A Comparison of Conditionally-Secure and Heuristic Symmetric Encryption

by
Reid Wiggins

A thesis presented to the Honors College of Middle Tennessee State University in partial fulfillment of the requirements for graduation from the University Honors College

A Comparison of Conditionally-Secure and Heuristic Symmetric Encryption

by
Reid Wiggins

APPROVED:

Dr. Chrisila Pettey
Chairperson, Computer Science
Project Advisor

Dr. Yi Gu
Assistant Professor of Computer Science
Reader

Dr. Drew Sieg
Assistant Professor of Biology
Resident Honors Scholar

Abstract

We compare conditionally-secure and heuristic constructions in symmetric cryptography on the basis of performance, security level, and feasibility while simultaneously including the requisite mathematical background to understand the comparison.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 General Background

Modern-day cryptography is the scientific study of the creation of tamper-resistant systems through the use of mathematical tools [KL07]. In the information age, cryptography is widely used—it secures credit card transactions, stored passwords, military GPS, telephone calls, and any other sensitive digital information. Thus, while an end-user of a system rarely notices the use of cryptography, one cannot overstate its importance in the modern world.

Historically, the field of *cryptology* was comprised of the creation of secret systems (cryptography) and the breaking of secret systems (cryptanalysis). However, since creating secret systems requires a detailed understanding of how they are broken, many authors roll cryptography and cryptology into the same term (cryptography) while still using cryptanalysis to refer specifically to the breaking of secret systems. We use this terminology in this thesis.

Simplest in cryptography is the notion of *encryption*. Encryption is the process of "scrambling" data in such a way that an eavesdropper cannot determine useful information about the data (notably, what the data means). As an example, cellphone calls are encrypted, so an adversary with a radio who tries to eavesdrop on a call will instead hear only static. See figure 1.1 for more information.

Cryptographers use a variety of methods to build tamper-resistant systems. At the lowest level, we use *cryptographic primitives*. These primitives form the basis for higher-level constructs. For example, we may use a random number generator (RNG) to build an encryption cipher. This RNG would be called a cryptographic primitive. On the other hand, we might also build a cipher from hand without the use of a lower-level primitive. In this scenario, the cipher itself would be known

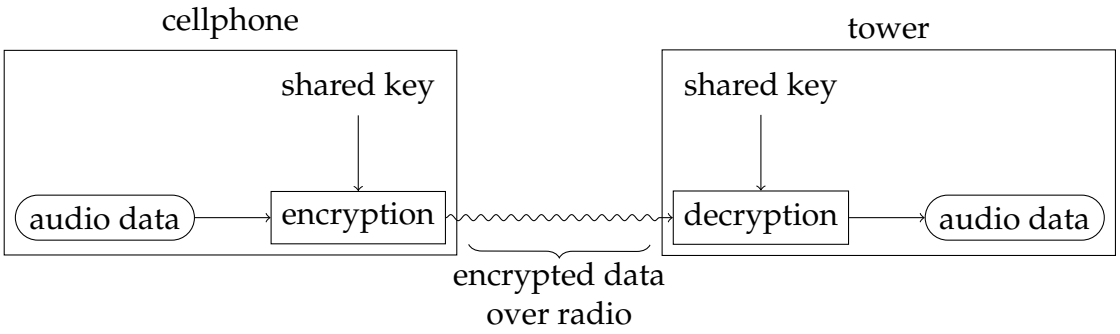

Figure 1.1: Diagram showing encrypted cellphone communication. Note that the data transmitted over radio would be vulnerable to eavesdropping if not for encryption. The shared key is prearranged between the two parties. Since the key is shared, this is known as *symmetric* encryption.

as a cryptographic primitive. From the cipher, we could build more sophisticated constructs; for instance, we might create a scheme to ensure a message has not been tampered with in transit.

Of particular interest are constructions that are *provably secure* contingent on the assumption that the underlying primitive is secure. For example, we might prove that our cipher is secure under the assumption that the primitive we built it on—say, a RNG—is secure. Such a scheme is also called *conditionally secure*, since it is secure under the condition that the primitive is secure [Gol00].

We may divide real-world cryptography into roughly two areas [KL07]. First is mathematical cryptography: this area tends to rely on assumptions from various areas of mathematics, most notably number theory, to build its primitives. Constructions from the realm of mathematical cryptography are almost always conditionally secure, but they usually suffer from being very slow to compute [Dei09]. We tend to leverage mathematical cryptography when our requirements demand we have some underlying mathematical structure for our scheme; for example, asymmetric cryptography concerns itself with the notion of having two keys, public and private, where public can only encrypt and private can only decrypt [KL07]. This structure requires the use of mathematics, as far as modern-day cryptographers can ascertain.

On the other hand, we have the realm of *ad hoc* or *heuristic* cryptography. Here, cryptographers often construct "higher-level" primitives directly instead of basing them on some mathematical assumption. For instance, the widely-renowned block cipher Rijndael [DR02], standardized by the National Institute of Standards and Technology (NIST) in 2001 as the "Advanced Encryption Standard" (AES) [Nat01],

is a primitive all its own. Our belief in the security of Rijndael/AES does not come from some proof or mathematical structure; instead, cryptanalysts (those who study cryptographic structures and attempt to break them) have studied AES extensively and have not yet found any significant flaws.

Mathematics is still used heavily in heuristic cryptography; rather, the divide is determined by what we use as a basis for security. In mathematical cryptography, we use some mathematical problem (say, the discrete logarithm problem) as the basis for security, by assuming it is intractable and then proving that, under that assumption, the construction is secure. On the other hand, in heuristic cryptography, we build some construction and let loose the cryptanalysts upon it. If the construction survives, we have faith in its security.

1.2 Mathematical Preliminaries

1.2.1 Binary

In the real world, we use the *decimal* number system—under which our numbers have "places" that are worth a certain power-of-ten values. For example, the "ones place" is worth 10^0, the "tens place" is worth 10^1, the "hundreds place" is worth 10^2, and so on. So, a number like 1234 is equivalent to the sum $1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$ or $1000 + 200 + 30 + 4$. Thus, our decimal number system is called *base 10*, and numbers written in decimal are sometimes followed by a subscript 10 (for example: 1234_{10}). Here, *deci-* means "ten." Note that each base-10 digit may have a value from 0 to 9.

Modern computers, on the other hand, rely on a *base 2* number system. In this number system, each place is worth a *power of 2*, not a power of 10 like in decimal. Instead of each digit having a value between 0 and 9, each digit instead has a value of 0 or 1. Thus, an example binary number is 10_2, which is equivalent to the (decimal) sum $1 \cdot 2^1 + 0 \cdot 2^0$. So, we have $10_2 = 2_{10}$. That is, 10 in binary is two in decimal.[1] As another example, 1010_2 is equivalent to $2^3 + 2^1 = 8 + 2 = 10$ in decimal.

Each *binary digit* is called a *bit*. Thus, a bit is a single 0 or 1. A group of 8 bits is called a *byte*, which is usually the smallest unit of information in modern-day computing. Multiple bits are sometimes called a *bit string* or *bit stream*, and a bit

[1] This is the basis of the old joke: "There are 10 types of people in the world: those that understand binary and those that don't."

string with n bits is usually called an n-bit string. (For example, a byte may be called an 8-bit string.)

Understanding binary is a fundamental prerequisite for working with modern-day cryptography. Cryptography works exclusively with bits and bytes because we (1) use computers to do cryptography and (2) are interested in the most efficient implementations possible.

Character Encoding

All data ultimately is stored as a string of bits/bytes in a computer, so how is actual text (like this document) stored? The answer is a *character-encoding scheme*. An example character-encoding scheme, once widely used, is ASCII. In ASCII, the capital letter 'A' is defined to be the same as decimal 65. That can be represented in binary as 1000001_2. Other letters, punctuation, and so on have the same style of definition. So long as everyone agrees on a standard character-encoding scheme, everyone can read each others' documents.

Since cryptography is concerned with bits, the details of character encoding are not important. However, since cryptography is frequently used to protect textual data, it is important to know that working with bits does not mean we cannot also work with textual data.

Exclusive-or Operation

The exclusive-or (XOR) operation is used in virtually every symmetric encryption scheme, and so it is represented by the special symbol $\oplus$. It is defined as:

$\oplus$	0	1
0	0	1
1	1	0

Table 1.1: Operation table for exclusive-or

So, the XOR operation returns 1 if the two inputs are different (e.g., $0 \oplus 1 = 1$) and 0 if they are the same (e.g., $1 \oplus 1 = 0$).

To XOR two bit strings together, we simply XOR each corresponding pair of bits individually, much like adding two numbers; see figure 1.2.

