CERIAS Tech Report 2007-20

DYNAMIC CRYPTOGRAPHIC HASH FUNCTIONS

by William Speirs

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

DYNAMIC CRYPTOGRAPHIC HASH FUNCTIONS

A Thesis

Submitted to the Faculty

of

Purdue University

by

William Robert Speirs II

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2007

Purdue University

West Lafayette, Indiana

To my parents,

Jeffrey and Ellen

ACKNOWLEDGMENTS

First and foremost I would like to thank my adviser, Samuel S. Wagstaff, Jr. His advice, guidance, and patience helped me to gain a deeper understanding of the field of cryptography and research in general. I would also like to thank only one of my committee members Eugene Spafford for supporting me and my efforts to complete this Ph.D. A special thanks to Bart Preneel (Katholieke Universiteit Leuven) for providing early ideas and agreeing to be part of my preliminary examination committee. I would also like to thank Moses Liskov (The College of William and Mary) for providing me with valuable insight and comments towards the end of my work.

Thanks to The Center for Education and Research in Information Assurance and Security (CERIAS) and the Computer Science faculty and staff of Purdue University for providing me with all the necessary resources and an environment that allowed me to complete this work. A special thanks goes to the Pikewerks Corporation and the Air Force Research Laboratory for providing me with funding to accomplish additional research not directly related to this dissertation.

Finally, I would like to thank my family and friends. My family has always shown me unconditional support through sometimes tumultuous periods and always encouraged me to pursue this degree. My friends always supported me and endured my sometimes negative discourse about this dissertation. A special thanks goes to my roommates Barry Wittman and Armand Navabi who were always there to humor my ideas and help me especially in those final trying weeks; stick with it guys.

> "This thesis is dedicated to all the professors that wanted to see me fail, to all the companies that supported me while I was trying to make some money to support my daughter, and all the graduate students in the struggle." – Christopher George Latore Wallace (modified)

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

Pre, Sec and Col are used for both traditional and dynamic hash functions; context will disambiguate.

Pre	Preimage Resistance
Sec	Second Preimage Resistance
Col	Collision Resistance
PCol	Security Parameter Collision Resistance
Dig	Digest Resistance
sSec	Strong Second Preimage Resistance
sCol	Strong Collision Resistance
dPre	Dynamic Preimage Resistance
dSec	Dynamic Second Preimage Resistance
dCol	Dynamic Collision Resistance
wdPre	Weak Dynamic Preimage Resistance
wdSec	Weak Dynamic Second Preimage Resistance
wdCol	Weak Dynamic Collision Resistance
sdPre	Strong Dynamic Preimage Resistance
sdSec	Strong Dynamic Second Preimage Resistance
sdCol	Strong Dynamic Collision Resistance
wPCol	Weak Security Parameter Collision Resistance
sPCol	Strong Security Parameter Collision Resistance
wDig	Weak Digest Resistance

ABSTRACT

Speirs II, William Robert Ph.D., Purdue University, May, 2007. Dynamic Cryptographic Hash Functions. Major Professor: Samuel S. Wagstaff, Jr.

This dissertation introduces a new type of cryptographic hash function, the *dynamic cryptographic hash function*. Dynamic cryptographic hash functions differ from traditional hash functions because they require a second parameter, the *security parameter*. The security parameter controls both the method used to calculate a digest and the size of the digest produced. Dynamic cryptographic hash functions are motivated by the need for a hash function that can match the level of expected security of the protocols in conjunction with which they are used.

The properties that dictate the security of a dynamic cryptographic hash function are explored. The traditional properties of preimage resistance, second preimage resistance, and collision resistance are modified to accommodate the security parameter and expanded into dynamic versions that dictate a dynamic cryptographic hash function must be secure even if the attacker is able to choose a different security parameter. Two additional properties are defined, *security parameter collision resistance* and *digest resistance*. These properties ensure that two digests created from the same message using different security parameters are unrelated.

Finally, the dynamic cryptographic hash function construction is presented, which creates a dynamic cryptographic hash function from a traditional compression function. The construction is able to create digests larger than the compression function's output.

1 INTRODUCTION

Cryptographic hash functions are a necessary evil. It is almost impossible to construct an efficient and secure cryptographic hash function, yet they are required for the security of numerous protocols and systems. Recently research in cryptographic hash functions has increased due to attacks, at all levels, against the cryptographic hash functions currently in wide spread use. The National Institute for Standards and Technology has opened a competition for the next cryptographic hash function standard. This dissertation provides one possible avenue for creating the next generation of cryptographic hash function.

1.1 A Brief History of Cryptographic Hash Functions

Hash functions, in the cryptographic sense, arose from a need for one-way functions. One-way functions were needed for computer login procedures, first described by R.M. Needham [21]. The desire was for a function whose description was publicly known, and yet calculating the inverse of the function was difficult because of some unknown piece of information such as a key. In [21] they explain that the *noninvertible* property required was not the standard notion of a single range point having multiple domain points.

In [18] Ivan Damgård explored claw free functions and provided formal definitions for what it means for a function to be one-way and collision resistant. These definitions were updated and the field of hash functions explored in more depth by Bart Preneel in [59]. Preneel's dissertation explored the formal definitions of the security properties (preimage resistance and collision resistance) that are required for a hash function to be considered cryptographically secure in both the information theoretic and complexity theoretic settings. His dissertation further explored hash functions

based on block ciphers, hash functions based on modular arithmetic and dedicated hash functions. The properties of Boolean functions were also discussed with respect to hash functions. Preneel's dissertation is one of the most complete works, to date, of cryptographic hash functions.

The basis for constructing most dedicated hash functions is the Merkle-Damgård construction discovered independently by Ralph Merkle [49] and Ivan Damgård [19] in 1989. Their construction expands a finite domain function to one of infinite domain. Their construction changed the focus from building a secure hash function to building a secure finite domain function.

One method for creating a finite domain function is by using a block cipher. In 1992 Lai and Massey explored creating hash functions using block ciphers [42]. This work was systematically explored by Preneel, Govaerts and Vandewalle in [63] and extended by Knudsen and Preneel in 1996 [38]. Black, Rogaway and Shrimpton explored the work of [63] in their 2002 paper [10]. The result of this work is a number of ways to construct secure finite domain functions from block ciphers, including the most commonly used Davies-Meyer and Miyaguchi-Preneel.

Attacks against the Merkle-Damgård construction began with Antoine Joux's paper on multi-collisions [35]. While other attacks such as the length extension attack and the fixed-point attack were known before 2004, it was Joux's paper that sparked a series of attacks against the Merkle-Damgård construction. Attacks such as the herding attack, long message second preimage attack and others, seriously question the continued use of the Merkle-Damgård construction.

In 1989 Naor and Yung introduced the notion of Universal One-Way Hash Functions [54]. This notion came from the work of Carter and Wegman who first introduced universal classes of hash functions in 1977 [12]. Carter and Wegman even commented on the cryptographic uses of such classes in 1981 [76]. A complete exploration of the different notion of security with respect to cryptographic hash functions was given by Rogaway and Shrimpton in 2004 [69]. They also explored the implications between these notions of security [69].

Numerous dedicated hash functions have been introduced and attacks against most of them have been discovered. One lineage of cryptographic hash functions stem from the Message Digest algorithms of Rivest [67] [68]. The National Institute of Standards and Technology released modifications to MD4 as the Secure Hash Algorithm. This algorithm was modified to fix "a minor technical flaw" and is now known as SHA-1 [56]. The RIPEMD family of hash functions were developed as part of the RIPE (RACE Integrity Primitives Evaluation) project. The successor to RIPEMD is RIPEMD-160 [24]. Attacks against all of these functions have been published by numerous authors, most notedly Xiaoyun Wang [74, 75].

Today the security of SHA-1 is seriously questioned. Alternatives such as SHA-256 and SHA-512 exist, but are designed from the same basic principles as SHA-1 [56]. Other hash functions such as Whirlpool [65] are built using already considered secure primitives. Currently there is no consensus as to which hash function should be used in implementations today.

1.2 Current Applications of Cryptographic Hash Functions

Cryptographic hash functions are used in numerous ways, but their primary use is to protect data integrity. The two settings in which a cryptographic hash function is able to protect data integrity are with and without a secure channel. If a secure channel exists, then the hash or digest of the data can be computed and sent via the secure channel. The data is sent via an insecure channel and the recipient of both the data and digest is able to recompute the digest to see if it matches the digest that was sent.

If a secure channel does not exist, then hash functions can be used in conjunction with signature schemes. The digest of the data is computed and signed using the signature scheme. If the signed digest is modified during transmission through the insecure channel, the signature will not verify. If the data is modified during trans-

mission, then the computed digest will not equal the digest that was signed. Signing a digest is in some ways analogous to sending the digest via a secure channel.

Cryptographic hash functions can also be used in conjunction with a trusted third party to provide data integrity. The two common applications of this type of use are for commitment schemes and to ensure data corruption has not occurred during transmission. In a commitment scheme that uses a cryptographic hash function, the user committing to a value will compute the digest of the value and send it to a trusted third party.[1] This third party is often a newspaper or other public media. The user committing to the value can verify that no change occurred during the publishing stage. The user verifying the commitment is able to record the digest of the value that has been committed. At a later stage the value is revealed and the verifier can recompute the digest and compare it to the recorded digest.

Cryptographic hash functions can also be used to ensure that data corruption has not been caused by a malicious or an error prone communication channel. The scheme works by the user receiving data through some insecure or faulty communication channel. The digest of the data is then computed and is compared to the digest of the original data sent by the trusted third party. For example, the digest of a CD-ROM ISO can be published on a website. After a user has downloaded the ISO, the digest can be computed and checked against the digest posted on the website. This scheme is similar to the commitment scheme except the goal of the scheme is slightly different.

While the applications of hash functions explained above all rely on collision resistance, applications of the one-way property of cryptographic hash functions can be found in user authentication schemes. Passwords are stored securely on a system by storing only their digests. Each time a user authenticates with the system the digest of the provided password is compared with the stored digest. It is imperative

[1]One should note that information is possibly leaked about the value if the domain of values is smaller than the set of possible digests.

that attackers are unable to reverse the digest of a password and discover the original password.

Unfortunately, cryptographic hash functions are also often used as random number generators, pseudorandom functions, or entropy generators. Often times implementers will use a cryptographic hash function to hash data in a haphazard manner to create randomness. This type of use can be found in almost all modern operating systems such as the `/dev/random` device found on most Unix and Linux systems. Most weaknesses are not due to the hash function [30]. However, using hash functions to generate entropy is not the intended use and therefore should be used with care or not at all.

1.3 Motivation for Dynamic Hash Functions

Creating a new class of cryptographic hash functions, dynamic cryptographic hash functions, is advantageous for a number of reasons. First, dynamic cryptographic hash functions have the ability to more evenly match the relative security of any protocol or scheme with which they are used. For example, the block cipher AES (Advanced Encryption Standard) has three different key sizes [55]. To accommodate the different key sizes, three different[2] hash functions were designed to complement the relative level of security [56]. Instead of requiring multiple hash functions, one for each key size of the block cipher being used, one hash function should have the ability to produce different size digests.

This same logic applies to digital signature schemes. Most signatures schemes in use today (RSA, DSA, or ElGamal) have the ability to select the size of the key down to the bit. However, each of these signature schemes use a fixed size hash function to compute the digest of the message. Instead, a hash function should be used that can create digests of different sizes so the relative level of security is the same.[3]

[2] Actually, SHA-384 is the truncation of SHA-512 with a different initial value.

[3] Increasing the size of a key or digest does not always relate to increased security.

Second, incorporating dynamic hash functions into protocol design and implementation allows for easier changes of the function used for a given protocol. As discussed in [6], algorithm agility is extremely important in protocol design. It is not feasible to upgrade all of the systems used today at once. Instead of needing to implement a new hash function when an attack is discovered, a security parameter can be changed such that the function dynamically changes how a digest is computed.

Finally, the use of a dynamic cryptographic hash function allows designers to more easily test functions by scaling down the number of rounds and the size of the digest to a manageable number. Attacks can be launched against a reduced version in an attempt to find weaknesses in the full version. This technique has been used numerous times in the past [20, 22, 23], and was even suggested during the second workshop on hash functions held by the National Institute of Standards and Technology [66].

1.4 Scope of This Dissertation

While cryptographic hash functions occupy a small sliver of cryptography, inside this sliver are a few different areas each addressing slightly different problems. Because the main function of this dissertation is to present a new type of cryptographic hash function, the dynamic cryptographic hash function, all aspects related to cryptographic hash functions are not investigated.

One such area not thoroughly investigated is Message Authentication Codes. Message Authentication Codes (or MAC) are functions that work in the same manner as cryptographic hash functions except that they require a key. These functions are of great importance to the subject of cryptography and even cryptographic hash functions, so they are briefly covered in the following section. For a complete reference on Message Authentication Codes, see John Black's dissertation [9].

1.4.1 Message Authentication Codes

A message authentication scheme contains an algorithm (MAC) that generates a tag for a message using a secret key.[4] The algorithm is used in the following way to provide message authentication. Say a user, Alice, wants to send a message to Bob and provide Bob with some level of confidence that the message truly came from Alice. Assume that Alice and Bob already share a secret key k and communicate over a channel that has an active adversary with full control over the channel. The goal of the adversary is to create a message and tag that appears to have come from Alice. The goal of the adversary is not to intercept and read messages sent between Alice and Bob. In this situation only the authenticity of the message is being protected.

Using the MAC algorithm and the secret key k, Alice is able to compute a tag for the message: $t = MAC(M, k)$ where t is the tag and M is the message. The tag is then sent over an insecure channel to Bob along with the original message:

$$\text{Alice} \rightarrow (t, M) \rightarrow \text{Bob}.$$

Bob, or anyone else possessing the secret key k, is able to verify that the message originated from Alice by recomputing the tag: $MAC(M, k) = t'$ and checking to ensure that $t = t'$. If the adversary attempts to change either the message, tag, or both while in transit the message authentication algorithm should ensure that $t \neq t'$ with a high degree of probability.

Several Message Authentication Codes have been proposed, some built upon cryptographic hash functions. The two most famous MACs built upon cryptographic hash functions are NMAC and HMAC. These are of interest to this work because they demonstrate that a secure MAC can be created from a secure hash function [1, 2, 41]. The form of NMAC and HMAC are provided with no further discussion to demonstrate how easily a MAC can be built from a cryptographic hash function. Assume $h : \{0,1\}^n \times \{0,1\}^* \rightarrow \{0,1\}^n$ is a collision resistant function that iterates over a message M starting with an initial value IV. The cryptographic hash function is

[4] The key generation and verification algorithms are ignored for simplicity's sake.

denoted by $H(M) = h(IV, M)$ or $H^*(K, M) = h(K, M)$ where K specifies the initial value. NMAC and HMAC are then defined as follows where $K = K_1 \parallel K_2$:

$$\begin{aligned} NMAC(M, K_1 \parallel K_2) &= H^*(K_1, H^*(K_2, M)) \\ HMAC(M, K_1 \parallel K_2) &= H(K_1 \parallel H(K_2, M)). \end{aligned}$$

HMAC is used in practice and is specified in IETF RFC [41] and NIST FIPS. The results for NMAC can be shown to be lifted to HMAC [2], and is used in theoretical work to prove the security of both [1].

1.5 Contributions of This Dissertation

This dissertation makes three contributions to the field of cryptographic hash functions. First, it defines a new class of cryptographic hash functions, dynamic cryptographic hash functions. A formal definition for a dynamic cryptographic hash function is provided. The three security properties that make traditional hash functions cryptographically secure are modified to apply to dynamic cryptographic hash functions. Additional security properties are explored and formally defined, and implications between the properties are proved.

Second, a survey of viable constructions for creating cryptographic hash functions from compression functions is given. The constructions discussed are suggested as replacements for or modifications to the Merkle-Damgård construction. These new constructions attempt to thwart many of the attacks against the Merkle-Damgård construction, which are also presented in this dissertation.

Finally, a construction that creates a dynamic cryptographic hash function from a traditional compression function is described. The dynamic hash function construction is proved to have all of the security properties necessary for a dynamic hash function to be considered cryptographically secure. The technique used to create larger digests is also investigated.

1.6 Organization of This Dissertation

Chapter 2 introduces cryptographic hash functions. The chapter is titled "Traditional Cryptographic Hash Functions" only to disambiguate the information contained in the chapter from dynamic cryptographic hash functions, the main focus of this dissertation. The chapter provides a definition for a traditional cryptographic hash function including the three main security parameters that separate cryptographic hash functions from regular hash functions. The chapter continues by defining the different notions of security with respect to these three main properties, and the implications found between these notions.

Chapter 3 introduces dynamic cryptographic hash functions. This chapter closely mirrors Chapter 2. The definition of a dynamic hash function is given. The three main properties that differentiate a traditional hash function from a regular hash function are redefined to fit a dynamic hash function. The chapter includes variations on the three traditional properties that only apply to dynamic hash functions and introduces two new security properties. All of the security properties, both new and old, are extended to various notions of security with implications between them explored.

Chapter 4 explores how traditional hash functions are constructed. The Merkle-Damgård construction is presented along with numerous attacks against this construction. A number of new constructions are also presented that attempt to thwart some of the attacks against the Merkle-Damgård construction. Some of these constructions motivate many of the design decisions used in creating the dynamic cryptographic hash function construction.

Chapter 5 presents the dynamic hash function construction. The construction builds a dynamic hash function from a traditional compression function. Proofs that the

dynamic hash function construction possesses all of the security properties defined in Chapter 3 are provided. An investigation into the methods for attacking the technique used to create larger digests is given.

Chapter 6 provides concluding remarks on the work and recommendations for future work.

