption completes, the construction switches to squeezing mode and the state is squeezed until a message digest of length ℓ is acquired.

Without any structural changes, several other applications can be derived from the sponge construction. These following applications are of considerable relevance to the ultimate goal of authenticated encryption.

MAC Generation

A Message Authentication Code (MAC) function is essentially a keyed hash function. We denote a MAC function under a given key K and initialization vector IV by

$$\mathbf{MAC}_{K,IV} \colon \mathbb{Z}_2^k \times \mathbb{Z}_2^v \times \mathbb{Z}_2^* \to \mathbb{Z}_2^\ell,$$

where k is the length of the key, v is the length of the IV, and ℓ is the desired length of the tag (MAC). In this case, $K||IV$ is absorbed first and then the padded message is absorbed directly after as usual. Squeezing works the same as in a hash computation. 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, 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.

Bitstream Encryption

The previous two direct applications of the sponge construction were 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. We denote a stream cipher under a given key K and initialization vector IV by

$$\mathbf{STREAM}_{K,IV} \colon \mathbb{Z}_2^k \times \mathbb{Z}_2^v \to \mathbb{Z}_2^\infty,$$

where k is the length of the key and v is the length of the IV. The codomain of such a stream cipher is the set of infinite bitstreams; in practice, the output is truncated to provide just enough keystream material to encrypt a given message M. 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. Notice again how the overall structure of the algorithm remains the same as it was for its original application of hashing.

3.1.3 Generic Security

It is typical to employ Kerckhoffs' principle when discussion the security of a cryptosystem. This principle states that a cryptosystem should be secure regardless of any knowledge the adversary may have about the system (excluding the key). Stated in a different way, the adversary is assumed to know every detail of the system except for the key [25]. Indeed, we even presume that an attacker has access to the cryptosystem. In the case of a sponge construction, the adversary knows f (and f^{-1} if f is a permutation). The easiest way for him or her to gain knowledge about f that may differentiate it from a random transformation is to simply make calls to it (and f^{-1} if a permutation) [24].

A *random oracle* model is often useful as a framework for security proofs. A random oracle (RO), which lies at the core of most sponge security proofs, is a theoretical ideal function

$$\mathbf{RO} \colon \mathbb{Z}_2^* \to \mathbb{Z}_2^\infty$$

such that every bit of the output, for every possible input, is chosen uniformly and independently. We state security proofs in the context of *indistinguishability* in the random oracle model. We suppose that an adversary is able to query both a random oracle and the pseudorandom function (PRF) that we are testing. The adversary then decides which of the systems is the PRF. If it is "hard" for the adversary to do this, then the PRF is called indistinguishable from random. This is what we desire. Hardness is given in terms of computational complexity and is subject to interpretation. For example, if the adversary is able to accurately distinguish the two systems after only 2^{40} queries, then the PRF is not indistinguishable from random since a complexity of 2^{40} is computable with today's technology. If, however, it takes a minimum of 2^{128} queries, then the system *is* indistinguishable from random since this is not computable (i.e. such a computation would not finish within any reasonable time frame) and will not be computable in the foreseeable future [25].

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 [24].

3.1.4 Security of Keyed Sponges

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.

In [26] it is proven that the *advantage* of distinguishing a keyed permutation-based sponge from a random oracle is

$$\max\left(1 - \exp\left(-\frac{\frac{M^2}{2} + 2MN}{2^c}\right), \frac{N}{2^{|K|}}\right),$$

where M is the data complexity, N is the time complexity, c is the capacity, and $|K|$ is the size of the key. The advantage is the probability of success of a generic attack.

We can take N as the number of queries to the permutation f and its inverse. The data complexity M is typically considered to be upper bounded by the implementation under a given key K. For example, it is very reasonable and common practice to enforce that no more than, say, $M = 2^{40}$ operations be performed under a given key. In [26] this exponent is called a *usage exponent* and is denoted as a. The assumption is made that $a \ll c/2$ and the requirement (as is typical) is that no attacks are faster than a brute force search on the keyspace. This ultimately leads to our metric of interest: a lower bound on the capacity c such that there are no generic attacks (that allow differentiation from a random oracle) that are faster than exhausitive search. This lower bound is given as

$$c \geq |K| + a + 1.$$

Jovanovic et. al [27] further improved on these results in 2014 by proving that the generic security level of keyed sponge constructions is lower bounded by

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

3.2 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". Other than this, switching from the sponge to the duplex construction is simply a matter of adjusting how inputs and outputs are handled. The duplex mode has several applications, with authenticated encryption being the one of obvious interest to us.

3.2.1 Duplex Parameters

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 **duplexing**:

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

Figure 3.2 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 **duplexing** for which no input is provided ($|\sigma_i| = 0$). A *mute call* is a call for which no output is requested ($\ell_i = 0$). The reasons for these types of calls will soon be apparent.

Simplified Duplex

In addition to the padding required for the sponge construction, the duplex construction also requires *domain separation*. This can be generally defined as a mechanism that allows differentiation
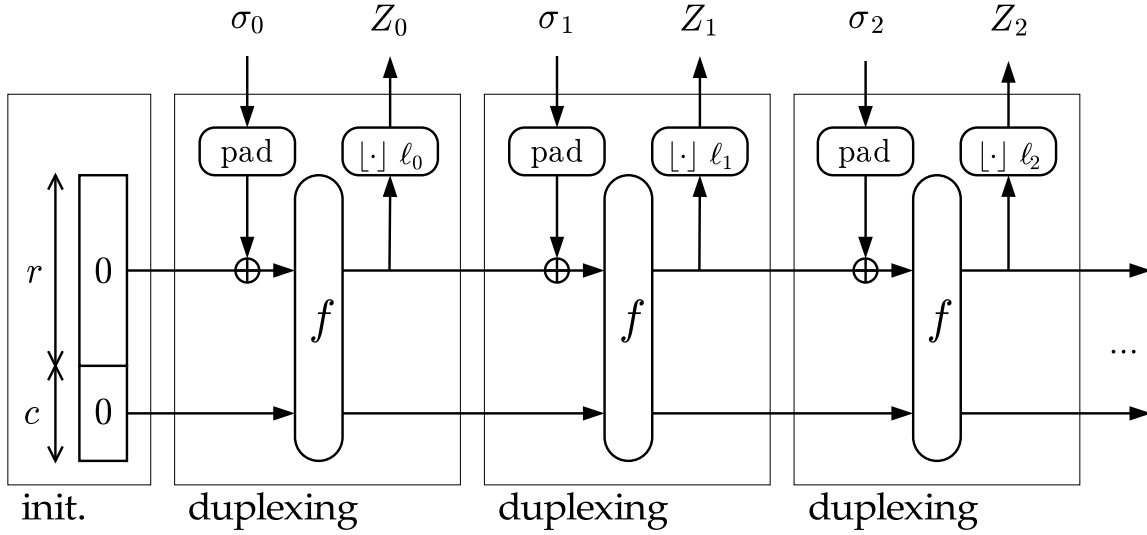
Figure 3.2: The duplex construction $\mathbf{duplex}[f, \mathbf{pad}, r]$ [24]

between varying types of inputs (e.g. between header and body data). In other words, it eliminates ambiguity on the receiving end that naturally arises from support for arbitrary length inputs. The KECCAK designers use a *frame bit* appended to the end of every input (see Figure 3.3) in their original definition of the duplex construction [28]. The value of this bit toggles for every input type so that the receiver can differentiate between outputs. Since the order of the inputs (and thus outputs) is always the same, a single frame bit is sufficient.

We also note that a frame bit is not the only way to achieve domain separation. In general one can append a *frame string*, denoted γ_i, to the end of every input. The only requirements are that $|\gamma_i| \geq 1$ and $\gamma_i \neq \gamma_{i+1}$ for all i. For example, each frame string could be a byte instead of a single bit.

For the simplified duplex construction, we can make the assumption that both the padding and domain separation are accomplished at some higher level if needed. This again allows us to focus more design effort towards the underlying sponge function.

3.2.2 Duplex for Authenticated Encryption

Authenticated encryption is easily achieved using the duplex construction. It can be modeled under a given key K as

$$\mathbf{AE}_K \colon \mathbb{Z}_2^k \times (\mathbb{Z}_2^*)^2 \to \mathbb{Z}_2^* \times \mathbb{Z}_2^\ell,$$

where k is the length of the key and ℓ is the length of the MAC desired.

Figure 3.3 shows the duplex construction being used in an AE use case. Frame bits are shown, but recall that these are ignored in the simplified duplex construction. 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 $D.\textbf{duplexing}$. 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 $D.\textbf{duplexing}$ after all header and body inputs have been processed.
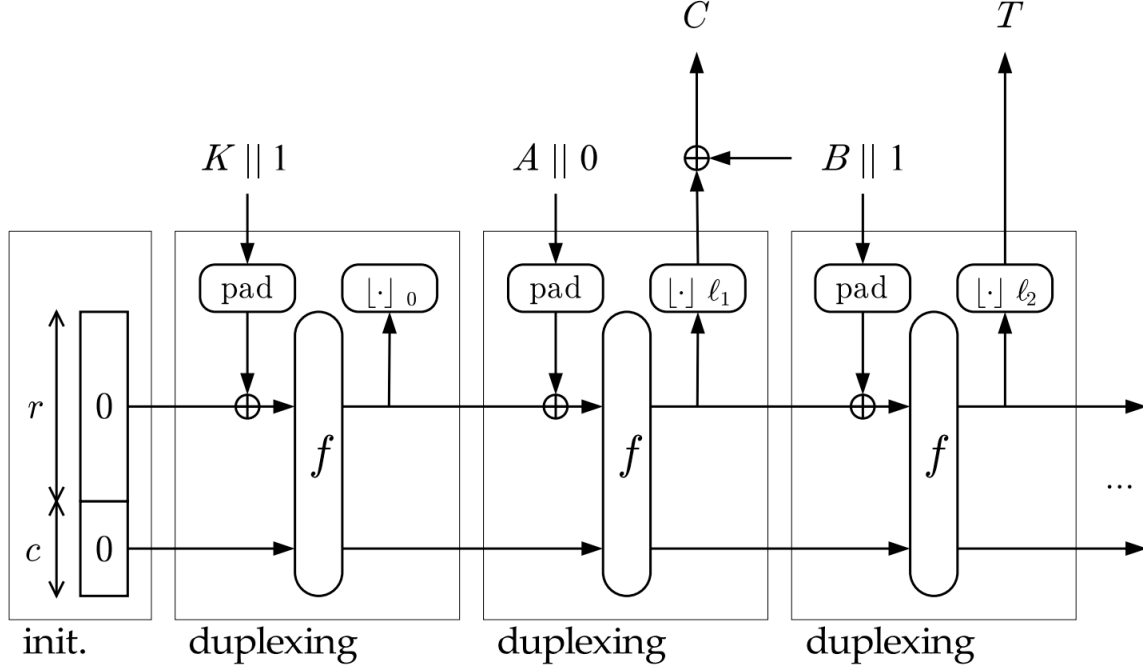
Figure 3.3: The duplex construction as used for authenticated encryption [29]

A more general case is shown in a slightly modified view in Figure 3.4. In this view, the duplex object D is shown as a block and the different header and body inputs (A_i and B_i respectively) are absorbed over a series of calls to $D.\textbf{duplexing}$. 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.
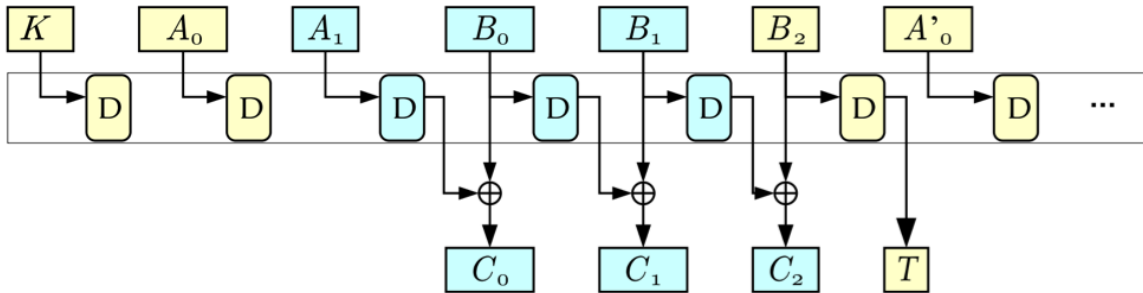
Figure 3.4: The duplex construction as used for authenticated encryption (general case) [29]

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

The main disadvantage to using the duplex construction for AE is that it is not easily parallelizable as given here. This should not be a concern most of the time since data (e.g. IP traffic) is often received and sent in a serial fashion for AE applications. Compared to other methods of achieving AE, the duplex construction is clearly superior in the majority of applications.

3.2.3 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 concrete example, consider the i-th duplexing call to a duplex object D:

$$Z_i = D.\textbf{duplexing}(\sigma_i, \ell_i)$$
$$= \textbf{sponge}(\textbf{pad}(\sigma_0) \,||\, \textbf{pad}(\sigma_1) \,||\, \ldots \,||\, \textbf{pad}(\sigma_i))$$

Or in the case of a simplified duplex object D, we have:

$$Z_i = D.\textbf{duplexing}(\sigma_i, \ell_i)$$
$$= \textbf{sponge}(\sigma_0 \,||\, \sigma_1 \,||\, \ldots \,||\, \sigma_i),$$

where σ_i may be key, header, or body material. Since this is true in general, it is clear that the duplex reduces to the sponge. For a more rigorous proof, we refer to [28].

Chapter 4

Algorithm Specification

Our authenticated encryption algorithm is based on the 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.

For Known Answer Tests (KATs) corresponding to this specification, see Appendix D.

4.1 Duplex Parameters

We allow two key sizes: 128 bits and 256 bits. These are the NIST recommended symmetric key sizes as of 2011 [30]. 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 Chapter 3, we know from [27] that the generic security level is

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

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

4.2 Permutation f

Our underlying sponge function f is a permutation, so it has the advantage of being fully entropy-preserving. Since it is bijective, f^{-1} exists by definition. We specify both f and f^{-1} here. While f^{-1} is never used in practice, it may be useful for cryptanalysis and verification purposes.

The permutation consists of a number of rounds. Figure 4.1 shows a diagram of a single round of the forward permutation. Figure 4.2 shows a single inverse round. 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. These are represented respectively as **S**, π, **M**, and $\oplus$ in the diagrams.