The XOR operation has two significant properties. First, a bit string XOR'd with itself simply leads to all zeroes; see figure 1.3.

$$\begin{array}{rr} & 1010 \\ \oplus & 1110 \\ \hline & 0100 \end{array}$$

Figure 1.2: Example of XOR operation with two bit strings

$$\begin{array}{rr} & 1010 \\ \oplus & 1010 \\ \hline & 0000 \end{array}$$

Figure 1.3: Example of XOR operation canceling

Likewise, if you XOR an all-zero bit string with any other bit string, you get the original bit string; see figure 1.4.

$$\begin{array}{rr} & 0000 \\ \oplus & 1010 \\ \hline & 1010 \end{array}$$

Figure 1.4: Example of XOR operation identity element

These two properties may be written algebraically as $x \oplus x = 0$ and $x \oplus 0 = x$. Since XOR is so frequently used in cryptography, it is critical that these two properties be kept in mind.

1.2.2 Permutations

A *permutation* is a function that represents a "shuffling" of the elements in a set. To illustrate what properties are required for a permutation, let us take a simplified card deck $\{1, 2, 3, 4\}$ that has no suits. A permutation is a simple re-ordering of this set. Some possible permuted values are $\{4, 3, 2, 1\}$, $\{3, 1, 2, 4\}$, $\{1, 3, 2, 4\}$, and so on. Elements cannot be missing from a permuted set: $\{1, 2, 3\}$ is not a valid permuted set. Elements also cannot be repeated.

A permutation is a function, however; $\{3, 1, 2, 4\}$ is a set that was created by running each element of the input set $\{1, 2, 3, 4\}$ through a permutation. In this case, if we call the permutation p, then we have

$$p(1) = 3, p(2) = 1, p(3) = 2, p(4) = 4$$

as the permutation. We can also permute the set $\{3, 1, 2, 4\}$ by the same permutation again to get $\{p(3), p(1), p(2), p(4)\} = \{2, 3, 1, 4\}$. And again on that set to arrive at $\{1, 2, 3, 4\}$.

Our above p is just one permutation. How many permutations does a set with n elements have? For the first slot of the output, one may pick any of the n elements to place there. However, the second slot cannot have the same element as the first—that would be a repeated element—so one can only pick from the remaining $n-1$ elements. The third slot cannot have the same element as the first or second slots, so one can only pick from the remaining $n-2$ elements. This argument continues until we get to the last slot available: at that slot, we have already chosen an element for every other slot in the output, so there is a single element remaining.

Thus, the number of permutations for a set with n elements is:

$$n(n-1)(n-2)(n-3)\cdots(3)(2)(1) = n!$$

to be read "n factorial." So, for our above set $\{1,2,3,4\}$ there are $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$ possible permutations.

Consider the set of all possible n-bit strings. Each bit in the string has a value of either 1 or 0, so there are 2^n possible n-bit strings. For instance, there are $2^8 = 256$ different 8-bit strings (bytes). Therefore, there are $(2^8)! = 8.578177753 \times 10^{506}$ different permutations on the set of 8-bit strings. As a preview of coming attractions, we will be working with permutations of 128-bit strings, and $(2^{128})!$ is quite a large number.

1.2.3 Modular Arithmetic

Modular arithmetic is sometimes known as clock arithmetic. In a 12-hour clock, our count looks like:

$$1, 2, 3, \ldots, 11, 12, 1, 2, \ldots, 11, 12, 1, 2, \ldots$$

In other words, our count "rolls over" once it hits 13. Modular arithmetic captures this notion mathematically. In this case, we would be operating *modulo 12*, except that our count starts at zero instead of one:

$$0, 1, 2, 3, \ldots, 10, 11, 0, 1, 2, 3, \ldots$$

The two are equivalent. So, under this system, we would say that 13 is *congruent* to 1 modulo 12, because our count rolls over to 0 at 12. Notationally, we write this:

$$13 \equiv 1 \pmod{12}$$

(The parentheses and their proceeding space are sometimes omitted, depending on context.) Another way to view modular arithmetic is remainder upon division.

When dividing 13 by 12, we have a remainder of 1. With this view, we can say that 24 is congruent to 0 modulo 12 ($24 \equiv 0 \bmod 12$), or that 512 is congruent to 8 modulo 12 ($512 \equiv 8 \bmod 12$).

Naturally, mathematicians are not interested specifically in arithmetic modulo 12. Instead, we consider arithmetic modulo some arbitrary integer depending on context—or in number theory, arithmetic modulo n. As a concrete example, modern computers have fixed-width integers, typically 64 bits wide (they cannot have an infinite number of bits, of course, so we have to stop somewhere). When a computer adds two integers together, the resultant sum may actually be wider than 64 bits ($2^{64} - 1 + 2$ is, for example). This case is known as *overflow*, and typically the result "wraps around" starting again at zero. This is actually modular arithmetic in action: addition in a 64-bit architecture is done modulo 2^{64}.

Modular arithmetic is also used frequently in cryptography. Some areas of cryptography use algebraic number theory to build and study constructions. Since algebraic number theory has roots in modular arithmetic, modular arithmetic forms the foundation of some very widely-used cryptosystems, like RSA and Diffie-Hellman [KL07].

1.2.4 Cyclic Groups

I hesitate to devote much space to cyclic groups, given that they are an advanced topic in abstract algebra, but the bare basics allow for simple one-way functions later on.

In essence, a group is a set with an operation on it, like multiplication, that fulfills some axiomatic definitions. A cyclic group is a group where there is a *generator* element that "creates" the rest of the group. An example best illustrates this notion (without getting too heavy into the theory).

Take the set of integers modulo 7 excluding 0: $\{1, 2, 3, 4, 5, 6\}$. In this set, we can multiply two numbers together modulo 7; for example, $2 \cdot 5 \equiv 3 \bmod 7$. We can also exponentiate numbers, such as $2^3 = 8 \equiv 1 \bmod 7$. A generator is an element where all the powers of that element make up the entire group. In the case of the

integers modulo 7, the generator of the group is 3. Thus, we have:

$$\begin{aligned} 3^0 &\equiv 1 \pmod 7 \\ 3^1 &\equiv 3 \pmod 7 \\ 3^2 &\equiv 2 \pmod 7 \\ 3^3 &\equiv 6 \pmod 7 \\ 3^4 &\equiv 4 \pmod 7 \\ 3^5 &\equiv 5 \pmod 7 \end{aligned}$$

Therefore the set of powers of 3 modulo 7 forms the set of integers modulo 7 precisely. Because such an element exists, the group of integers modulo 7 is defined to be cyclic.

Another way to view this property would be: $f(x) = 3^x \bmod 7$ is a permutation on the integers modulo 7. Because of this property, 3 is called a generator of that group. As it happens, all sets of integers modulo p where p is a prime number are cyclic groups under multiplication, and so if we let g be the generator of that group, the function $f(x) = g^x \bmod p$ is a permutation. (Note: g is not necessarily the same for every p. The choice of p will affect the value of g.)

The precise mathematical details of cyclic groups are beyond the scope of this thesis, and they do not much matter for the following content. For more information, see [Hof+08].

1.2.5 Computational Feasibility

In algorithmic analysis, algorithms are measured in the number of operations or steps needed to solve a particular problem. Modern-day cryptography does not use any constructions that are secure in the presence of an adversary with unlimited resources—though such constructions do exist, as we will see shortly. Instead, we use algorithms that have *computational security*: they are secure against realistic attackers with limited time and memory.

Formally speaking, encryption schemes have an adjustable security level—usually raised and lowered by making the secret key longer or shorter. What makes a construction secure is the asymptotic behavior of adversaries and valid parties as the security level increases. That is, when the security level is low enough, we anticipate the scheme being insecure, and we raise the security level (key length)

to a level where it becomes computationally infeasible for an adversary to break the encryption scheme.

Cobham's thesis, a core assumption in computational complexity theory [AB09], states that efficient algorithms are those that take polynomial time. In this case, specifically, that means time polynomial in the security level. The goal of secure encryption is having valid parties (those with the secret key) able to efficiently use the encryption scheme while having invalid parties (adversaries) unable to efficiently break the encryption scheme.

Thus, we say that a secure encryption scheme must require at most polynomial time from valid parties, and it must require at least super-polynomial-time for adversaries to break. As a concrete example, suppose our security level is x. An encryption scheme might require x^2 operations from a valid party and 2^x operations to break the scheme by an adversary. That scheme would be secure: as the security level x increases, the adversary's 2^x algorithm blows up exponentially while the valid parties' algorithm increases merely quadratically. Because of the growth difference, it costs adversaries dearly when we raise the security level, while the cost to valid parties is next to nothing in comparison. We can keep raising the security level to bankrupt adversaries far before us, even if they have significantly more resources.

This means that if *any* polynomial-time algorithm exists to break the scheme, it is considered insecure. So, if it takes an adversary x^3 operations to break the scheme, that means there exists an efficient (polynomial-time) algorithm to do so. As such, that encryption scheme would be broken.

In this way, we limit ourselves to "realistic" adversaries, since only polynomial-time algorithms are considered "realistic" (or feasible). That is the essence of computational security.

1.2.6 Computational Indistinguishability

Computational indistinguishability is a core principle in theoretical cryptography. Intuitively, computationally indistinguishable functions cannot be "distinguished" by an adversary in polynomial-time (that is, the best adversaries are super-polynomial time). The formal definition of indistinguishability uses a *security game* [KL07].