1.7 Notation

Throughout this dissertation certain symbols are used to aid in describing functions, operations, etc. Whenever possible the symbols match that which is most commonly used in the field. Table 1.1 lists the symbols used in this dissertation.

A number of variables are used to describe hash functions, compression functions and messages. These variables are listed in Table 1.2 with their description. A number of the theorems and proofs in this dissertation are modified from the originals so that the names of all the variables are consistent throughout the dissertation. Any variations from the naming convention is clearly noted.

In many of the theorems in this dissertation the big-Oh notation is used to denote an expected value. This is a slight abuse of the notation; however, it is done to be consistent with published work. At all opportunities the expected value itself is used instead of the asymptotic bound.

Table 1.1
Symbols and their description.

Symbol	Description
$\|x\|$	If x is a binary string, the length of the string.
	If x is a set, the size of the set.
Σ	The binary alphabet.
Σ^x	The set of all binary strings of length x.
Σ^*	The set of all binary strings, including the empty string.
$\times$	The Cartesian product of two sets.
$\mathbb{N}$	The set of natural numbers.
$\Pr[x]$	The probability of event x occurring.
$\mathbf{Adv}_H(A)$	The adversarial advantage of adversary A against function H.
$[x, y]$	The following set: $\{i \in \mathbb{N} : x \leq i \leq y\}$.
$x \xleftarrow{\$} y$	If y is a set, an element is randomly chosen from y and assigned to x.
	If y is an algorithm, the algorithm is randomized and its result is x.
$\mathcal{O}$	The standard asymptotic upper bound.
Ω	The standard asymptotic lower bound.

Table 1.2
Variables and their description.

Variable	**Description**
n	The output length, in bits, of a compression function.
b	The block size, in bits, of a compression function.
g	A compression function of the form $\Sigma^n \times \Sigma^b \rightarrow \Sigma^n$.
M	The message being hashed.
l	The length of the message being hashed.
$\mathcal{M}$	The set of all possible messages.
m_i	The i^{th} message block of the message M.
k	The number of message blocks in the message M.
s	The security parameter of a dynamic hash function.
$\mathcal{S}$	The set of all possible security parameters.
d	The function that determines the digest's size.
λ	The function that determines the smallest security parameter.
υ	The function that determines the largest security parameter.
H	A hash function; context will disambiguate its type.
D	The domain of the hash function.
R	The range of the hash function.
$\mathcal{K}$	The set of all possible keys for the hash function.
IV	The initial value of a hash function construction.
h_i	The i^{th} internal value of a hash function construction.
A	The adversary.
t	The time it takes for an adversary to run.
ϵ	An upper limit on the adversarial advantage of an adversary.
	It is desirable for t/ϵ to be large.

2 TRADITIONAL CRYPTOGRAPHIC HASH FUNCTIONS

There are numerous definitions in the literature for a cryptographic hash function, including [19, 48, 59, 69]. While superficially these definitions are different, they all define essentially the same type of function. For the purposes of this dissertation, a hash function will be defined as follows.

Definition 2.0.1 (Hash Function) *A hash function is a function of the form*

$$H : \Sigma^* \rightarrow \Sigma^n.$$

This definition defines H, a hash function, as a function from binary strings of arbitrary length to strings of a fixed length. The input to H is called a message, and the output is called the digest of the message.

While the above definition describes what a hash function looks like, it does not mention what is required of a hash function to be considered cryptographically secure, making it a cryptographic hash function. There are six informal requirements for a hash function to be considered cryptographically secure. The first three are commonly referenced when discussing cryptographic hash functions and are formally defined later in this chapter. All six requirements are enumerated below for completeness, taken from [48].

1. *Preimage Resistance* - For essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output, i.e., to find any preimage x' such that $h(x') = y$ when given any y for which a corresponding input is not known.

2. *2nd-Preimage Resistance* - It is computationally infeasible to find any second input which has the same output as any specified input, i.e., given x, to find a 2nd-preimage $x' \neq x$ such that $h(x) = h(x')$.

3. *Collision Resistance* - It is computationally infeasible to find any two distinct inputs x, x' which hash to the same output, i.e., such that $h(x) = h(x')$.[1]

4. *Non-Correlation* - Input bits and output bits should not be correlated. Related to this, an avalanche property similar to that of good block ciphers is desirable whereby every input bit affects every output bit. [28] (This rules out hash functions for which preimage resistance fails to imply 2nd-preimage resistance simply due to the function effectively ignoring a subset of input bits.)

5. *Near-Collision Resistance* - It should be hard to find any two inputs x, x' such that $h(x)$ and $h(x')$ differ in only a small number of bits.

6. *Partial-Preimage Resistance* or *Local One-Wayness* - It should be as difficult to recover any substring of x as to recover the entire input x, given $h(x)$. Moreover, even if part of the input is known, it should be difficult to find the remainder (e.g., if t input bits remain unknown, it should take on average 2^{t-1} hash operations to find these bits.)

Unfortunately, precise definitions for "essentially all" and "pre-specified outputs" do not exist in these definitions. Section 2.4 provides formal definitions for preimage resistance, second preimage resistance, and collision resistance. In Section 2.5 the definitions in Section 2.4 are expanded to seven notions of security for preimage resistance, second preimage resistance and collision resistance.

The final requirement of a cryptographic hash function does not deal with security but practical application. A cryptographic hash function must be easy and quick to compute. The difference between the time required to read data from a disk and computing the digest while doing so should be negligible. This requirement usually rules out number theoretic functions because of their high cost of computing.

[1]There is a free choice in both x and x' by the adversary.

2.1 Function Families

The informal definitions for preimage resistance, second preimage resistance, and collision resistance given in [48] are somewhat difficult to describe formally. Before defining preimage resistance, second preimage resistance and collision resistance, a hash function family must be defined. The reason is that one could imagine a probabilistic polynomial time algorithm which has two messages encoded in the algorithm for a specific hash function. This algorithm can be used to compute a collision for the hash function. The algorithm simply outputs the two messages that cause a collision [62]. Creating such an algorithm is extremely difficult, but once constructed it would run in polynomial time. However, defining a function family as an infinite family of finite sets of hash functions prevents such an algorithm from succeeding for all hash functions in the family, because there are infinitely many. The definition of a hash function family that follows is taken, in modified form, from [5, 18, 50, 59, 69].

Let $D = \Sigma^l$, or the domain of the function. Let $R = \Sigma^n$, or the range of the function. Let $\mathcal{K}$ be the set of all possible keys.[2] Therefore, for a hash function family $\mathcal{H}$, each hash function is of the form $H : \mathcal{K} \times D \to R$ or $H_k : D \to R$.

Definition 2.1.1 (Hash Function Family) *A hash function family $\mathcal{H}$ is a infinite set of functions where each function in the family is indexed by a key K, and each function is of the form*

$$H_K : \Sigma^l \to \Sigma^n.$$

There are three requirements imposed on the hash function family $\mathcal{H}$ [18, 59].

1. $\mathcal{H}$ is accessible, that is, there is a probabilistic polynomial time algorithm, that on input K outputs an instance H_K.

2. D is samplable, that is, there is a probabilistic polynomial time algorithm, that selects an element uniformly from D.

[2] Theoretically this set is infinite. In practice it is finite.

3. H_K is polynomial time computable, that is, there is a polynomial time algorithm (polynomial in l) that on input $M \in D$ computes $H_K(M)$.

To clarify the definition of a hash function family, SHA-1 [56] is used as an example. First, it is important to note that SHA-1 is a single instance of a hash function, not a family. However, SHA-1 can be modified to construct a finite family of functions. In [5] SHA-1 is modified such that the key specifies the constants used in the four round functions. In this case the size of the key is 128 bits in length, 4 32-bit words. Therefore, $\mathcal{K} = \Sigma^{128}$, $D = \Sigma^{2^{64}}$ and $R = \Sigma^{160}$.[3]

One should note that an instance of a hash function family does not directly correlate with the definition of a hash function given. The form of a hash function in Definition 2.0.1 is $H : \Sigma^* \rightarrow \Sigma^n$; whereas, an instance of a hash function in Definition 2.1.1 is $H_K : \Sigma^l \rightarrow \Sigma^n$. This is done because for all of the security properties it is required that the domain be uniformly sampled in polynomial time [50]. A family $\mathcal{H}$ can always be constructed with an appropriate size domain to accommodate any message because all the strings in Σ^* are finite.

2.2 Adversarial Model

Before defining the properties that a hash function must possess to be cryptographically secure, an adversarial model must be constructed. Without an appropriate adversarial model it is impossible to tell if a given hash function possesses a security property or not. The adversarial model used in this dissertation is a RAM model similar to the one found in [69] and [60]. The adversary is a program, in some fixed programming language, that runs on the RAM model. The adversary, or program, can take any number of inputs.

There are three important features of this adversarial model. First, the RAM model has pointers. Pointers allow the adversary to query the i^{th} bit of some argument x by writing (i, x) in some distinguished register. The result of this query is returned

[3]The specification of SHA-1 is modified slightly to only accept strings that are up to 2^{64} bits in length.

in unit time. This prevents artificially slow adversaries that might need to read through an extremely long input discarding all but the last bit, for example.

Second, the RAM model has random bits the adversary can access. This is much the same as a probabilistic Turing machine traditionally used in cryptography [60]. Random bits allow the adversary to be a randomized algorithm. Access to a random integer in the range $[1, n]$ requires the expected time, $\Theta(\log n)$.

Third, the adversary has access, in unit time, to the hash function it is designed to break. For example, if the adversary needs to compute the digest of a given message, this is performed by the RAM model in unit time. This prevents constructing a "secure" hash function by simply requiring the hash function to take exponential time to compute a single digest. The same is true for any underlying piece of the hash function, such as a compression function. This allows for a flexible adversary that is not inhibited by time consuming hash functions.

The resource used to determine the success of an adversary is time, t. An adversary is considered successful if it returns the correct result, as determined by some experiment, in the time allowed. To prevent time-memory trade-offs, a common trick used to break cryptographic functions, the running time of the adversary is computed as the running time of the program, plus the size of the program. This prevents an adversary that stores the precomputed values for all domain and range points for a given function. In this model the same amount of time is required for an adversary to build the table as to store it in the program.

2.3 Methods for Defining Security Properties

With an adversarial model fixed, a method for defining the security properties must be chosen. There are three ways to define a given security property with slight variations for each method. The first method was used by Preneel [59] and Damgård [18] to define one-way functions (preimage resistant) and collision resistant functions. The definitions are formed in a complexity theoretic framework where an adversary

is defined that attacks a certain property of the function. For example, to define one-way functions, an adversary A is described that takes as input $H_k(M) \in R$ and outputs $A(H_K(M)) \in D$. Then a description of how successful an adversary can be is given in relation to some polynomial Q as follows:

$$\Pr\big[H_K(A(H_K(M))) = H_K(M)\big] < \frac{1}{Q(n)}.$$

The main problem with this type of a definition is that it does not explicitly state how each piece is selected. In the above example, as with the definition in [59] there is no mention how the hash function is selected from the function family or the message from the set of possible messages. As described in Section 2.5, the way in which the function is selected changes the notion of security that is considered. For this reason this type of definition was not used.

The other two methods for defining properties are very similar. Both describe the property being defined in two parts: an experiment and a description of the success of an adversary. The experiment describes precisely what is required of an attacker and how each piece of the experiment is chosen. This allows for numerous notions of security such as those where the function is specified or chosen at random. The difference between the two methods lies in the description of the success of the adversary. In the fixed parameter framework, parameters for time and the adversarial advantage are given. On the other hand, in the asymptotic framework the values for time and the adversarial advantage are related to the output size of the hash function. It is for this reason that the asymptotic framework is used to describe all of the properties required by traditional hash functions and dynamic hash functions.

2.3.1 Experiment Descriptions

Experiments describe the steps taken by an adversary and the *environment* for a particular security property with respect to a hash function. Experiments are parameterized by the adversary, and possibly another parameter, so that the same experiment can be used to describe the success of any adversary. The notation $\mathbf{Exp}_H^{\mathtt{xxx}}(A)$

labels the experiment describing property XXX using hash function H and adversary A.

Each step in the experiment description is performed sequentially. If a step does not involve the adversary it is assumed that the step is performed by the environment. The adversary only has knowledge of those values explicitly passed as inputs. The result of an experiment is binary. If the adversary is able to satisfy the condition in the experiment, then a one (1) is returned. If the adversary fails to satisfy the condition, then a zero (0) is returned.

2.3.2 The Fixed and Asymptotic Frameworks

As mentioned above, the description of how successful an adversary is can be described in two ways. The first method is with fixed parameters for both time and the adversarial advantage. The fixed parameter framework dictates that for all adversaries that run in time less than t, the adversarial advantage $\mathbf{Adv}_H^{\mathtt{XXX}}(A)$ must be less than ϵ for the function to posses the property XXX described by experiment $\mathbf{Exp}_H^{\mathtt{XXX}}(A)$. The t and ϵ are usually not defined in the fixed parameter framework. The fixed parameter framework is often used to compare the relative values for t and ϵ for different security properties. For this reason the fixed parameter framework was chosen for this dissertation. Using the fixed parameter framework allows definitions for traditional and dynamic hash functions to be compared. Also, it allows for notions of security to be discussed.

The asymptotic framework describes the success of an adversary related to the output size of the hash function. Because the upper bound on the amount of work needed to break any security property is dictated by the output size of the hash function, it is natural to relate the success of the adversary to the output size of the function. In the asymptotic framework it is required that t/ϵ be as large as possible and yet $t/\epsilon \leq 2^n$.[4] Taken to the extreme, if there is an almost zero advantage

[4]For collision resistance, $t/\epsilon \leq 2^{n/2}$.

by the adversary, $\epsilon \approx 0$, then $t \approx 2^n$ for preimage resistance and $t \approx 2^{n/2}$ for collision resistance. The trade-off between time and adversarial advantage should be determined by the application in which the function is being used. However, in the general case it is desirable for $t/\epsilon = 2^n$ for preimage resistance and $t/\epsilon = 2^{n/2}$ for collision resistance.

2.4 Formal Definitions of Security Properties

The properties a hash function must possess to be considered cryptographically secure are defined by experiments. These experiments define a series of steps taken by the environment. The result of each experiment is either a 1 or 0. A 1 denotes a successful experiment. Stated differently, a result of 1 means that an adversary was able to successfully attack the property the experiment tests. A result of 0 denotes the failure of an adversary to successfully attack the property. While not explicitly stated for each definition, the experiment is carried out multiple times using the same adversary. The probability is the average of the results unless otherwise noted.

The advantage an adversary has over a generic or brute force attack is called the *adversarial advantage*. For an adversary A, a function family $\mathcal{H}$, and a property $\texttt{XXX}$, the adversarial advantage is denoted by: $\mathbf{Adv}_H^{\texttt{XXX}}(A)$.

2.4.1 Preimage Resistance

The formal definition for a preimage resistant hash function family is constructed using the experiment from Figure 2.1. A hash function family is preimage resistant if inverting more than a negligible fraction of the domain points is a hard problem. Definition 2.4.1 formally defines preimage resistance.

Experiment $\mathbf{Exp}_H^{\mathrm{Pre}}(A)$
$K \xleftarrow{\$} \mathcal{K}$
$M_1 \xleftarrow{\$} \mathcal{M}$
$Y \leftarrow H_K(M_1)$
$M_2 \xleftarrow{\$} A(K, Y)$
if$(Y = H_K(M_2))$
return 1
else
return 0

Figure 2.1. Preimage resistance experiment.

Definition 2.4.1 (Preimage Resistant Hash Function Family) *A hash function family $\mathcal{H}$ is (t, ϵ)-preimage resistant if $\boldsymbol{Adv}_H^{Pre}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{Pre}(A) = Pr\big[\boldsymbol{Exp}_H^{Pre}(A) = 1\big],$$

and the probability is taken over all $K \in \mathcal{K}$, $M \in \mathcal{M}$ and the random choices of A.

Definition 2.4.1 formally defines what it means for a preimage resistant hash function family to be "hard to invert for essentially all pre-specified outputs".

2.4.2 Second Preimage Resistance

A hash function family is second preimage resistance if given a digest and a message, it is a hard problem to find a second message that hashes to the same digest. It has been shown in numerous places ([69], [48], etc) that collision resistance, explained next, implies second preimage resistance. The formal definition for a second preimage resistant hash function family is constructed using the experiment from Figure 2.2.

Experiment $\mathbf{Exp}_H^{\text{Sec}}(A)$

$K \xleftarrow{\$} \mathcal{K}$

$M_1 \xleftarrow{\$} \mathcal{M}$

$Y \leftarrow H_K(M_1)$

$M_2 \xleftarrow{\$} A(K, M_1, Y)$

if$(Y = H_K(M_2)$ **and** $M_1 \neq M_2)$

  return 1

else

  return 0

Figure 2.2. Second preimage resistance experiment.

Definition 2.4.2 (Second Preimage Resistant Hash Function Family) *A hash function family $\mathcal{H}$ is (t, ϵ)-second preimage resistant if $\boldsymbol{Adv}_H^{Sec}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{Sec}(A) = Pr\big[\boldsymbol{Exp}_H^{Sec}(A) = 1\big],$$

and the probability is taken over all $K \in \mathcal{K}$, $M \in \mathcal{M}$ and the random choices of A.

Definition 2.4.2 states exactly how hard it is to find "any second input which hash the same output as any specific input".

2.4.3 Collision Resistance

A hash function family is collision resistant if finding two different messages that hash to the same digest is a hard problem. The formal definition for a collision resistant hash function family is constructed using the experiment from Figure 2.3.