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 most substitution-based symmetric key algorithms [25].
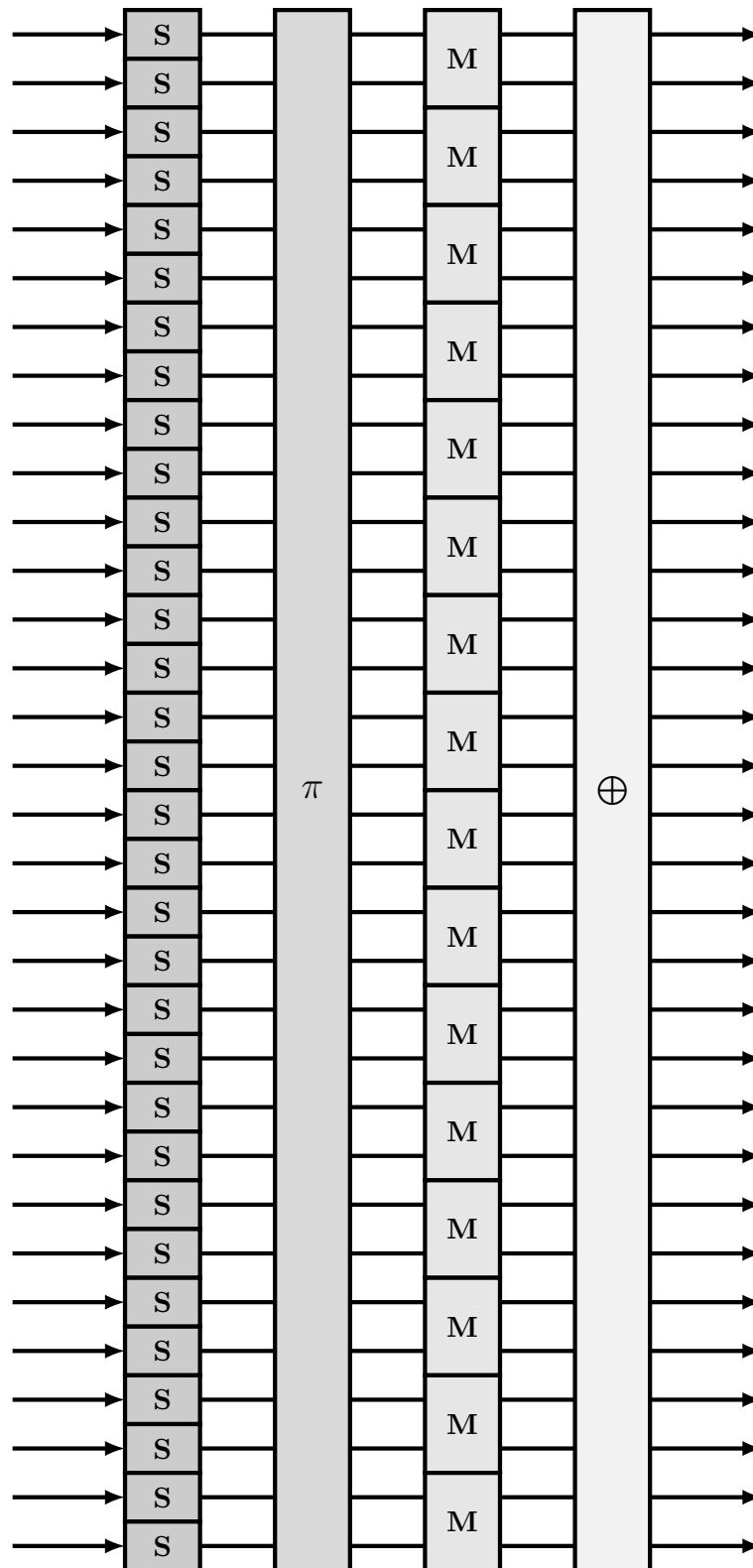
Figure 4.1: A single round of the sponge permutation f. Each line represents a 16-bit word.
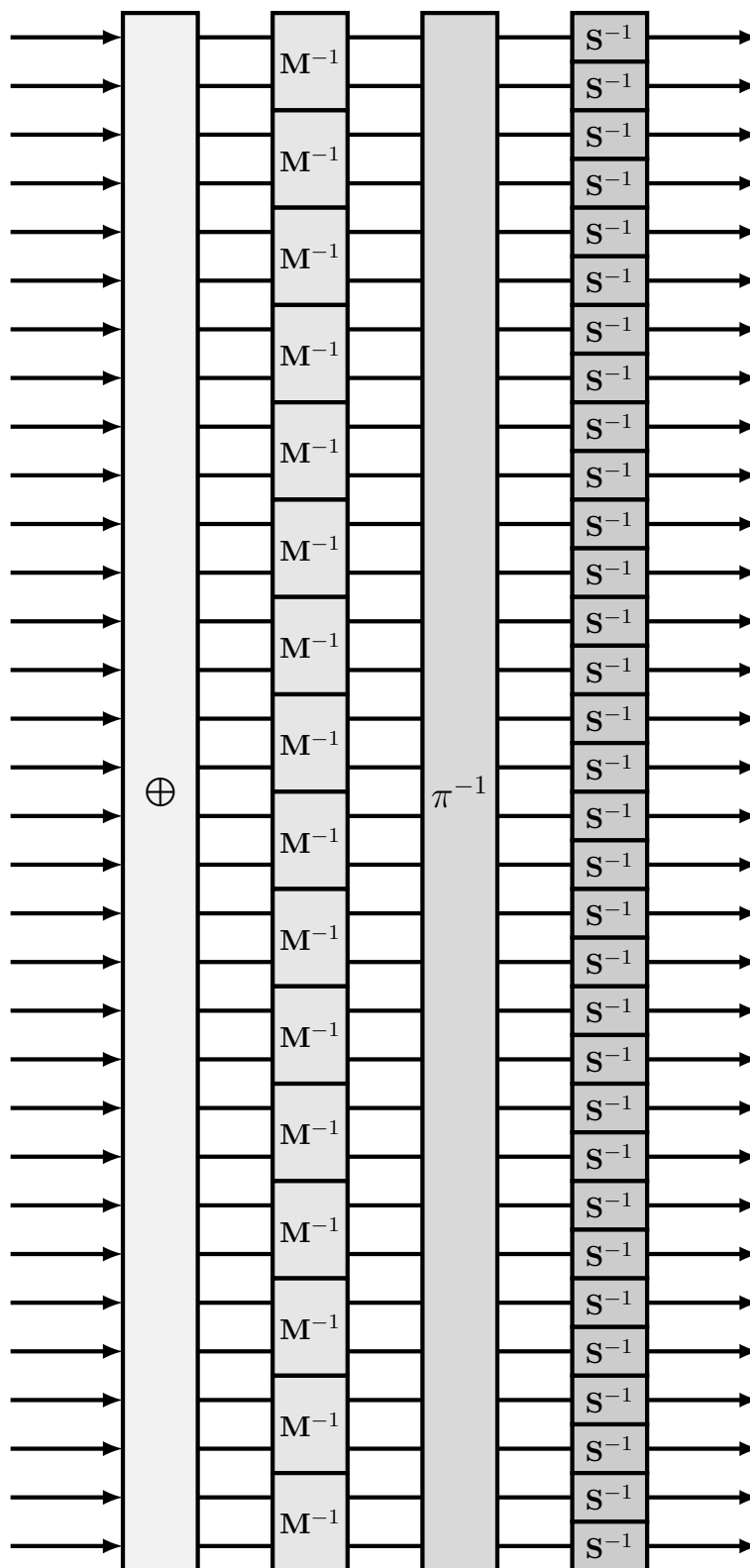
Figure 4.2: An inverse round of the permutation f. Each line represents a 16-bit word.

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) [31][32].

Our S-box is an AES-inspired design taken directly from Wood's thesis on the subject [33]. 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 an actual circuit which performs the operations rather than use the corresponding (and prohibitively large) look-up table.

Specifically, we use the reference C implementation provided in Appendix C of the aforementioned thesis. 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

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

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

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

and the inverse is

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

A hardware implementation for this particular S-box requires just 1238 XOR gates and 144 AND gates. However, No concrete implementation (e.g. in VHDL) is provided; this is an area for future work.

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 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.

We denote the bitwise permutation function

$$\pi \colon \mathbb{Z}_{512} \to \mathbb{Z}_{512},$$

where it operates on the *index* of a given bit, $x \in \mathbb{Z}_{512}$.

To understand our specific choice of π, it is useful to consider a poor choice first. For this we can turn our attention to the lightweight block cipher PRESENT which operates on a 64-bit state over 31 rounds. PRESENT uses the bitwise permutation

$$\pi_P \colon \mathbb{Z}_{63} \to \mathbb{Z}_{63}$$

given by the linear function

$$\pi_P(x) = 16x \bmod 63.$$

Since it operates in Z_{63}, an augmented mapping $\pi_P(63) = 63$ is required for the last bit [34].

The particular structure of π_P led to an attack on PRESENT in 2009 [35]. The attack leverages the following undesirable properties of π_P:

1. There are four fixed points: $x = 0, 21, 41, 63$

2. The order is only three: $\pi_P^3(x) = \pi_P(x)$

The combination of these properties results in a decrease in the lower bound on the number of active S-boxes across four rounds since it is possible to construct a trail that branches back into itself after π_P is iterated only three times. See Figure A.1 for a diagram that illustrates this trail.

A simple way to avoid fixed points is to use an affine function rather than a linear function. An affine function over $\mathbb{Z}_{512}$ is of the form

$$\pi(x) = \alpha x + \beta \bmod 512.$$

For π to be bijective, we require $\gcd(\alpha, 512) = 1$. Since the prime factorization of 512 is simply 2^9, it is equivalent to say that α must be odd [25].

A low order bit is defined as a bit that has order less than the order of the overall bitwise permutation. It is cumbersome to mathematically characterize all permutations that satisfy the property that no bits have low order. Instead, we wrote a script (see Appendix E) to search for such permutations. The script also identifies if a given permutation satisfies all other properties that we require. We found 384 permutations defined by affine functions over Z_{512} that satisfy all properties. A complete listing is provided in Table B.1.

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 particular bitwise permutation has order 32 and its inverse is given by

$$\pi^{-1}(x) = 479(x - 15) \bmod 512$$

since $31^{-1} \equiv 479$ in $\mathbb{Z}_{512}$.

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. See Chapter 5 for more detail on branch numbers. 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 mix step is invertible because the matrix is invertible; its inverse is given by

$$\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} a & b \\ b & c \end{pmatrix} \begin{pmatrix} A' \\ B' \end{pmatrix}$$

where

$$a = x^{15} + x^{14} + x^{12} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x + 1$$
$$b = x^{14} + x^{13} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^3 + 1$$
$$c = x^{15} + x^{13} + x^{12} + x^{10} + x^9 + x^7 + x^6 + x^3 + x.$$

This can be verified by using the provided $GF(2^{16})$ C library. 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. Notice that the outputs A' and B' can be written as

$$A' = A \oplus Bx$$
$$B' = Ax \oplus Bx \oplus B$$

since addition in $GF(2^{16})$ is simply the XOR operation. Figure 4.3 shows how this matrix multiplication is implemented.
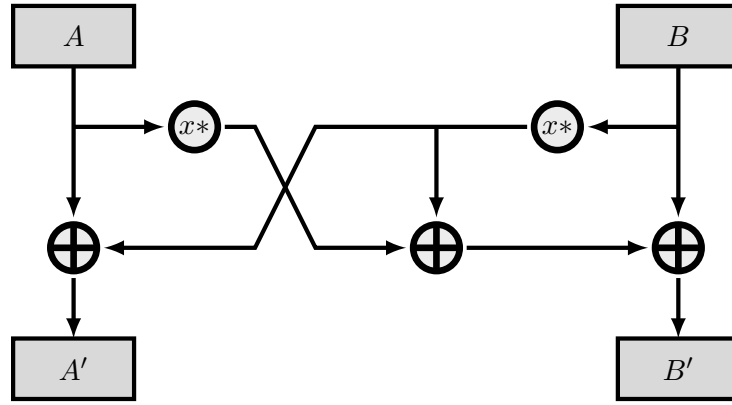
Figure 4.3: Hardware implementation of the forward mixer function.

The $x*$ operation is a multiplication by x in $GF(2^{16})$. Its implementation, which is shown in Figure 4.4, is very simple. Notice that a multiplication by x is simply a left rotation followed by a reduction if the MSB was one. The reduction is derived from the fact that

$$x^{16} \equiv x^5 + x^3 + x^2 + 1 \bmod p(x)$$

for our particular irreducible polynomial $p(x)$, and it is implementable using just three XOR gates.

4.2.4   Add Round Constant Step

The add round constant step is the simplest by far, and it is its own inverse. A constant 512-bit value is added 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

$$RC_i = \textbf{SHA3-512}(\textbf{ASCII}(i)),$$
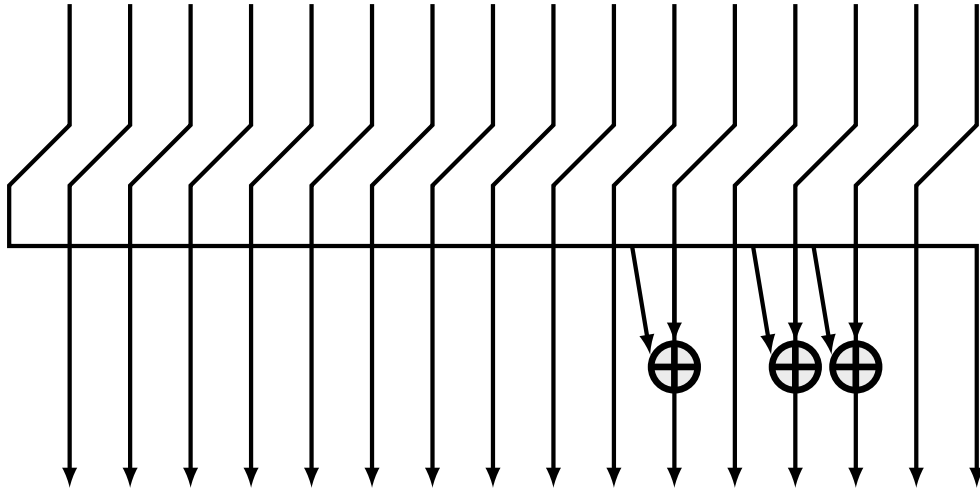
Figure 4.4: Hardware implementation of the $x*$ function. The leftmost bit is the MSB.

where $\textbf{ASCII}(i)$ is a function that provides the one or two byte ASCII representation of i and **SHA3-512** is the SHA-3 hash function that outputs a 512-bit message digest. Table 4.1 provides the values of RC_i up to $i = 16$.

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 Chapter 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.

Constant	Hex Value
RC_1	00197a4f5f1ff8c356a78f6921b5a6bfbf71df8dbd313fbc5095a55de756bfa1 ea7240695005149294f2a2e419ae251fe2f7dbb67c3bb647c2ac1be05eec7ef9
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 4.1: Round constants for up to 16 rounds

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 [33]. 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.

4.4.3 Bitwise Permutations

The bitwise permutation provided in the specification is subject to the constraints explained before. There are many permutations that satisfy these constraints. We chose to use a permutation that is easily definable by an affine function as obtained by our script; this is not a requirement. A user could generate their own bitwise permutation subject to the given constraints via exhaustive search.

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 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; see Chapter 5)

3. The transformation should be efficiently implementable in hardware

Note that the inverse transformation need not be efficiently implementable. Like the addition of a new S-box, any new mixer (i.e. matrix) introduced to the algorithm should be analyzed to ensure it meets these constraints. A transformation defined by 2×2 matrix multiplication that does meet the second requirement, for example, would lower our security margin and possibly require increasing the number of rounds significantly.

It should be noted that before settling on a mixer based on matrix multiplication, many mixers based on modular addition, XOR, and rotation operations were tested. Systems or operations that rely mainly on these operations are typically called *ARX-based*. None of the ARX-based mixers we tested met our requirement of increasing the branch number from two to three. However, for completeness, we enumerate all of the mixers examined in this work in Appendix C.

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.

There are also other ways of injecting asymmetry into the algorithm using constants. For example, the round constants could actually be rotation constants that define the amount to rotate each word on the inputs to mixers.

Chapter 5

Cryptanalysis

We have already discussed the security of this construction against generic attacks in Chapter 3. The only requirement remaining is to assess the security of the underlying sponge permutation f.

We present a survey of potentially relevant cryptanalysis techniques here. Focus is placed on differential and linear cryptanalysis since these techniques are so general, powerful, and prevalent. Resistance against these techniques should result in resistance against many other less general techniques. There is far too much to be said about various cryptanalysis techniques, and there are far too many to discuss here. 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, [36]. 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 [37].

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. Note that in an actual attack, high

probability 1-bit-to-1-bit differentials are of most interest since these will likely be the easiest to propagate through the overall system.

A *differential trail* or *characteristic* is the propagation of non-zero differentials throughout a system (i.e. across rounds). Figure A.1 shows an example trail through four rounds of PRESENT. 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. For example, if Figure A.1 shows a differential trail, then there are six differentially active S-boxes.

Proving resistance against differential cryptanalysis for a substitution-permutation network requires coming up with 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 [38][31].

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` written with Kaminsky's Parallel Java 2 library [39][40].

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 [41]. 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 [39]. The Boolean satisfiability problem is known to be NP-complete [42]; however, our CNFs were small enough that it finished within a minute for all cases tried (see Appendix C for failed mixer designs).

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 matrix multiplication-based 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 5.1, 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 5.1: 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, [43]. 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.

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

where $X_i \in \mathbb{Z}_2$. Favoring typical convention found in the literature, X_1 is the MSB [37]. This notation allows us to easily represent linear approximations in the form

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

or equivalently

$$\left(\bigoplus_{i=1}^{16} X_i \right) \oplus \left(\bigoplus_{j=1}^{16} Y_j \right) = 0.$$

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.

S-boxes that are involved in a linear approximation of a system are called *linearly active* S-boxes. A linear approximation across rounds is called a *linear trail*; it involves several linearly active S-boxes. Our goal, as with differential cryptanalysis, consists of trying to maximize the minimum number of linearly active S-boxes. 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 [31], 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 [43]:

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

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 5.2.

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.

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 5.2: Worst case linear biases and linear attack complexities over increasing rounds

5.3 Differential and Linear Cryptanalysis Variants

There have been numerous efforts to extend on the foundations of differential and linear cryptanalysis over the years in order to produce attacks that are potentially more powerful. We present a brief overview of the some of the most prevalent variants here.