In the security game, we pick randomly between two functions and allow the adversary to make polynomially-many queries to that function, receiving back the

result of their queries. After they are finished with their queries, we allow them polynomial time to do computation. Once that is over, the adversary must declare which function we allowed access to.

If the adversary can determine the correct answer with odds non-negligibly greater than 50%, then the two constructs are considered distinguishable (usually, this means one of them is broken). Note that the adversary always has at least a 50% chance of guessing correctly, which is why the requirement is that they have a *non-negligibly greater* than 50% chance [KL07].

As an example, there are many constructions in theoretical cryptography that are supposed to be *indistinguishable from random*. What this means is that one of the functions we might give the adversary access to is a real random object—something that genuinely is random. The other is the construction whose security we are trying to prove. From there, one might prove that the adversary cannot possibly have non-negligibly greater than a 50% chance.

This is purely a theoretical definition. Genuinely-random constructions, as a general rule, cannot be efficiently created by mere mortals, and so while we may use in a theoretical sense to prove certain facts, we usually cannot "implement" our security game.

1.3 Definition of Symmetric Encryption

It is somewhat difficult to precisely define encryption. An encryption cipher is a function that takes two inputs: a *plaintext* and a *key*. The plaintext is raw, unprotected data. The key is a secret value that is somehow used to "scramble" the plaintext. The output of the cipher is known as the *ciphertext*—it is the scrambled data that the attacker (hopefully) cannot read. Typically, the plaintext, key, and ciphertext are all strings of bits. The cipher must also have a way to "reverse" the scrambling, given the secret key—otherwise we would never be able to unscramble the data, revealing what it said. (That is the entire point of encryption, of course…)

The difficulty arises when attempting to define *secure* encryption. Take the *ciphertext-only* attack, where an attacker is given just the ciphertext output of a cipher. Is the cipher secure if the attacker cannot recover the entire plaintext? No: it may be that the attacker can learn quite a lot without recovering the entire plaintext, such as (for example) all but a single digit of a social security number.

Instead, we want to formalize the notion that an attacker cannot learn *anything* useful about the plaintext by looking at the ciphertext. The ciphertext should

be a totally opaque string of bits. The standard way to do this is to create an indistinguishability security game [KL07]. In this game, the attacker generates two plaintexts of identical length (the attacker may make the content of the plaintexts whatever they desire). The attacker gives both plaintexts to us. We pick one of the plaintexts uniformly at random, encrypt it with a random secret key, and give the attacker back the ciphertext. If the attacker can determine which plaintext was encrypted with greater than 50% odds, then the cipher is considered broken [KL07]. (The attacker can always guess one or the other, so they always have a 50% chance. The requirement states that they must have a chance non-negligibly greater than 50%.)

Why does this game capture the notion of not being able to learn anything about the plaintext given a ciphertext? Consider an example: Suppose an attacker, by looking at the ciphertext, can learn if the plaintext (which was in binary) was even or odd. Then when creating the two plaintexts in the game, they would make one which was even and the other which was odd. When they are given back the ciphertext, they can determine if the plaintext was even or odd—and thus distinguish which plaintext was encrypted. Therefore, that cipher would be considered broken despite the fact that the attacker can only learn marginally useful information.

In other words, secure encryption seeks to protect *all* information about the plaintext from the attacker. Some information that leaks may not actually impact the security of an application; for example, in the above even-odd case, bank transactions would be mostly unaffected (who cares if an attacker can learn if you deposited an even or odd amount of money into your account). Nonetheless, the point is that, ahead of time, cryptographers cannot be sure what information leaks actually matter: we'd rather be safe and protect everything.

1.4 The One-Time Pad

The one-time pad is a special encryption scheme. It has the property of being *unconditionally* secure (or *information-theoretically*) secure—that is, the one-time pad is *provably* secure [KL07].

In the one-time pad, the input plaintext p is a string of bits. The key k is a uniformly, truly random string of bits that is equal in length to the plaintext—and the key *cannot* ever be re-used (it is *one-time*). Then the ciphertext is simply $p \oplus k$. That is, figure 1.2 could also be an example of the one-time pad with 1010 as the

plaintext and 1110 as the key. Given a ciphertext $c = p \oplus k$ and the key k, we can compute $c \oplus k = p \oplus k \oplus k = p \oplus 0 = p$ to recover the plaintext. Thus, our encryption scheme works.

Why is the one-time pad secure? Assuming that the key is never re-used, an attacker simply does not have enough information to determine potential plaintexts from the ciphertext. Given a ciphertext c, an attacker can find a key k for any possible plaintext p. That is, *all plaintexts are equally likely to be correct*. Even if an attacker enumerates through all possible keys, they cannot determine what putative plaintext "looks" correct—this is sometimes called *perfect secrecy*. This property is possible because the key is exactly the same length as the plaintext and is uniformly, truly randomly chosen. For a more detailed explanation, refer to any standard text such as [KL07].

1.4.1 Limitations of the One-Time Pad

Historically, three limitations are identified for the one-time pad. First, the key must be precisely the same length as the plaintext. However, since both parties in communication need the key, how does one distribute the key? If we have some secure channel by which we can communicate the key, why not just communicate the message instead? However, this limitation applies to all symmetric encryption schemes.

Second, since the key must be as long as the plaintext and must never be re-used, a large amount of keying material is needed, so if messages are exchanged frequently, large keys must be pre-arranged frequently. For every bit that is securely sent, there must be a key bit. This ties into the third issue: how does one generate so many truly-random bits? In general, generating truly random data quickly is difficult, especially on resource-limited platforms like smartphones.

These limitations are all direct results of the key length being equal to the plaintext length. Claude Shannon proved that this condition must be met to have unconditionally-secure encryption [Sha49], so no better solution exists for unconditional security. Unfortunately, the limitations of the one-time pad make it impractical for daily widespread use. However, supposedly, communication between the US and Russian presidents was once secured by a one-time pad [MVO96].

1.5 Realistic Symmetric Ciphers

Since unconditional security is impractical, we turn toward *computational security*. With computational security, we abandon the notion of an unlimited attacker and instead consider a much more realistic model: *limited* attackers—ones that are restricted to polynomial time.

Two types of symmetric ciphers are used frequently. The first is known as a stream cipher. Stream ciphers seek to take a small input key (256 bits or so) and "expand" that key into a long, pseudorandom string that is then XOR'd with the plaintext string [MVO96]. This is directly inspired by the one-time pad, but since the pseudorandom string is not truly random, it does not qualify as a one-time pad, and thus is not unconditionally secure.

Block ciphers are the second commonly used symmetric cipher. A block cipher takes an input key k and a *block* of input bits b bits wide and outputs a ciphertext block that is b bits wide. A block cipher is actually a family of permutations on the set of b-bit strings. (How one applies a block cipher to *multiple* blocks of text is a tricky matter; see block cipher modes of operation in [KL07] for more information.)

In the modern day, the line between stream ciphers and block ciphers is very blurred: stream ciphers have block-cipher-like properties while block ciphers are frequently used in modes of operation that turn them into a stream cipher. Whether to use a stream cipher versus a block cipher mostly depends on the precise hardware characteristics of the machine the cipher will be running on, how "modern" a cryptographer wants their cipher to be, and if any security certifications are required (e.g., one of the few ciphers certified for working with United States TOP SECRET data is AES [DR02], which we will study in chapter 3).

In this thesis, we will focus on a discussion of conditionally-secure constructions of ciphers (chapter 2) versus heuristic constructions of ciphers (chapter 3). A formal comparison of their efficiency, security, and availability can be found in chapter 4.

Chapter 2

Conditionally-Secure Constructions

Conditionally-secure constructions are those which have a concise, well-stated mathematical assumption by which their security is proven. They are so-called because, on the condition that the assumption is true, the construction is secure.

Such constructions are often-used in asymmetric cryptography. In asymmetric cryptography, we consider two different keys that are mathematically related: usually one of these is called the public key and the other the private key [KL07]. An example is a protocol known as Diffie-Hellman Key Exchange, invented by Whitfield Diffie, Martin Hellman, and Ralph Merkle in 1976 [Sin99]. Diffie-Hellman key exchange uses the assumption that given $g^x \mod p$ and $g^y \mod p$, it is computationally infeasible to find $g^{xy} \mod p$—this assumption is known as the Diffie-Hellman problem [Hof+08], for obvious reasons.

However, conditionally-secure schemes can be created for symmetric cryptography as well, by introducing *one-way functions* [Gol00]. The theoretical implications of this fact are huge: as Katz and Lindell say, "*the existence of one-way functions is equivalent to the existence of all (non-trivial) [symmetric] cryptography.* This is one of the major contributions of modern cryptography" [KL07] (emphasis theirs).

In this chapter, we describe one-way functions and show how a candidate one-way function can be used to construct a stream cipher and a block cipher.

2.1 One-way Functions

A one-way function f is a function that is easy to compute but difficult to invert. Formally, given x, $f(x)$ may be computed in polynomial time; on the other hand, given $f(x)$, an x' such that $f(x) = f(x')$ cannot be found in polynomial time (with non-negligible probability) [KL07]. See figure 2.1.