Experiment $\mathbf{Exp}_H^{\mathrm{Col}}(A)$
$K \xleftarrow{\$} \mathcal{K}$
$(M_1, M_2) \xleftarrow{\$} A(K)$
if$(H_K(M_1) = H_K(M_2)$ **and** $M_1 \neq M_2)$
return 1
else
return 0

Figure 2.3. Collision resistance experiment.

Definition 2.4.3 (Collision Resistant Hash Function Family) *A hash function family $\mathcal{H}$ is (t, ϵ)-collision resistant if $\boldsymbol{Adv}_H^{Col}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{Col}(A) = Pr\big[\boldsymbol{Exp}_H^{Col}(A) = 1\big],$$

and the probability is taken over all $K \in \mathcal{K}$ and the random choices of A.

Definition 2.4.3 states exactly how hard it is to find "any two distinct inputs which hash to the same output".

2.5 Notions of Security

In each definition if the probability of success is maximized, instead of averaged, over some parameter in the experiment, then the definition can be expanded into several notions of security. Seven notions of security can be defined with respect to preimage resistance, second preimage resistance and collision resistance. These seven notions are summarized in Table 2.1. The prefix *a* or *e* represents *always* or *everywhere* respectively.[5] If a function family is secure for any fixed function, then

[5]These terms are taken from [69].

Table 2.1
The seven notions of security for traditional hash functions [69].

Name	Given	Aliases
Pre	Random H, Random $H(M)$	One-Way-Function
aPre	Fixed H, Random $H(M)$	
ePre	Random H, Fixed $H(M)$	
Sec	Random H, Random M	Weak Collision Resistance
aSec	Fixed H, Random M	
eSec	Random H, Fixed M	Universal One-Way-Function
Col	Random H	Strong Collision Resistance

it is always secure. If a function family is secure for any fixed challenge, then it is everywhere secure.

2.5.1 Preimage Resistance

Using Definition 2.4.1 as the basis, the different notions of preimage resistance are defined as follows, taken from [69]. The experiments for the notions of always and everywhere are defined in Figure 2.4.

Definition 2.5.1 (Always Preimage Resistant) *A hash function family $\mathcal{H}$ is (t, ϵ)-always preimage resistant if $\boldsymbol{Adv}_H^{aPre}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{aPre}(A) = \max_{K \in \mathcal{K}} \left\{ Pr\left[\boldsymbol{Exp}_H^{aPre}(A, K) = 1\right] \right\},$$

and the probability is maximized over all $K \in \mathcal{K}$ for each $M \in \mathcal{M}$ and the random choices of A.

Always preimage resistant, aPre, is a stronger notion of security than preimage resistant. Always preimage resistant relates to a particular hash function such as

Experiment $\mathbf{Exp}_H^{\text{aPre}}(A, K)$	Experiment $\mathbf{Exp}_H^{\text{ePre}}(A, Y)$
$M_1 \xleftarrow{\$} \mathcal{M}$	$K \xleftarrow{\$} \mathcal{K}$
$Y \leftarrow H_K(M_1)$	$M \xleftarrow{\$} A(K, Y)$
$M_2 \xleftarrow{\$} A(K, Y)$	**if**$(Y = H_K(M))$
if$(Y = H_K(M_2))$	**return** 1
return 1	**else**
else	**return** 0
return 0	

Figure 2.4. Notions of preimage resistance.

SHA-1, where the key is fixed. The difference between aPre and Pre is that the probability is not averaged over all $K \in \mathcal{K}$ and $M \in \mathcal{M}$. Instead the probability is taken over all $M \in \mathcal{M}$ for each $K \in \mathcal{K}$ and only the maximum probability is used as the measure of how secure the function is.

Definition 2.5.2 (Everywhere Preimage Resistant) *A hash function family $\mathcal{H}$ is (t, ϵ)-everywhere preimage resistant if $\boldsymbol{Adv}_H^{ePre}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{ePre}(A) = \max_{Y \in R} \left\{ Pr\left[\boldsymbol{Exp}_H^{ePre}(A, Y) = 1\right] \right\},$$

and the probability is maximized over all $Y \in R$ for each $K \in \mathcal{K}$ and the random choices of A.

Everywhere preimage resistant relates to finding the preimage of a particular digest. One should note that instead of selecting a message to generate the digest, all digests (range points) are considered, even those that might not be the image of a message. For ePre, probabilities are calculated over all $K \in \mathcal{K}$ for each $Y \in R$.

2.5.2 Second Preimage Resistance

Based on Definition 2.4.2, the different notions of second preimage resistance are defined as follows using the experiments from Figure 2.5. These notions are analogous to the those for preimage resistance, but apply to second preimage resistance.

Experiment $\mathbf{Exp}_H^{\text{aSec}}(A, K)$	Experiment $\mathbf{Exp}_H^{\text{eSec}}(A, M_1)$
$M_1 \xleftarrow{\$} \mathcal{M}$	$K \xleftarrow{\$} \mathcal{K}$
$Y \leftarrow H_K(M_1)$	$Y \leftarrow H_K(M_1)$
$M_2 \xleftarrow{\$} A(K, M_1)$	$M_2 \xleftarrow{\$} A(K, M_1)$
if$(Y = H_K(M_2))$	**if**$(Y = H_K(M_2))$
return 1	**return** 1
else	**else**
return 0	**return** 0

Figure 2.5. Notions of second preimage resistance.

Definition 2.5.3 (Always Second Preimage Resistant) *A hash function family $\mathcal{H}$ is (t, ϵ)-always second preimage resistant if $\boldsymbol{Adv}_H^{aSec}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{aSec}(A) = \max_{K \in \mathcal{K}} \left\{ Pr\left[\boldsymbol{Exp}_H^{aSec}(A, K) = 1\right] \right\},$$

and the probability is maximized over all $K \in \mathcal{K}$ for each $M \in \mathcal{M}$ and the random choices of A.

Always second preimage resistance states that functions such as SHA-1 are second preimage resistant. The probability is calculated over all $M \in \mathcal{M}$ for each $K \in \mathcal{K}$, the same as is done for preimage resistance.

Definition 2.5.4 (Everywhere Second Preimage Resistant) *A hash function family $\mathcal{H}$ is (t, ϵ)-everywhere second preimage resistant if $\boldsymbol{Adv}_H^{eSec}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{eSec}(A) = \max_{M_1 \in \mathcal{M}} \left\{ Pr\left[\boldsymbol{Exp}_H^{eSec}(A, M_1) = 1\right] \right\},$$

and the probability is maximized over all $M \in \mathcal{M}$ for each $K \in \mathcal{K}$ and the random choices of A.

Everywhere second preimage resistance relates to the idea that it is computationally difficult to find a second message for a particular message that will result in the same digest. Here the probability is calculated over all $K \in \mathcal{K}$ for each $M \in \mathcal{M}$. The maximum probability is used as the indicator of the strength of the function.

2.5.3 Collision Resistance

For collision resistance there is no notion for maximizing over all $K \in \mathcal{K}$. While it would be hard for an adversary to construct a program to find a collision, the algorithm would be efficient [69]. One might argue that an adversary could simply possess a lookup table for each $K \in \mathcal{K}$ and then run the efficient algorithm for that particular K. As stated in Section 2.2, all definitions are parameterized by the time used by the adversary. In such an attack the size of the adversary's program is included in the overall time complexity of the adversary.

2.5.4 Implications Between Notions of Security

The notions of security for preimage resistance and second preimage resistance have natural implications to the standard definitions for preimage resistance and second preimage resistance. Because the probability is maximized over a particular parameter of the experiment, a stronger notion of security is defined. When the maximum probability is used all other probabilities must be smaller, and therefore the average of the probabilities cannot be larger than the largest. The opposite

implication however is not true. All of the probabilities for some parameter could be potentially very small except for one. This single large probability is the measure of the notion, providing a larger advantage to the adversary than the traditional property.

Theorem 2.5.1 *Given a hash function H, the following implications are true for all of the notions of security for a hash function:*

1. *If H possesses the a`XXX` security property, then H also possesses the `XXX` security property.*

2. *If H possesses the e`XXX` security property, then H also possesses the `XXX` security property.*

The proof of this theorem is given in [69].

3 DYNAMIC CRYPTOGRAPHIC HASH FUNCTIONS

This chapter introduces a new type of cryptographic hash function and the foundation of this dissertation, the *dynamic cryptographic hash function*. On a superficial level, the only difference between a traditional cryptographic hash function and a dynamic cryptographic hash function is that a dynamic cryptographic hash function takes a second input. The second input, the *security parameter*, specifies the level of security expected from the function. Increasing the security parameter should increase the security of the digest produced by dynamically changing how the digest is computed, hence the name. This dissertation presents the first formal definition of a dynamic hash function and investigates the security properties of dynamic cryptographic hash functions.

3.1 Background and Introduction

While the concept of a hash function that requires two inputs is not new, this type of function is unique. Dynamic cryptographic hash functions are the only type of hash function in which the second parameter specifies how the digest is computed and the size of the digest. It is the fact that the security parameter changes both of these attributes that makes this type of function unique.

Message authentication codes, for example, require two inputs (see Section 1.4.1) however they are inherently different from dynamic hash functions. One difference is that the size of a message authentication code does not change when the key changes. Also, most message authentication code constructions work by concatenating the key to the message in some secure manner. This is inherently different from changing the way a digest is computed. It is not enough to simply concatenate the security parameter to the message being hashed to create a dynamic hash function. The

security parameter must be related to the expected level of security of the digest. As the security parameter increases, the expected level of security of the digest should also increase. The same concept does not exist for message authentication codes.

Other hash functions, such as HAVAL [78], VSH [15], and functions constructed from expander graphs [14], have been designed to create different size digests. While such functions change the size of the digest created, the core way in which the digest is computed does not change. For example, if the hard problem used in the VSH function turns out to be easy, modifying the size of the digest does not help to keep the function secure. The same is true for functions such as HAVAL.

A dynamic hash function must change how a digest is computed based on the security parameter in two ways. First, the bit length of the digest is a function of the security parameter. Second, the computation of the digest must depend on the security parameter. These requirements ensure, among other things, that the digests computed using two different security parameters and the same message are unrelated.

To enable users to select the expected level of security, the bit length of the digest is a function of the security parameter. While the size of the digest is not equal to the complexity required to break the function, it does provide an upper bound on the amount of work needed to break the function. The larger the digest, the longer it will take, using a brute force approach, to find two messages that collide or a preimage of a given digest. By being able to choose the expected level of security, schemes that use hash functions (such as hash-then-sign) can now appropriately select an expected level of security that is comparable to the expected level of security of the rest of the protocol.

3.2 Definition of a Dynamic Cryptographic Hash Function

A dynamic cryptographic hash function is the same as a traditional cryptographic hash function except that a security parameter specifies how the function computes

the digest and the size of the digest. The definition of a traditional hash function is modified to define a dynamic hash function as follows.

Definition 3.2.1 (Dynamic Hash Function) *Let d, λ and υ be monotone increasing functions from $\mathbb{N} \to \mathbb{N}$ such that $d(s) > 0$ and $0 < \lambda(l) \leq \upsilon(l)$. A dynamic hash function is a function*

$$H : \Sigma^* \times \mathbb{N} \to \Sigma^* \cup \{\text{“undefined”}\},$$

such that when $|M| = l$ and $\lambda(l) \leq s \leq \upsilon(l)$, $|H(M, s)| = d(s)$. If it is not true that $\lambda(l) \leq s \leq \upsilon(l)$, then $H(M, s)$ is undefined. The functions λ, υ and d all run in polynomial time.

To enable a dynamic hash function to be quickly computed, there may be restrictions on the values of s and $d(s)$. For example, $d(s)$ might be restricted to a multiple of the word size of the target architecture. In the broadest sense of the definition these restrictions are not imposed.

Before defining the security properties for a dynamic hash function, a dynamic hash function family must be defined. The reason is the same as that discussed in Section 2.4 for traditional hash functions. If a function family is not defined, then a probabilistic polynomial time algorithm could exist which has a table of (message, security parameter, digest) triples encoded in it for a specific dynamic hash function. An attacker can use this algorithm to break the function [62]. However, defining a function family as an infinite family of finite sets of hash functions prevents such an algorithm from running in polynomial time for all hash functions in the family.

Definition 3.2.2 (Dynamic Hash Function Family) *A dynamic hash function family $\mathcal{H}$ is an infinite set of functions where each function in the family is indexed by a key K, and each function is of the form*

$$H_K : \Sigma^l \times \mathbb{N} \to \Sigma^* \cup \{\text{“undefined”}\},$$

so that $\forall s \in [\lambda(l), \upsilon(l)]$, $H_K(\cdot, s) : \Sigma^l \to \Sigma^{d(s)}$.

The same three requirements imposed on a traditional hash function family are also imposed on the dynamic hash function family $\mathcal{H}$.

1. $\mathcal{H}$ is accessible, that is, there is a probabilistic polynomial time algorithm, that on input K outputs an instance H_K.

2. $D = \Sigma^l$ is samplable, that is, there is a probabilistic polynomial time algorithm, that selects an element uniformly from D.

3. H_K is polynomial time computable, that is, there is a polynomial time algorithm (polynomial in s and in $|M|$) that computes $H_K(M, s)$.

3.3 Traditional Properties for Dynamic Hash Functions

One should note that a dynamic cryptographic hash function creates a family of traditional cryptographic hash functions, each possessing the required security properties of a traditional cryptographic hash function.

Because a dynamic cryptographic hash function takes a second parameter that determines the digest's size and how the function is computed, the definitions for the traditional properties must be modified appropriately. Preimage resistance, second preimage resistance, and collision resistance are defined for dynamic hash functions in Definition 3.3.1. The experiments are the same as for traditional hash functions except a dynamic hash function is used instead. Figures 3.1, 3.2, and 3.3 define the experiments for these properties for dynamic hash functions.

The adversarial model used to define the security properties for dynamic hash functions is similar to the one used for traditional hash functions. The only difference between the adversarial model described in Section 2.2 and the one used in the following definitions is access to the function d. The adversarial model for traditional hash functions attack hash functions where the digest is the same fixed size. However, with dynamic hash functions the size is determined by a function, d. As with access

Experiment $\mathbf{Exp}_H^{\text{Pre}}(A, s)$

$K \xleftarrow{\$} \mathcal{K}$

$M_1 \xleftarrow{\$} D$

$Y \leftarrow H_K(M_1, s)$

$M_2 \xleftarrow{\$} A(K, Y, s)$

if$(Y = H_K(M_2, s))$

  return 1

else

  return 0

Figure 3.1. Preimage resistance experiment for dynamic hash functions.

Experiment $\mathbf{Exp}_H^{\text{Sec}}(A, s)$

$K \xleftarrow{\$} \mathcal{K}$

$M_1 \xleftarrow{\$} D$

$Y \leftarrow H(M_1, s)$

$M_2 \xleftarrow{\$} A(K, M_1, Y, s)$

if$(Y = H_K(M_2, s)$ **and** $M_1 \neq M_2)$

  return 1

else

  return 0

Figure 3.2. Second preimage resistance experiment for dynamic hash functions.

to the hash function itself, the adversary has access to the algorithm for computing the digest size and can compute a digest size in unit time.

Experiment $\mathbf{Exp}_H^{\mathrm{Col}}(A, s)$
$K \xleftarrow{\$} \mathcal{K}$
$(M_1, M_2) \xleftarrow{\$} A(K, s)$
if$(H_K(M_1, s) = H_K(M_2, s)$ **and** $M_1 \neq M_2)$
return 1
else
return 0

Figure 3.3. Collision resistance experiment for dynamic hash functions.

Definition 3.3.1 *A dynamic hash function family $\mathcal{H}$ is (t, ϵ)-preimage resistant if there exists an s so that $\boldsymbol{Adv}_H^{Pre}(A, s) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{Pre}(A, s) = Pr\big[\boldsymbol{Exp}_H^{Pre}(A, s) = 1\big].$$

A dynamic hash function family $\mathcal{H}$ is (t, ϵ)-second preimage resistant if there exists an s so that $\boldsymbol{Adv}_H^{Sec}(A, s) < \epsilon$ for all adversaries A with a running time less than t, where

$$\boldsymbol{Adv}_H^{Sec}(A, s) = Pr\big[\boldsymbol{Exp}_H^{Sec}(A, s) = 1\big].$$

A dynamic hash function family $\mathcal{H}$ is (t, ϵ)-collision resistant if there exists an s so that $\boldsymbol{Adv}_H^{Col}(A, s) < \epsilon$ for all adversaries A with a running time less than t, where

$$\boldsymbol{Adv}_H^{Col}(A, s) = Pr\big[\boldsymbol{Exp}_H^{Col}(A, s) = 1\big].$$

For each experiment the probability is taken over all $K \in \mathcal{K}$, $M \in \mathcal{M}$, and the random choices of A.

The inclusion of a second parameter in a dynamic hash function dictates the need for additional security properties. If the modified versions of the traditional properties

were the only properties considered for dynamic cryptographic hash functions, then they would be improperly used. For example, the traditional properties do not provide statements about the security of two digests created from the same message but with different security parameters. By the definition of a dynamic cryptographic hash function, one would expect these digests to be unrelated. Unfortunately, this idea is never explicitly stated in the traditional properties.

It is important to note that the security parameter of a dynamic hash function is not a key. Therefore, one cannot expect that it is infeasible to compute the security parameter given a digest. The security parameter, by definition, determines the size of the digest produced by the dynamic hash function. Therefore, determining the security parameter might be as easy as counting the number of bits in the digest.