5.3.1 Differential-Linear Cryptanalysis

As the name suggests, differential-linear cryptanalysis is a combination of differential and linear cryptanalysis in which a differential trail is followed by a linear trail. It was introduced in 1994 by Langford and Hellman [44], who demonstrated a differential-linear attack on 8-round DES that requires far fewer plaintexts than previous attacks. The attack is based on a differential trail over the first three rounds that holds with probability 1 followed by a high magnitude bias linear approximation for the following rounds. While it was a requirement for this attack to use a unity probability differential trail over the initial rounds, Biham et al. [45] were able to generalize differential-linear attacks in 2002 to differential trails with probability less than 1. Recall from Chapter 1 that these results were used in 2007 by Wu and Praneel [16] to break the stream cipher Phelix, which had promising applications to authenticated encryption prior to this.

   We believe that any differential-linear attack on our permutation f would be limited to a few rounds at best due to the extremely low maximum differential probability of our S-box. For example, our absolute worst case differential probability over the first two rounds is bounded by 2^{-57}. Even without considering the linear approximation that must follow this, the complexity of such an attack is already approaching the limits of feasibility after these two rounds.

5.3.2 Truncated and Higher-Order Differentials

Both truncated and higher-order differential attacks were introduced in Knudsen's 1995 paper on the subject [46]. Knudsen shows that there exist systems which are provably secure against regular differential attacks yet susceptible to truncated or higher-order differential attacks.

   A differential in which only some of the bits are known is called a *truncated differential*. The idea is that because fewer bits are being estimated, the propagation should be easier. However, the propagation of truncated differentials relies largely on *strong alignment* of the system. For a complete treatment of this, we refer to the KECCAK team's excellent coverage of the subject in [47]. We note only that a particular conclusion is that bit-oriented transformations have weak alignment,

thus truncated differentials become extremely difficult to propagate. Because of the effect of our bitwise permutation and the very small differential probabilities of our S-box, we believe that any attack based on truncated differentials would not exceed a few rounds of our underlying permutation at worst.

A *higher-order differential* is a differential that consists of several differences. It could be thought of as a difference of differences. One of the conclusions of [46] is that a higher-order differential attack will only be effective against a system of sufficiently low algebraic degree. This is why, for example, such attacks have yet to be effective against AES. Because of this, as we will elaborate on in the next section, we believe that our system will be resistant to higher-order differential attacks.

5.4   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. [48] 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. We discuss a few potentially relevant algebraic attacks here that follow a similar idea.

5.4.1   XL and XSL Attacks

There has been some ongoing work related to reducing the cryptanalysis of cryptosystems to the solution of multivariate quadratic (MQ) equations. The first attempted attack to leverage this work, called the eXtended Linearization (XL) attack, was introduced by Courtois et al. in 2000 [49]. This attack works by transforming a system of MQ equations into a much larger overdefined system of linear equations. The idea is that while we are bad at solving nonlinear systems, linear systems are quite easy for us to solve. However, XL is still an impractical attack because after linearization the system of equations is simply too large to solve.

In 2002, Courtois et al. [50] introduced the eXtended Sparse Linearization (XSL) attack as an attempt to improve on XL. This attack leverages the fact that the complexity of XL drops significantly if the MQ system is sparse and regularly structured in addition to being overdefined. The authors shows that for Serpent (an AES candidate) and AES itself, the MQ system does end up satisfying these properties. Still, no practical attack has risen out of these efforts due to the fact that the complexity of such an attack remains computationally infeasible.

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 XSL 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 [33]) leads us to conjecture that our permutation would still be resistant.

5.4.2 Cube Attacks

Cube attacks were fairly recently introduced by Dinur and Shamir in 2008 [51]. This cryptanalysis method is intriguing because it can be applied to cryptosystems which are treated as a black box; that is, the internal structure of the system does not need to be known. For an attack to work, at least one output bit must be able to be represented by a polynomial of relatively low degree. Cube attacks have had minor success. For example, Lathrop demonstrated a successful attack on four rounds of a 224-bit hash variant of KECCAK in his 2009 thesis on the subject [52]. He estimates that this attack could be practical up to seven rounds.

Given that KECCAK is also based on the sponge construction, it may be fair to question whether cube attacks are applicable to our algorithm. However, KECCAK is unique in the sense that its round function has a very low degree of only two. Systems like AES which use sufficiently large S-boxes, as ours does, do not generally have the property that output bits can be represented as polynomials of small degree. In fact, the degrees of any such polynomials are typically extremely high [53]. Therefore we conclude that our algorithm is immune to cube attacks.

5.5 Other Cryptanalysis

There are several other attacks that are very dissimilar from both differential and linear attacks as well as algebraic attacks. We present the two that we believe to be the most prevalent and relevant in this section.

5.5.1 Slide Attacks

Slide attacks were introduced in 1999 by Biryukov and Wagner [54]. A slide attack has the interesting property that it is independent of the number of rounds of the system. Therefore the common method of having a high number of simple rounds to provide security is no longer valid if proper care is not taken. These attacks work by exploiting the *self-similarity* of a system with respect to the behavior of its rounds. For example, a block cipher with a periodic key schedule is at risk because the groups of rounds belonging to a period of the key schedule can be abstractly considered as "larger rounds" within the slide attack methodology. This similarity is then exploited.

One of the key advantages that we repeatedly mention in this thesis is that the sponge and duplex constructions do not require a key schedule since keys are treated the same as any other input data. Therefore, we do not even have to consider any weaknesses that may be presented by poor subkey generation. For unkeyed permutation design, one of the simplest methods to prevent against slide attacks is to XOR round constants into the state every round. This disrupts self-similarity across rounds as long as the round constants never repeat, which ours do not. In addition, our round constants are completely independent of each other and no relation can be drawn between them except for the fact that they are all generated by the SHA-3 hashing function. These properties easily lead to the conclusion that our permutation f is not susceptible to slide attacks.

5.5.2 Integral Attacks

Integral cryptanalysis was initially presented in 1997 by Daemen et al. [55] as a dedicated attack on the block cipher Square (interestingly, in the same paper as the initial specification of Square). As

such, it is sometimes called the Square attack. However, it has since been extended to attacks on other systems.

Integral cryptanalysis analyzes the propagation of special sets (or multisets) of chosen plaintexts. In the basic version, all of the bitstrings belonging to a set are related by the property that most of the bits are held constant while a contiguous block is varied through all 2^n possibilities, where n is the length of the block. For example, if $n = 8$ (a byte), then a set will contain $2^8 = 256$ unique bitstrings that iterate through all possibilities of values for that byte. These bitstrings necessarily have the property that the bitwise XOR sum of the varied bytes will be equal to zero. It is the behavior of this sum that is analyzed as it propagates throughout the system, and any significant non-random behavior can lead to an attack.

While the classical form of this attack has even been extended to AES with some success [56], we argue that our bitwise permutation step eliminates any chance of success against our underlying f function. This technique will clearly break down for systems that are not entirely block-based, so our hybrid design approach of using both bit-oriented and word-oriented operations excels here. Indeed, this is the same argument the KECCAK team initially makes in [22].

Z'aba et. al generalized integral attacks in 2008 to bit-oriented systems with some success [57]. While arguably not as powerful as the basic block-oriented form of the attack, it has, for example, broken up to five rounds of PRESENT with only 80 chosen plaintexts compared to the 2^{20} required for a differential attack. However, they note at the conclusion of their paper that bit-oriented integral attacks simply do not extend, even with low probability, past the first few rounds of such systems. This is in great contrast to differential and linear cryptanalysis, which always extend but with greatly reduced probability. The exact resistance of our algorithm to this new bit-oriented integral attack would have to be studied more in depth, but given the aforementioned results, we believe that it is highly unlikely that an integral attack would extend past the initial rounds of our permutation.

Chapter 6

Statistical Testing

In addition to the previously described cryptanalysis techniques, it can be insightful to analyze the output of a cryptosystem purely on a statistical basis. Two methods of statistical testing were explored here: using the NIST Statistical Tool Suite for randomness testing, and creating our own tool to check that we meet the avalanche criterion. Both of these methods help provide evidence that the system behaves randomly. However, they are certainly not a replacement for the previously described cryptanalysis techniques.

6.1 NIST STS Testing

6.1.1 Overview

NIST created and maintains the Statistical Tool Suite (STS) for randomness testing of binary data [58]. While its primary purpose is for testing random and pseudorandom number generators, it can be applied to general symmetric key cryptosystem testing if proper measures are taken. For example, Soto (in affiliation with NIST) used an earlier version of the STS in 1999 to analyze the AES candidates [59].

The STS requires very long bitstrings as input in order to ensure valid results. The NIST recommendation is at least 55 bitstrings each of length 2^{20} bits (1 Mb) for most tests in the suite. Because block ciphers such as the AES candidates have small, fixed length outputs, Soto had to concatenate many "derived" output blocks together to form one input bitstring for the STS. The derived blocks come from using the block cipher, for example, in various modes of operation such as CBC. The validity of this method is debatable since the STS ends up testing the mode of operation of the block cipher rather than the block cipher itself. A better statistical test for block ciphers and other symmetric key cryptosystems with fixed length outputs would be the coincidence test based on a Bayesian approach as described by Kaminsky in [60].

For this reason, we chose a different method to generate STS inputs. Because the STS is intended for random and pseudorandom number generators, we used the sponge construction in the keystream generation mode detailed in Section 3.1.2. We absorb a 128-bit key (with no IV, for simplicity) and squeeze out 1 Mb of keystream material. This is very similar to a pseudorandom number generator because a good keystream generator will behave as a cryptographically secure pseudorandom bitstream generator. While this does not directly test the duplex construction, we remind the reader that the duplex construction can be reduced to the sponge construction as shown in Section 3.2.3. Therefore we believe that this is the best method to use the NIST STS for statistical testing of our cryptosystem.

There is one input file generated for every number of rounds up to the maximum of 16. Each input file consists of 128 bitstrings, each of which is acquired by flipping a unique bit of the key and squeezing as previously described.

6.1.2 STS Test Descriptions

The NIST STS runs a battery of 15 different tests on an input file in an attempt to discover any non-random behavior. We very briefly describe those tests here. Interested readers may refer to [58] for a much more in-depth description of each test.

1. **Frequency:** Determines whether the number of ones and zeros in a bitstring is approximately the same as it would be for a truly random bitstring. All subsequent tests rely on the passing of this test.

2. **Block Frequency:** Determines whether the number of ones and zeros in M-bit blocks of a bitstring is approximately the same as it would be for a truly random bitstring. We use the default recommended value for our input size: $M = 128$.

3. **Runs:** Determines whether the number of runs of ones and zeros of various length is approximately the same as it would be for a truly random bitstring. A run is an uninterrupted sequence of identical bits.

4. **Longest Run of Ones in a Block:** Determines whether the longest run of ones in M-bit blocks of a bitstring is approximately the same as it would be for a truly random bitstring. (A deviation from random for longest run of ones also indicates a deviation for longest run of zeros, thus only one test is needed.) The preset value for M is 10^4 for bitstrings of length greater than $750,000$; this is what we use.

5. **Binary Matrix Rank:** Determines whether the linear independence (i.e. rank) of disjoint $M \times Q$ matrices built from an input bitstring is approximately the same as it would be for a truly random bitstring. The preset values $M = Q = 32$ are used.

6. **Discrete Fourier Transform (Spectral):** Determines whether the spectral frequency of a bitstring is approximately the same as it would be for a truly random bitstring.

7. **Non-Overlapping Template Matching:** Determines whether the number of occurrences of an m-bit aperiodic pattern within a bitstring is approximately the same as it would be for a truly random bitstring. Patterns are not allowed to overlap. We use the default recommended value for our input size: $m = 9$. This test is actually comprised of 148 separate tests for different preset patterns.

8. **Overlapping Template Matching:** Determines whether the number of occurrences of an m-bit all-one pattern within a bitstring is approximately the same as it would be for a truly random bitstring. The pattern may overlap. We use the default recommended value for our input size: $m = 9$.

9. **Maurer's "Universal Statistical" Test:** Determines whether the achievable level of compression of a bitstring is approximately the same as it would be for a truly random bitstring. A bitstring that can be compressed significantly is considered to exhibit non-random behavior.

10. **Linear Complexity:** Determines whether the length of a Linear Feedback Shift Register (LFSR) that adequately models M-bit blocks in a bitstring is approximately the same as it would be for a truly random bitstring. A block that can be adequately modeled by a small LFSR is considered to exhibit non-random behavior. We use the default recommended value for our input size: $M = 500$.

11. **Serial:** Determines whether the frequency of *all* possible overlapping m-bit patterns is appoximately the same as it would be for a truly random bitstring. Note that for $m = 1$, this is equivalent to the previously described Frequency test. We use the default recommended value for our input size: $m = 16$.

12. **Approximate Entropy:** Determines whether the frequency of all possible m- and $(m+1)$-bit patterns is approximately the same as it would be for a truly random bitstring. We use the default recommended value for our input size: $m = 10$.

13. **Cumulative Sums (Cusums):** Determines whether the (forward and reverse) cumulative sums of a bitstring are approximately the same as they would be for a truly random bitstring. A cumulative sum that is too large or small indicates too many ones or zeros in the early (for forward) or late (for reverse) stages of the bitstring.

14. **Random Excursions:** Determines whether the distribution of the number of visits of a one-dimensional random walk to a certain state is approximately the same as it would be for a truly random bitstring. This test is actually comprised of eight separate tests, one for each of the states in $[-4, -1] \cup [+1, +4]$.

15. **Random Excursions Variant:** Determines whether the distribution of the number of visits across many one-dimensional random walks to a certain state is approximately the same as it would be for a truly random bitstring. This test is actually comprised of 18 separate tests, one for each of the states in $[-9, -1] \cup [+1, +9]$.

To interpret the results of each test, one needs to understand some very basic probability theory. The *null hypothesis* is the hypothesis that there exists no statistical significance in a given set of observations. We *accept* or *reject* the null hypothesis based on the results of each test on each bitstring. For our purposes, accepting the null hypothesis means that the bitstream in question exhibits no significant non-random behavior; this is what we desire. Each STS test outputs a *P-value* (between 0 and 1) that summarizes the strength of the evidence against the null hypothesis. It indicates the probability that a perfectly random bitstring would have produced a sequence less random than the one under test. For example, if a test returns a *P-value* of 1, then the bitstring under test exhibits perfect randomness.

In order to translate this to a pass/fail test, we must choose a *significance level*, denoted α, such that if *P-value* $\geq \alpha$ then the null hypothesis is accepted. We use the default NIST-recommended value of $\alpha = .01$ here. This means that an acceptance of the null hypothesis has a 99% confidence level. In other words, we expect that one out of every 100 truly random bitstrings is falsely rejected.

6.1.3   Results

NIST recommends two approaches for interpreting the results of a statistical test for all bitstrings in an input file. The first is based on the proportion of *P-values* which pass (i.e. for which the null

hypothesis is accepted for that bitstring). The STS directly provides support for this. The second is based on the distribution of *P-values*. The STS provides some support for this, but further work is needed to come up with an overall pass/fail result for each test. For this, we wrote the script provided in Section E.3.

P-value Proportions

The first method of interpreting test results for an entire input file is based on the proportion of bitstrings that generate *P-values* which pass the test. Acceptable proportions are given by the confidence interval

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}},$$

where $\hat{p} = 1 - \alpha = 0.99$ and $m = 128$ is the sample size. In our case, the lower bound on the confidence interval is approximately 0.9636. If a proportion is below this, the test on that input file fails. The Random Excursions Variant test does not always use all $m = 128$ inputs it is given and thus the lower bound may be decreased in some cases. The details behind this are considered out of scope of this thesis. However, we do note that the minimum lower bound for this particular test occurs at approximately 0.9576 when $m = 66$ is used.

We present in Table 6.1 the pass/fail results for every number of rounds up to the maximum of 16 as provided directly by the NIST STS output. If a test failed, we indicate which test it was and the corresponding *P-value*. For tests which consist of numerous subtests, we list exactly which test failed. For example, a failed Non-Overlapping Template test includes the template number. A failed Random Excursion (Variant) test includes the state it failed on.

At first, it may seem alarming that statistical tests are failing at all. However, one must take into account that the proportion for a failed test at most exceeds the lower bound less than .03. Furthermore, there are no significantly *repeated* failures; that is, no specific test fails a high number of times and therefore raises a concern. While the Non-Overlapping Template test does fail more often than others, it represents the vast majority of the overall tests run and so this is expected. In addition, it does not repeatedly fail on a specific template.

We contrast these results with those found for the AES candidate HPC in [59]. This was the only AES candidate found to exhibit definitively non-random statistical behavior. HPC failed 23.96% of the time in multiple tests, which is far greater than our failure rate. In the same year it was discovered that HPC suffers from equivalent keys [61]. Our results are similar to the rest of the AES candidate algorithms, for which there are no real statistical anomalies.