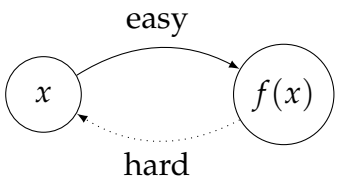

Figure 2.1: Diagram showing the difficulty of computing one value given another in the context of a one-way function f. To underscore this picture's importance, it is on the cover of [Gol00].

We do not know if any one-way functions exist [KL07]. Proving that they exist would be a massive breakthrough in computational complexity theory. For context, consider the famous $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ problem. It is one of the seven illustrious Millennium Prize Problems, each of which has a $1 million prize from the Clay Mathematics Institute (CMI) for its solution; as the CMI says: "The Prizes were conceived to record some of the most difficult problems with which mathematicians were grappling at the turn of the second millennium" [Ins00]. Only a single Millennium Prize Problem has been solved—and that was not $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$. So, the $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ problem is extraordinarily difficult. Yet despite the difficulty of the $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ problem, proving that $\mathcal{P} \neq \mathcal{NP}$ would *still* be insufficient to prove that one-way functions exist [KL07]. Thus, if one of the most difficult problems in computer science is strictly weaker than a proof of one-way functions, we truly have little hope of a proof in the near future.

With this in mind, we must instead consider *candidate* one-way functions. Fortunately, cryptographers have many candidates. Given a prime p and a generator g of the multiplicative group of integers modulo p, the function $f(x) = g^x \pmod{p}$ is conjectured to be a one-way function [KL07]—and at that, also a permutation. For our purposes, this candidate one-way permutation will do nicely, but others exist; see [Gol00; KL07] for others. (As a general rule, one may construct candidate one-way functions from practically every popular mathematical discipline, so there are *many* candidates.)

If we assume our above f is a one-way function, then as per [Gol00; KL07], we may construct the whole of symmetric cryptography using f, and moreover, those constructions will be conditionally secure. As mentioned previously, we will use f to construct a stream cipher and a block cipher.

2.2 Constructing a Stream Cipher

Before constructing the stream cipher, we must understand two more concepts—the hard-core predicate and the pseudorandom generator.

2.2.1 Hard-core Predicates

With a one-way function f, if we are given an $f(x)$, it is hard to find an x' such that $f(x') = f(x)$. Does that mean f conceals all information about x? Absolutely not. For example, let g be a one-way function. Then let $h(x) = (g(x), p(x))$ where

$$p(x) = \begin{cases} 0 & \text{if } x \text{ is even} \\ 1 & \text{if } x \text{ is odd} \end{cases}$$

This h is also a one-way function—yet it very clearly leaks information about x, namely whether x is even or odd.

A hard-core predicate is a single-bit-function (like p above) that captures the information "hidden" by f. Formally speaking, a predicate $b(x)$ is a hard-core predicate of a one-way function f if any adversary given $f(x)$ for some random x cannot compute $b(x)$ with probability non-negligibly greater than $1/2$ in polynomial time [Gol00]. See figure 2.2 for a visual explanation.

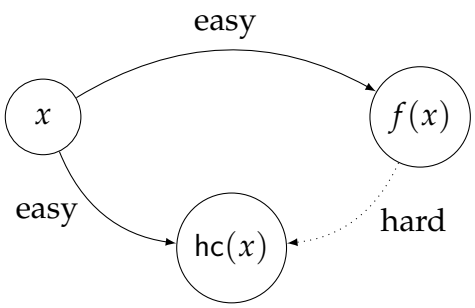

Figure 2.2: Diagram showing the difficulty of computing certain values relative to another in the context of a one-way function f with a hard-core predicate $\mathsf{hc}(x)$.

As a general rule, intuitive guesses at what might be a hard-core predicate for a given one-way function tend to be wrong [KL07]. However, we can prove that hard-core predicates for *any* one-way function exist [Gol00], and we know several "specialized" hard-core predicates for common number-theoretic problems [Gol00].

For our one-way function case study, $f(x) = g^x \bmod p$, we have

$$\mathsf{hc}(x) = \begin{cases} 0 & x \leq p/2 \\ 1 & x > p/2 \end{cases}$$

as a simple, provable hard-core predicate [Gol00]. Thus, we can say that f *hides* whether the original x was greater than or less than half the prime we're using—or at least, it does if f is a one-way function.

2.2.2 Pseudorandom Generators

A pseudorandom generator (PRG) seeks to take an input and *expand* that input with bits that are indistinguishable from random (in polynomial time and so forth). A PRG has an *expansion factor* $\ell(n)$. This function represents how *much* the PRG will expand the input. For example, a PRG with expansion factor $\ell(n) = n + 1$ will add a single pseudorandom bit to the input. On the other hand, $\ell(n) = 2n$ will double its input.

With a PRG, we are building toward a stream cipher. Recall that a stream cipher takes a small key and expands that key, somehow, into a stream that is indistinguishable from random—thereby using the same security argument as the one-time pad. The difference between a generic PRG and a stream cipher is that a stream cipher is supposed to be able to generate an unbounded length pseudorandom stream, whereas a PRG has a specific expansion factor.

Given a one-way permutation f, we may easily construct a provably-secure pseudorandom generator [KL07]. The standard construction, as given in [KL07], is given a uniformly randomly distributed n-bit string s, consider

$$G(s) = f(s)||\mathsf{hc}(s)$$

where f is a one-way permutation and hc is a hard-core predicate for f. (Here, the vertical bars $||$ mean concatenation, i.e., simply "gluing" the two functions together as bit strings.)

The intuitive explanation for the security of this construction, again given in [KL07], is:

> [Note] first that the initial n bits of the output of $G(s)$ (i.e., the bits of $f(s)$) are truly uniformly distributed when s is uniformly distributed, by virtue of the fact that f is a permutation. Next, the fact that hc is a hard-core predicate of f means that $\mathsf{hc}(s)$ "looks random"—i.e., is

pseudorandom—even given $f(s)$ (assuming again that s is uniform). Putting these observations together, we see that the entire output of G is pseudorandom.

Note that our $G(s)$ has an output length of $n+1$ but an input length of n. This means we have created a pseudorandom generator with expansion factor $\ell(n) = n+1$. This construction may also be formally proven secure, assuming that f is a one-way permutation, of course; see [Gol00].

With this G, we already have the means by which to securely encrypt $n+1$ bits given an n-bit key: let $\mathsf{Enc}_k(x) = G(k) \oplus x$ where k is $n-1$ bits wide and x is n bits wide. This Enc is a fixed-length stream cipher—one that is conditionally secure—that can encrypt $n+1$ bits given an n-bit key. However, the overhead of using this method to encrypt $n+1$ bits is exorbitant: it would be better to generate an extra bit and use a one-time pad instead. Nonetheless, this G will serve as the basis for PRGs with an arbitrary-length expansion factor.

To expand $\ell(n)$ to an arbitrary polynomial-length expansion factor, we use a fixed beginning *seed* value that is uniformly random. We run that seed value through a PRG G with expansion factor $\ell(n) = n+1$. For every round, we take the extra bit and emit it as a pseudorandom bit, then we take the seed from that round as the seed for the next round. At the end of the rounds, we have emitted n bits plus the width of the last round's seed, thereby creating a PRG with an arbitrary expansion factor. See figure 2.3 for a visual explanation.

2.2.3 The Stream Cipher

Armed with this algorithm, we can create an arbitrary-length stream cipher. To encrypt a plaintext w-bit string p, we first select a uniformly random key k. This k will be the initial seed for the pseudorandom generator. We continually make calls to our expansion factor $\ell(n) = n+1$ pseudorandom generator to build up a w-bit pseudorandom stream[1]. Then we encrypt the plaintext by computing $p \oplus G_w(k)$, where $G_w(k)$ represents the w-bit output stream of the arbitrary expansion factor PRG G. So long as $G_w(k)$ is indistinguishable from random, this has the same security guarantees as the one-time pad, yet with a much smaller key. A proof of the algorithm's security under the assumption that f is a one-way permutation can be found in [KL07].

[1] When used for encryption, a PRG's output stream is sometimes called a *keystream*.

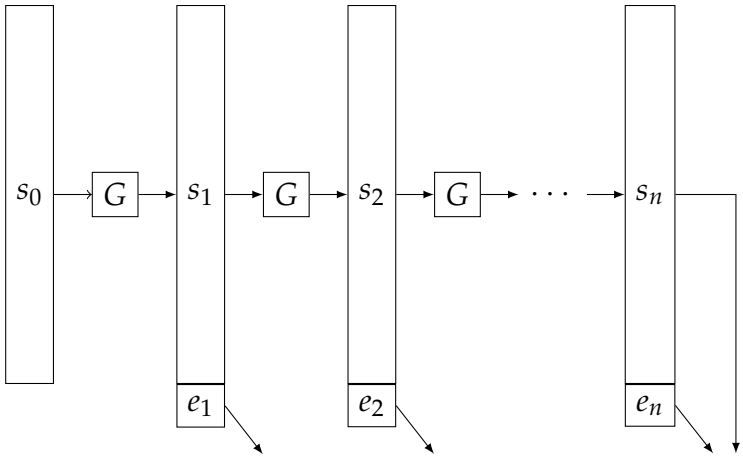

Figure 2.3: Diagram showing how one can create an arbitrarily-long expansion factor for a pseudorandom generator using a pseudorandom generator G with expansion factor $\ell(n)$. In this case, for each extra bit e_i that we create, we have another evaluation of the original G. In each round, the seed for that round is used as the seed for the next round's computation. Algorithm and figure were inspired by a combination of [Gol00; KL07].