3.4 Dynamic Versions of the Traditional Properties

The traditional properties defined in Section 3.3 require that the same security parameter is used for the digests being compared. Because the security parameter dictates the way the digest is computed, properties must be defined for digests created using different security parameters. These properties are defined by allowing the adversary to choose the value of a second security parameter. While the security parameter also dictates the size of the digest, it does not make sense to compare digests of different sizes. This restricts the adversary to those security parameters that keep the digests the same size.

Definition 3.4.1 (Dynamic Preimage Resistance) *A dynamic hash function family $\mathcal{H}$ is (t, ϵ)-dynamic preimage resistant if $\boldsymbol{Adv}_H^{dPre}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{dPre}(A) = Pr\big[\boldsymbol{Exp}_H^{dPre}(A) = 1\big],$$

and the probability is taken over all $K \in \mathcal{K}$, $M \in \mathcal{M}$, $s \in [\lambda(l), \upsilon(l)]$, and the random choices of A.

Experiment $\mathbf{Exp}_H^{\mathrm{dPre}}(A)$

$K \xleftarrow{\$} \mathcal{K}$

$M_1 \xleftarrow{\$} \mathcal{M}$

$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$

$Y \leftarrow H_K(M_1, s_1)$

$(M_2, s_2) \xleftarrow{\$} A(K, Y, s_1)$

if$(Y = H_K(M_2, s_2)$ **and** $d(s_1) = d(s_2))$

  return 1

else

  return 0

Figure 3.4. Dynamic preimage resistance experiment.

Definition 3.4.1 states that the probability of an adversary successfully finding a message and a security parameter that will hash to the given digest is negligible.

Dynamic preimage resistance is needed in the following scenario where a dynamic hash function is used for the Linux password system rather than a traditional hash function. In Linux systems there are two files that dictate user authentication: `passwd` and `shadow`. Instead of a traditional hash function, a dynamic hash function can be used with only slight modifications to the authentication system. The `passwd` file stores user name and security parameter pairs. The `shadow` file stores user name and digest pairs, where the digest is computed using the security parameter in the `passwd` file and the user's password. Everyone has read access to the `passwd` file and only the administrator has access to the `shadow` file.

Assume Alice is a user on the system with a password of P. The digest of her password is stored in the `shadow` file using security parameter s. Through a vulnerability in the system, Mallory is able to gain write access to the `passwd` file and read access to the `shadow` file. Mallory is now able to read Alice's digest from the `shadow` file and the security parameter from the `passwd` file.

If the dynamic hash function used on the system does not have dynamic preimage resistance, then Mallory can compute a password P' and security parameter s' such that $H(P, s) = H(P', s')$.[1] Because Mallory has write access to the `passwd` file, she can change Alice's security parameter from s to s'. Mallory can now use the new password P' to authenticate with the system as if she were Alice. When Mallory uses the user name "Alice" and the password P', the system reads the security parameter s' from the `passwd` file and computes the digest $H(P', s')$. The digest is then compared with the one stored in the `shadow` file, and Mallory is granted access to the system as Alice.

Experiment $\mathbf{Exp}_H^{\text{dSec}}(A)$

$K \xleftarrow{\$} \mathcal{K}$
$M_1 \xleftarrow{\$} \mathcal{M}$
$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$Y \leftarrow H_K(M_1, s_1)$
$(M_2, s_2) \xleftarrow{\$} A(K, M_1, Y, s_1)$
if$(Y = H_K(M_2, s_2)$ **and** $M_1 \neq M_2$ **and** $d(s_1) = d(s_2))$
  return 1
else
  return 0

Figure 3.5. Dynamic second preimage resistance experiment.

Definition 3.4.2 (Dynamic Second Preimage Restance) *A dynamic hash function family $\mathcal{H}$ is (t, ϵ)-dynamic second preimage resistant if $\boldsymbol{Adv}_H^{dSec}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{dSec}(A) = Pr\big[\boldsymbol{Exp}_H^{dSec}(A) = 1\big],$$

[1]Note that P may or may not be the same as P' and s may or may not be the same as s'.

and the probability is taken over all $K \in \mathcal{K}$, $M \in \mathcal{M}$, $s \in [\lambda(l), \upsilon(l)]$, *and the random choices of* A.

Definition 3.4.2 states that the probability of an adversary successfully finding a different message and a security parameter that will hash to the given digest is negligible.

Dynamic second preimage resistance is required in the following scenario where digests are compared manually and assumptions are made about the security parameter that is used. Assume a key server with a web interface and a protocol for downloading keys, such as a PGP key server. Users can search, by e-mail address, for another user's public key. The web page displays e-mail addresses and the corresponding digest of that user's public key when hashed with a dynamic hash function using security parameter s. When a user clicks on an e-mail address a file is downloaded that contains the public key and the security parameter used to compute the digest. The security parameter is included in the file to provide algorithm agility, one of the main reasons for using a dynamic hash function. If certain security parameters are found to be insecure in the future, the security parameter is change in the files and the website is updated to reflect the new digests. These changes can all be done without the user updating the encryption software on his or her computer.

Assume Mallory is able to control the files that are downloaded from the website, but has no control over the web pages that are generated when a search is performed. Alice visits the website in search of Bob's public key, K_{Bob}. She searches for Bob's e-mail address, finds it and clicks on the link to download his key. Instead of downloading the proper file, Mallory is able to intercept it and change it to a file she has precomputed. The new files contains her public key $K_{Mallory}$ and a different security parameter, s'. Alice's encryption program computes $H(K_{Mallory}, s')$ and displays this digest on the screen for Alice to check against the digest associated with Bob's e-mail address on the website.

If the dynamic hash function that was used in the following scenario does not have dynamic second preimage resistance, then the digest of Mallory's key and new security

parameter will be the same as the one on the web-page: $H(K_{Mallory}, s') = H(K_{Bob}, s)$. Alice will think that she has Bob's public key after verifying the digest computed by her encryption program is the same as the one on the website. Actually, the key that Alice has is Mallory's. Mallory can now read any message sent from Alice to Bob. Mallory can also re-encrypt the message with Bob's actual public key and send it to Bob as if nothing has happened. The same scenario can occur in reverse so that Mallory can intercept any message sent from Bob to Alice.

Experiment $\mathbf{Exp}_H^{\text{dCol}}(A)$

$K \xleftarrow{\$} \mathcal{K}$

$l \xleftarrow{\$} \mathbb{N}$

$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$

$(M_1, M_2, s_2) \xleftarrow{\$} A(K, s_1)$

if$(H_K(M_1, s_1) = H_K(M_2, s_2)$ **and** $M_1 \neq M_2$ **and** $d(s_1) = d(s_2))$

  return 1

else

  return 0

Figure 3.6. Dynamic collision resistance experiment.

Definition 3.4.3 (Dynamic Collision Resistance) *A dynamic hash function family $\mathcal{H}$ is (t, ϵ)-dynamic collision resistant if $\boldsymbol{Adv}_H^{dCol}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{dCol}(A) = Pr\big[\boldsymbol{Exp}_H^{dCol}(A) = 1\big],$$

and the probability is taken over all $K \in \mathcal{K}$, $M \in \mathcal{M}$, $s \in [\lambda(l), \upsilon(l)]$, and the random choices of A.

Definition 3.4.3 states that the probability of an adversary successfully finding two messages and a security parameter that will hash to the same digest using different security parameters is negligible.

Dynamic collision resistance is needed in the following contract signing scenario between Mallory and Bob. Mallory agrees to buy Bob's car for $1,000. Bob sends Mallory a security parameter s to use for computing the digest of the contract in which Mallory agrees to buy his car for $1,000. Mallory creates a contract C stating the price she will pay for the car and sends it to Bob. Bob computes the digest $H(C, s)$ and sends a digitally signed copy of the digest back to Mallory: $\textsc{Sign}(H(C, s))$. Before Mallory pays Bob for his car she contests Bob's version of the contract, C, with a trusted mediator Trent.

If the dynamic hash function used in this scenario does not have dynamic collision resistance, then Mallory is able produce a second contract C', which states that she will only pay Bob $10 for his car, and a second security parameter s' such that $H(C', s') = H(C, s)$. Because the two digests are the same, the two signatures will be the same: $\textsc{Sign}(H(C, s)) = \textsc{Sign}(H(C', s'))$. The trusted mediator Trent will verify Bob's signature and compute the digest of the second contract C' with the second security parameter s'. The two digests will be the same, and Mallory will be awarded Bob's car by Trent for only $10.

The fact that only small changes are needed for these new security properties to be defined is not surprising. The dynamic versions of the traditional security properties are the generalized versions of the traditional properties. In fact, there is a natural implication between the dynamic versions of the traditional security properties and the traditional security properties. These implications are discussed in Section 3.6.

3.5 Properties Without Traditional Analogs

While the properties in Section 3.4 are analogous to the traditional properties, there are two new properties, *security parameter collision resistance* and *digest resistance*, that are not. These properties are only applicable to dynamic hash functions because they directly relate to the function's ability to dynamically generate digests for different security parameters.

3.5.1 Security Parameter Collision Resistance

Security parameter collision resistance ensures that one cannot find a message that will result in the same digest for two different security parameters. It is the property of security parameter collision resistance that dictates dynamic hash functions are different from hash functions where the security parameter only affects the size of the digest. This property avoids having a weak hash function where if given $H(M, s_1) = H(M', s_2)$, it is probably the case that $M = M'$.

Experiment $\mathbf{Exp}_H^{\mathrm{PCol}}(A)$

$K \xleftarrow{\$} \mathcal{K}$
$l \xleftarrow{\$} \mathbb{N}$
$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$(M, s_2) \xleftarrow{\$} A(K, s_1)$
if$(H_K(M, s_1) = H_K(M, s_2)$ **and** $s_1 \neq s_2$ **and** $d(s_1) = d(s_2))$
    return 1
else
    return 0

Figure 3.7. Security parameter collision resistance experiment.

Definition 3.5.1 (Security Parameter Collision Resistance) *A dynamic hash function family $\mathcal{H}$ is (t, ϵ)-dynamic security parameter collision resistant if $\boldsymbol{Adv}_H^{PCol}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{PCol}(A) = Pr\big[\boldsymbol{Exp}_H^{PCol}(A) = 1\big],$$

and the probability is taken over all $K \in \mathcal{K}$, $M \in \mathcal{M}$, $s \in [\lambda(l), \upsilon(l)]$, and the random choices of A.

Definition 3.5.1 states that the probability of an adversary successfully finding a message and a security parameter that will hash to the same digest as the same message and the given security parameter is negligible.

Dynamic hash functions construction without security parameter collision resistance are vulnerable when used in the following type of scenario. Assume a computer system with $r = |\mathcal{S}|$ levels of access. Each level has a security parameter associated with it. For simplicity the security parameters are the integers $\{1, 2, \ldots, r\}$. Let P be a password, l be one of the r levels, and SIGN be the signature of the computer system. When an account is created on the system, the administrator assigns the access level and the user picks a password. The system computes and sends to the user $\text{SIGN}(H(P, l))$. A user authenticates with the system by sending P, l, and $\text{SIGN}(H(P, l))$ to the system. The server checks that the signature and the digest are both valid. Assume Alice is given an account on the computer system at level one. Alice chooses the password P for her account and the system computes and sends to Alice $\text{SIGN}(H(P, 1))$.

If the dynamic hash function used in the computer system does not have security parameter collision resistance, then Alice can choose a higher level of access l' and a password P such that $H(P, 1) = H(P, l')$. Because the two digests are the same, the signatures will be the same: $\text{SIGN}(H(P, 1)) = \text{SIGN}(H(P, l'))$. Alice can now authenticate with the system by sending P, l' and $\text{SIGN}(H(P, l'))$. The system will check the signatures and the digest, and authenticate Alice at the new level. Alice is now able to operate at an increased level of access than the level intended by the administrator.

3.5.2 Digest Resistance

Digest resistance ensures that it is not easy to create one digest from another. This property is motivated by the following situation. Suppose Alice has created a secret document. She posts the digest and the security parameter used to compute

the digest of her document on the Internet, staking her claim that she created the document. Bob argues that he is the original creator of the document. He has a digest using a different security parameter on his website which he claims proves he is the creator of the document. Carol, acting as a trusted mediator, has both Alice and Bob recompute the digest of their documents using a different security parameter that Carol chooses. Because the security parameter will change how the digest is computed, the procedure will allow Carol to determine if Alice and Bob actually have the same document without revealing to Carol what the two documents contain.

Experiment $\mathbf{Exp}_H^{\text{Dig}}(A)$
$K \xleftarrow{\$} \mathcal{K}$
$M \xleftarrow{\$} \mathcal{M}$
$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$Y_1 \leftarrow H_K(M, s_1)$
$(Y_2, s_2) \xleftarrow{\$} A(K, Y_1, s_1)$
if$(Y_2 = H_K(M, s_2)$ **and** $s_1 \neq s_2)$
return 1
else
return 0

Figure 3.8. Digest resistance experiment.

Definition 3.5.2 (Digest Resistance) *A dynamic hash function family $\mathcal{H}$ is (t, ϵ)-dynamic digest resistant if $\boldsymbol{Adv}_H^{Dig}(A) < \epsilon$ for all adversaries A with a running time less than t, where*

$$\boldsymbol{Adv}_H^{Dig}(A) = Pr\big[\boldsymbol{Exp}_H^{Dig}(A) = 1\big],$$

and the probability is taken over all $K \in \mathcal{K}$, $M \in \mathcal{M}$, $s \in [\lambda(l), \upsilon(l)]$, and the random choices of A.

Definition 3.5.2 states that the probability of an adversary successfully finding the digest of an unknown message and a security parameter, given a digest of the same message with a different security parameter, is negligible.

This property ensures, among other things, that a dynamic cryptographic hash function is not constructed by concatenating or truncating a standard cryptographic hash function. Another motivation for this property is to protect against attacks that would leverage the ability to reduce the digest space to a manageable size and then launch another attack against the hash function. For example, it is insecure for the 50-bit digest of a message to be constructed from the 160-bit digest of the same message. If this property is not ensured the following attack could be launched against a password authentication scheme.

Assume that a two computer system stores the hash of users' passwords in a central database. The security parameter for system A produces a 160-bit digest and the security parameter for system B produces a 100-bit digest. The only terminals to log into either system are connected to the central database via a secure communication link. The authentication is done by having the terminal compute the digest of the password and send it to the database for comparison. Suppose that Alice has accounts on both systems and that Bob is able to discover the hash of Alice's password for system A.

If the dynamic hash function used to compute the digest a of user's passwords does not have digest resistance, then Bob can compute Alice's 100-bit digest from her 160-bit digest. Because the digest and user name are sent from the secure terminal to the server, Bob can send the 100-bit digest and the user name "Alice" to authenticate with the system B without ever knowing Alice's password.

3.6 Implications Between Security Properties

The properties in Sections 3.3, 3.4, and 3.5 can be strengthened or weakened by allowing the adversary to choose the security parameter(s) or by providing the

security parameter(s). When the adversary is able to choose the security parameter(s) the property is strengthened. On the other hand, dictating the security parameter(s) to the adversary creates a weaker version of the security property. Experiments for the strengthened and weakened versions of all the security properties can be found in Appendix A.

Most of the security properties for dynamic hash functions are weaker or stronger versions of base properties. A natural implication is present between these classes of security properties. The implication follows from a general rule: properties that give more control to an adversary imply properties that give less control to an adversary. The rule is true because if an adversary is unable to show the function is insecure, with respect to some property, when complete control is given, then the adversary will also be unable to break the function when some of the control is taken away. Let `XXX` denote a security property. Let s`XXX` and w`XXX` denote the stronger and weaker versions of the security property `XXX` respectively. A function is broken, with respect to property `XXX`, if the adversary can find an example where the property does not hold. The function is secure, with respect to property `XXX`, if no counter example can be found.

Theorem 3.6.1 *Given a dynamic hash function H, the following implications are true for all of the security properties for a dynamic hash function:*

1. *If H possesses the s`XXX` security property, then H also possesses the `XXX` security property.*

2. *If H possesses the `XXX` security property, then H also possesses the w`XXX` security property.*

Proof The proof of both statements is done the same way. The only difference between the s`XXX` property and the `XXX` property is the loss of control of some parameter p by the adversary. The same is true for `XXX` and w`XXX`. A proof for both is provided simultaneously.

Let H be a dynamic hash function. Assume there does not exist an algorithm to break the s`XXX` (`XXX`) property of H; therefore, H possesses this property. Further assume that there is an algorithm, A, that breaks the `XXX` (w`XXX`) property of H. Stated differently, it is assumed that H has property s`XXX` (`XXX`) but this does not imply `XXX` (w`XXX`).

Using algorithm A, algorithm A' can always be constructed to break the s`XXX` (`XXX`) property of H. Algorithm A' is constructed by choosing at random a parameter p from the set of possible parameters. Algorithm A is run using the chosen parameter p as input. Algorithm A breaks the `XXX` (w`XXX`) property; therefore, A' will break the s`XXX` (`XXX`) property. Because an algorithm exists to break the s`XXX` (`XXX`) property of H, it is not secure. Therefore, the assumption that s`XXX` (`XXX`) does not imply `XXX` (w`XXX`) must be rejected and the theorem is proved correct. ■

While a dynamic hash function requires a security parameter, the same implication for collision resistance and second preimage resistance that applies for traditional hash functions still exists. This implication, along with the stronger and weaker versions of properties, are shown in Figure 3.9. An arrow from one property to another means that one property implies the other. For example, Col → Sec means that Col implies Sec.

Figure 3.9 also shows implications between the dynamic and traditional versions of preimage resistance, second preimage resistance, and collision resistance. These implications exist because the dynamic versions of these properties are the generalized versions of the traditional properties. It is clear that if a dynamic hash function possesses the generalized version of a property, then it will also possess the more specific version of the same property.