P-value Distribution

It is also important to verify that the distribution of *P-values* for each test is uniform enough. Even if a test passed in the previous section, a uniformity failure on the *P-values* indicates non-random behavior. The uniformity test NIST recommends involves computing a χ^2 statistic with nine degrees of freedom on the "binned" *P-values* for each test. NIST provides minimal support for this by binning the *P-values* into 10 equally spaced intervals for each test. The χ^2 statistic is given by

$$\chi^2 = \sum_{i=1}^{10} \frac{(F_i - \frac{s}{10})^2}{\frac{s}{10}},$$

# Rounds	# Tests Passed	# Tests Failed	Failed Tests & Proportions
1	188	1	Spectral (0.9609)
2	188	1	Random Excursions Variant [+3] (0.9529)
3	187	2	Non-Overlapping Template [#58] (0.9531) Random Excursions Variant [+2] (0.9529)
4	187	2	Non-Overlapping Template [#54] (0.9609) Overlapping Template (0.9609)
5	187	2	Non-Overlapping Template [#32] (0.9609) Non-Overlapping Template [#86] (0.9609)
6	187	2	Non-Overlapping Template [#4] (0.9531) Non-Overlapping Template [#100] (0.9609)
7	186	3	Non-Overlapping Template [#22] (0.9609) Non-Overlapping Template [#68] (0.9609) Random Excursions [+4] (0.9420)
8	187	2	Non-Overlapping Template [#126] (0.9453) Non-Overlapping Template [#137] (0.9609)
9	187	2	Non-Overlapping Template [#12] (0.9609) Random Excursions [-1] (0.9467)
10	188	1	Non-Overlapping Template [#103] (0.9609)
11	187	2	Non-Overlapping Template [#45] (0.9453) Non-Overlapping Template [#109] (0.9609)
12	186	3	Spectral (0.9609) Non-Overlapping Template [#104] (0.9609) Non-Overlapping Template [#110] (0.9609)
13	189	0	N/A
14	188	1	Non-Overlapping Template [#79] (0.9531)
15	189	0	N/A
16	186	3	Non-Overlapping Template [#81] (0.9609) Non-Overlapping Template [#102] (0.9609) Non-Overlapping Template [#130] (0.9531)

Table 6.1: NIST Statistical Test Suite results

where F_i is the number of *P-values* in bin i and s is the sample size. A new valued called *P-value*$_\mathrm{T}$ is then calculated as follows:

$$P\text{-}value_\mathrm{T} = \mathbf{igamc}(\frac{9}{2}, \frac{\chi^2}{2})$$

where **igamc** is the incomplete gamma function given by

$$Q(a, x) = \frac{1}{\Gamma(a)} \int\limits_{x}^{\infty} e^{-t} t^{a-1} dt$$

and the gamma function Γ is

$$\Gamma(z) = \int\limits_{0}^{\infty} t^{z-1} e^{-t} dt.$$

The resulting *P-value*$_\mathrm{T}$ is compared to a new significance level $\alpha = .0001$ as recommended by NIST. If *P-value*$_\mathrm{T} \geq \alpha$ then the uniformity test passes.

Our Python script uses the SciPy library [62] to help compute *P-value*$_\mathrm{T}$ for every test and for every number of rounds. There were zero uniformity failures for each number of rounds tested up to the maximum of 16.

6.2 Avalanche Testing

The *avalanche criterion* (AC) is stated as follows: when a single bit of the input changes, on average half of the output bits should change. For our construction, it makes most sense to test for the AC at the permutation level. In particular, we are interested in how many rounds it takes for the AC to be satisfied. At this number of rounds, the permutation f is said to achieve *full diffusion*. Intuitively, it is good practice to maximize the number of full diffusions achieved within the number of rounds used for the algorithm.

As part of our permutation test code given in Section E.1.5, we provide a function for computing the average Hamming distance (i.e. number of bits changed) between the input and output for every number of rounds up to the maximum of 16. This is done by, for each number of rounds, consecutively flipping one bit at a time of the initial state and running the permutation, keeping track of Hamming distances between the inputs and outputs along the way. The minimum and maximum Hamming distances are also computed for completeness. The results of this experiment are shown in Table 6.2.

From this table, it is clear that full diffusion is achieved over the entire 512-bit state after just three rounds of the underlying permutation. This means that for a 128-bit key with 10 rounds, three full diffusions are achieved. For a 256-bit key with 16 rounds, five full diffusions are achieved.

As with the NIST STS results, this does not at all imply resistance against powerful attacks such as linear and differential cryptanalysis. In fact, linear and differential cryptanalysis have been shown to be very effective against cryptographic primitives which satisfy the AC [63]. We present these results merely as further proof of the excellent diffusive capabilities of our bitwise permutation and mixing steps.

# Rounds	Average HD	Min HD	Max HD
1	21.94	10	36
2	180.29	113	242
3	256.28	223	283
4	256.10	226	287
5	256.40	220	289
6	256.55	227	293
7	255.50	211	287
8	255.65	220	289
9	256.20	219	289
10	256.05	219	295
11	256.87	218	287
12	255.39	224	291
13	255.18	218	289
14	256.82	224	288
15	256.04	223	290
16	255.41	221	286

Table 6.2: Avalanche test results for the permutation f

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we designed an authenticated encryption algorithm based on the duplex construction that we believe to be highly secure. This algorithm can be customized on a per-user or per-application basis by following the guidelines we presented as part of its specification. We believe that this is the only customizable authenticated encryption algorithm to date. Furthermore, we believe that this is the first cryptosystem to use a 16-bit S-box. We have shown that this large S-box, when combined with good diffusion, provides superior resistance against differential and linear cryptanalysis due to its small differential probabilities and linear biases. Because this S-box is based on bijective power mappings over a Galois field, it can be efficiently implemented in hardware (like the rest of our algorithm) and does not require a look-up table.

Following the algorithm specification, we presented a survey of state of the art cryptanalysis techniques and assessed our algorithm's resistance against them. Additionally, we provided a comprehensive statistical evaluation of the cryptosystem through the use of existing tools as well as our own. Our algorithm demonstrates no significant non-random statistical behavior and its underlying permutation achieves full diffusion on the entire 512-bit state after only three rounds.

7.2 Future Work

The scope of this thesis is quite wide. As a result, there are many items that we would love to research further but have so far been unable to. Several of these items are provided here.

1. **Cryptanalysis:** The duplex construction we use guarantees security against generic attacks, and our underlying permutation is designed for resistance against many powerful cryptanalysis techniques. However, as with all cryptosystems, further cryptanalysis is always welcome and encouraged. In particular, it is likely possible to prove better bounds for our resistance against differential and linear cryptanalysis. So far, we have proved only the absolute worst case. Furthermore, it could be interesting to dig deeper into the fairly new bit-oriented integral attacks. While we strongly believe our permutation to be secure against such an attack, it would be a worthwhile exercise to attempt to prove this resistance.

2. **Hardware Design:** Due to time constraints we have only provided a software model of the algorithm in this thesis. In practice, our algorithm is intended for hardware (e.g. FPGA) implementation. One of the natural next steps is to implement the construction in hardware and benchmark its performance.

3. **S-boxes:** The particular instantiation of our algorithm described in Chapter 4 uses a specific S-box based on bijective power mappings. We analyzed this S-box to determine its maximum differential probability and maximum linear bias. There are several other similar S-boxes provided in [33]. It would be helpful to also analyze these other S-boxes in order to determine if their statistical properties are much different. If the maximum probabilities and biases are very close to those of our particular S-box, these other S-boxes could be swapped in as a customization without the need for determining new security margins (and possibly adjusting the number of rounds). In addition, it would be extremely fruitful to find completely new cryptographically secure 16-bit S-boxes that are efficiently implementable in hardware.

4. **Large Mixers:** Part of the reason that we use a 2×2 matrix multiplication for our mixer is that it is extremely efficient to implement in hardware. It is well known that the maximum branch number of an $m \times m$ matrix is $m + 1$. We achieve the maximum differential and linear branch number of three with this mixer. Larger matrices are capable of achieving higher branch numbers, and so it would be interesting to further explore them. In particular, Maximum Distance Separable (MDS) matrices such as the one used in AES are capable of achieving the maximum branch number. However, the construction of large MDS matrices that are efficiently implementable is a difficult problem [38]. An in-depth analysis of the tradeoff between a large MDS matrix with fewer rounds and a small matrix with more rounds is welcome for future work.

5. **ARX-Based Mixers:** As part of the work that did not make it into the final AE algorithm presented here, we analyzed many ARX-based mixers. None of these mixers were able to achieve a branch number greater than two (the worst possible). So far it is unclear what an ARX-based mixer with branch number equal to three would look like, if it even exists. While we believe that our matrix multiplication-based mixer is an excellent choice, it would still be interesting to further analyze ARX-based mixers in an attempt to generalize their characteristics with respect to differential and linear cryptanalysis.

Bibliography

[1] N. Borisov, I. Goldberg, and D. Wagner, "Intercepting Mobile Communications: The Insecurity of 802.11," in *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pp. 180–189, ACM, 2001.

[2] E. Tews, R.-P. Weinmann, and A. Pyshkin, "Breaking 104 Bit WEP in Less Than 60 Seconds," in *Information Security Applications*, pp. 188–202, Springer, 2007.

[3] S. Vaudenay, "Security Flaws Induced by CBC Padding—Applications to SSL, IPSEC, WTLS...," in *Advances in Cryptology—EUROCRYPT 2002*, pp. 534–545, Springer, 2002.

[4] 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.

[5] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering*. Indianapolis, IN: Wiley Publishing, 2010.

[6] C. S. Jutla, "Encryption Modes with Almost Free Message Integrity," in *Advances in Cryptology —- EUROCRYPT 2001*, pp. 529–544, Springer, 2001.

[7] P. Rogaway, M. Bellare, J. Black, and T. Krovetz, "OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption," *ACM Transactions on Information and System Security (TISSEC)*, vol. 6, no. 3, pp. 365–403, 2003.

[8] D. Whiting, R. Housley, and N. Ferguson, "Counter with CBC-MAC (CCM)." IETF Request for Comments: 3610, 2003.

[9] P. Rogaway and D. Wagner, "A Critique of CCM," 2003. `http://cs.ucdavis.edu/~rogaway/papers/ccm.pdf`.

[10] T. Kohno, J. Viega, and D. Whiting, "CWC: A high-performance conventional authenticated encryption mode," in *Fast Software Encryption*, pp. 408–426, Springer, 2004.

[11] D. McGrew and J. Viega, "The Galois/Counter Mode of Operation (GCM)," 2004. `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf`.

[12] S. Gueron and M. E. Kounavis, *Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*, 2010. `http://intel.com/content/www/us/en/processors/carry-less-multiplication-instruction-in-gcm-mode-paper.html`.

[13] M. Bellare, P. Rogaway, and D. Wagner, "The EAX Mode of Operation," in *Fast Software Encryption*, pp. 389–407, Springer, 2004.

[14] D. Whiting, B. Schneier, S. Lucks, and F. Muller, "Helix: Fast Encryption and Authentication in a Single Cryptographic Primitive," *ECRYPT Stream Cipher Project Report*, vol. 27, no. 200, p. 5, 2005.

[15] D. Whiting, B. Schneier, S. Lucks, and F. Muller, "Phelix: Fast Encryption and Authentication in a Single Cryptographic Primitive," 2005. `http://schneier.com/paper-phelix.pdf`.

[16] H. Wu and B. Preneel, "Differential-Linear Attacks Against the Stream Cipher Phelix," in *Fast Software Encryption*, pp. 87–100, Springer, 2007.

[17] D. Bernstein, "CAESAR Submissions." `http://competitions.cr.yp.to/caesar-submissions.html`, 2014. Accessed: 2014-06-10.

[18] J. A. Gallian, *Contemporary Abstract Algebra*. Cengage Learning, 2nd ed., 2012.

[19] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, 1948.

[20] C. E. Shannon, "Communication Theory of Secrecy Systems," *Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[21] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley, 1996.

[22] 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`.

[23] 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`.

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

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

[26] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "On the Security of the Keyed Sponge Construction," 2011. `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/VANASSCHE_SpongeKeyed.pdf`.

[27] 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/`.

[28] 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.

[29] 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`.

[30] E. Barker and A. Roginsky, "Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths." NIST Special Publication 800-131A, January 2011. `http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf`.

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

[32] 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`.

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

[34] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," in *Cryptographic Hardware and Embedded Systems-CHES 2007*, pp. 450–466, Springer, 2007.

[35] J. Nakahara Jr, P. Sepehrdad, B. Zhang, and M. Wang, "Linear (Hull) and Algebraic Cryptanalysis of the Block Cipher PRESENT," in *Cryptology and Network Security*, pp. 58–75, Springer, 2009.

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

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

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

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

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

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

[42] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, 3rd ed., 2012.

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

[44] S. K. Langford and M. E. Hellman, "Differential-Linear Cryptanalysis," in *Advances in Cryptology—CRYPTO'94*, pp. 17–25, Springer, 1994.

[45] E. Biham, O. Dunkelman, and N. Keller, "Enhancing Differential-Linear Cryptanalysis," in *Advances in Cryptology—ASIACRYPT 2002*, pp. 254–266, Springer, 2002.

[46] L. R. Knudsen, "Truncated and Higher Order Differentials," in *Fast Software Encryption*, pp. 196–211, Springer, 1995.

[47] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "On alignment in KECCAK," May 2011. `http://keccak.noekeon.org/KeccakAlignment.pdf`.

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

[49] N. Courtois, A. Klimov, J. Patarin, and A. Shamir, "Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations," in *Advances in Cryptology—EUROCRYPT 2000*, pp. 392–407, Springer, 2000.

[50] N. T. Courtois and J. Pieprzyk, "Cryptanalysis of Block Ciphers with Overdefined Systems of Equations," in *Advances in Cryptology—ASIACRYPT 2002*, pp. 267–287, Springer, 2002.

[51] I. Dinur and A. Shamir, "Cube Attacks on Tweakable Black Box Polynomials," in *Advances in Cryptology-EUROCRYPT 2009*, pp. 278–299, Springer, 2009.

[52] J. Lathrop, "Cube Attacks on Cryptographic Hash Functions," Master's thesis, Rochester Institute of Technology, 2009.

[53] B. Schneier, "Adi Shamir's Cube Attacks." `http://schneier.com/blog/archives/2008/08/adi_shamirs_cub.html`, August 2008.

[54] A. Biryukov and D. Wagner, "Slide Attacks," in *Fast Software Encryption*, pp. 245–259, Springer, 1999.

[55] J. Daemen, L. R. Knudsen, and V. Rijmen, "The Block Cipher Square," in *Fast Software Encryption*, pp. 149–165, Springer, 1997.

[56] N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting, "Improved Cryptanalysis of Rijndael," in *Fast Software Encryption*, pp. 213–230, Springer, 2001.

[57] M. R. Z'aba, H. Raddum, M. Henricksen, and E. Dawson, "Bit-Pattern Based Integral Attack," in *Fast Software Encryption*, pp. 363–381, Springer, 2008.

[58] A. Rukhin, J. Soto, J. Nechvatal, E. Barker, S. Leigh, M. Levenson, D. Banks, A. Heckert, J. Dray, S. Vo, *et al.*, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications." NIST Special Publication 800-22, Revision 1a, April 2010.

[59] J. S. Jr., "Randomness Testing of the Advanced Encryption Standard Candidate Algorithms." NIST Internal Report 6390, September 1999.

[60] A. Kaminsky, "The Coincidence Test: a Bayesian Statistical Test for Block Ciphers and MACs." `http://www.cs.rit.edu/~ark/parallelcrypto/cryptostat/coincidence.pdf`, September 2013.

[61] C. D'Halluin, G. Bijnens, B. Preneel, and V. Rijmen, "Equivalent Keys of HPC," in *Advances in Cryptology-ASIACRYPT'99*, pp. 29–42, Springer, 1999.

[62] SciPy Developers, "SciPy Library." `http://scipy.org/scipylib`, 2014.

[63] H. M. Heys and S. E. Tavares, "Avalanche Characteristics of Substitution-Permutation Encryption Networks," *IEEE Transactions on Computers*, vol. 44, no. 9, pp. 1131–1139, 1995.

[64] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The Skein Hash Function Family," *NIST SHA-3 Submission Document*, October 2010. `http://schneier.com/skein1.3.pdf`.

[65] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SI-MON and SPECK Families of Lightweight Block Ciphers." Cryptology ePrint Archive, Report 2013/404, 2013. `http://eprint.iacr.org/`.