While that security guarantee is predicated on the assumption that we have a one-way function f, because we have candidate one-way functions that are as yet unbroken, this is still a potentially useful construction—certainly so from a theoretical standpoint. Another useful consequence of studying pseudorandom generators is that they are the theoretical model for real world stream ciphers; by studying PRGs, we can provide formal security definitions and analyses for stream ciphers.

Goldwater takes a broader view of PRGs in [Gol00]: many cryptographic applications require high-quality randomness, but high-quality randomness is computationally expensive to generate. In any case that requires such randomness, we may instead "amplify" a small amount of randomness into a large, indistinguishable-from-random stream of randomness without any thought for the consequences (since the scheme is conditionally secure)!

Unfortunately, this scheme is not without its costs. Since $G(s) = f(s)||\mathsf{hc}(s)$, if we want an expansion factor of $\ell(n) = n + m$, we must evaluate both the one-way permutation and the hard-core predicate m times (see figure 2.3); this evaluation is, as a typical rule, not cheap. Even for a relatively simple one-way permutation like $f(x) = g^x \bmod p$, generating m extra bits requires m modular exponentiations and m comparisons. Modular exponentiation can be done in $\mathcal{O}(\log e)$ time (where e is

the exponent), which is not too bad, but this will still not compare favorably to a real-world stream cipher—as we will see in chapter 4.

2.3 Constructing a Block Cipher

To create a block cipher, we can use the constructs described in section 2.2, Constructing a Stream Cipher. Specifically, using a one-way function f and a PRG G, it is possible to construct a conditionally-secure block cipher. To do this, we need two more constructions—a pseudorandom function and a pseudorandom permutation.

2.3.1 Pseudorandom Functions

Pseudorandom functions are motivated by the stream nature of a pseudorandom generator's output. Imagine a system encrypted an entire hard-disk drive with a pseudorandom generator as a stream cipher. In a non-encrypted scenario, if the system wishes to retrieve a file that is stored 75% of the way through the disk, the system may merely request that the disk seek to that location, read the sector, and return it. Crucially, however, in the case of a pseudorandom generator, the system cannot decrypt that sector in disk without computing the portion of the keystream that was used to encrypt that sector because each bit of the keystream depends on the bits that came before it. That is, the system may be forced to compute nearly the entire keystream to access data near the end of the ciphertext.

This situation is obviously non-optimal: it transforms a medium that can seek to other locations with relative quickness (a hard-disk drive) into a medium that is now forced to sequentially seek. One might as well use a cassette tape. Even worse, the advantages of expensive truly random-access media like solid-state drives would be eliminated. This is too high a cost for encryption.

Pseudorandom functions are essentially an efficient way to index the keystream produced by a pseudorandom generator. Instead of being forced to generate the entire keystream to read off blocks near the end, a pseudorandom function allows the system to directly compute portions of the keystream near the end—or at least, it makes doing so much more efficient than it would be otherwise.

With that motivation, we may define a pseudorandom function. A pseudorandom function F_k is a fixed-length input, fixed-length output *keyed* function that is indistinguishable from a truly random function. (A PRF cannot be used without a key, which is why we notationally write F_k instead of F.) We may implement

a PRF by creating a way to efficiently index the output from a PRG—although the definition does not require such an implementation, as a PRF is meant to be a concept of its own.

In section 2.2.2, we explained a method by which we can have an arbitrary-expansion-factor PRG. To build a PRF, we will consider a PRG with expansion factor $\ell(n) = 2n$, which still fits the polynomial expansion factor requirement from section 2.2.2. Let this PRG be called G. This G will double the input received. Our PRF will accept n bits of input and output n bits as well.

First, split the PRG's output into two blocks:

$$G_{2n}(s) = H_0(s)||H_1(s)$$

where again $||$ means concatenation and $H_0(s)$ and $H_1(s)$ are the same length. Since $\ell(n) = 2n$, the length of both $H_0(s)$ and $H_1(s)$ must be n. As an example, if $s = 110$ and $G_{2n}(s) = 101011$, then we have

$$\underbrace{101011}_{G_{2n}(s)} = \underbrace{101}_{H_0(s)}\underbrace{011}_{H_1(s)}$$

Next, consider the input s in binary. Each bit will be either a 0 or 1, as usual. We may use this—along with our split-in-half $G_{2n}(s)$—to construct a binary tree. At the root of our binary tree will be our key k: see figure 2.4.

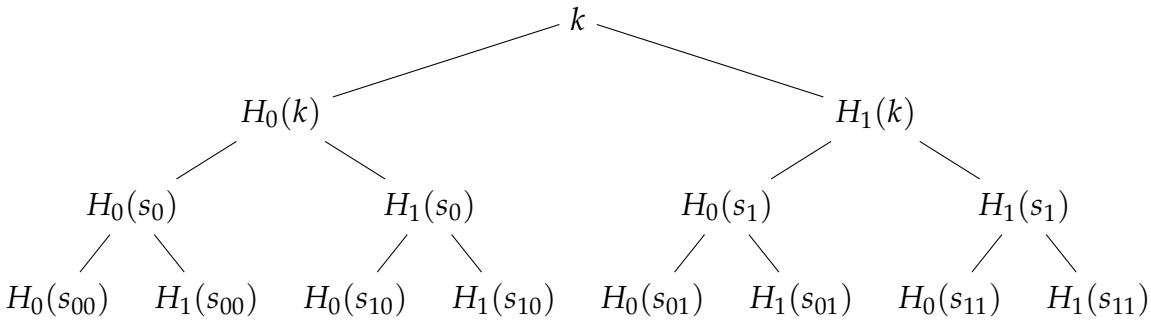

Figure 2.4: This diagram represents the construction of an n-bit indistinguishable-from-random string given a random key k and an input. At each junction of the tree, we use a bit from the input to decide whether to go left or right. Each node of the tree is computed from that node's parent. An example input might be 011, in which case we would have the result $H_0(H_1(H_1(k)))$. Figure inspired by [Gol00; KL07].

In the case of disk or file encryption, this scheme allows us to compute blocks of the keystream with only n evaluations of G. Recall that each extra bit of output

from a PRG, using our construction, requires another evaluation of the underlying one-way permutation. So, we have n evaluations of the underlying one-way permutation each time G is evaluated, and n evaluations of G for every evaluation of F_k, our PRF. So, evaluating F_k on an input takes n^2 evaluations of the underlying one-way permutation, where n is the bit length of the input/output.

We may use this PRF to implement a "seekable" stream cipher, as above. However, we have another option: We may also use the PRF to implement a pseudorandom permutation, which is the theoretical model of a block cipher.

2.3.2 Pseudorandom Permutations

A pseudorandom permutation (PRP) is the theoretical model of a block cipher. It acts identically to a pseudorandom function except that it is also a permutation, and a PRP can be built out of a PRF relatively simply.

Pseudorandom permutations are useful because they are a flexible construction, specifically because they may easily be used to build other more complex constructions. For example, cryptographic hash functions (while not covered here) are widely used in the industry. One method of building a cryptographic hash function is the Merkle-Damgård construction [KL07], which requires the use of a one-way compression function. A pseudorandom permutation (or block cipher) can be transformed into a one-way compression function in many different ways, with the most popular being the Davies-Meyer, Matyas-Meyer-Oseas, and Miyaguchi-Preneel constructions [DR02; MVO96], all of which are provably secure under the assumption that the block cipher is ideal [MVO96]. This is but one example of the utility of a block cipher.

To construct a pseudorandom permutation from a pseudorandom function F_k, we use a three-round Feistel network [KL07], as demonstrated in figure 2.5. Unfortunately, the proof of the Feistel network's correctness is beyond the scope of this thesis; it may be found in either [Gol00; KL07].