Theorem 3.6.2 *For each of the three traditional properties (Pre, Sec and Col), the dynamic versions of these properties imply the traditional properties.*

Proof The only difference between the traditional version and the dynamic version of a property is that in the dynamic version the adversary is able to choose a new

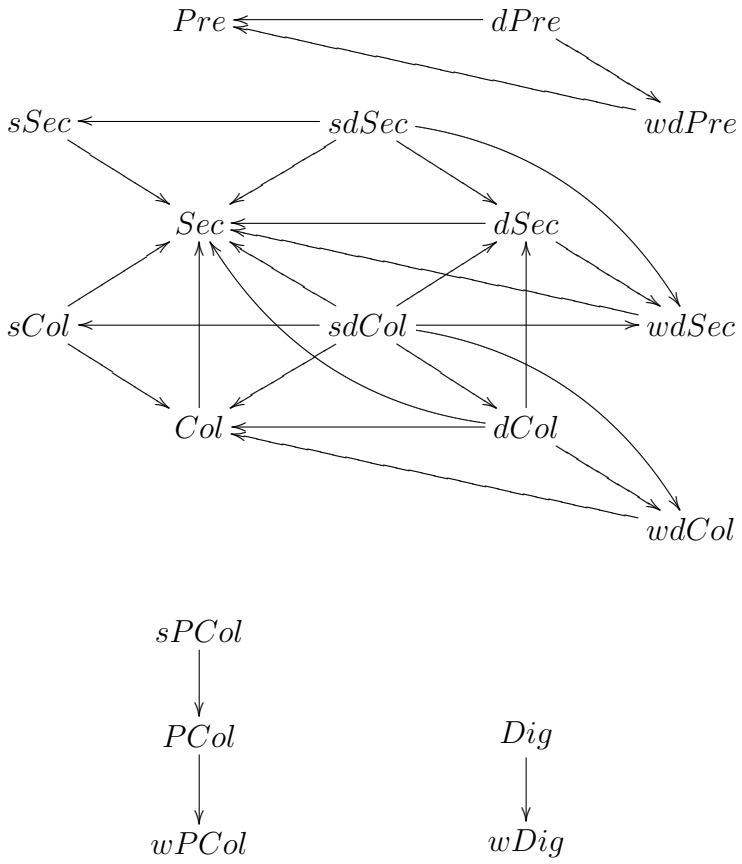

Figure 3.9. Implications between security properties for dynamic hash functions.

security parameter for the second digest. A proof by contradiction can be established by assuming that the dynamic hash function has the d`XXX` property but there is an algorithm A to break the `XXX` property, where `XXX` is one of Pre, Sec, or Col.

Using algorithm A, algorithm A' can be constructed to attack the d`XXX` property of the dynamic hash function. Algorithm A' sets $s' = s$ and then runs algorithm A. Because the dynamic versions do not require that $s \neq s'$ this will always work, if algorithm A breaks the `XXX` property. Therefore, the assumption that the dynamic hash function possesses the d`XXX` property but that this does not imply the `XXX` property must be rejected. ■

The final set of implications state that the weak dynamic version of a property implies the traditional property. This implication is slightly different from the others. The weak version of a dynamic properties specifies that both s and s' are given to the adversary. On the other hand, in the traditional property only s is given and the adversary is not allowed to choose an s'. The implication occurs because a proper subset of the instances of the weak dynamic version exists when $s = s'$. The instances in this subset are equivalent to the traditional security properties. Because all instances of a given property must be considered to determine if a dynamic hash function has the property or not, this implies that the weak dynamic version of a property implies the traditional version.

Theorem 3.6.3 *Let `XXX` represent one of the three traditional security properties: Pre, Sec, or Col. The weak dynamic version of `XXX`, wd`XXX`, implies the traditional security property `XXX`.*

Proof The traditional version of a property is a specific instance of the weak dynamic version. In the weak dynamic version both security parameters, s_1 and s_2, are given to the adversary. For certain instances of the experiment $s_1 = s_2$. When this occurs, the wd`XXX` property is the same as `XXX`. If there is no adversary that can break this instance of the wd`XXX` property, then there is no adversary that can break the `XXX` property. Therefore, wd`XXX` implies `XXX`. ■

3.7 Notions of Security

The same notions of security described in Section 2.5 can be applied to dynamic hash functions. Dynamic hash function families, like traditional hash function families, are infinite families of finite sets. A particular dynamic hash function can be chosen at random so that the notions of *always* and *everywhere* can be established in the same way. However, in defining these notions of security for dynamic hash functions, the security parameter must be considered.

Because dynamic hash functions require a security parameter, each notion must include the selection of a security parameter in the experiment that defines the adversary's advantage. While this security parameter must be chosen for each experiment, it does not make sense for notions of security to exist across different security parameters because the size of the digest is a function of the security parameter. The size of the digest bounds the complexity of attacking the hash function and therefore makes it nonsensical to evaluate notions of security for different digest sizes.

The traditional notions of security have been modified for dynamic hash functions and are defined in the same manner. One should note the change to ePre. The change is required because a digest must be chosen from the set of all digests of a specified length. Instead of choosing a random size range point, only valid domain points are used to construct range points. Otherwise it might be impossible for an adversary to find a preimage simply because the function does not create digests of that size with that security parameter.

The definitions for the adversarial advantage for aPre, ePre, aSec, and eSec are the same as those found in Definitions 2.5.1, 2.5.2, 2.5.3, and 2.5.4. The only change is in the experiment.

The new security properties described in Sections 3.4 and 3.5 can also be expanded into notions of security. The dynamic properties are analogous to the traditional ones but compare digests created with different security parameters. The adversary in these notions pick the second security parameter. Also, the original security pa-

Experiment $\mathbf{Exp}_H^{\text{aPre}}(A, K)$	Experiment $\mathbf{Exp}_H^{\text{ePre}}(A, M_1)$
$M_1 \xleftarrow{\$} D$	$K \xleftarrow{\$} \mathcal{K}$
$s \xleftarrow{\$} [\lambda(l), \upsilon(l)]$	$s \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$Y \leftarrow H_K(M_1, s)$	$Y \leftarrow H_K(M_1, s)$
$M_2 \xleftarrow{\$} A(K, Y, s)$	$M_2 \xleftarrow{\$} A(K, Y, s)$
if$(Y = H_K(M_2, s))$	**if**$(Y = H_K(M_2, s))$
return 1	**return** 1
else	**else**
return 0	**return** 0

Figure 3.10. Notions of preimage resistance for dynamic hash functions.

Experiment $\mathbf{Exp}_H^{\text{aSec}}(A, K)$	Experiment $\mathbf{Exp}_H^{\text{eSec}}(A, M_1)$
$M_1 \xleftarrow{\$} D$	$K \xleftarrow{\$} \mathcal{K}$
$s \xleftarrow{\$} [\lambda(l), \upsilon(l)]$	$s \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$Y \leftarrow H_K(M_1, s)$	$Y \leftarrow H_K(M_1, s)$
$M_2 \xleftarrow{\$} A(K, M_1, s)$	$M_2 \xleftarrow{\$} A(K, M_1, s)$
if$(Y = H_K(M_2, s))$	**if**$(Y = H_K(M_2, s))$
return 1	**return** 1
else	**else**
return 0	**return** 0

Figure 3.11. Notions of second preimage resistance for dynamic hash functions.

rameter must be given to the adversary. Again, the definitions are the same as the traditional ones. The only change is to the experiment. Figures 3.12 and 3.13 define the experiments for adPre, edPre, adSec, and edSec.

Experiment $\mathbf{Exp}_H^{\text{adPre}}(A, K)$	Experiment $\mathbf{Exp}_H^{\text{edPre}}(A, M_1)$
$M_1 \xleftarrow{\$} D$	$K \xleftarrow{\$} \mathcal{K}$
$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$	$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$Y \leftarrow H_K(M_1, s_1)$	$Y \leftarrow H_K(M_1, s_1)$
$(M_2, s_2) \xleftarrow{\$} A(K, Y, s_1)$	$(M_2, s_2) \xleftarrow{\$} A(K, Y, s_1)$
if$(Y = H_K(M_2, s_2))$	**if**$(Y = H_K(M_2, s_2))$
return 1	**return** 1
else	**else**
return 0	**return** 0

Figure 3.12. Notions of dynamic preimage resistance.

Experiment $\mathbf{Exp}_H^{\text{adSec}}(A, K)$	Experiment $\mathbf{Exp}_H^{\text{edSec}}(A, M_1)$
$M_1 \xleftarrow{\$} D$	$K \xleftarrow{\$} \mathcal{K}$
$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$	$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$Y \leftarrow H_K(M_1, s_1)$	$Y \leftarrow H_K(M_1, s_1)$
$(M_2, s_2) \xleftarrow{\$} A(K, M_1, s_1)$	$(M_2, s_2) \xleftarrow{\$} A(K, M_1, s_1)$
if$(Y = H_K(M_2, s_2))$	**if**$(Y = H_K(M_2, s_2))$
return 1	**return** 1
else	**else**
return 0	**return** 0

Figure 3.13. Notions of dynamic second preimage resistance.

There are no additional notions of security for security parameter collision resistance because the property is similar to collision resistance except that the collision is for the security parameter and not the message.

The two notions of security are applicable for digest resistance. Because the adversary is trying to form one digest from another, the probability can be measured

when the message used to create the original digest is fixed or random, along with when the function is fixed or random. The definition for the adversarial advantage is the same as previous notions, with Figure 3.14 defining the experiments.

Experiment $\mathbf{Exp}_H^{\text{aDig}}(A, K)$	Experiment $\mathbf{Exp}_H^{\text{eDig}}(A, M)$
$M \xleftarrow{\$} \mathcal{M}$	$K \xleftarrow{\$} \mathcal{K}$
$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$	$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$Y_1 \leftarrow H_K(M, s_1)$	$Y_1 \leftarrow H_K(M, s_1)$
$(Y_2, s_2) \xleftarrow{\$} A(K, Y_1, s_1)$	$(Y_2, s_2) \xleftarrow{\$} A(K, Y_1, s_1)$
if$(Y_2 = H_K(M, s_2)$ **and** $s_1 \neq s_2)$	**if**$(Y_2 = H_K(M, s_2)$ **and** $s_1 \neq s_2)$
return 1	**return** 1
else	**else**
return 0	**return** 0

Figure 3.14. Notions of digest resistance.

3.7.1 Implications Between Notions of Security

Figure 3.15 shows the implication graph between the notions of security for dynamic hash functions. The stronger and weaker versions of each property were not included to reduce the complexity of the graph. As in the case of the traditional notions of security, the *everywhere* and *always* security notions imply the standard notions of security even for the dynamic versions of the traditional properties. Also, the same logic that was used to form the implications for the traditional notions can be extended to the digest resistance property.

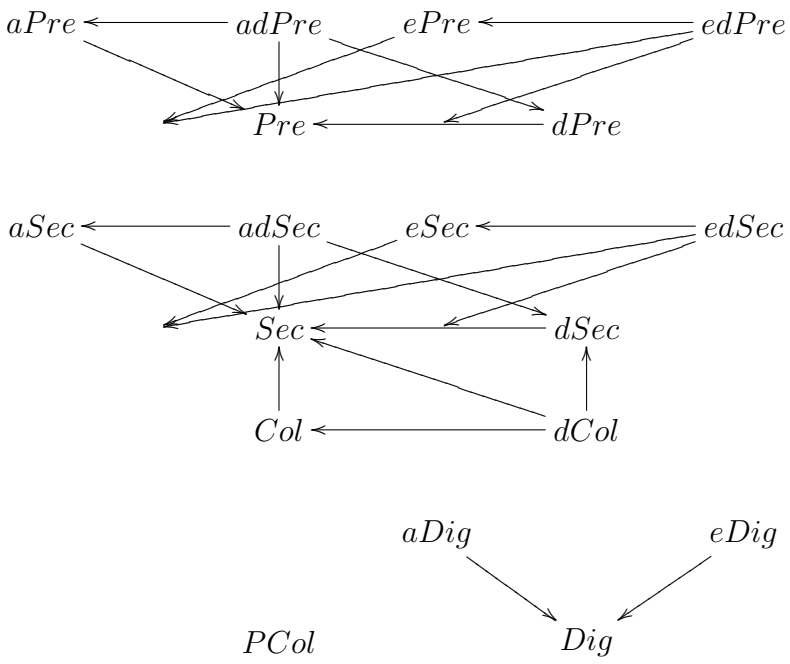

Figure 3.15. Implications between notions of security for dynamic hash functions.

3.8 Using Dynamic Hash Functions in Practice

Dynamic cryptographic hash functions are the next logical step in the evolution of hash functions. A dynamic cryptographic hash function can be used in all instances where a traditional hash function is used by simply hard-coding the security parameter. However, to take full advantage of a dynamic hash function, the security parameter must be incorporated into the design of the protocol. One should note that care needs to be taken when designing and implementing protocols so that an attacker cannot negotiate an artificially small security parameter or a security parameter that is not allowed by the function. Simple checks by the implementation can alleviate these types of problems.

With the ever advancing attacks against traditional hash functions, the need for more secure and dynamic functions is obvious. Instead of requiring an implementation change when an attack is discovered, certain vulnerable security parameters can be blacklisted as attacks are discovered. This is an important advantage of dynamic hash functions that should be leveraged by system designers. To properly leverage the flexibility inherent with dynamic hash functions, a mechanism is needed to blacklist security parameters that are deemed insecure. The mechanism, and its implementation, has certain security issues that are outside of the scope of this dissertation to discuss at length. However, the most obvious problem is one of policy. The policy used to update the blacklist must be well known and carefully thought out. If anyone is able to update the blacklist, then security parameters that are secure could be blacklisted causing a denial of service. Also, security parameters that are known to be vulnerable might never be blacklisted (or removed from the blacklist).

3.8.1 Expected Security for an Ideal Dynamic Hash Function

In the design of current systems, designers have an expected level of security for each cryptographic primitive used, with respect to some type of an attack. The security of a traditional hash function is usually measured by the expected number

of messages that must be tested before either a preimage or a collision is found. The same method can be applied to the five properties of a dynamic hash function. For the rest of this section it is assumed that the dynamic hash function is ideal.

The expected number of messages that must be tested for an attacker to discover a preimage, second preimage or collision for a dynamic hash function is the same as the expected number of messages for a traditional hash function. Because a dynamic hash function allows for variable size digests, the expected number of messages that must be tested varies as well. However, the expected number of messages is fixed for each security parameter. For dynamic preimage resistance, second preimage resistance and collision resistance, the expected number of messages that must be tested are $2^{d(s)}$, $2^{d(s)}$ and $2^{d(s)/2}$ respectively.

The additional properties are more difficult to quantify succinctly. The problem is that there is no guarantee such a message can ever be found to break the property. For example, if all of the security parameters produce different size digests, then security parameter collision resistance can never be broken. This is an important point to note for system designers. To ensure a dynamic hash function has security parameter collision resistance one need only pick a dynamic hash function such that the size of each digest is different for each security parameter.

Assuming that there are multiple security parameters that produce the same size digest, the expected number of messages that must be tested to break security parameter collision resistance is $2^{d(s)}$. The expected value comes from the properties of an ideal hash function. The two different security parameters used simulate two different hash functions. Lemma 5.2.1 provides a proof for this situation.

Digest resistance for an ideal dynamic hash function reduces to randomly choosing digests and testing to see if they are correct. The problem is that an attacker does not know when the correct digest is found (see Figure 3.8). Assuming the attacker was able to learn if a given digest is correct or not, the attacker would need to produce $2^{d(s)}$ digests before finding the correct one. The expected value is derived from the

question, "How many objects must be drawn at random from a bucket until a specific object is found?" Appendix B addresses this question.

3.8.2 Choosing the Right Security Parameter

When used properly, dynamic hash functions allow the proper level of security for each instantiation of a protocol. For example, a packet that has a lifetime of a few seconds probably does not require a digest that is 160 bits long. However, a document that must exist for twenty or thirty years would require a digest much larger than 160 bits. Instead of being forced to use two static hash functions, a single dynamic hash function can be used that can create a digest that is appropriate for both situations. The issue is then which security parameter to choose for each situation.

Assuming a general dynamic hash function, where no guidelines have been provided by the function designer, the selection of the security parameter should be related to the size of the digest required. If multiple security parameters provide the same size digest, it should be assumed that all of the security parameters produce the same level of security for that digest size. The only problem that remains is how big a digest should be for a certain scenario.

Each scenario is different and unfortunately no single answer is correct. When hash functions, dynamic or traditional, are used in conjunction with some other protocol, like hash-then-sign, the expected level of security provided by the hash function should match the expected level of security provided by the rest of the protocol. For example, collision resistance is usually the property desired when using a hash function for digital signatures. If the signature scheme requires an expected 2^{100} trials before it is broken, the digest produced by the dynamic hash function should be 2^{200}, or $d(s) = 200$.

Using the expected number of messages for each property described in Section 3.8.1, the same logic can be applied whenever a dynamic hash function is used in conjunction with a protocol. The more difficult question to answer is what size a

digest should be when used by itself. For example, if a secure channel exists to send the digest of each packet sent on an insecure channel, the integrity of the packet's data relies only on the hash function used. In this type of a situation how long data integrity will need to be checked will determine the digest's size. Using packet integrity as an example, most packets are short-lived. The data in them is not relevant a few minutes (or seconds) later. For example, the packets that carry the information for a web page are usually not needed after the user has read the web page. Therefore, one should choose a digest size that cannot be broken for the amount of time needed. In the situation of short-lived packets, a 160-bit digest should be adequate.