Appendix A

PRESENT Trail Illustration
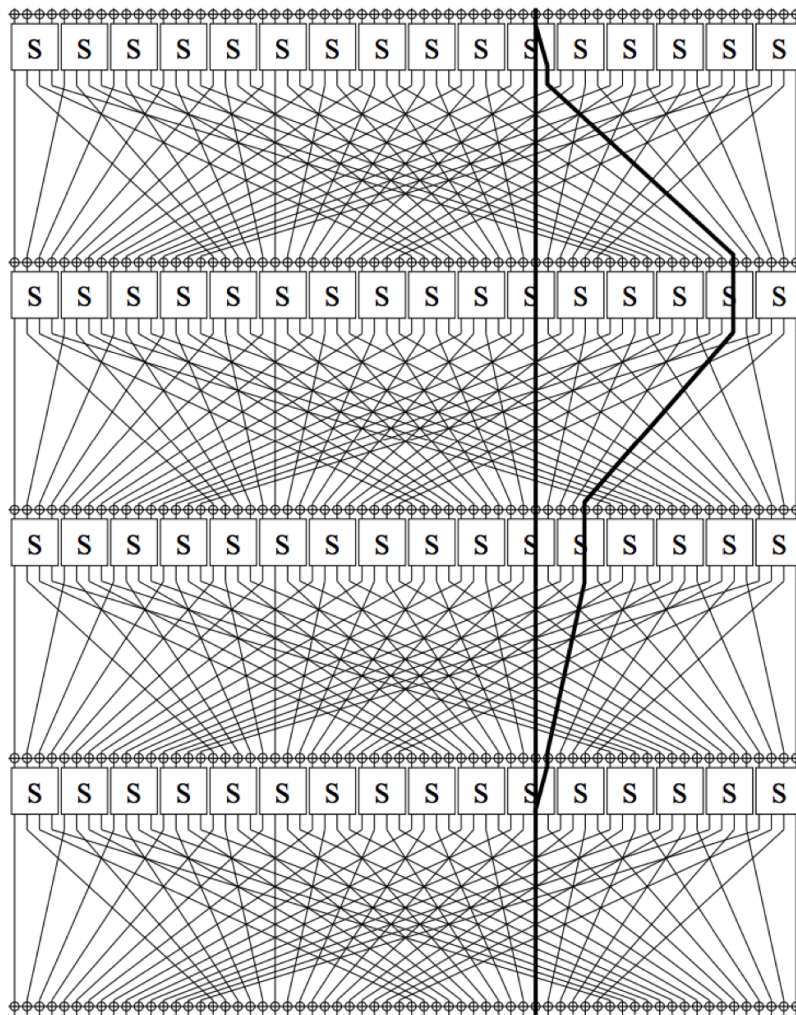
Figure A.1: Example of an attack on PRESENT that arose due to poor selection of a bitwise permutation. The combination of fixed-points and a low-order permutation results in a trail which minimizes the number of active S-boxes across four rounds [35].

Appendix B

Bitwise Permutation Listing

The following bitwise permutations defined by the affine function

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

satisfy all required properties listed in Chapter 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
	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	

Corresponding β values

Table B.1: Bitwise permutations satisfying all desired properties

Appendix C

ARX-Based Mixers

As part of the design process for the cryptosystem described in this thesis, many ARX-based mixers were analyzed before deciding to switch to a mixer based on matrix multiplication in a Galois field. None of these ARX-based mixers were able to increase the linear and differential branch numbers from two to three, which is one of our requirements. For completeness, we enumerate all of the mixers we analyzed here.

One of the mixers is based on the ARX structure used in Threefish, the block cipher used within SHA-3 candidate Skein [64]. Another is based on the recently released lightweight block cipher Speck that was created by the National Security Agency [65]. From these ideas, more complex ARX structures were constructed and analyzed.

In all diagrams shown here, the **ROT** operation represents all possible nontrivial left and right rotations on a single word. Mixers corresponding to every single combination of rotations were analyzed. The $\boxplus$ symbol denotes addition modulo 2^{16}. It is a nonlinear operation due to effect of the carry bit. The $\oplus$ symbol denotes XOR, as usual.
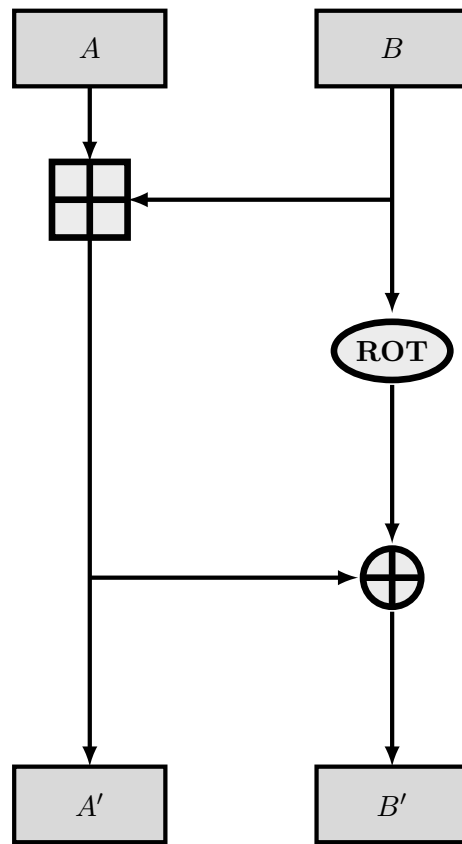
Figure C.1: Candidate mixer inspired by Threefish
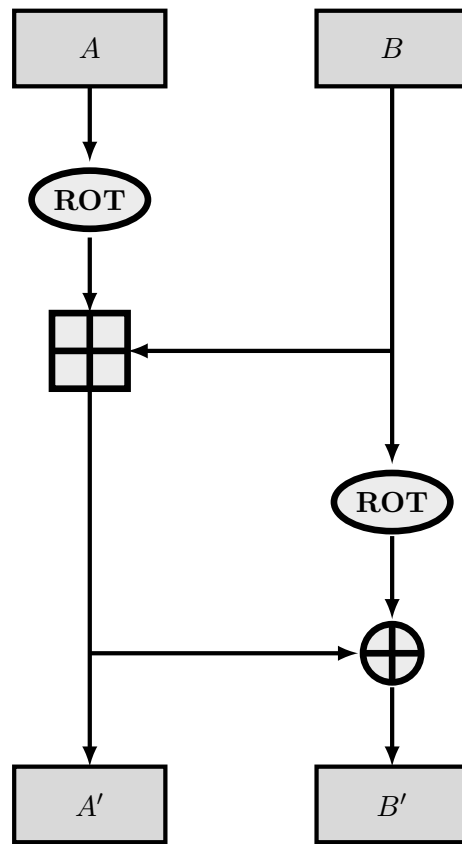
Figure C.2: Candidate mixer inspired by Speck
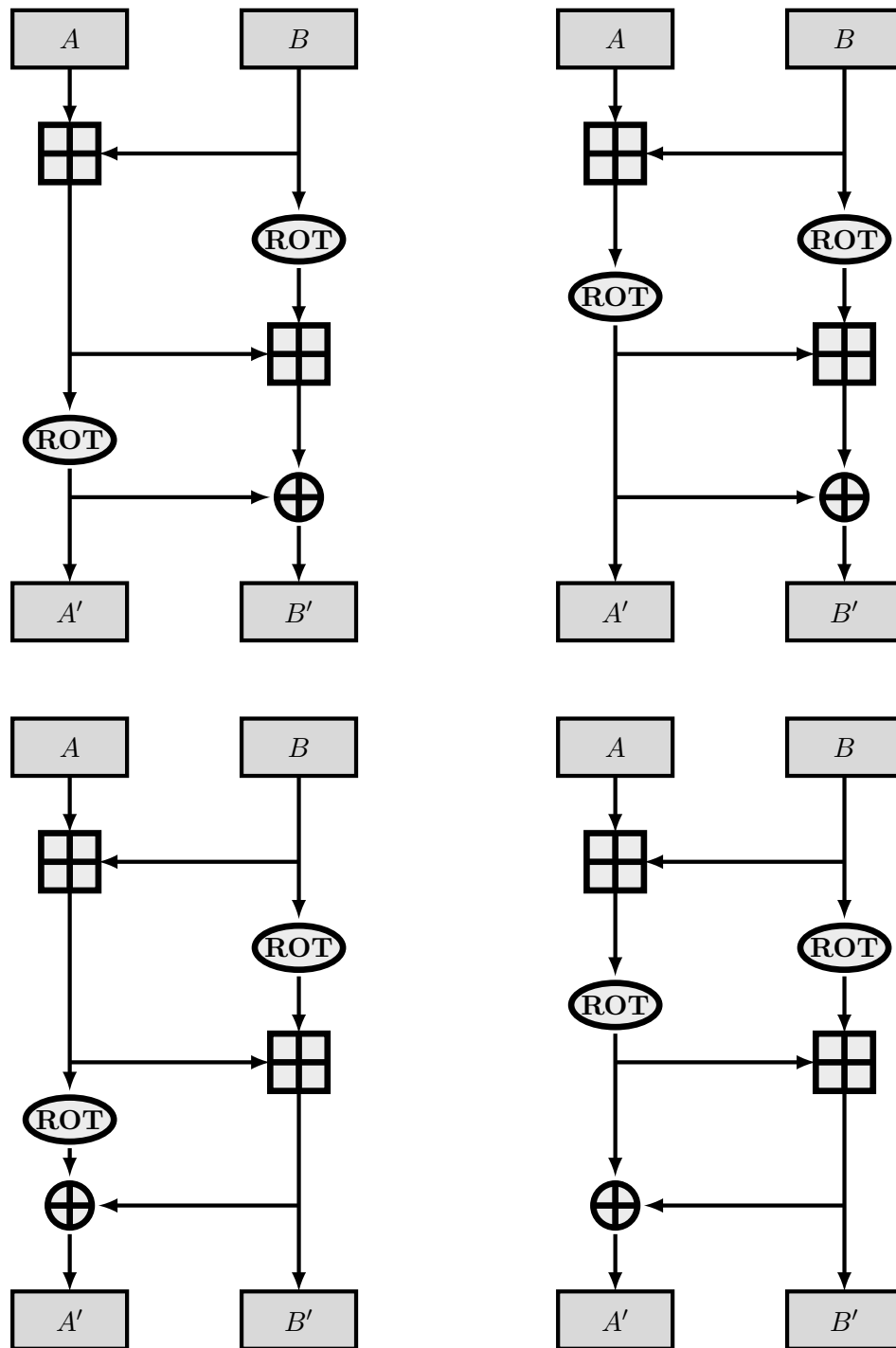
Figure C.3: Custom candidate mixers

Appendix D

Known Answer Tests

Key (128)	00000000000000000000000000000000
IV	000000000000000000000000000000
A (header)	000000000000000000000000000000000000000000000000000000000000000000000000
B (body)	000000000000000000000000000000000000000000000000000000000000000000000000 000000000000000000000000000000000000000000000000000000000000000000000000
Ciphertext	55851b98d902192aa6e0f23738205b74802c7702b5b88bd44de711f4188d5a43 a5fcfdd178e6e8e05a28761723974fe978c1c83d378a40490bf5e944f862b452
Tag (MAC)	2a0ae2db865ec69dc7a075d089986466

Key (128)	00000000000000000000000000000000
IV	000000000000000000000000000000
A (header)	000000000000000000000000000000000000000000000000000000000000000000000000 000000000000000000000000000000000000000000000000000000000000000000000000
B (body)	000000000000000000000000000000000000000000000000000000000000000000000000 000000000000000000000000000000000000000000000000000000000000000000000000 000000000000000000000000000000000000000000000000000000000000000000000000 000000000000000000000000000000000000000000000000000000000000000000000000
Ciphertext	a5fcfdd178e6e8e05a28761723974fe978c1c83d378a40490bf5e944f862b452 9f671ea19fbc2611d372b98ba6eda6f46ef6b6123ec2c1512e7c9e0cfd878a87 46726b4fe11801f76457be25f0ba2b662a0ae2db865ec69dc7a075d089986466 f360b07d5d532261868d610ffa26cc4d8cbdc43cf9f777ce7cb3861b2ff35022
Tag (MAC)	b42652acb41d263633b554bb15bad007

Key (128)	00000000000000000000000000000000
IV	000000000000000000000000000000
A (header)	(empty)
B (body)	0000000000000000000000000000000000000000000000000000000000000000
Ciphertext	daa223c25a6dacc0934c82a7c0bf45e730d4bc43cb07e36703a34e24655c4c65
Tag (MAC)	802c7702b5b88bd44de711f4188d5a43

Key (128)	00000000000000000000000000000001
IV	000000000000000000000000000000
A (header)	(empty)
B (body)	0000000000000000000000000000000000000000000000000000000000000000
Ciphertext	e77f18fee964bc2c4a3419a3cf97c558b38d5649af7877a31793cb7850ec8d61
Tag (MAC)	51c466c4d46e4a94d627452d1802a257

Key (128)	a110c8b01dc0ffeedea110c8a11decaf
IV	123456789012345678901234567890012
A (header)	(empty)
B (body)	b0a710ad50fc0c0a5ca1ab1eca55e77e
Ciphertext	f80da4aa7a9a54541902747e4eef2977
Tag (MAC)	8f81ba4c838d9f5ee93bbef40752a0f8

Key (256)	0000000000000000000000000000000000000000000000000000000000000000
IV	000000000000000000000000000000000000
A (header)	00000000000000000000000000000000000000000000000000000000000000000000
B (body)	0000000000000000000000000000000000000000000000000000000000000000000000 000000000000000000000000000000000000000000000000000000000000000000000
Ciphertext	53fe8b7d1cfefc5f12fe27b42efe8669d1f15b0ecfb0b8adcc0aeeee50f170d7 ce5a0e25d9b03dc55057e093fbaad8e5b83cb0fbc3249e3e38ec9bc7966b177e
Tag (MAC)	1770e24ce3ddeaed27a817c41d6af97b

Key (256)	0000000000000000000000000000000000000000000000000000000000000000
IV	000000000000000000000000000000000000
A (header)	0000000000000000000000000000000000000000000000000000000000000000000000 000000000000000000000000000000000000000000000000000000000000000000000
B (body)	0000000000000000000000000000000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000000000000000000000000000000 000000000000000000000000000000000000000000000000000000000000000000000
Ciphertext	ce5a0e25d9b03dc55057e093fbaad8e5b83cb0fbc3249e3e38ec9bc7966b177e 0fc577cf886970f3638628130c7e0b0264ca2017d5f150cd9a2c0866008319ce f9feb4a28983be315cfb538fcb79f30c1770e24ce3ddeaed27a817c41d6af97b 531d26ae76128bfc19fa632472f010d57c14a906fef7801110125a0e1ef3fd9a
Tag (MAC)	d29be6e3b6013168e83421743a82fed2

Key (256)	0000000000000000000000000000000000000000000000000000000000000000
IV	000000000000000000000000000000000000
A (header)	(empty)
B (body)	0000000000000000000000000000000000000000000000000000000000000000
Ciphertext	ed2376dfe1268a96f10cad1f97f7a9d38838ff1abc53459a1f395170f552f3ec
Tag (MAC)	d1f15b0ecfb0b8adcc0aeeee50f170d7

Key (256)	0000000000000000000000000000000000000000000000000000000000000001
IV	000000000000000000000000000000000
A (header)	(empty)
B (body)	00000000000000000000000000000000000000000000000000000000000000000
Ciphertext	b1fc1266e17b4aac3bf582ce759c1f21c16e30d893e455f0246eaf948a83d9d3
Tag (MAC)	558b474275fc8910a6b713eed680c5d6

Key (256)	a110c8b01dc0ffeedea110c8a11decafa110c8b01dc0ffeedea110c8a11decaf
IV	123456789012345678901234567890
A (header)	(empty)
B (body)	b0a710ad50fc0c0a5ca1ab1eca55e77e
Ciphertext	d1c39640e6e42373c54a727d4ed45cd6
Tag (MAC)	2e15194003f7ed304557991dc19e4e88