2.3.3 The Block Cipher

Given a pseudorandom permutation P, we may define a block cipher as $P_k(x)$. This will take a uniformly random key and an input plaintext block and return an output ciphertext block. Using $P_k(x)$ to directly encrypt text is actually ill-advised: how one handles plaintexts consisting of multiple (possibly incomplete) blocks is a matter left unsolved, and careful thought must be given to such a scenario. See [KL07] for

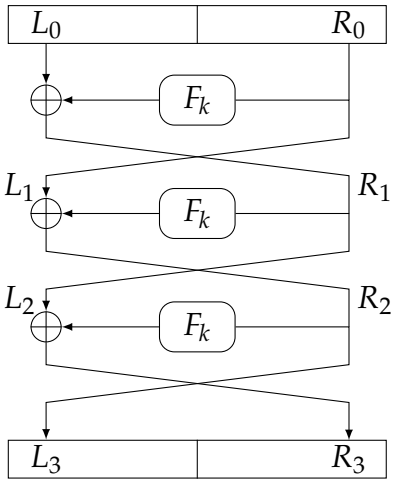

Figure 2.5: Diagram showing use of a Feistel network to construct a pseudorandom permutation from a pseudorandom function. We split the original input into two halves, run the right half through the PRF, XOR the result with the left half, swap the halves, and repeat. Figured inspired by [KL07].

a discussion of block cipher modes of operation and padding schemes, which are required to safely encrypt data using a block cipher.

Notice that we evaluate the pseudorandom function F_k three times in the above scheme. Recall that in section 2.3.1, we derived that evaluating F_k takes n^2 evaluations of the underlying one-way function. In the case of a PRP, we evaluate F_k three times, so we have $3n^2$ evaluations of the underlying one-way function for each evaluation of P_k.

Chapter 3

Heuristic Constructions

Heuristic constructions are those that cryptographers build directly without using a compact mathematical assumption (like some f being a one-way function). Instead, as an example, one might assume that an ad hoc function they build is itself a pseudorandom function, and then from there use that PRF in a Feistel network to build a PRP. Alternately, no assumption may be made at all, and the construction is considered secure simply because no cryptanalysts have been able to break it yet.

Heuristic constructions have many advantages, but primarily, they are faster than the conditionally-secure constructions we presented in chapter 2. In this chapter, we will present a brief overview of some common heuristic constructions so as to demonstrate their nature, mostly by selectively quoting the authoritative works for each construction.

3.1 Salsa20

Salsa20 is a state-of-the-art stream cipher designed by Daniel Bernstein; detailed information may be found in its design paper, [Ber08b]. No one summarizes the algorithm better than Bernstein himself in [Ber08b]:

> Salsa20 expands a 256-bit key and a 64-bit nonce (unique message number) into a 2^{70}-byte stream. It encrypts a b-byte plaintext by xor'ing the plaintext with the first b bytes of the stream and discarding the rest of the stream. It decrypts a b-byte ciphertext by xor'ing the ciphertext with the first b-bytes of the stream.
>
> There is no feedback from the plaintext or ciphertext into the stream. Salsa20 generates the stream in 64-byte (512-bit) blocks. Each block is an

> independent hash of the key, the nonce, and a 64-bit block number; there is no chaining from one block to the next. The Salsa20 output stream can therefore be accessed randomly, and any number of blocks can be computed in parallel.

This summary serves as the first real introduction into the world of "concrete" cryptography in this thesis. One may see that seeking in PRGs is already addressed by Bernstein. Note how many bytes can be safely generated by use of a single 256-bit key: 2^{70} bytes—one full zebibyte! This is not far off from being able to encrypt all human data on Earth.

Salsa20 has no underlying security assumption. Rather, Daniel Bernstein is a well-known cryptographer, and so if he builds a construction that he himself cannot break, that is a good indication that it is probably secure—this is enough to generate the initial interest in the scheme. From there, other well-known cryptanalysts attempt to break the cipher. If they do not succeed in creating even semi-practical attacks, then the construction is considered secure. That is the case with Salsa20.

We will present a brief taste of Salsa20, but not a full algorithmic description. Salsa20 has a 4×4 matrix of internal state, where each element in the matrix is 4 bytes in size, for a total state size of $4^3 = 64$ bytes. The initial matrix state is built from the key, nonce, block counter, and a few constants. Next, a series of rounds are executed using the state matrix. Salsa20's round primitive is the function

$$b := b \oplus ((a \boxplus c) \lll k))$$

where $\oplus$ is the XOR operation, $\boxplus$ is addition modulo 2^{32}, and $\lll$ is a left rotation. Here, k is not a key, but rather a round-specific constant (hard-coded into the algorithm itself). This round primitive is used to update 8 elements of the state matrix every round; those 8 elements form a diagonal pattern in the state matrix. In particular, the b element has its value XOR'd with the modular addition of two other elements a and c after left-rotation by a constant k.

This process of updating 8 different elements is repeated 20 times, and then the original state matrix is added to the final post-round matrix. This forms a 64-byte block.

As one can see, a heuristic primitive like Salsa20 has a much less clearly understood mathematical structure than something akin to the PRG construction we saw back in section 2.2.2. Yet despite this, every element of Salsa20's design was carefully considered, even down to the specific rotation constants in the round primitive, by Bernstein.

In 2008, Bernstein published called ChaCha20 [Ber08a]. In ChaCha20, Bernstein changes the round primitive to

$$b := (b \oplus (a \boxplus c)) \lll k$$

in an attempt to increase bit-diffusion and re-defines the ordering of the state matrix bytes in an attempt to thwart trail-width analysis. As one can see, in heuristic constructions, there is much less emphasis on proofs of security and much more emphasis on addressing real-world cryptanalytic techniques.

3.2 Data Encryption Standard

The Data Encryption Standard (DES) is a block cipher developed and standardized in the 1970s by IBM and the NIST (at the time, known as the National Bureau of Standards) [KL07]. It is widely known as being the first high-grade cipher completely available to the public [MVO96].

DES has many interesting facets to discuss and is of great historical significance. First of all, DES only has a 56-bit key, which is far too small for modern-day use. Some believe that at the time of its design, its key size was intentionally reduced by the National Security Agency (NSA) from 64 bits to 56 bits to enable easier attacks [KL07; MVO96].

DES is a Feistel network, as discussed in section 2.3.2. However, DES is a 16-round Feistel network instead of a 3-round (like the theoretically perfect PRP). DES's core function, built much the same way as Salsa20 (heuristically) and demonstrated in figure 3.1, is simply assumed to be a pseudorandom function. As it turns out, this assumption is not very good: we can pretty easily distinguish the core function f from random, and there are (albeit theoretical) weaknesses in DES because of the core function's weaknesses.

In DES, in each round of the Feistel structure, a different round subkey is used. These round subkeys are generated by an algorithm known as the *key schedule* for DES. A secure key schedule is another key component in creating a cipher to stand the test of time, as an insecure key schedule may lead to loss of security if not all bits of the key are fully utilized. DES's key schedule is demonstrated in figure 3.2.

In the core function, eight sets of 6-bits go through a substitution step in which they are each replaced with 4-bits via a lookup table. This lookup table contains what are known as S-boxes (demonstrated as the boxes containing s_i in figure 3.1), which map 6-bit strings to 4-bits; they form the core of the security for DES, since

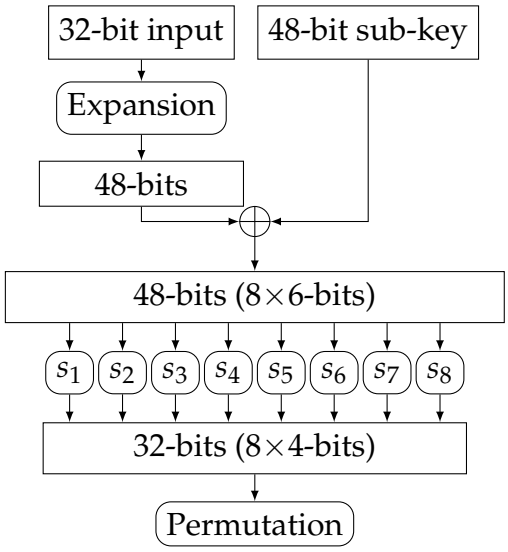

Figure 3.1: Diagram demonstrating the core function for DES, sometimes known as the DES mangler function. This function is assumed to be a pseudorandom function and is used as the PRF in a 16-round Feistel network.

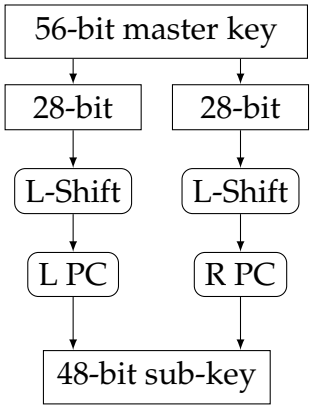

Figure 3.2: Diagram showing the key schedule of DES. The Left Permuted Choice and Right Permuted Choice boxes indicate the selection of particular bits from the input to those boxes; 24 bits of each 28 bit key-half are selected in any given round.

without the S-boxes, DES would be trivially breakable. Those S-boxes are hard-coded into the algorithm design.

The S-boxes have two historically interesting properties. First of all, when differential cryptanalysis was discovered by the general public in the early 1990s, researchers found that DES (designed in the 1970s) was already resistant to it [KL07]. This led to speculation that IBM had independently discovered differential cryptanalysis before the general public and had kept its existence secret. IBM researchers later admitted that this was the case [KL07].

Secondly, the NSA made unexplained changes to the S-boxes before DES was released to the general public. At the time, cryptographers were concerned that these unexplained changes might negatively affect the strength of DES. As it happens, however, the NSA's S-box changes ended up improving DES's resistance to linear cryptanalysis, which was discovered in the mid-1990s [KL07].