Beyond security, the final consideration for choosing a security parameter is the amount of computational power required to produce the digest. In most situations the larger the security parameter the more computational power is required to produce the digest. This is because the function that determines the size of the digest is monotonically increasing. Therefore, as larger security parameters are used, larger (or the same size) digests are produced. The simple action of copying the digest into the return buffer will require more time for larger digests. Therefore as a general rule, larger security parameters will require more computational power to compute.

4 CRYPTOGRAPHIC HASH FUNCTION CONSTRUCTIONS

Constructing a hash function that possesses all of the security properties discussed in Chapter 2 is extremely difficult. There are very few guidelines for constructing a good cryptographic hash function. In fact, while the desired properties are well known there have been very few published papers on how to create a cryptographic hash function that possesses these properties. Part of the problem is that testing a function to see if it possesses a security property is extremely difficult. It is easy to prove that a function does not posses a property by constructing an algorithm that breaks the function with respect to the property. However, for a function that seems to be secure how can one know that it actually is? The only method currently available is to publish the function and let cryptographers attempt to break it. This method has been quite successful at proving some functions are not secure [75], and doubt has even been cast on the current standard [7, 13, 74].

To make the problem of designing a secure hash function easier a construction was designed by Ralph Merkle and Ivan Damgård that takes a compression function and extends the domain to binary strings of arbitrary length. This construction, appropriately named the Merkle-Damgård construction, is *property preserving* for preimage resistance and collision resistance. Property preservation of preimage resistance and collision resistance means that if the underlying compression function is preimage resistant and collision resistant, then the hash function created using the Merkle-Damgård construction is also preimage resistant and collision resistant. This reduces the work of creating a secure hash function to creating a secure compression function. While this task is still extremely difficult, it is more manageable than constructing an entire hash function from scratch.

Merkle and Damgård developed the construction independently and presented their work at the 1989 CRYPTO conference [58]. Merkle's paper [49] discussed a *meta*

method which describes the construction and a method for padding an arbitrarily long message to an appropriate length. Damgård's paper [19] discusses the same construction with the same padding method, also providing proofs for the security of the construction.

4.1 The Merkle-Damgård Construction

Most hash functions use the Merkle-Damgård construction. The construction leverages a compression function that takes a fixed size input and produces a fixed size output, where the output is smaller than the input. The input to the compression function is a block of the message to be hashed and the output of the previous compression function. The entire message is processed by iterative calls to the compression function. The output of one call is fed forward as input to the next call. The output of the last call to the compression function is the digest of the message. Figure 4.1 is a diagram of the Merkle-Damgård construction as it was originally described. Definition 4.1.1 defines the Merkle-Damgård construction algorithmically.

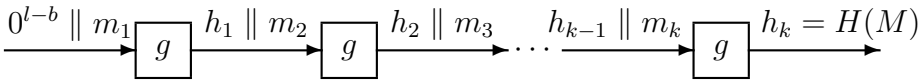

Figure 4.1. The Merkle-Damgård Construction.

Definition 4.1.1 (Merkle-Damgård Construction) *Let β be the length in bits of the input to the compression function g and n be the size of the output in bits. The compression function g has the form: $g : \Sigma^{\beta} \rightarrow \Sigma^{n}$. Let M be a message broken into blocks $m_1, m_2, \ldots m_k$, each of size $\beta - n$ bits, after appending a single 1 bit and enough 0s so that the total message is 64 bits short of a multiple of β. The size of the*

message, as a 64-bit integer, is then appended to the end of the message. The hash function $H : \Sigma^* \rightarrow \Sigma^n$ *for the message* M *is constructed as follows:*

$$\begin{aligned} h_1 &= g(0^n \parallel m_1) \\ h_i &= g(h_{i-1} \parallel m_i) \quad for\ i = 2, 3, \ldots, k \\ H(M) &= h_k \end{aligned}$$

In practice the construction is modified slightly by redefining the compression function to take two inputs instead of one: $\Sigma^n \times \Sigma^b \rightarrow \Sigma^n$. The first input is the previous output or intermediate value and the second input is the message block to be compressed by the function. The compression function is redefined because most hash functions are constructed from a block cipher using the Davies-Meyer or Miyaguchi-Preneel construction [63]. Also, a specified value, instead of 0s is used as the initial value [56, 67, 68]. These modifications do not impact the security of the construction, only aid in creating secure compression functions.

4.1.1 Proofs for Preimage Resistance and Collision Resistance

Two theorems that focus on the security of the Merkle-Damgård construction are presented by Merkle and Damgård [19, 49] and Lai and Massey [42]. Theorem 4.1.1 states a sufficient condition for g in order for H to be collision resistant. Proposition 4.1.1 states necessary and sufficient conditions for g in order to obtain a secure hash function [61].

Theorem 4.1.1 (Merkle-Damgård) *If the compression function* g *used in Definition 4.1.1 is collision resistant and* $\beta - n > 1$*, then the Merkle-Damgård hash function in Definition 4.1.1 is a collision resistant hash function.*

Proof Proof by induction on k, the number of blocks, is used to show that the construction in Definition 4.1.1 is collision resistant if the compression function is collision resistant. First, it is assumed that M and M' are two messages of the same

length and $M \neq M'$.

Base Case: For $k = 1$, $H(M) = g(0^n \parallel m_1)$. If $H(M) = H(M')$ then $g(0^n \parallel m_1) = g(0^n \parallel m'_1)$ which breaks the assumption that g is collision resistant because $0^n \parallel m_1 \neq 0^n \parallel m'_1$.

Induction: Assume the property holds for k and must show that it holds for $k+1$. Because the property holds for k it must be the case that $g(H_{k-1} \parallel m_k) = g(H_{k-1} \parallel m'_k)$. Therefore, $m_1, \ldots, m_k = m'_1, \ldots, m'_k$, and because $M \neq M'$, it must be true that $m_{k+1} \neq m_{k+1}$. The only way to cause a collision is for $g(H_k \parallel m_{k+1}) = g(H'_k \parallel m'_{k+1})$. However, this breaks the assumption that g is collision resistant because $H_t \parallel m_{k+1} \neq H'_k \parallel m'_{k+1}$.

General Case: In the general case the length of the two messages might not be equal. Assume, without loss of generality, that $|M| < |M'|$. There are two cases 1) M is a prefix of M', and 2) M is not a prefix of M'. Both cases are proved in the same manner.

For a collision to occur the last iteration of g on both M and M' must result in the same output. However, there must exist a difference in their inputs at some point because $|M| \neq |M'|$ and the final block of both M and M' contain their respective sizes. Therefore, two different inputs have resulted in the same output from the compression function g which breaks the assumption that g is collision resistant.

This means the only way for $H(M) = H(M')$ is for $M = M'$, assuming g is collision resistant. ■

Provided as a proposition without proof in [42], the following states the relation between the security of the underlying compression function and the overall hash function constructed using the Merkle-Damgård construction.

Proposition 4.1.1 (Lai-Massey) *For an iterated hash function, any successful attack on its compression function implies a successful attack of the same type on the iterated hash function with the same computational complexity.*

The proof for this proposition is somewhat difficult to show because the scope of attacks has not been defined. For example, the common attacks against a hash function are finding a preimage for a given digest and finding a collision; however, there might be a situation where it is advantageous to find a digest that when treated as an integer is a prime number. Certainly if one can find preimages, then one can mount this attack; however, finding a digest that is a prime might be a lot easier than finding a preimage. Therefore, only a proof a sketch is given for most attacks.

Proof Sketch Because the size of the message is not mentioned in the proposition, one can assume that if the size of the message, with padding, is a single block then the proof of the proposition is trivial. In the situation where a message after padding is a single message block the hash function is a single iteration of the compression function and therefore the computation complexity would be the same.

For messages that are larger than a single block the type of a attack is highly dependent; however, the attack can usually be applied to only the last iteration of the compression function. For example, if one can find a collision for the last iteration of the compression function, then a message can be constructed with any prefix, the internal value calculated and a collision found. The same can be said for a preimage or the prime finding attack explained earlier. Essentially the root of the proposition is that the last iteration of the compression function is the output of the construction and therefore attacks on one implies attacks on the other. □

Combining Proposition 4.1.1 with other propositions given in [42], Preneel provided the following theorem, without proof, about the relation between second preimage resistance for the underlying compression function g and the hash function H [61].

Theorem 4.1.2 *Assume that the padding contains the length of the input string, and that the message M (without padding) contains at least 2 blocks. Then finding a second preimage for H with a fixed IV requires 2^n operations if and only if finding a second preimage for g with arbitrarily chosen h_{i-1} requires 2^n operations.*

Proof The proof is by construction in both directions. First, if finding a second preimage for H with a fixed IV requires 2^n operations, then finding a second preimage for g with arbitrarily chosen h_{i-i} requires 2^n operations. Without loss of generality, assume both messages are only 2 blocks long: $M = m_1 \parallel m_2$ and $M' = m'_1 \parallel m'_2$. If M' is a second preimage for H, then it must be the case that either $m_1 \neq m'_1$ or $m_2 \neq m'_2$ or both. In any situation, it is the case that a second preimage is found for the compression function g on the last iteration of the compression function or an arbitrarily chosen h_{i-1}.

Second, if finding a second preimage for g with arbitrarily chosen h_{i-1} requires 2^n operations, then finding a second preimage for H with a fixed IV requires 2^n operations. It is clear that a second preimage for H can be constructed by finding a second preimage for g after the first message block. This would create two messages $M = m_1 \parallel m_2$ and $M' = m_1 \parallel m'_2$ where $m_2 \neq m'_2$. Both messages would hash to the same digest, $H(M) = H(M')$, and yet M' would be a second preimage, given M. ■

4.2 Attacks Against the Merkle-Damgård Construction

While there have been numerous attacks that specifically target the compression functions of dedicated hash functions, only a few attacks have been launched against the Merkle-Damgård construction. This is a testament to the construction's strength and durability through the years. However, recent attacks, such as the multi-collision attack, have raised questions about the continued use of the construction for modern hash functions. Before discussing attacks against the Merkle-Damgård construction, the generic birthday attack is presented because it is used in a number of other attacks.

4.2.1 The Birthday Attack

The birthday attack is a generic attack against any function whose output is smaller than its input. This attack describes the expected number of random messages that must be tested before a preimage or collision is discovered with a probability greater than 50%. The attacks against the Merkle-Damgård construction will often use the birthday attack as a basis for describing the expected number of messages that must be tested before the attack is sucessful. However, if a more efficient attack is known for a particular hash function or compression function, then that attack can usually be leveraged to reduce the expected number of trials. This is a direct result of Proposition 4.1.1.

The birthday attack was first introduced, in relation to hash functions, by Yuval in [77]. The attack specifies the expected number of random messages one must try before finding a preimage for a given digest or a collision between two messages. The following theorems provide the expected values with proofs given in Appendix B.

Theorem 4.2.1 (Birthday Attack) *Assume H is an ideal hash function. Given a digest $H(M)$ of size n, the expected number of messages that need to be tested before finding a preimage is 2^n. The expected number of messages that need to be tested before finding a collision is $2^{n/2}$.*

In [47] Mckinney provides a more generalized version of the birthday attack in which more than two messages collide. This type of collision is called a k-way collision or k-collision. The expected number of messages that must be hashed before finding k messages that collide is given in the following theorem with the proof provided in Appendix B.

Theorem 4.2.2 (k-collisions) *For an ideal hash function H that produces an n-bit digest, the expected number of messages hashed before finding k messages that collide is $2^{n(k-1/k)}$.*

While a k-collision is usually not the goal of an attacker, it is an extremely helpful tool in talking about the strength or weakness of a construction. The attacks in Sections 4.2.3, 4.2.4, and 4.2.5 all use a k-collision to mount some other form of attack.

4.2.2 The Length Extension Attack

One of the simplest and well-known attacks against the Merkle-Damgård construction is the length extension attack. In this attack an adversary is able to extend the length of an unknown message by computing the digest of the concatenation of the unknown message and a suffix [27]. Stated differently, if an attacker is provided $H(M)$ and the length of the message, then the attacker is able to compute $H(M \parallel S)$, for any suffix S, without knowing M [43]. This is possible because the digest $H(M)$ is also the intermediate value for calculating $H(M \parallel S)$.

While theoretically this attack is damaging, there are some hurdles that make the attack less effective in practice. First, the original unknown message M is not hashed directly, but padded to include the message's length. Let $M = X \parallel Y$ where X is the original message and Y is the padding. Therefore to compute $H(M \parallel S)$, the prefix of S must be Y. While all of the information needed to construct such an S is public, constructing a meaningful S of that form is usually difficult. However, if implementations do not check for this sort of attack, then the attack is quite damaging to certain protocols.

4.2.3 The Multi-Collision Attack

The most surprising attack against the Merkle-Damgård construction is Joux's multi-collision attack. This attack shows how to create 2^k collisions for a hash function built using the Merkle-Damgård construction in $k \times 2^{n/2}$ hash computations instead of $2^{n(2^k-1)/2^k}$ as expected [35]. The attack works by finding local collisions for message

blocks, and then using any of the 2^k "paths" through the two messages to create a collision.

Theorem 4.2.3 (Joux Multi-Collision) *If H is a hash function constructed using the Merkle-Damgård construction, then finding 2^k collisions for H requires an expected $k \times 2^{n/2}$ evaluations of H, where n is the size of the output of H.*

Proof The proof is by construction. Let $M = m_1 \parallel m_2 \parallel \cdots \parallel m_k$ and $M' = m'_1 \parallel m'_2 \parallel \cdots \parallel m'_k$ and for all i, $m_i \neq m'_i$. For each message block in M and M' find a local collision using $\mathcal{O}(2^{n/2})$ evaluations of g to find each collision.

$$\begin{aligned} g(IV, m_1) &= g(IV, m'_1) \\ g(g(IV, m_1), m_2) &= g(g(IV, m'_1), m'_2) \\ \vdots &= \vdots \\ g(g(\cdots, m_{k-1}), m_k) &= g(g(\cdots, m'_{k-1}), m'_k) \end{aligned}$$

This results in $\mathcal{O}(k \times 2^{n/2})$ time being spent to find all k local collisions.

For each intermediate value there are two possible message blocks that result in the same intermediate value, m_i and m'_i. Therefore, there are 2^k combinations of messages that all hash to the same digest:

$$\begin{aligned} H(m_1 \parallel m_2 \parallel \cdots \parallel m_k) &= H(m'_1 \parallel m_2 \parallel \cdots \parallel m_k) \\ &= H(m_1 \parallel m'_2 \parallel \cdots \parallel m_k) \\ &= \vdots \\ &= H(m'_1 \parallel m'_2 \parallel \cdots \parallel m_k) \\ &= H(m'_1 \parallel m'_2 \parallel \cdots \parallel m'_k). \end{aligned}$$

■

This attack can also be applied to a cascading hash function where $H(M) = H^1(M) \parallel H^2(M)$ and H^1 and H^2 are independent n-bit hash functions. If both H^1 and H^2 are random oracles, then the expected number of queries to the oracles to

find a collision is 2^n. However, if either H^1 or H^2 is a function built using the Merkle-Damgård construction, then the expected number of computations of the underlying compression function is only $(n/2) \times 2^{n/2}$ [35].

Theorem 4.2.4 *If H is a hash function constructed by concatenating two n-bit independent hash functions $H(M) = H^1(M) \parallel H^2(M)$ and either H^1 or H^2 is built using the Merkle-Damgård construction, then finding a collision for H requires only an expected $(n/2) \times 2^{n/2}$ evaluations of the compression function of H^1 or H^2, whichever is built using the Merkle-Damgård construction.*

Proof Without loss of generality, let H^1 be built using the Merkle-Damgård construction. By application of Theorem 4.2.3, $2^{n/2}$ collisions for H^1 can be found in an expected $(n/2) \times 2^{n/2}$ evaluations of H^1. By application of the Birthday Attack, one of the $2^{n/2}$ collisions for H^1 will also collide with H^2. Such a collision will also collide with H as $H(M) = H^1(M) \parallel H^2(M)$. ■

This theorem states that concatenating two hash functions of the same size does not increase the security as originally thought. Before this was known, a common technique to increase security was to concatenate two functions. For example, MD4 and MD5 were concatenated to increase the security of the overall digest to an expected 2^{128} evaluations. However, as this theorem shows, the actual security is only that of 64×2^{64} expected evaluations.

This attack also has implications on efficiently finding second preimages. Instead of requiring an expected $2^k \times 2^n$ evaluations to find 2^k second preimages, only an expected 2^n evaluations are needed [35]. This works by finding 2^k colliding messages, and then searching for a single preimage to force all 2^k messages to result in the target digest.

Theorem 4.2.5 *If H is a hash function built using the Merkle-Damgård construction, then finding 2^k second preimages for a target digest Y requires only an expected 2^n evaluations of H, where n is the size of the output of H.*

Proof The proof is by construction. By application of Theorem 4.2.3 finding 2^k colliding messages requires an expected $k \times 2^{n/2}$ evaluations of H. Let X be the intermediate value after processing the k^{th} block of any of the 2^k colliding messages. Search for a message block(s) M' that will result in $g(X, M') = Y$. Finding the preimage will take an expected 2^n evaluations of the function H.

Because all of the 2^k messages result in the same intermediate value X, all of the messages will result in the target digest Y when M' is concatenated to the end of them. The overall number of expected evaluations of H is $k \times 2^{n/2} + 2^n = \mathcal{O}(2^n)$ when $k \ll 2^{n/2}$. ■

4.2.4 Herding Attack

The herding attack is an attack on the Merkle-Damgård construction that allows an attacker to force a particular digest when given a prefix to a message by choosing the appropriate suffix [36]. In this attack an attacker chooses a target digest T, then a prefix P for a message is given to the attacker and the attacker must create a suffix S such that $H(P \parallel S) = T$. Using a naïve approach would require finding a preimage for the compression function where the initial value is not fixed, but determined by the prefix P.[1] In the naïve approach the work required would be $\mathcal{O}(2^n)$ to find such a preimage.