Appendix E

Source Code Listings

E.1   AE Algorithm Software Implementation

E.1.1   State.h

```
/********************************************************************
 * A 512-bit state split into 32 16-bit words.
 * Provides word and bit access.
 * Includes all necessary functions for absorbing, squeezing, copying, etc.
 *
 * Author: Matt Kelly
 * Date:   June 2014
 ********************************************************************/
#ifndef STATE_H
#define STATE_H

#include <stdint.h>
#include <stdio.h>
#include <stdexcept> // std::out_of_range
#include <string.h> // memcpy

#define WIDTH 512
#define WORD_SIZE 16
#define NUM_WORDS WIDTH / WORD_SIZE

class State {
private:
    uint16_t state[NUM_WORDS];

public:

    /**
     * Constructor - create new all-zero state
     */
    State()
    {
        for( int i = 0; i < NUM_WORDS; ++i )
            state[i] = 0;
    }

    /**
     * Copy constructor
     */
    State(const State &other)
    {
        memcpy(state, other.state, sizeof(state));
    }

    /**
     * Copy contents of this state into given output array,
     * starting from given offset.
     */
    void copy(uint16_t output[], int numWords, int offset)
    {
```

```cpp
    memcpy(&output[offset], state, sizeof(uint16_t) * numWords);
}

/**
 * Return reference to word at given index in state
 */
uint16_t& operator[](const unsigned int x)
{
    if( x > NUM_WORDS − 1 )
        throw std::out_of_range("Invalid_index");
    return state[x];
}

/**
 * Return reference to word at given index in state
 */
const uint16_t& operator[](const unsigned int x) const
{
    if( x > NUM_WORDS − 1 )
        throw std::out_of_range("Invalid_index");
    return state[x];
}

/**
 * Return true if this state is equal to other state
 */
const bool operator==(const State& other)
{
    for( int i = 0; i < NUM_WORDS; ++i )
        if( state[i] != other[i] ) return false;
    return true;
}

/**
 * Return true if this state is different from other state
 */
const bool operator!=(const State& other) { return !operator==(other); }

/**
 * Get bit at given index
 */
int getBit(unsigned int x) const
{
    return (state[x / 16] >> (x % 16)) & 1;
}

/**
 * Clear bit at given index
 */
void clearBit(unsigned int x)
{
    state[x / 16] &= ~(1 << (x % 16));
}

/**
 * Set bit at given index
 */
void setBit(unsigned int x)
{
    state[x / 16] |= (1 << (x % 16));
}

/**
 * Invert bit at given index
 */
void invertBit(unsigned int x)
{
    state[x / 16] ^= (1 << (x % 16));
}

/**
 * Set outer r = 128 bits of state to given value
```

```
 */
void setOuterState(uint16_t value[8])
{
    memcpy(state, value, sizeof(uint16_t) * 8);
}


/**
 * XOR outer r = 128 bits of state with given value
 * Assumption: value is correct length
 */
void xorOuterState(uint16_t value[8])
{
    for( int i = 0; i < 8; ++i )
        state[i] ^= value[i];
}


/**
 * Set inner c = 384 bits of state to given value
 */
void setInnerState(uint16_t value[24])
{
    memcpy(&state[8], value, sizeof(uint16_t) * 8);
}


/**
 * Clear all bits (set state to 0x000...0)
 */
void clearAll()
{
    for( int i = 0; i < NUM_WORDS; ++i )
        state[i] = 0x0000;
}


/**
 * Set all bits (set state to 0xFFF...F)
 */
void setAll()
{
    for( int i = 0; i < NUM_WORDS; ++i )
        state[i] = 0xFFFF;
}


/**
 * Get the hamming weight of this state
 */
int getHW()
{
    int hw = 0;
    for( int i = 0; i < NUM_WORDS; ++i ) {
        uint16_t x = state[i];
        while( x ) {
            if( x & 1 ) hw++;
            x >>= 1;
        }
    }
    return hw;
}


/**
 * Get the hamming distance from this state to another state
 */
int getHammingDistance(State other)
{
    int hd = 0;
    for( int i = 0; i < WIDTH; ++i )
        if( getBit(i) != other.getBit(i) ) hd++;
    return hd;
}


/**
 * Print state in words with nice formatting
 */
```

```
    void print(FILE *out = stdout)
    {
        for( int i = 0; i < NUM_WORDS; ++i ) {
            if( i > 0 && i % 8 == 0 ) fprintf(out, "\n");
            fprintf(out, "%04hx ", state[i]);
        }
        fprintf(out, "\n\n");
    }

    /**
     * Print state in bits with nice formatting
     */
    void printBits(FILE *out = stdout)
    {
        for( int i = 0; i < WIDTH; ++i ) {
            if( i > 0 && i % 128 == 0 ) fprintf(out, "\n");
            if( i > 0 && i % 16 == 0 && i % 128 != 0)
                fprintf(out, " ");
            fprintf(out, "%d", getBit(i));
        }
        fprintf(out, "\n\n");
    }

    /**
     * Print state in bits with no formatting
     */
    void dumpBits(FILE *out = stdout)
    {
        for( int i = 0; i < WIDTH; ++i )
            fprintf(out, "%d", getBit(i));
        fprintf(out, "\n");
    }

};

#endif /* STATE_H */
```

E.1.2 MixerGF.h

```
/***********************************************************************
 * Mathematical functions for GF(2^16)
 * The irreducible polynomial x^16 + x^5 + x^3 + x^2 + 1 is used.
 *
 * Author: Matt Kelly
 * Date:   June 2014
 ***********************************************************************/
#ifndef MIXER_GF_H
#define MIXER_GF_H

#include <stdint.h>
#include <stdio.h>
#include "string.h"

/* Field polynomial = x^16 + x^5 + x^3 + x^2 + 1 */
#define FIELD_POLY 0x002d

/*
 * Multiply two polynomials in GF(2^16), using the
 * LSB first algorithm
 */
uint16_t gf_multiply(uint16_t a, uint16_t b)
{
        uint16_t i;
        uint16_t acc = 0x0000; /* Accumulator */
        uint16_t msb; /* Current MSB of a */

        for(i = 0; i < 16; ++i) {
                /* If LSB of b is 1, add a to accumulator */
                if( b & 0x0001 ) acc ^= a;
```

```
                           /* Store MSB of a, then shift it off */
                           msb = a & 0x8000;
                           a <<= 1;
                           /* If MSB of a was 1, add field polynomial to a */
                           if ( msb ) a ^= FIELD_POLY;
                           /* Advance to next bit of b */
                           b >>= 1;
                }
                return acc;
}

/*
 * Divide two polynomials (b / a) in GF(2^16)
 */
void gf_divide(uint16_t a, uint16_t b, uint16_t *q, uint16_t *r, uint16_t field_poly)
{
                uint16_t i = 0;
                uint16_t j;
                uint16_t quotient = 0x0000; /* Quotient */
                uint16_t rem = b; /* Remainder */
                uint16_t msb_a; /* Current MSB of a */

                /* Perform initial alignment */
                msb_a = a & 0x8000;
                while ( !msb_a && i < 16 ) {
                           a <<= 1;
                           msb_a = a & 0x8000;
                           i++;
                }

                /* Take care of field polynomial MSB (bit 8) */
                if (field_poly) {
                           quotient |= (1 << (i+1));
                           rem ^= (a << 1);
                }

                /* Compute for remaining bits after alignment */
                /* From this point msb_a is always 1 */
                for ( j = 0; j < i+1; ++j ) {
                           if ( ( rem << j ) & 0x8000 ) {
                                      /* Reduce if needed */
                                      quotient |= (1 << (i-j));
                                      rem ^= a;
                           }
                           a >>= 1;
                }
                *q = quotient;
                *r = rem;
}

/*
 * Find multiplicative inverse of a number in GF(2^16)
 * using the Extended Euclid Algorithm.
 */
uint16_t gf_inverse(uint16_t a)
{
                /* Map 0x0000 to 0x0000 */
                if ( a == 0x0000 ) {
                           return 0x0000;
                }

        uint16_t b = FIELD_POLY;

                uint16_t rem[16];
                uint16_t aux[16];
                uint16_t q, r;
                uint16_t i;

                rem[0] = b;
                rem[1] = a;
                aux[0] = 0;
                aux[1] = 1;
```

```
        i = 1;
        while( rem[i] > 0x0001 ) {
                i++;
                if( i == 2 )
                        gf_divide(rem[i-1], rem[i-2], &q, &r, 1);
            else
                        gf_divide(rem[i-1], rem[i-2], &q, &r, 0);
                rem[i] = r;
                aux[i] = gf_multiply(q, aux[i-1]) ^ aux[i-2];
        }

        /* Inverse of A(x) is in current index of aux */
        return aux[i];
}

/*
 * Calculate the inverse of a 2x2 matrix
 * Input matrix is assumed to be invertible
 */
void gf_2x2_inverse(uint16_t M[2][2])
{
    uint16_t i;
    uint16_t A[2][4] = {{M[0][0], M[0][1], 1, 0},
                        {M[1][0], M[1][1], 0, 1}};

    // Set A[0][0] to 1
    uint16_t a00_inv = gf_inverse(A[0][0]);
    for( i = 0; i < 4; ++i )
        A[0][i] = gf_multiply(a00_inv, A[0][i]);

    // Set A[1][0] to 0
    uint16_t a10 = A[1][0];
    for( i = 0; i < 4; ++i )
        A[1][i] = A[1][i] ^ gf_multiply(a10, A[0][i]);

    // Set A[1][1] to 1
    uint16_t a11_inv = gf_inverse(A[1][1]);
    for( i = 0; i < 4; ++i )
        A[1][i] = gf_multiply(a11_inv, A[1][i]);

    // Set A[0][1] to 0
    uint16_t a01 = A[0][1];
    for( i = 0; i < 4; ++i )
        A[0][i] ^= gf_multiply(a01, A[1][i]);

    // Return in M
    M[0][0] = A[0][2];
    M[0][1] = A[0][3];
    M[1][0] = A[1][2];
    M[1][1] = A[1][3];
}

#endif /* MIXER_GF_H */
```

E.1.3 SboxGF.h

This code was written by Christopher Wood as part of his thesis work in [33]; we merely condensed it for brevity here.

```
/**
 * Author: Christopher A. Wood, caw4567@rit.edu
 *
 * Modified by Matt Kelly in June 2014 for brevity.
 */
#ifndef SBOXGF_H
#define SBOXGF_H
```

```c
#include <stdint.h>
#include <stdio.h>

// Quotient and remainder struct
typedef struct
{
        uint16_t q;
        uint16_t r;
        uint8_t error;
} QR;

// Bit masks for the MSB and LSB
#define MSB_16   0x8000
#define HMSB_16  0x10000
#define LSB      0x1

// Standard definitions (for sbox16.c)
#define PX_16   0x002B
#define FPX_16  0x1002B

uint16_t g16_add(uint16_t x, uint16_t y)
{
        return x ^ y;
}

uint16_t g16_sub(uint16_t x, uint16_t y)
{
        return x ^ y;
}

uint16_t g16_mul(uint16_t x, uint16_t y)
{
        uint16_t accum = 0;
        uint16_t msb = 0;
        uint16_t i;
        for (i = 0; i < 16; i++)
        {
                if (y & LSB) accum ^= x;
                msb = (x & MSB_16); // fetch the MSB
                x <<= 1;
                if (msb) x ^= PX_16;
                y >>= 1;
        }
        return accum;
}

/**
 * Polynomial division in GF(2^16).
 */
QR g16_div(uint32_t ai, uint16_t b)
{
        uint16_t a = (uint16_t)ai;
        int msb = MSB_16;
        int d = 0;
        QR result = {0, 0};

        // Align the denominator with the numerator
        while (b > 0 && !(b & MSB_16)) {
                ++d;
                b <<= 1;
        }

        // If the polynomial MSB is set (17th bit), increment
        // the quotient and reduce the numerator.
        if (ai & HMSB_16) {
                result.q ^= 1 << (d+1);
                a ^= b << 1;
        }

        for (;d > -1; d--) {
                if ((a & msb) && (b & msb)) {
                        result.q ^= 1 << d;
```

```
                                      a  ^= b;
                        }
                    msb >>= 1;
                    b >>=l;
            }

            result.r = a;
            return result;
}

/**
 * Modular inverse in GF(2^16) using the EEA algorithm.
 */
uint16_t g16_inv(uint16_t x)
{
            // Trivial special cases.
            if (x == 0) return 0;
            if (x == 1) return 1;

            uint16_t r0 = PX_16; // rem[i − 2]
            uint16_t r1 = x;     // rem[i − 1]
            uint16_t a0 = 0;     // aux[i − 2]
            uint16_t a1 = 1;     // aux[i − 1]
            uint16_t tmp;
            QR qr;

            int firstRun = 0;
            while (r1 > 0)
            {
                    if (firstRun != 0) qr = g16_div(r0, r1);
                    else
                    {
                            qr = g16_div(FPX_16, r1);
                            firstRun++;
                    }
                    r0 = r1; r1 = qr.r;
                    tmp = a0; a0 = a1;
                    a1 = g16_add(tmp, g16_mul(qr.q, a1));
            }

            return a0;
}

uint16_t g16_change_basis(uint16_t x, uint16_t* M)
{
            int32_t i;
            uint16_t y = 0;

            for (i = 15; i >= 0; i−−)
            {
                    if (x & 1) y ^= M[i];
                    x >>= 1;
            }

            return y;
}

#endif /* SBOXGF_H */
```

E.1.4   Sbox16.h

This code was written by Christopher Wood as part of his thesis work in [33]; we merely condensed it for brevity here.

```
/**
 * Author: Christopher A. Wood, caw4567@rit.edu
 *
 * Modified by Matt Kelly in June 2014 for brevity.
```

```c
 */
#ifndef SBOXGF_H
#define SBOXGF_H

#include <stdint.h>
#include <stdio.h>

// Quotient and remainder struct
typedef struct
{
        uint16_t q;
        uint16_t r;
        uint8_t error;
} QR;

// Bit masks for the MSB and LSB
#define MSB_16   0x8000
#define HMSB_16 0x10000
#define LSB       0x1

// Standard definitions (for sbox16.c)
#define PX_16   0x002B
#define FPX_16 0x1002B

uint16_t g16_add(uint16_t x, uint16_t y)
{
        return x ^ y;
}

uint16_t g16_sub(uint16_t x, uint16_t y)
{
        return x ^ y;
}

uint16_t g16_mul(uint16_t x, uint16_t y)
{
        uint16_t accum = 0;
        uint16_t msb = 0;
        uint16_t i;
        for (i = 0; i < 16; i++)
        {
                if (y & LSB) accum ^= x;
                msb = (x & MSB_16); // fetch the MSB
                x <<= 1;
                if (msb) x ^= PX_16;
                y >>= 1;
        }
        return accum;
}

/**
 * Polynomial division in GF(2^16).
 */
QR g16_div(uint32_t ai, uint16_t b)
{
        uint16_t a = (uint16_t)ai;
        int msb = MSB_16;
        int d = 0;
        QR result = {0, 0};

        // Align the denominator with the numerator
        while (b > 0 && !(b & MSB_16)) {
                ++d;
                b <<= 1;
        }

        // If the polynomial MSB is set (17th bit), increment
        // the quotient and reduce the numerator.
        if (ai & HMSB_16) {
                result.q ^= 1 << (d+1);
                a ^= b << 1;
        }
```

```
        for (; d > -1; d--) {
                if ((a & msb) && (b & msb)) {
                        result.q ^= 1 << d;
                        a ^= b;
                }
                msb >>= 1;
                b >>=l;
        }

        result.r = a;
        return result;
}

/**
 * Modular inverse in GF(2^16) using the EEA algorithm.
 */
uint16_t g16_inv(uint16_t x)
{
        // Trivial special cases.
        if (x == 0) return 0;
        if (x == 1) return 1;

        uint16_t r0 = PX_16; // rem[i - 2]
        uint16_t r1 = x;     // rem[i - 1]
        uint16_t a0 = 0;     // aux[i - 2]
        uint16_t a1 = 1;     // aux[i - 1]
        uint16_t tmp;
        QR qr;

        int firstRun = 0;
        while (r1 > 0)
        {
                if (firstRun != 0) qr = g16_div(r0, r1);
                else
                {
                        qr = g16_div(FPX_16, r1);
                        firstRun++;
                }
                r0 = r1; r1 = qr.r;
                tmp = a0; a0 = a1;
                a1 = g16_add(tmp, g16_mul(qr.q, a1));
        }

        return a0;
}

uint16_t g16_change_basis(uint16_t x, uint16_t* M)
{
        int32_t i;
        uint16_t y = 0;

        for (i = 15; i >= 0; i--)
        {
                if (x & 1) y ^= M[i];
                x >>= 1;
        }

        return y;
}

#endif /* SBOXGF_H */
```