DES has also received far-and-away the most scrutiny of any cipher—ever. Despite that scrutiny, in practice, the most realistic attack on DES is still a brute-force enumeration of all possible 2^{56} keys [KL07]. Thus, DES is an extraordinarily well-designed block cipher, though it has been superseded in practice thanks to the small key size. There are methods by which one may extend the key length of DES (by using multiple invocations of DES cascaded on one another, like Triple DES), but these methods suffer from extreme slowness.

3.3 Advanced Encryption Standard

The Advanced Encryption Standard (AES) is the replacement for DES; it was standardized in 2001 by the NIST. A detailed design justification can be found in the book written by its creators, John Daemen and Vincent Rijmen, *The Design of Rijndael* [DR02]. AES was selected out of many candidate block ciphers in a competition held by the NIST; its original name was Rijndael, a play on the name of the two authors.

AES accepts three different key sizes (128, 192, and 256 bits) and has a block size of 128 bits. Depending on the key size, the cipher has a different number of rounds: for 128 bits, there are 10 rounds; for 192 bits, there are 12; for 256 bits, there are 14. Let the number of rounds be called N_r in the spirit of [DR02]. Like Salsa20, there is a 4×4 state matrix that is used in each round; this matrix is initialized with the input to the cipher (the plaintext), which is 128 bits or 16 bytes.

Then the AES algorithm is comprised of the following four steps:

(1) Key expansion: We create $N_r + 1$ round-specific keys from the master key using an algorithm known as the *key schedule*. Call each round specific key K_i, so we have $K_0, K_1, \ldots, K_{N_r}$.

(2) Add round key: We XOR the first round-specific key K_0 with the state matrix, byte by byte. When XORing a derived key with the state, we are using portions of the master key to directly provide security for the cipher.

(3) Rounds: We perform $N_r - 1$ rounds on the state matrix with each round using a separate round-specific key (K_1 through K_{N_r-1}). Each round does four steps:

    (1) SubBytes: We use a set of nonlinear S-boxes, hard-coded into the algorithm, to substitute each byte for another byte in the state. This step is similar to DES's S-boxes.

    (2) ShiftRows: We shift each row by a fixed amount of spaces per round to help bytes "move about" in the state. These fixed amounts are hard-coded into the algorithm, much like the fixed-amount rotations in Salsa20.

    (3) MixColumns: We multiply each column of the state matrix with a fixed matrix in order to mix the column's data. This promotes diffusion of the bits in the state matrix.

    (4) AddRoundKey: We XOR the bytes of K_i with the bytes of the state matrix, introducing actual security into the cipher.

(4) Final round: We perform the final round with K_{N_r}. The final round is the same as the above rounds, but omits the MixColumns step.

This description is essentially a high-level distillation of the content presented in [DR02]. As with the other heuristic constructions we have seen, each component of the cipher has careful design justification, but at no point is the cipher actually proven secure relative to some compact assumption.

AES is considered to be the ultimate choice of block cipher at the present time. It has received (as of the time of this writing) 16 years of scrutiny by cryptographers; the only attack that affects the security of AES is so inconsequential that AES is practically untouched at this point. There are no indications that AES will be broken any time soon [KL07].

3.4 Security of Heuristic Constructions

As said above, at no point is a heuristic construction ever proven secure with some simple assumption. Some heuristic constructions, like DES, make the assumption that a core function is one of the "perfect" theoretical constructs like a PRF, and then prove their security based on that—but that assumption is large and difficult to test in the design phase.

Instead, heuristic constructions are considered secure by the cryptanalytic results borne out by cryptanalysts. This is a much more practical view of security: no theoretical proof is necessarily needed so long as there is no one who is capable of breaking the construction. Because of this, our confidence in a construction is directly correlated with the amount of scrutiny it has received. Thus, the "gold standard" constructions have received intense focus from cryptanalysts over an extremely long period of time. The easiest example of a gold standard is AES.

In recent years, the cryptographic community has grown fond of competitions. These competitions usually seek to standardize one or perhaps a few excellent constructions for a particular purpose. For example, the Advanced Encryption Standard competition from 1997 through 2001 by the NIST sought to standardize one fantastic block cipher [DR02]. The AES competition received fifteen different candidate block ciphers, and over the course of several years, each of those candidates was subjected to intense study. As put in [KL07], this competition was ingenious because all of the best cryptographers were submitting candidates—and so those cryptographers had a vested interest in finding even the smallest flaws in every other candidate.

At the end of the AES competition, there were five remaining finalists: Rijndael, Serpent, Twofish, RC6, and Mars, with cryptographers resoundingly preferring Rijndael over the others (per a poll) [DR02]. And the rest is history—NIST selected Rijndael as the AES.

Emboldened by the success enjoyed by the AES competition, cryptographers have sought to recreate the conditions of this competition. In 2004, the European Network of Excellence for Cryptology (ECRYPT) launched the eSTREAM competition, which sought to create two "portfolios" of stream ciphers (one for use in software, the other in hardware) [Exc12]. After four years and multiple phases of testing, eight stream ciphers were selected—four for each portfolio. One of those stream ciphers was Salsa20.

Yet another competition was held in 2007 by the NIST to select a new crypto-

graphic hash function to be the Secure Hash Algorithm 3 (SHA-3). This competition ran for five years, had sixty-four entries, and three rounds of testing/analysis. At the competition's conclusion, an entry known as Keccak (pronounced "ketch-ack") was selected as SHA-3 [ST12].

There are currently two widely-known cryptographic competitions ongoing. One is the Password Hashing Competition [Org13], which will select one or more password hashing schemes sometime in the second quarter of 2015. The other is CAESAR [Ber15], which will tentatively select a portfolio of authenticated ciphers in 2017.

With these competitions in mind, one can see how the security of most heuristic constructions is judged: against many other competitors. Competitions also have the benefit of ruling out candidates that, while secure, are suboptimal in some other way—for example, some AES candidates were ruled out because they had the same security level as one of the finalists but were merely less efficient [DR02].

Chapter 4

Comparison of Heuristic and Conditionally-Secure Constructions

With both heuristic and conditionally-secure constructions explored and explained, we may finally turn toward their comparison.

4.1 Efficiency

One area where heuristic constructions handily dominate conditionally-secure constructions is efficiency.

Let's take Diffie-Hellman key exchange, which uses the familiar candidate one-way permutation

$$f(x) = g^x \pmod{p}.$$

In Diffie-Hellman key exchange, when computing a shared secret, three values are computed: $g^x \bmod p$, $g^y \bmod p$, and $g^{xy} \bmod p$. ECRYPT runs a public domain cryptographic benchmarking suite, which includes a suite for public-key cryptography [ECR15a]. In this suite, on the Titan supercomputer, the median CPU clock cycles required per shared secret computation for the `sclaus2048` primitive[1] is $1,147,528$ cycles. This means that each modular exponentiation (i.e., each evaluation of $f(x) = g^x \bmod p$ for a realistic p) takes *roughly* $1,147,528/3 \approx 380,000$ CPU clock cycles.

Recall from section 2.3.2 that each evaluation of a Feistel network using our construction took $3n^2$ evaluations of the underlying one-way permutation f. Thus,

[1] `sclaus2048` refers to the particular group used in the Diffie-Hellman key exchange. In this case, `sclaus2048` uses a realistically-sized prime p and regular multiplicative modular arithmetic: i.e., sclaus2048 is a realistic system.

a single evaluation of a pseudorandom permutation with a 128-bit width, with our construction at least, would take $3(128)^2 \cdot 380,000 \approx 1.9 \times 10^{10}$ CPU clock cycles on the Titan supercomputer. Since the Titan supercomputer node used in the above benchmark has a clock rate of 3.501 GHz, that PRP evaluation would take $(1.9 \times 10^{10})/(3.501\,\text{GHz}) \approx 5.5\,\text{s}$. An alternate measurement is CPU cycles per byte encrypted—in that case, our scheme would take $(1.9 \times 10^{10}\ \text{cycles})/(16\ \text{bytes}) \approx 1.19 \times 10^9$ cycles per byte.

On the other hand, we can look at the same project's benchmark of ChaCha20 on the same machine [ECR15b] and see that for "long messages," the median value for ChaCha20 is 1.21 cycles per byte. No, scientific notation was not accidentally omitted. That's one point two one cycles per byte. Even on 8-byte messages—the worst case for ChaCha20—it takes (on average) 47.50 cycles per byte (or 13.57 ns per byte).

Even though this is comparing a stream cipher to a pseudorandom permutation (block cipher), the comparison doesn't really matter. When heuristic constructions can take less than 100 cycles per byte to encrypt data, no one-way function that takes 380,000 clock cycles has any chance. A single evaluation of that OWF is too expensive in comparison, and we are talking of possibly having to evaluate it n or n^2 or $3n^2$ times for a PRG, PRF, or PRP, respectively.

Suffice it to say that heuristic constructions obliterate naïve conditionally-secure constructions on the grounds of efficiency.