Using the herding attack reduces the amount of work required to $\mathcal{O}(2^{n-k-1} + 2^{(n+k)/2+2})$ evaluations of the compression function, where the length of the suffix created is $k + 1$.[2] This is accomplished by building a diamond structure that has $2^{(k+1)} - 1$ intermediate hash values, requiring $\mathcal{O}(2^{(n+k)/2+2})$ evaluations of the compression function. Then a linking message block is found requiring an additional $\mathcal{O}(2^{n-k-1})$ evaluations of the compression function.

[1] It is assumed that H is built using the Merkle-Damgård construction.

[2] This is slightly different than what is published in the paper because this version of the attack considers searching all intermediate nodes and not applying the "expandable message" strategy. This allows for the reduced work of $\mathcal{O}(2^{n-k-1})$ for searching without incurring the additional cost of searching for $\lg(k) + 1$ message blocks.

Theorem 4.2.6 (Herding Attack) *For a hash function H built using the Merkle-Damgård construction, a Herding attack can be launched in $\mathcal{O}(2^{n-k-1} + 2^{(n+k)/2+2})$ evaluations of the compression function, where n is the size in bits of the output of g and $k+1$ is the length of the suffix in message blocks.*

Proof The proof is by construction in two parts. First, a diamond structure is built requiring $\mathcal{O}(2^{(n+k)/2+2})$ evaluations of the compression function. Second, a linking message is found requiring $\mathcal{O}(2^{n-k-1})$ evaluations of the compression function. The overall work is the stated $\mathcal{O}(2^{n-k-1} + 2^{(n+k)/2+2})$.

Building the diamond structure works by generating 2^k intermediate values and then finding 2^{k-1} intermediate values such that pairs of messages are used to collide two of the previous level's intermediate values. The overall structure looks like a binary tree turned on its side. To go from one level to another requires $\mathcal{O}(2^{(n+k+1)/2})$ evaluations of the compression function. This is calculated by trying $2^{(n+k+1)/2}$ messages for each 2^k starting values, resulting in $2^{(n+k+1)/2-k}$ messages per starting value. Between any two starting values it is expected that $(2^{(n+k+1)/2-k})^2 \times 2^{-n} = 2^{n+k+1-2k-n} = 2^{-k+1}$ collisions occur. Therefore, about $2^{-k+k+1} = 2$ other hash values collide with any given starting value [36]. This results in $\mathcal{O}(2^{(n+k)/2+2})$ evaluations of the compression function.

To find the linking value, 2^n messages must be generated to find a message block that will link to a particular starting node. However, if all nodes in the diamond structure are considered instead of only the 2^k starting nodes, then on average 2^{n-k-1} messages must be created. This results in $\mathcal{O}(2^{n-k-1})$ evaluations of the compression function.

Summing the amount of work for both parts results in the overall number of evaluations stated in the theorem: $\mathcal{O}(2^{n-k-1} + 2^{(n+k)/2+2})$. ■

To minimized the overall amount of work, the derivative with respect to k can be taken, set equal to zero and then k solved for.

$$w = 2^{n-k-1} + 2^{(n+k)/2+2} \tag{4.1}$$

$$\frac{\partial}{\partial k}\left(2^{n-k-1} + 2^{(n+k)/2+2}\right) = \left(\frac{2^{n-k}}{2} - 2^{n/2+k/2+1}\right)\lg(1/2) \tag{4.2}$$

Setting Equation 4.2 equal to zero and solving for k results in the value of k that minimizes the work.

$$0 = \left(\frac{2^{n-k}}{2} - 2^{n/2+k/2+1}\right)\lg(1/2) \tag{4.3}$$

$$k = \frac{n-4}{3} \tag{4.4}$$

$$\text{Plugging } k \text{ back in, } w = 2^{n-\frac{n-4}{3}-1} + 2^{(n+\frac{n-4}{3})/2+2} \tag{4.5}$$

$$w = 3 \times 2^{(2n+1)/3} \tag{4.6}$$

Therefore, when $k = (n-4)/3$, the overall work is minimized resulting in $\mathcal{O}(2^{2n/3})$ evaluations of the compression function or $2^{n/3}$ less work than the naïve approach.

4.2.5 Long Message Second Preimage Attack

The expected number of messages that must be hashed to find a second preimage for a given digest is equal to the number of messages that must be hashed to find a preimage or $\mathcal{O}(2^n)$ where the digest is n bits in length. Kelsey and Schneier demonstrate in [37] that only $\mathcal{O}(k \times 2^{n/2+1} + 2^{n-k+1})$ messages, of k blocks in length, must be tried before finding a second preimage. While these messages are too long[3] for practical use, this challenges the security of the Merkle-Damgård construction.

The attack works by constructing *expandable messages* which are messages of varying length that all result in the same intermediate value before applying any padding. Using an expandable message, a second preimage for a target message k-blocks long is created in much less work than $\mathcal{O}(2^n)$. Before stating the theorem, the following lemma is provided to aid in proving the theorem.

[3] An example message that is $2^{64} - 1$ bits in length is given for SHA-1

Lemma 4.2.1 *Creating messages of* $[k, k+2^k-1]$*-blocks in length which all result in the same final intermediate value (before padding) requires* $\mathcal{O}(k \times 2^{(n/2)+1})$ *evaluations of the underlying compression function* g*.*

Proof The proof is by construction. If a pair of messages of 1 block and α blocks can be constructed that result in the same final intermediate value in time $\mathcal{O}(\alpha - 1 + 2^{(n/2)+1})$, then with k calls to this function the overall number of evaluations of the compression function required is the stated $\mathcal{O}(k \times 2^{(n/2)+1})$.

To create a pair of messages of size 1 block and α blocks that result in the same intermediate value, the following algorithm is used, taken from [37].

FindCollision(α, h_{in}) – Finds 2 messages of 1 block and α blocks in length that collide, both starting with h_{in}

1. Compute $\alpha - 1$ dummy message blocks resulting in h_{tmp} as the intermediate value using q, a fixed random message, as the message block.
   (a) $h_{tmp} = h_{in}$
   (b) $h_{tmp} = F(h_{tmp}, q)$ for $i = 0 \ldots \alpha - 2$
2. Build lists A and B as follows where m_i is a distinct random message block:
   (a) $A[i] = g(IV, m_i)$ for $i = 0 \ldots 2^{n/2} - 1$
   (b) $B[i] = g(h_{tmp}, m_i)$ for $i = 0 \ldots 2^{n/2} - 1$
3. Find i, j such that $A[i] = B[j]$
4. Return the colliding messages (m_i and $q \parallel q \parallel \ldots \parallel m_j$) and the intermediate value $g(h_{in}, m_i)$

The overall work for this algorithm is $\mathcal{O}(\alpha - 1 + 2^{(n/2)+1})$ evaluations of the compression function g. By calling FindCollision k times as follows the result is a table of messages which all have the same final intermediate value.

$$(C[k-i-1][0], C[k-i-1][1], h_{tmp}) = \text{FindCollision}(2^i + 1, h_{tmp})$$

where $i = 0 \ldots k-1$, and $h_{tmp} = h_{in}$ for the first iteration. The total number of evaluations of the compression function for this algorithm is $\mathcal{O}(k \times 2^{(n/2)+1})$. ■

Theorem 4.2.7 (Long Message Second Preimage Attack) *Given a target message k-blocks long and the digest of the message, a second preimage can be constructed in $\mathcal{O}(k \times 2^{(n/2)+1} + 2^{n-k+1})$ evaluations of the underlying compression function if the hash function is built using the Merkle-Damgård construction.*

Proof The proof is by construction. First, expandable messages of lengths $[k, k + 2^k - 1]$ are created using Lemma 4.2.1 requiring $\mathcal{O}(k \times 2^{(n/2)+1})$ evaluations of the compression function.

Next a message block is found that links the expandable message to one of the intermediate values for the target message after the k^{th} block. Using the the message blocks after the one that collides with the linking block, a second message is constructed using the appropriately sized expandable message. Because the length of these two messages up to this point is the same, the padding will have no affect on the resulting digest. The result is a second preimage for the target message. The time required to find the linking message block is $\mathcal{O}(2^{n-k+1})$ evaluations of the underlying compression function. Therefore, the overall number of evaluations of the compression function g is $\mathcal{O}(k \times 2^{(n/2)+1} + 2^{n-k+1})$. ■

4.2.6 Fixed Point Attack

While all of the attacks discussed thus far treat the compression function as a black box, the fixed point attack works on compression functions built using a block cipher according to the Davies-Meyer [64] principle. The compression function is defined using a block cipher E, where the notation $E_k(M)$ is used for the message M being encrypted using the key k. The symbol $\oplus$ is used to denote any group operation over Σ^n:

$$g(h_{i-1}, m_i) = E_{m_i}(h_{i-1}) \oplus h_{i-1}.$$

To find a fixed point for a given message block m_i, the identity element of the group is decrypted resulting in a fixed point for the message block. In the case of addition, the identity element is a string of zero bits:

$$E_{m_i}^{-1}(0 \ldots 0) = h_{i-1}$$

where E^{-1} denotes decryption. Using this fixed point allows one to improve the attack in Theorem 4.2.7. Creating an extremely long message that hashes to the same digest as shorter versions of the message.

4.3 New Constructions

With attacks against the Merkle-Damgård construction discovered, a number of new constructions have been proposed to thwart these attacks. Most of the new constructions still work in an iterative manner, but modify the input or output in some way to prevent the above attacks from being successful. Almost all of the constructions presented in this section assume an underlying compression function g of the form: $\Sigma^n \times \Sigma^b \rightarrow \Sigma^n$ where n is the size of the digest in bits and b is size of a message block in bits. Any other functions used in the constructions will be explicitly defined. The new constructions are presented in the following sections in no particular order.

4.3.1 The Wide-Pipe Hash and Double-Pipe Hash

In [43, 44] Lucks introduces the notion of a wide-pipe hash function where the intermediate value of the compression function is w bits long while the output remains n bits long, where $w > n$. To have the overall hash function result in n bits, a second

compression function is used to compress w bits down to n bits, $c : \Sigma^w \rightarrow \Sigma^n$. The wide-pipe construction is then defined as follows:

$$\begin{aligned} h_1 &= g(IV, m_1) \\ h_i &= g(h_{i-1}, m_i) \quad for\ i = 2, 3, \ldots, k \\ H(M) &= c(h_k) \end{aligned}$$

The design of this construction is motivated by the Joux Multi-Collision attack. Instead of performing $2^{n/2}$ work to find an internal collision, $2^{w/2}$ work is required. This requires more work to mount the attack because $w > n$.

Theorem 4.3.1 *If g and c are modeled as random oracles, then finding 2^k-collisions for the wide-pipe construction takes an expected $\min\{k \times 2^{w/2}, 2^{n(2^k-1)/2^k}\}$ queries to the oracles.*

Proof There are two cases to consider, when 2^k collisions are found for function g, and when 2^k collisions are found for function c. The case for 2^k collisions for g will require a running time of $k \times 2^{w/2}$. Constructing the 2^k collisions for c requires a running time of $2^{n(2^k-1)/2^k}$.

CASE 1: 2^k collisions for g

The compression function g produces a w-bit output; therefore, by application of Theorem 4.2.3 it requires an expected $k \times 2^{w/2}$ messages to find 2^k collisions for g.

CASE 2: 2^k collisions for c

The compression function c produces an n-bit output, so by application of Theorem 4.2.2 it requires an expected $2^{n(2^k-1)/2^k}$ messages to find 2^k collisions.

Therefore the total amount of work required to find 2^k collisions for the wide-pipe construction takes time $\Omega(\min\{k \times 2^{w/2}, 2^{n(2^k-1)/2^k}\})$. ■

To ensure that the construction is asymptotically as secure against the multi-collision attack as an ideal hash function, $w \geq 2n$ [43]. This requires a new compression function of the form $\Sigma^w \times \Sigma^b \rightarrow \Sigma^w$. Instead of creating a new dedicated compression function, Lucks provides a new construction, the double-pipe construction which simulates a wider compression function. The construction simultaneously computes two lines or Merkle-Damgård iterations. The output from one line is mixed with the input of the other to simulate a larger compression function.

Let $h_{x,y}$ denote the output of the x^{th} iteration of the compression function in line y. The double-pipe construction is defined as follows.

$$\begin{aligned}
h_{1,1} &= g(IV_1, IV_2 \parallel m_1) \\
h_{1,2} &= g(IV_2, IV_1 \parallel m_1) \\
h_{i,1} &= g(h_{i-1,1}, h_{i-1,2} \parallel m_i) \quad for\ i = 2, 3, \ldots, k \\
h_{i,2} &= g(h_{i-1,2}, h_{i-1,1} \parallel m_i) \quad for\ i = 2, 3, \ldots, k \\
H(M) &= g(IV_3, h_{k,1} \parallel h_{k,2})
\end{aligned}$$

The following theorem, taken from [44], states the time required to find cross collisions and strict collisions for the double-pipe hash. A cross collision is a collision such that $h_{i,1} \neq h_{i,2}$ but $g(h_{i-1,1}, h_{i-1,2} \parallel m_i) = g(h_{i-1,2}, h_{i-1,1} \parallel m_i)$. A strict collision is a collision such that $h_{i,1} \neq h_{i,2}$, $h_{j,1} \neq h_{j,2}$, $m_i \neq m_j$, and $i \neq j$ but $g(h_{i-1,1}, h_{i-1,2} \parallel m_i) = g(h_{j-1,1}, h_{j-1,2} \parallel m_j)$ and $g(h_{i-1,2}, h_{i-1,1} \parallel m_i) = g(h_{j-1,2}, h_{j-1,1} \parallel m_j)$.

Theorem 4.3.2 *If g is modeled as a random oracle, then finding cross collisions for g requires time $\Omega(2^n)$, and finding strict collisions for g requires time $\Omega(2^n)$.*

Proof The proof is done in two parts, one for cross collisions and the other for strict collisions, taken from [44].

Cross Collisions: Any triple $(h_{i,1}, h_{i,2}, m_i)$ can only be part of a cross collision if $h_{i,1} \neq h_{i,2}$ and $g(h_{i-1,1}, h_{i-1,2} \parallel m_i) = g(h_{i-1,2}, h_{i-1,1} \parallel m_i)$, i.e., with a probability of

2^{-n}. Thus, $\Omega(2^n)$ oracle queries are expected to find a cross collision.

Strict Collisions: For any triple $(h_{i-1,1}, h_{i-1,2}, m_{i-1})$ with $h_{i-1,1} \neq h_{i-1,2}$, the pair $(h_{i,1}, h_{i,2}) \in \Sigma^{2n}$ is a uniformly distributed $2n$-bit random value, chosen independently from all other $g(\cdot, \cdot \parallel \cdot)$-values. If the adversary chooses q different tuples and makes q queries to the oracle, then the probability of success is $\Sigma_{0 \leq j < q} j/2^{2n} = \Omega(q^2/2^{2n})$. Thus, it is expected that $q = \Omega(2^n)$ oracle queries are made to find a strict collision. ■

Because the only way to mount the multi-collision attack against the double-pipe construction is through a combination of strict collisions and double collisions[4], the following corollary can be stated.

Corollary 4.3.1 *The double-pipe construction requires $\Omega(2^{n(k-1)/k})$ evaluations of the compression function g to create a k-collision.*

4.3.2 Prefix-Free Merkle-Damgård

In [17] it is shown that if the underlying compression function acts as a random oracle, then any hash function built using the Merkle-Damgård construction does not act as a random oracle. To fix this problem a number of solutions were presented including two methods for prefix-free encoding messages before the message is processed by the hash function. The definition of a prefix-free code follows, taken from [17].

Definition 4.3.1 (Prefix-Free Code) *A prefix-free code[5] over Σ^b is an efficiently computable injective function $f : \Sigma^* \rightarrow \left(\Sigma^b\right)^*$ with the following conditions:*

1. *$\forall x, \forall y, x \neq y, f(x)$ is not a prefix of $f(y)$*
2. *Given only $f(x)$ it should be easy to recover x*

[4]The final compression function is not considered as it would require $\Omega(2^{n(k-1)/k})$ queries to find a k-collision.

[5]Many authors call this a Prefix Code.

The two prefix-free codes defined in the paper, $f_1(M)$ and $f_2(M)$, are applied to the message M before it is fed into the compression function. Because both of the codes work on one message block at a time, the prefix-free encoding can be done in parallel with the application of the compression function. The two methods for prefix-free encoding defined are as follows for the message $M = m_1 \parallel m_2 \parallel \ldots \parallel m_k$ where m_k has the padding for the Merkle-Damgård construction.

$$\begin{aligned} f_1(M) &= |M| \parallel m_1 \parallel m_2 \ldots \parallel m_k, \text{ where } |m_i| = b \\ f_2(M) &= 0 \parallel m_1 \parallel 0 \parallel m_2 \parallel \ldots 0 \parallel m_{k-1} \parallel 1 \parallel m_k, \text{ where } |m_i| = b - 1 \end{aligned}$$

One should note that the first method $f_1(M)$ has an important drawback that makes it infeasible for use in real-world applications. In $f_1(M)$ the entire length of the message must be known before processing begins. In applications where a digest must be computed for streaming data this type of an encoding will not work. The second method does not suffer from this same drawback. $f_2(M)$ can be run in parallel with the compression function as data is streamed.