E.1.5 Permutation.h

```
/********************************************************************
 * The underlying sponge permutation f.
 *
 * Author: Matt Kelly
```

```cpp
 * Date:    June 2014
 **********************************************************************/
#ifndef PERMUTATION_H
#define PERMUTATION_H

#include "State.h"
#include "Sbox16.h"
#include "MixerGF.h"

class Permutation {
private:
    int _numRounds;

    const uint16_t RC[16][32] =
    // SHA3-512(ASCII(1))
    {{0x0019,0x7a4f,0x5f1f,0xf8c3,0x56a7,0x8f69,0x21b5,0xa6bf,0xbf71,0xdf8d,0xbd31,0x3fbc,0x5095,0xa55d,0xe756,0xbfa1,
      0xea72,0x4069,0x5005,0x1492,0x94f2,0xa2e4,0x19ae,0x251f,0xe2f7,0xdbb6,0x7c3b,0xb647,0xc2ac,0x1be0,0x5eec,0x7ef9},
    // SHA3-512(ASCII(2))
     {0xac3b,0x6998,0xac9c,0x5e2c,0x7ee8,0x3300,0x10a7,0xb0f8,0x7ac9,0xdee7,0xea54,0x7d4d,0x8cd0,0x0ab7,0xad1b,0xd5f5,
      0x7f80,0xaf2b,0xa711,0xa9eb,0x137b,0x4e83,0xb503,0xd24c,0xd766,0x5399,0xa487,0x34d4,0x7fff,0x324f,0xb745,0x51e2},
    // SHA3-512(ASCII(3))
     {0xce4f,0xd406,0x8e56,0xeb07,0xa6e7,0x9d00,0x7aed,0x4bc8,0x257e,0x1082,0x7c74,0xee42,0x2d82,0xa29b,0x2ce8,0xcb07,
      0x9fea,0xd81d,0x9df0,0x513b,0xb577,0xf3b6,0xc478,0x43b1,0x7c96,0x4e7f,0xf8f4,0x198f,0x3202,0x7533,0xeaf5,0xbcc1},
    // SHA3-512(ASCII(4))
     {0x5058,0xcb97,0x5975,0xceff,0x027d,0x1326,0x4889,0x12e1,0x99b7,0x9b91,0x6ad9,0x0a3f,0xe2fd,0x0150,0x8cd7,0xd7c0,
      0x1bc8,0xaaaa4,0xd21a,0x8473,0xfb15,0xf3b1,0x51ab,0x9e44,0x172e,0x9ccb,0x70a5,0xea04,0x495a,0xf3ec,0x03b5,0x153e},
    // SHA3-512(ASCII(5))
     {0x84da,0x272d,0x13a4,0x4f08,0x98ee,0x4ea5,0x3334,0xc255,0xd894,0xcc54,0xd357,0xc554,0x66d7,0x60de,0xbde4,0x82a2,
      0x44c1,0x28df,0x641e,0x8067,0x3a8b,0xc34a,0x1620,0xd880,0xb796,0x5e54,0x9f31,0x3ddc,0xcfd5,0x06b0,0x7341,0x3b87},
    // SHA3-512(ASCII(6))
     {0xbb93,0xaaa2,0x3b38,0xea96,0xc934,0x6ef9,0x1e18,0x4982,0xbf50,0xe910,0x33f4,0x354e,0xcb20,0xd3c7,0x390c,0x2b41,
      0x862e,0x8825,0xec3d,0x0fee,0x0a6f,0x9788,0x81f9,0x0728,0xc674,0x8e4a,0xed8b,0x7323,0x5007,0x5d6c,0x2bdd,0x8e4b},
    // SHA3-512(ASCII(7))
     {0xfe32,0xf3eb,0xa766,0x26de,0xdf36,0x622b,0xfdc5,0xccd3,0x3db2,0xf3e0,0xdd7c,0x3c12,0x8298,0xea78,0xc1cc,0x7fee,
      0x1a14,0x0edb,0x8e57,0xcd58,0x24c7,0xf4b8,0x17c0,0xfc94,0xe70d,0xa5b9,0x399f,0xaaf9,0xa848,0xa46a,0xd306,0x79e9},
    // SHA3-512(ASCII(8))
     {0x952b,0xa024,0x86b8,0x18fe,0xbc0e,0xc985,0x59df,0x27c7,0x9357,0x838f,0x011b,0x1e5b,0xc11f,0x2cfb,0x6fc0,0x573e,
      0x5459,0x78c2,0xbc5b,0x390f,0x4490,0x7f8d,0xa0df,0xd682,0x06fe,0x4521,0xf86b,0xa6c8,0x79ec,0x1e69,0xcaed,0x9533},
    // SHA3-512(ASCII(9))
     {0xb41e,0x6bb4,0xed20,0x2940,0x1639,0x9c26,0x8da6,0xbf88,0xc89e,0x2dc1,0x18a3,0x61b3,0x560e,0xe8da,0xed97,0x3a8f,
      0x9778,0xdf40,0xe308,0xc120,0x6fa4,0x2f97,0xf3fd,0x3f63,0xd2b4,0xb3b5,0x7eb5,0xbcbe,0xc6ad,0x64d4,0x6216,0xb692},
    // SHA3-512(ASCII(10))
     {0x6954,0xa418,0xcecc,0x4363,0x3bd5,0x26c2,0x499d,0xfc16,0xb832,0xf58b,0x216b,0x9a8b,0x226a,0x6a0b,0x7918,0xd364,
      0xa793,0x9004,0x339d,0xe0ba,0x08e2,0xb547,0xe64d,0xc562,0x2e24,0xb0c4,0xf8f4,0x15d9,0xe0a8,0x4cb9,0x4b6c,0x5f3f},
    // SHA3-512(ASCII(11))
     {0x2e4b,0x9ad3,0x7091,0xe3e5,0xa218,0xc5e5,0x7b33,0xed34,0x70ba,0x4f31,0xfbcf,0x1642,0x4684,0xfdd5,0xcde3,0x8e88,
      0x9eae,0x3f01,0x8b37,0xaf58,0xc24c,0xcc8a,0xf57a,0xbc2c,0x6911,0x408d,0xd20e,0xf643,0x5e44,0x94a3,0xe659,0x9a06},
    // SHA3-512(ASCII(12))
     {0xaa42,0xaca7,0x3bd7,0xf8a1,0x7e98,0x7f28,0x1422,0xb266,0xe44f,0x0de1,0x615d,0x2d39,0x3c62,0x0c8c,0x5a2c,0x80b4,
      0xf061,0x78c8,0x455b,0xf981,0x7960,0x3f2f,0x1bcb,0x30b2,0x559f,0x282c,0x799e,0x4053,0x3b06,0x65f9,0x7a2a,0x706a},
    // SHA3-512(ASCII(13))
     {0x969c,0x39ae,0x2dc1,0x6834,0x3103,0x44c0,0x579d,0x0ffd,0xfde0,0x1772,0xdbf9,0xa4ca,0xb984,0x953c,0x395d,0x7791,
      0x1510,0xf39e,0x5f37,0x295e,0x3611,0xa1d4,0x6101,0x460d,0xaf73,0x1ddb,0xdab1,0xec1b,0xbc51,0x2edc,0x4468,0x0d8d},
    // SHA3-512(ASCII(14))
     {0x8a1e,0x6ce3,0x1f0b,0x526d,0x884b,0x584a,0xa1a5,0xae42,0x94fc,0xf85f,0xd2e5,0x25f9,0x59ed,0x1a54,0x2333,0x59c7,
      0xc5fe,0xce6d,0x2477,0x5e7d,0x4a9a,0xd97c,0x2632,0xa3be,0x5b33,0x1a8f,0x580f,0x557b,0x269e,0x7b65,0x123a,0x5992},
    // SHA3-512(ASCII(15))
     {0x9bd6,0x4a93,0x2f09,0x672d,0xef04,0xb6a9,0x4753,0xa3e4,0x087a,0x1c38,0x9507,0x8dc7,0x0927,0xfcd7,0x7488,0x8dfd,
      0x400b,0x95fd,0x1c6a,0x0b2a,0x91a1,0xba44,0xeea0,0x9f51,0x63db,0xa4df,0xa9da,0x7b8e,0xb97d,0x791c,0xab56,0x6437},
    // SHA3-512(ASCII(16))
     {0x4840,0x1f65,0xc2d2,0xd9e7,0x1fe4,0x7bd8,0x0b28,0xd834,0xeee8,0xfff3,0xbe9a,0xa460,0x8cba,0x33e6,0xfedc,0xe0b1,
      0x693c,0x80cd,0xc36d,0xb7f5,0x04e4,0xabea,0x23cc,0xc672,0x9a03,0x0f5b,0x3e03,0x5fb5,0x9c2c,0x7882,0x15cf,0x84a8}};

public:

    /**
     * Construct a permutation with the given number of rounds.
     */
    Permutation(int numRounds)
    {
        _numRounds = numRounds;
    }
```

```
/**
 * Performs substitution using 16x16-bit S-box
 */
uint16_t sub16(uint16_t a)
{
    return sbox_forward(a);
}

/**
 * Performs inverse substitution using 16x16-bit S-box
 */
uint16_t sub16Inverse(uint16_t a)
{
    return sbox_inverse(a);
}

/**
 * Performs addition modulo 65536 on two 16-bit inputs
 */
uint16_t add(uint16_t a, uint16_t b)
{
    return (uint16_t)(((uint32_t)(a + b)) % 65536);
}

/**
 * Rotate left by given amount
 */
uint16_t rotl(uint16_t a, int rotConst)
{
    return ((a << rotConst) | (a >> (16 - rotConst)));
}

/**
 * Rotate right by given amount
 */
uint16_t rotr(uint16_t a, int rotConst)
{
    return ((a >> rotConst) | (a << (16 - rotConst)));
}

/**
 * Apply mixing function based on matrix multiplication in GF
 */
void mix(uint16_t &a, uint16_t &b)
{
    uint16_t a_in = a;
    uint16_t M[2][2] = {{0x0001, 0x0002}, {0x0002, 0x0003}};
    a = gf_multiply(a_in, M[0][0]) ^ gf_multiply(b, M[0][1]);
    b = gf_multiply(a_in, M[1][0]) ^ gf_multiply(b, M[1][1]);
}

/**
 * Apply inverse mixing function based on matrix multiplication in GF
 */
void mixInverse(uint16_t &a, uint16_t &b)
{
    uint16_t a_in = a;
    uint16_t M[2][2] = {{0x0001, 0x0002}, {0x0002, 0x0003}};
    gf_2x2_inverse(M);
    a = gf_multiply(a_in, M[0][0]) ^ gf_multiply(b, M[0][1]);
    b = gf_multiply(a_in, M[1][0]) ^ gf_multiply(b, M[1][1]);
}

/**
 * Perform a bitwise permutation defined by the affine function
 *      pi(x) = alpha * x + beta
 * where gcd(alpha, 512) = 1 to ensure invertibility.
 */
void permuteBits(State &state, int alpha, int beta)
{
    State stateCopy(state);
```

```
    for ( int i = 0; i < WIDTH; ++i ) {
        int newIndex = (alpha*i + beta) & (WIDTH-1);
        if ( stateCopy.getBit(i) )
            state.setBit(newIndex);
        else
            state.clearBit(newIndex);
    }
}

/**
 * Perform inverse bitwise permutation defined by the inverse affine function
 *      pi^{-1}(x) = alphaInverse * (x - beta)
 */
void permuteBitsInverse(State &state, int alphaInverse, int beta)
{
    State stateCopy(state);
    for ( int i = 0; i < WIDTH; ++i ) {
        // i & (n-1) == i mod n for n power of 2
        int newIndex = (alphaInverse*(i-beta)) & (WIDTH-1);
        if ( stateCopy.getBit(i) )
            state.setBit(newIndex);
        else
            state.clearBit(newIndex);
    }
}

/**
 * Permutation step.
 */
void permute(State &state)
{
    permuteBits(state, 31, 15);
}

/**
 * Inverse permutation step.
 */
void permuteInverse(State &state)
{
    permuteBitsInverse(state, 479, 15);
}

/**
 * Add RC_i to the state.
 */
void addRC(State &state, int roundNum)
{
    for (int i = 0; i < NUM_WORDS; ++i)
        state[i] ^= RC[roundNum][i];
}

/**
 * A forward round.
 */
void round(State &state, int roundNum, bool verbose = false)
{
    // S-box
    for ( int i = 0; i < NUM_WORDS; ++i )
        state[i] = sbox_forward(state[i]);
    if ( verbose ) {
        printf("After_S-box:\n");
        state.print();
    }

    // Permute
    permute(state);
    if ( verbose ) {
        printf("After_permute:\n");
        state.print();
    }

    // Mix
```

```
        int k = 0; // Mixer number
        for( int j = 0; j < NUM_WORDS; j += 2 )
            mix(state[j], state[j+1]);
        if( verbose ) {
            printf("After_mix:\n") ;
            state.print();
        }

        // Add round constant
        addRC(state, roundNum);
        if( verbose ) {
            printf("After_addRC:\n") ;
            state.print();
        }
    }

    /**
     * An inverse round.
     */
    void roundInverse(State &state, int roundNum, bool verbose = false)
    {
        // Add round constant
        addRC(state, roundNum);
        if( verbose ) {
            printf("After_addRC_inverse:\n") ;
            state.print();
        }

        // Mix inverse
        int k = 0; // Mixer number
        for( int j = 0; j < NUM_WORDS; j += 2 )
            mixInverse(state[j], state[j+1]);
        if( verbose ) {
            printf("After_mixInverse:\n");
            state.print();
        }

        // Permute inverse
        permuteInverse(state);
        if( verbose ) {
            printf("After_permute_inverse:\n");
            state.print();
        }

        // S-box inverse
        for( int i = 0; i < NUM_WORDS; ++i )
            state[i] = sbox_inverse(state[i]);
        if( verbose ) {
            printf("After_S-box_inverse:\n");
            state.print();
        }
    }

    /**
     * Run the forward permutation.
     */
    void forward(State &state, bool verbose = false)
    {
        for( int i = 0; i < _numRounds; ++i )
            round(state, i, verbose);
    }

    /**
     * Run the inverse permutation.
     */
    void inverse(State &state, bool verbose = false)
    {
        for( int i = 0; i < _numRounds; ++i )
            roundInverse(state, _numRounds - i - 1, verbose);
    }

}; // Permutation
```

```
#endif /* PERMUTATION_H */
```

E.1.6 Sponge.h

```cpp
/**********************************************************************
 * The simplified sponge construction.
 * Padding, if necessary, is assumed to be done at some
 * higher level in the overall system.
 *
 * Author: Matt Kelly
 * Date:   June 2014
 **********************************************************************/
#ifndef SPONGE_H
#define SPONGE_H

#include <assert.h>
#include "State.h"
#include "Permutation.h"

class Sponge {
private:
    State state;
    Permutation f;

public:

    /*
     * Construct a new Sponge with the given number of rounds.
     */
    Sponge(int numRounds)
        : f(numRounds) {}

    /**
     * Absorb input data.
     * Assumptions: input array is correct size
     *              input is a multiple of r = 128
     */
    void absorb(uint16_t input[], int numWords)
    {
        assert( numWords % 8 == 0 );
        int offset = 0;
        while( numWords > 0 ) {
            state.xorOuterState(&input[offset]);
            f.forward(state);
            numWords -= 8;
            offset += 8;
        }
    }

    /**
     * Squeeze output data to desired length.
     * Assumption: output array is correct size
     */
    void squeeze(uint16_t output[], int numWords) {
        // We can take r = 128 bits (8 words) per squeeze
        // after the first 128 bits
        if( numWords <= 8 ) {
            state.copy(output, numWords, 0);
        } else {
            // First 8 words
            state.copy(output, 8, 0);
            numWords -= 8;

            int offset = 8;
            while( numWords >= 8 ) {
                f.forward(state);
                state.copy(output, 8, offset);
                numWords -= 8;
```

```
                offset += 8;
            }

            // Leftover
            if( numWords > 0 ) {
                f.forward(state);
                state.copy(output, numWords, offset);
            }
        }
    }

}; /* SPONGE_H */

#endif
```

E.1.7 Duplex.h

```
/********************************************************************
 * The simplified duplex construction.
 * Padding and domain separation (frame bit handling), if necessary,
 * are assumed to be done at some higher level in the overall system.
 *
 * Author: Matt Kelly
 * Date:   June 2014
 *
 ********************************************************************/

#ifndef DUPLEX_H
#define DUPLEX_H

#include "Sponge.h"

class Duplex {
private:
    Sponge _sponge;
    int _keySize;
    int _isInitialized;

public:

    /**
     * Construct a new Duplex object with the given key size.
     */
    Duplex(int keySize, int numRounds)
        : _sponge(numRounds)
    {
        assert(keySize == 128 || keySize == 256);
        _keySize = keySize;
        _isInitialized = 0;
    }

    /**
     * Set the key by absorbing it with a mute call.
     */
    void setKey(uint16_t key[])
    {
        int numWords = _keySize / 16;
        _sponge.absorb(key, numWords);
    }

    /**
     * Set the IV by absorbing it with a mute call.
     */
    void setIV(uint16_t iv[8])
    {
        _sponge.absorb(iv, 8);
    }

    /**
```

```
     * Set the key and IV with two mute calls.
     */
    void initialize(uint16_t key[], uint16_t iv[8])
    {
        setKey(key);
        setIV(iv);
        _isInitialized = 1;
    }

    /**
     * Duplexing call:
     *    Input may be any size, including 0
     *    Output may be any size, including 0
     * Assumptions: sigma is correct size
     *               sigma is multiple of r = 128
     *               z is correct size
     */
    void duplexing(uint16_t sigma[], int sigmaSize, uint16_t z[], int zSize)
    {
        assert( _isInitialized );
        if( sigmaSize > 0 )
            _sponge.absorb(sigma, sigmaSize);
        if( zSize > 0 )
            _sponge.squeeze(z, zSize);
    }

}; // Duplex

#endif
```