4.2 Effective Security

Effectively, which should be trusted more: conditionally-secure or heuristic constructions? Surely something with a proof of security, even if only based on an assumption, has the upper hand, right? Not necessarily.

Inverting the one-way function $f(x) = g^x \bmod p$ would certainly require a breakthrough in our understanding of algebraic number theory. At the same time, however, defeating AES would require a breakthrough in the field of cryptanalysis—entirely new techniques as yet unknown must be discovered before AES will ever be vulnerable.

If there were some way to determine which area is more likely to have a breakthrough, we might consider one more secure than the other. We may possibly speculate that since algebraic number theory is the older field, it is better stud-

ied, and so sudden breakthroughs are less likely. However, this is not a rigorous argument.

Side-channel attacks are another angle that must be considered. Side-channel attacks attempt to measure some physical quantity (time taken, power drawn, frequency of sound waves emitted, electromagnetic radiation, etc.) while the cryptographic system is being operated, often with plaintexts or ciphertexts that may be controlled by the attacker—a common-day scenario with the advent of the internet. Timing attacks, in particular, are widely studied, since they can be measured at range: an attacker can send a message over the internet and measure how long it takes before a response is received. Other forms of side channel attacks, such as differential power analysis, usually require physical access to the hardware, which is a scenario cryptographers don't usually defend against.

Side-channel attacks are particularly fearsome because they do not cryptographically attack the system. Instead, they attack the real-world implementation of the system. Proofs of security do not address the possibility of side-channel attacks. In modern day systems, like Salsa20, the scheme is carefully designed with timing attacks already in mind [Ber08b], mostly by having enough real-world experience with timing attacks in already-existing systems to know what types of operations are vulnerable to timing attacks.

On the other hand, the theoretical constructions we've presented for PRGs, PRFs, and PRPs do not take possible side-channel attacks into account. It may be that a simple timing attack is capable of defeating one of the constructions without much difficulty. Alternately, it may be possible to implement those constructions in such a way that timing attacks are not possible—but such an implementation may be difficult to do. (For example, AES is famously difficult to implement in software without timing attacks.)

Because we don't have an existing knowledge base of side-channel attacks as they apply to our theoretical constructions—a result of those theoretical constructions not being used in practice—we cannot say for sure if they are vulnerable to side-channel attacks. Instead, we would need skilled cryptographers familiar with side-channel attacks to investigate whatever implementations that might be produced. This sort of analysis takes a large amount of effort: effort that has already been put forth for heuristic constructions.

4.3 Availability

So, in order to make conditionally-secure schemes worth using, we must find some way to efficiently implement them, perhaps by choosing a better one-way function or by simply making the theoretical constructions take fewer evaluations of the one-way function for the same security level. (One might use a hard-core *function* instead of a hard-core *predicate*, as discussed in [Gol00], but this is not enough.)

Next, we must actually implement the efficient theoretical construct. This alone is a nontrivial effort: it is much harder to write computer code to perform cryptographic operations than it is to discuss them in the abstract, especially if one is using a programming language that lends itself to performance-optimized code, like C. For instance, if the programmer wishes to use integers greater than 64 bits in width, they may no longer use the built-in mathematical functions in C: they must find a "BigNum" library that is capable of handling larger integers. These difficulties arise because our mathematical descriptions of constructions tend to be concise and information-dense, whereas implementing those constructions on a real-world computer requires the programmer to deal with thousands of small details. On top of this, the programmer should be an experienced applied cryptographer, one who is familiar with generic, non-algorithm-specific security problems.

If we want the implementation to be actually usable, instead of merely proof-of-concept, it must be carefully vetted by experienced cryptanalysts. When vetting a cryptographic library, cryptanalysts first try to find possible programmatic errors. Such programmatic errors are not cryptographic in nature; rather, they are merely an error in the translation of the mathematical specification to real-world code. An example might be the accidental omission of an XOR operation in a round function, or a typo in a conditional statement. If a programming error is severe enough to change the ultimate output of the algorithm, then usually programmers catch it quickly enough by comparing the code's output to a known-good example. However, since this conditionally-secure library would be the first of its kind, there are no known-good examples already existing; as a result, a manual scanning of the potentially thousands of lines of code would be required.

That is merely the first stage, however. Once cryptographers are convinced that the implementation is semantically correct, the hunt for possible side-channel attacks (especially timing attacks) begins. Hopefully, the most common culprits were already recognized by the programmer and eliminated, but the entire code-

base must be checked all the same. Cryptanalysts can then study the algorithms themselves for possible timing attacks.

These requirements may not happen in that exact order (some of them may occur in parallel, perhaps by different cryptographers), but they must all happen for the library to be considered trusted.

If we want to make conditionally-secure constructions usable in the real world, the above is what must be done. On the other hand, all of the above has *already* been done for heuristic constructions; there are a variety of pre-existing, vetted libraries for consumption by the general public. In other words, an absolutely huge amount of effort has already been poured into existing heuristic constructions by cryptographers. Attempting to supplant carefully-designed and carefully-vetted heuristic libraries with a fresh conditionally-secure library would require a large duplication of effort on the part of the entire security community. Non-cryptographers using the old libraries would be required to be re-trained to use the new ones. We would be abandoning decades of effort.

As is clear, as a community, we cannot recommend swapping to conditionally-secure constructions until they have *significant* benefits. Those benefits do not appear to be forthcoming any time in the near future.

Chapter 5

Conclusions

In our thesis, after familiarizing ourselves with the history, terminology, and mathematical concepts in the field of cryptography, we did an in-depth discussion and comparison of conditionally-secure constructions and realistic heuristic constructions.

Conceptually, conditionally-secure constructions *are* cleaner—the assumption on which they rest is clearly stated, mathematically examinable, and fully rigorous. Sometimes, those assumptions are based on problems studied for thousands of years (even if the only significant progress on those problems was made in the past forty years). In that sense, conditionally-secure functions are preferable, and they are certainly interesting from a theoretical point of view. However, clarity of concept is insufficient justification for the widespread adoption of conditionally-secure constructions in symmetric cryptography. Actual implementations would be extremely inefficient. There are no proofs that they would be secure against side-channel attacks. And, finally, there are no implementations currently available.

Realistic heuristic constructions are an area of extremely active study. They are already widely available in pre-existing libraries. Heuristic constructions are orders of magnitude more efficient than even the simplest conditionally-secure constructions. Since providing an equal level of security and efficiency with a conditionally-secure construction will take years of work, we conclude that heuristic constructions are the only viable choice at this time.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. New York, NY, USA: Cambridge University Press, 2009. ISBN: 0521424267.

[Ber08a] Daniel J Bernstein. *ChaCha, a variant of Salsa20*. 2008. URL: http://cr.yp.to/chacha/chacha-20080128.pdf.

[Ber15] Daniel J Bernstein. *Crypto Competitions: CAESAR Call for Submissions*. 2015. URL: http://competitions.cr.yp.to/caesar-call.html.

[Ber08b] Daniel J Bernstein. "The Salsa20 family of stream ciphers". In: *New Stream Cipher Designs*. Springer, 2008, pp. 84–97.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002. ISBN: 3540425802.

[Dei09] Wei Dei. *Speed Comparison of Popular Crypto Algorithms*. 2009. URL: http://www.cryptopp.com/benchmarks.html (visited on 11/15/2013).

[ECR15a] ECRYPT. *Measurements of public-key Diffie-Hellman secret-sharing systems, indexed by machine*. 2015. URL: http://bench.cr.yp.to/results-dh.html.

[ECR15b] ECRYPT. *Measurements of stream ciphers, indexed by machine*. 2015. URL: http://bench.cr.yp.to/results-stream.html.

[Exc12] European Network of Excellence for Cryptology. *eSTREAM: the ECRYPT Stream Cipher Project*. 2012. URL: http://www.ecrypt.eu.org/stream/index.html.

[Gol00] Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. New York, NY, USA: Cambridge University Press, 2000. ISBN: 0521791723.

[Hof+08] Jeffrey Hoffstein et al. *An Introduction to Mathematical Cryptography*. Springer, 2008.

[Ins00] Clay Mathematics Institute. *The Millennium Prize Problems*. 2000. URL: http://www.claymath.org/millennium-problems/millennium-prize-problems.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007. ISBN: 1584885513.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN: 0849385237.

[Nat01] National Institute of Standards and Technology. "FIPS 197". In: *National Institute of Standards and Technology, November* (2001), pp. 1–51. URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[Org13] Password Hashing Competition Organizers. *Password Hashing Competition*. 2013. URL: https://password-hashing.net.

[Sha49] Claude E. Shannon. "Communication Theory of Secrecy Systems". In: *The Bell System Technical Journal* 28.4 (Oct. 1949), pp. 656–715.

[Sin99] Simon Singh. *The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography*. Doubleday, 1999.

[ST12] National Institute of Standards and Technology. *The SHA-3 Cryptographic Hash Algorithm Competition*. 2012. URL: http://csrc.nist.gov/groups/ST/hash/sha-3/.