E.1.8 TestPermutation.cpp

```
#include <stdio.h>
#include <stdlib.h> /* exit */
#include "State.h"
#include "Permutation.h"

#define NUM_ROUNDS 16 // default number of rounds

/**
 * Check for avalanche criterion over increasing
 * numbers of rounds
 */
void testAvalancheCriterion(int maxRounds)
{
    for( int round = 1; round <= maxRounds; ++round ) {
        Permutation f(round);
        State zeroStateIn;
        State refStateOut;
        f.forward(refStateOut);
        float hdAvg = 0;
        int hdMax = 0;
        int hdMin = 512;
        for( int i = 0; i < WIDTH; ++i ) {
            State stateIn(zeroStateIn);
            stateIn.setBit(i);
            State stateOut = State(stateIn);
            f.forward(stateOut);
            int hd = refStateOut.getHammingDistance(stateOut);
            hdAvg += hd;
            if( hd > hdMax ) hdMax = hd;
            if( hd < hdMin ) hdMin = hd;
        }
        hdAvg /= 512;
        printf("------ %2d rounds ------\n\n", round );
        printf("Avg HD: %.2f\n", hdAvg );
        printf("Max HD: %d\n", hdMax );
        printf("Min HD: %d\n\n", hdMin );
    }
```

```
}

/**
 * Keep flipping a bit of the initial state and computing the
 * forward and inverse permutations.
 * If all of these tests pass, we have high confidence that the
 * the forward and inverse permutations are correct.
 */
void oneOffInverse()
{

    State state;
    Permutation f(NUM_ROUNDS);

    for( int i = 0; i < 512; ++i ) {
        for(int j = 0; j <= i; ++j )
            state.setBit(j);
        State stateCopy = State(state);
        f.forward(stateCopy, false);
        f.inverse(stateCopy, false);
        if( stateCopy != state ) {
            printf("ERROR: states do not match:\n\n");
            printf("Initial state:\n");
            state.print();
            printf("Final state (after round and roundInverse):\n");
            stateCopy.print();
            exit(1);
        }
        state.clearAll();
    }

    state.setAll();

    for( int i = 0; i < 512; ++i ) {
        for(int j = 0; j <= i; ++j )
            state.clearBit(j);
        State stateCopy = State(state);
        f.forward(stateCopy, false);
        f.inverse(stateCopy, false);
        if( stateCopy != state ) {
            printf("ERROR: states do not match:\n\n");
            printf("Initial state:\n");
            state.print();
            printf("Final state (after round and roundInverse):\n");
            stateCopy.print();
            exit(1);
        }
        state.setAll();
    }

    printf("All inverse tests passed!\n");

}

/**
 * Run tests.
 */
int main()
{
    oneOffInverse();
    testAvalancheCriterion(NUM_ROUNDS);
}
```

E.1.9   DuplexKAT.cpp

```
/*********************************************************************
 * Generates Known Answer Tests (KATs) for the Duplex construction.
 *
 * Author: Matt Kelly
```

```
 * Date:    June 2014
 *****************************************************************/
#include "Duplex.h"

/**
 * Print word array with nice formatting
 */
void printWords(uint16_t arr[], int numWords, FILE *out = stdout)
{
    if( numWords == 0 )
        fprintf(out, "____(empty)\n");
    else {
        for( int i = 0; i < numWords; ++i ) {
            if( i == 0 ) fprintf(out, "____");
            if( i > 0 && i % 8 == 0 ) fprintf(out, "\n____");
            fprintf(out, "%04hx_", arr[i]);
        }
        fprintf(out, "\n");
    }
}

/**
 * XOR arrays A and B together, outputting in D
 */
void xorArrays(uint16_t A[], uint16_t B[], uint16_t D[], int numWords)
{
    for( int i = 0; i < numWords; ++i ) D[i] = A[i] ^ B[i];
}

/**
 * Print a single KAT based on input data.
 * IVs are restricted to 128 bits for simplicity.
 */
void genKAT(uint16_t key[], int keySize, uint16_t iv[8],
            uint16_t A[], int headSize, uint16_t B[], int bodySize,
            int tagSize)
{
    uint16_t CT[bodySize]; // Ciphertext
    uint16_t T[8];         // Tag
    uint16_t Z[bodySize];  // Duplexing output

    assert(keySize == 128 || keySize == 256);

    // Initialize a new Duplex object
    Duplex D(keySize, keySize == 128 ? 10 : 16); // 128-bit key uses 10 rounds
    D.initialize(key, iv);
    // Absorb header and generate keystream
    D.duplexing(A, headSize, Z, bodySize);
    // XOR output with body to product ciphertext
    xorArrays(Z, B, CT, bodySize);
    // Absorb body and generate tag
    D.duplexing(B, bodySize, T, tagSize);

    printf("####################################\n");
    printf("Key:\n");
    printWords(key, keySize / 16);
    printf("IV:\n");
    printWords(iv, 8);
    printf("A_(header):\n");
    printWords(A, headSize);
    printf("B_(body):\n");
    printWords(B, bodySize);
    printf("CT:\n");
    printWords(CT, bodySize);
    printf("Tag:\n");
    printWords(T, tagSize);
    printf("####################################\n\n");
}

/**
 * Generate all KATs for 128-bit key
 */
```

```c
void genKAT_128() {
    uint16_t iv[8]  = {0}; // IV could be any length, but 128-bit is convenient
    uint16_t key[8] = {0};  // 128-bit key
    uint16_t A[64]  = {0}; // Header
    uint16_t B[64]  = {0}; // Body

    genKAT(key, 128, iv, A, 16, B, 32, 8);
    genKAT(key, 128, iv, A, 32, B, 64, 8);
    genKAT(key, 128, iv, A, 0, B, 16, 8);

    // Flip a bit of the key
    uint16_t key2[8] = {0x0000, 0x0000, 0x0000, 0x0000,
                        0x0000, 0x0000, 0x0000, 0x0001};
    genKAT(key2, 128, iv, A, 0, B, 16, 8);

    uint16_t key3[8] = {0xa110, 0xc8b0, 0x1dc0, 0xffee,
                        0xdea1, 0x10c8, 0xa11d, 0xecaf};
    uint16_t iv2[8] = {0x1234, 0x5678, 0x9012, 0x3456,
                       0x7890, 0x1234, 0x5678, 0x9012};
    uint16_t B2[8]  = {0xb0a7, 0x10ad, 0x50fc, 0x0c0a,
                       0x5ca1, 0xab1e, 0xca55, 0xe77e};
    genKAT(key3, 128, iv2, A, 0, B2, 8, 8);
}

/**
 * Generate all KATs for 256-bit key
 */
void genKAT_256() {
    uint16_t iv[8]   = {0}; // IV could be any length, but 128-bit is convenient
    uint16_t key[16] = {0}; // 256-bit key
    uint16_t A[64]   = {0}; // Header
    uint16_t B[64]   = {0}; // Body

    genKAT(key, 256, iv, A, 16, B, 32, 8);
    genKAT(key, 256, iv, A, 32, B, 64, 8);
    genKAT(key, 256, iv, A, 0, B, 16, 8);

    // Flip a bit of the key
    uint16_t key2[16] = {0x0000, 0x0000, 0x0000, 0x0000,
                         0x0000, 0x0000, 0x0000, 0x0000,
                         0x0000, 0x0000, 0x0000, 0x0000,
                         0x0000, 0x0000, 0x0000, 0x0001};
    genKAT(key2, 256, iv, A, 0, B, 16, 8);

    uint16_t key3[16] = {0xa110, 0xc8b0, 0x1dc0, 0xffee,
                         0xdea1, 0x10c8, 0xa11d, 0xecaf,
                         0xa110, 0xc8b0, 0x1dc0, 0xffee,
                         0xdea1, 0x10c8, 0xa11d, 0xecaf};
    uint16_t iv2[8] = {0x1234, 0x5678, 0x9012, 0x3456,
                       0x7890, 0x1234, 0x5678, 0x9012};
    uint16_t B2[8]  = {0xb0a7, 0x10ad, 0x50fc, 0x0c0a,
                       0x5ca1, 0xab1e, 0xca55, 0xe77e};
    genKAT(key3, 256, iv2, A, 0, B2, 8, 8);
}

/**
 * Generate some KATs.
 */
int main()
{
    genKAT_128();
    genKAT_256();
}
```

E.2   Permutation Analyzer

```
"""
Simple tool for analyzing properties of bitwise permutations
```

```python
defined by linear and affine functions.

Author: Matt Kelly
Date:    June 2014
"""
#!/usr/bin/env python

STATE_SIZE = 512
NUM_SBOXES = 32
NUM_MIXERS = 16
SBOX_SIZE  = STATE_SIZE / NUM_SBOXES
MIXER_SIZE = STATE_SIZE / NUM_MIXERS

def gcd(a, b):
    """ Simple GCD computation """
    while b != 0:
        (a, b) = (b, a % b)
    return a

class Bit:
    """
    A bit keeps track of its index history as it
    is modified by the permutation.
    """
    def __init__(self, initialIndex):
        """ Construct a new bit with the given initial index """
        self.initialIndex = initialIndex
        self.indices = [initialIndex]
        self.sboxes  = [initialIndex // SBOX_SIZE]
        self.mixers  = [initialIndex // MIXER_SIZE]
    def getOrder(self):
        """ Get the order of this bit """
        return self.indices[1:].index(self.initialIndex) + 1
    def reset(self):
        """ Reset this bit """
        self.indices = [self.initialIndex]

class Permutation:
    """
    A permutation can either be defined by a linear or an
    affine function.
    This class keeps track of information about the permutation.
    """
    def __init__(self, size, pType, alpha = 0, beta = 0):
        """ Construct a new linear or affine permutation """
        if pType not in ['linear', 'affine']:
            raise TypeError("Invalid permutation type!")
        self.size = size
        self.pType = pType
        self.alpha = alpha
        self.beta = beta
        self.order = 0
        self.bits = [Bit(i) for i in range(size)]
    def __str__(self):
        """ Get a string representation of this permutation """
        retStr = ''
        for bit in self.bits:
            retStr += "%3d: [ " % bit.initialIndex
            for i in range(len(bit.indices)):
                if i == len(bit.indices)-1:
                    retStr += "%3d (%d) " % (bit.indices[i], bit.sboxes[i])
                else:
                    retStr += "%3d (%d), " % (bit.indices[i], bit.sboxes[i])
            retStr += ']\n'
        return retStr
    def permute(self):
        """ Perform the permutation """
        done = False
        order = 0
        while not done and order < self.size:
            done = True
            if self.pType == 'linear':
```

```python
                self.permuteLinearOnce()
            elif self.pType == 'affine':
                self.permuteAffineOnce()
            order += 1
            for bit in self.bits:
                if bit.indices[-1] != bit.initialIndex:
                    done = False
        self.order = order
    def permuteLinearOnce(self):
        """ Perform the linear permutation only once """
        for bit in self.bits:
            # Last bit does not get permuted for linear permutation
            if bit.initialIndex == self.size-1:
                newIndex = bit.initialIndex
            else:
                newIndex = self.alpha * bit.indices[-1] % (self.size-1)
            bit.indices.append(newIndex)
            bit.sboxes.append(newIndex // NUM_SBOXES)
            bit.mixers.append(newIndex // NUM_MIXERS)
    def permuteAffineOnce(self):
        """ Perform the affine permutation only once """
        for bit in self.bits:
            newIndex = (self.alpha * bit.indices[-1] + self.beta) % self.size
            bit.indices.append(newIndex)
            bit.sboxes.append(newIndex // NUM_SBOXES)
            bit.mixers.append(newIndex // NUM_MIXERS)
    def getLowOrderBits(self):
        """ Get all bits with order less than the permutation order """
        lowOrderBits = []
        for bit in self.bits:
            order = bit.getOrder()
            if order < self.order:
                lowOrderBits.append(bit)
        return lowOrderBits
    def getFixedPoints(self):
        """ Get all bits with order 1 """
        fixedPoints = []
        for bit in self.bits:
            if bit.getOrder() == 1:
                fixedPoints.append(bit)
        return fixedPoints
    def hasMaxSboxSuccessors(self):
        """
        Check if this permutation sends all outputs of every S-box to
        different S-boxes
        """
        sboxes = [self.bits[i:i+SBOX_SIZE] for i in xrange(0, self.size, SBOX_SIZE)]
        for sbox in sboxes:
            nextSboxes = []
            for bit in sbox:
                nextSboxes.append(bit.sboxes[1])
            if len(set(nextSboxes)) != SBOX_SIZE:
                return False
        return True
    def hasMaxMixerSuccessors(self):
        """
        Check if this permutation sends all outputs of every S-box to
        different mixers
        """
        sboxes = [self.bits[i:i+SBOX_SIZE] for i in xrange(0, self.size, SBOX_SIZE)]
        for sbox in sboxes:
            nextMixers = []
            for bit in sbox:
                nextMixers.append(bit.mixers[1])
            if len(set(nextMixers)) != SBOX_SIZE:
                return False
        return True
    def isDerangement(self):
        """ This permutation is a derangement if it has no fixed points """
        return len(self.getFixedPoints()) == 0
    def reset(self):
        """ Reset all bits in this permutation"""
```

```
        for bit in self.bits:
            bit.reset()

def testAllLinear():
    """ Analyze all possible linear functions """
    for alpha in range(2, STATE_SIZE):
        if gcd(alpha, STATE_SIZE-1) == 1:
            p = Permutation(STATE_SIZE, 'linear', alpha)
            p.permute()

            lowOrderBits = p.getLowOrderBits()

            print "\na_=_%d:_Derange_=_%d,_Order_=_%d,_LowOrderBits_=_%d,\
MaxSboxSucc_=_%d,_MaxMixerSucc_=_%d\n" % (
                        alpha, p.isDerangement(), p.order, len(lowOrderBits),
                        p.hasMaxSboxSuccessors(), p.hasMaxMixerSuccessors())
            for bit in lowOrderBits:
                print "\t%d_(order_%d):_%s" % (bit.initialIndex, bit.getOrder(), bit.indices)

def testAllAffine():
    """ Analyze all possible affine functions, assuming state size is a power of 2 """
    for alpha in range(1, STATE_SIZE, 2):
        for beta in range(1, STATE_SIZE):
            p = Permutation(STATE_SIZE, 'affine', alpha, beta)
            p.permute()

            lowOrderBits = p.getLowOrderBits()

            print "\na_=_%d,_b_=_%d:_Derange_=_%d,_Order_=_%d,_LowOrderBits_=_%d,\
MaxSboxSucc_=_%d,_MaxMixerSucc_=_%d\n" % (
                        alpha, beta, p.isDerangement(), p.order, len(lowOrderBits),
                        p.hasMaxSboxSuccessors(), p.hasMaxMixerSuccessors())
            for bit in lowOrderBits:
                print "\t%d_(order_%d):_%s" % (bit.initialIndex, bit.getOrder(), bit.indices)

def main():
    #testAllLinear()
    testAllAffine()
if __name__ == '__main__':
    main()
```

E.3 *P-value* Uniformity Test

```
"""
Calculates the uniformity of p-values for each test
and prints numbers of failures.

See http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf
Section 4.2.2

Author: Matt Kelly
Date:   June 2014
"""
from scipy.special import gammainc

# If p-value_T < fail_threshold, uniformity test fails
fail_threshold = 0.0001

for rnd in range(1,17):
    fails = 0
    report_file = "results-stream128/stream128-%drnd/finalAnalysisReport.txt" % rnd
    with open(report_file) as f:
        lines = f.readlines()[7:195] # Only grab relevant lines
        for line in lines:
            pvals = [float(val) for val in line.split()[0:10]]
            samples = sum(pvals)

            chisq = sum([((float(pvals[i]) - (samples/10))**2 / (samples/10)) for i in range(len(pvals))])
```

```
            pval_T = gammainc(4.5, chisq / 2)
            if pval_T < fail_threshold:
                fails += 1

    print "%2d rounds: %d uniformity failures" % (rnd, fails)
```