Before stating the theorem that these prefix-free encodings result in a construction that is indifferentiable from a random oracle in the random oracle model, the definition of indifferentiable used in [17] is provided. The definition is based on [45] which is an extension of the ideas from [57] and [11]. The basic idea is that a simulator should exist such that no distinguisher can tell whether it is interacting with a simulator with access to a random oracle or a Turing machine with access to a random oracle. Essentially, the simulator should simulate the Turing machine.

Definition 4.3.2 (Indifferentiablity) *A Turing machine C with access to a random oracle $\mathcal{G}$ is (t_D, t_S, q, ϵ)-indifferentiable from a random oracle $\mathcal{F}$ if there exists a simulator S, such that for any distinguisher D one has:*

$$\left| Pr\left[D^{C,\mathcal{G}} = 1\right] - Pr\left[D^{\mathcal{F},S} = 1\right] \right| < \varepsilon$$

The simulator has oracle access to $\mathcal{F}$ and runs in time at most t_S. The distinguisher runs in time at most t_D and makes at most q queries. Similarly, $C^{\mathcal{G}}$ is computationally indifferentiable from $\mathcal{F}$ if ε is negligible.

When a distinguisher is successful it returns a 1 Definition 4.3.2 states that the probability that the distinguisher is able to distinguish the algorithm C with access to $\mathcal{G}$ from the simulator S with access to $\mathcal{F}$ should be negligible. For the prefix-free construction, the algorithm C is the Merkle-Damgård construction. The random oracle $\mathcal{G}$ represents the underlying compression function, with the prefix-free encoding applied first. The random oracle $\mathcal{F}$ represents the construction C is trying to emulate. From this the theorem for prefix-free encoding can be stated.

Theorem 4.3.3 *The Merkle-Damgård construction with a prefix-free encoding applied to a message first is $(t_D, t_S, q, \varepsilon)$-indifferentiable from a random oracle, in the random oracle model for the compression function, for any t_D, with $t_S = \ell \cdot \mathcal{O}(q^2)$, where ℓ is the maximum length of a query made by the distinguisher D.*

The proof of this theorem is given in [17].

4.3.3 Enveloped Merkle-Damgård

In [3] Bellare and Ristenpart presented a multi-property-preserving construction called the Enveloped Merkle-Damgård construction. The goal of their construction is to preserve the properties of collision resistance, pseudorandom function, and pseudo-random oracle if the underlying compression function possesses these properties. They prove that all the constructions presented in [17] do not preserve collision resistance when the underlying compression function is collision resistant. Their construction however is provably multi-property-preserving. Their construction achieves multi-property-preservation by taking the output from the second to last iteration of the compression function and concatenating it with the final message block and supplying it, along with a second initial value to the compression function. Appropriate padding of the last message block ensures that it is the proper length.

$$
\begin{aligned}
h_1 &= g(IV_1, m_1) \\
h_i &= g(h_{i-1}, m_i) \quad for\ i = 2, 3, \ldots, k-1 \\
h_k &= g(IV_2, h_{k-1} \parallel m_k) \\
H(M) &= h_k
\end{aligned}
$$

This transform utilizes two mechanisms to ensure that the required properties are preserved. The first mechanism is that of appending the length of the message to the final message block. This ensures that collision resistance is preserved, and is the major flaw in the constructions presented in [17]. As stated in [3] if the standard Merkle-Damgård strengthening technique of appending the message's length to the end of the message was applied to the constructions in [17], then they too would be multi-property-preserving.

The second mechanism is the use of the final application of the compression function *enveloping* the internal value of the previous applications of the compression function. This technique is similar to that of Maurer and Sjödin in [46]. Their technique is different from other enveloping techniques in that bits from the message and a second distinct initial value are used for the final application of the compression function. This technique provides domain separation from the previous applications of the compression function and the final application with high probability. It also is more efficient than perpending zeros to the message or some other sort of prefix free encoding scheme.

4.3.4 The Hash Iterative Framework

The Hash Iterative Framework (HAIFA) is a set of modifications to the Merkle-Damgård construction to provide resistance against a number of the attacks described in Section 4.2. HAIFA, designed by Biham and Dunkelman [8], also provides the ability to perform randomized hashing and securely truncate the underlying compression

function. HAIFA provides these properties by including a salt, the number of bits hashed so far, and a special IV for each digest size in the construction. The form of the underlying compression function is changed to accommodate these new parameters to the following,

$$f : \Sigma^n \times \Sigma^b \times \Sigma^t \times \Sigma^s \rightarrow \Sigma^n.$$

The intermediate values and message block sizes remain n and b bits long respectively. The size in bits of the number of message bits processed so far is t. The size of the salt in bits is s.

While the form of f is not the same as a traditional compression function, a traditional compression function can be modified to act as the type of function required for use in HAIFA. The first parameter of both functions is a value from the same size set Σ^n. The last three parameters of the HAIFA compression function can be concatenated together and supplied as the message block to a traditional compression function: $f(w, x, y, z) = g(w, x \parallel y \parallel z)$. While this reduces the efficiency of the overall function, it requires little modification to existing hash functions to use HAIFA.

To incorporate the size of the digest to be produced, a special initial value is created for each digest size. Let IV_d represent the initial value for a digest of size $d \leq n$. The initial value IV_d is created by applying the compression function to the desired size of the digest, a fixed initial value, zeros for the "bits processed", and a salt value. For example, if $d = 256$, then $IV_{256} = g(IV, 256, 0, 0)$.[6] One should note that digests can be precomputed to increase efficiency.

The overall form of a hash function created using HAIFA is,

$$H(M, d, S) : \Sigma^* \times \Sigma^n \times \Sigma^s \rightarrow \Sigma^d.$$

The function can be created as follows:

[6]This is a slight change from [8] as indicated by the author via e-mail [25].

$$
\begin{aligned}
IV_d &= f(IV, d, 0, salt) \\
h_1 &= f(IV_d, m_1, 0, salt) \\
h_i &= f(h_{i-1}, m_i, i \times b, salt) \quad for\ i = 2, 3, \ldots, k \\
H(M, d, S) &= h_k
\end{aligned}
$$

HAIFA also includes a modification to the padding algorithm used in the Merkle-Damgård construction. Instead of concatenating the size of the message to the end of the message, after padding with a single 1-bit and enough 0 bits, the size of the digest is also concatenated. All messages processed by HAIFA have the form $M \parallel 1 \parallel 0 \cdots 0 \parallel |M| \parallel d$.

Because HAIFA can be constructed using a traditional compression function and the overall iterative structure has not changed, the security proofs for the Merkle-Damgård construction described in Section 4.1.1 also apply to HAIFA [8]. However, because HAIFA is an iterative construction many of the same attacks that work on the Merkle-Damgård construction can also be applied to this construction. The multi-collision attack is successful against HAIFA, except that precomputation cannot be carried out until a salt has been chosen. This is an important point as it requires the Herding attack be carried out on-line [8]. As for the fixed point attack, this attack can no longer be carried out because the number of bits hashed so far is included in each iteration. Unfortunately, this same property does not prevent the length extension attack. Only when bits from the output of the compression function on the final iteration are truncated or obfuscated is the length extension attack prevented. Techniques for obfuscating the output of the final compression function are discussed in a number of places including [17] [43] and [27].

4.3.5 Randomized Hashing: RMX

The idea of randomized hashing or the RMX transform is proposed in two papers, [32] and [31]. The first, [32], explains the idea of randomized hashing which introduces randomization into messages before they are hashed to free digital signature schemes from relying on strong collision resistance. This provides a safety net in case the hash function used in a digital signature scheme is discovered to be less collision resistant than initially thought. The second paper, [31], talks about the implementation details of the RMX transform in the OpenSSL library.

The RMX transform works by generating a random binary string that is equal in length to a message block and then hashing that string first instead of the first message block. The random binary string is then XORed to each message block before the message block is processed by the underlying compression function. Finally, a truncated version of the random binary string is XORed with the final message block; however, not with the padding of the message.

Let r_1 be a random binary string of length b and r_2 be r_1 truncated to the size of the last message block before padding, then padded with zeros to the size of a message block. For example, if a message block is 4 bits long, and the last message block is 2 bits long; $r1 = 0110$ and then $r_2 = 0100$. This way the traditional padding used in the Merkle-Damgård construction is not XORed with the random string. The RMX scheme is applied to a message $M = m_1 \parallel m_2 \parallel \ldots \parallel m_k$, where m_k contains padding, as follows:

$$\begin{aligned} h_1 &= g(IV, r_1) \\ h_i &= g(h_{i-1}, r_1 \oplus m_i) \quad for\ i = 2, 3, \ldots, k-1 \\ h_k &= g(h_{k-1}, r_2 \oplus m_k) \\ H(M) &= h_{k+1} \end{aligned}$$

The intended application of the RMX transform is to ensure a secure digital signature even if an off-line collision can be generated for the hash function. To

enable a verifier to check the digest of a message the random value r_1 must be sent with the message so that the digest can be reconstructed.

The difference between this scheme and others that simply use a salt value, is that the RMX transform has Enhanced Target Collision Resistance. Enhanced Target Collision Resistance is defined as follows where $H_r(M)$ is the digest of M using the salt r.

Definition 4.3.3 (Enhanced Target Collision Resistance) *If an attacker is not able to win the following game with a non-negligible probability, then the function H is enhanced target collision resistant.*

1. *The attacker chooses a message M*
2. *The attacker receives a salt r*
3. *The attacker must find a second message and salt $(r, M) \neq (r', M')$ such that $H_r(M) = H'_r(M)$*

A formal proof for the RMX transform being enhanced target collision resistant can be found in [32].

4.3.6 3C and 3C-X

The 3C and 3C-X constructions were proposed by Gauravarm *et al.* in [29]. The 3C construction is a modification of the Merkle-Damgård construction which works as a pseudorandom function, message authentication code, and cryptographic hash function. This section will only focus on the 3C construction as it pertains to cryptographic hash functions.

The Merkle-Damgård construction is modified by the addition of another line of intermediate values calculated by applying a function to the output of the compression function. The name of the construction is derived from the number of applications of the compression function needed for the construction to work. The three applications

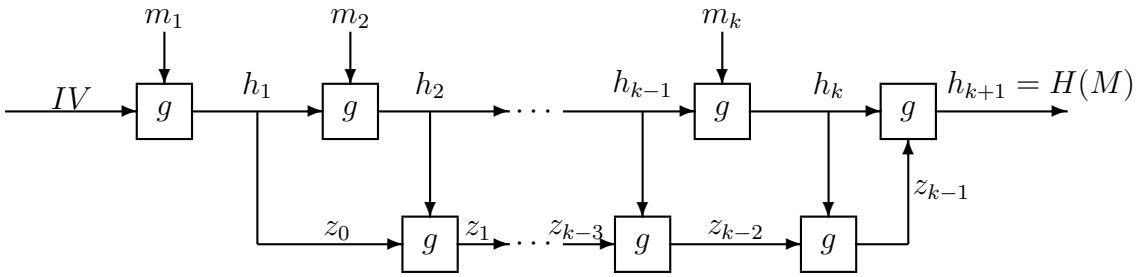

Figure 4.2. The 3C construction without padding for z_i.

come from compressing the first message block, the padded message block which encodes the message's size, and a final application of the compression function to thwart attacks like the length extension attack. Because the construction works on two lines, the traditional intermediate values are represented by h_i and the second line's intermediate values are represented by z_i. In the final step the two lines are combined resulting in a single value. The 3C construction is shown in Figure 4.2. For a message $M = m_1 \parallel m_2 \parallel \ldots \parallel m_k$ where $|m_i| = b$ and $PAD(x)$ pads the argument x with zeros to the appropriate bit length, the construction is defined as follows.

$$\begin{aligned} h_1 &= g(IV, m_1) \\ z_0 &= h_1 \\ h_i &= g(h_{i-1}, m_i) \quad for\ i = 2 \ldots k \\ z_{i-1} &= g(z_{i-2}, PAD(h_i)) \quad for\ i = 2 \ldots k \\ h_{k+1} &= g(h_k, PAD(z_{k-1})) \\ H(M) &= h_{k+1} \end{aligned}$$

The difference between the 3C construction and the 3C-X construction is the reuse of the compression function g. In the 3C-X construction the compression function

g is replaced by the XOR operation. This removes the need for padding each z_i, and reduces the overall computational effort needed to compute a digest. The XOR however does not replace the last iteration of the compression function so padding is always need for z_{k-1}. This modification, the authors claim, is the smallest modification which one can make to the Merkle-Damgård construction to increase its security without imposing a penalty on performance. It is also shown in the paper that this construction is collision resistant if the underlying compression function is collision resistant. The proof works in much the same way as the Merkle-Damgård construction's proof except that both lines must be considered.

Theorem 4.3.4 *If the compression function g used in the 3C (or 3C-X) construction is collision resistant, then the 3C (or 3C-X) construction results in a collision resistant hash function.*

Proof A sketch of the proof is provided for the last application of the compression function for both lines in the constructions. The same inductive approach used to prove this property for the Merkle-Damgård construction can be applied here to complete the proof.

Without loss of generality, assume that two messages M and M' are of equal length and their message blocks are identical except for the final block $m_k \neq m'_k$. For a collision to occur, $h_{k+1} = h'_{k+1}$, there are three cases.

CASE 1: Let $h_k = h'_k$ and $z_{k-1} \neq z'_{k-1}$. This case can never occur because the inputs to the compression function g are all the same, because the message blocks up to m_k and m'_k are all the same. The function g is deterministic; therefore, this is an invalid case.

CASE 2: Let $h_k \neq h'_k$ and $z_{k-1} = z'_{k-1}$. By definition of the construction, $g(h_k, PAD(z_{k-1})) = g(h'_k, PAD(z_{k-1'}))$. Because $h_k \neq h'_k$, a collision for g was discovered. This breaks the assumption that g is collision resistant.

CASE 3: Let $h_k \neq h'_k$ and $z_{k-1} \neq z'_{k-1}$. By definition of the construction, $g(h_k, PAD(z_{k-1})) = g(h'_k, PAD(z_{k-1'}))$. Because both $h_k \neq h'_k$ and $z_{k-1} \neq z'_{k-1}$, a collision for g was discovered. This breaks the assumption that g is collision resistant.

Therefore, the only way for $h_{k+1} = h'_{k+1}$ is for there to be a collision earlier in one or both of the lines of the construction. By induction the entire construction can be proved collision resistant.

For the case of the 3C-X construction all of the same cases apply except for Case 2. Because g is replaced by XOR in this construction with $z_{k-2} = z'_{k-2}$ and $h_k \neq h'_k$, there is no way for $z_{k-1} = z'_{k-1}$ by definition of the XOR operator. ■

It should be noted that even with the addition of a second line of intermediate values the complexity of finding multi-collisions is not reduced. The reason is that the input to the second line (the one producing z_is) comes from the h_is. Stated differently, the message does not directly impact how you compute a z_i. If a collision is found for an $h_i = h'_i$ for some $m_i \neq m_i$, then the inputs used to calculate z_{i-1} and z'_{i-1} rely only on h_i and h'_i respectively, assuming all message blocks are the same up to this point. Therefore, $z_{i-1} = z'_{i-1}$ when $h_i = h'_i$ for all i, and the computational cost to find the collision is still $\mathcal{O}(2^{n/2})$, when the output of g is n bits long. The same is true for the 3C-X construction. It is unclear what additional security is provided by either constructions.

5 A DYNAMIC HASH FUNCTION CONSTRUCTION

This chapter presents a construction that builds a dynamic cryptographic hash function from a traditional compression function. The construction is similar to the Merkle-Damgård construction in that a compression function is iteratively called over blocks of a message. The dynamic hash function construction forms a dynamic cryptographic hash function by incorporating the security parameter into the initial value, the padding of the message, and the size of the digest. Some of the ideas used to create the dynamic hash function construction are borrowed from [8], [3], and [43].

In [8] Dunkelman and Biham present HAIFA, or a Hash Iterative Framework. As noted in Section 4.3.4, their framework describes a method for creating a hash function that can produce digests of sizes less than or equal to the output size of the underlying compression function. Unlike HAIFA, the dynamic construction only produces a digest that is larger than or the same size as the output of the underlying compression function. The dynamic hash function construction uses some of the same techniques as HAIFA to securely create digests of various sizes.

In [3] Bellare and Ristenpart describe a multi-property preserving hash function construction. As noted in Section 4.3.3, the Enveloped Merkle-Damgård construction describes a modification to the Merkle-Damgård construction that provides provable property preservation with respect to collision resistance, pseudorandom oracles, and pseudorandom functions. The EMD construction accomplishes property preservation by feeding the output of the last iteration as a message into the compression function using a second initial value. The same enveloping technique is used in the dynamic hash function construction.

In [43] Lucks provides a construction that is resistant to the multi-collision attack described by Joux (see Section 4.3.1). Lucks's idea is to create a construction with two Merkle-Damgård lines. The output of one compression function from one line

is concatenated with a message block as input to the next iteration on the other line. This construction simulates a larger internal compression function such that the internal values are twice as large as the output of the traditional compression function. This same technique is used by the dynamic hash function construction to create digests that are larger than the underlying compression function.

5.1 Construction Description

There are four parts to the dynamic construction: initial value creation, message processing, message padding, and digest creation. Different initial values are created for each security parameter and for each line in the construction. Different initial values are used to ensure that the dynamic properties discussed in Chapter 3 are possessed by the resulting hash function. The message is processed in an iterative fashion by breaking the message up into blocks and processing blocks in a sequential manner. The message is padded to a multiple of the block size and strengthened by including both the message's size and the security parameter. Finally, the digest is constructed by enveloping the internal value of each line and concatenating together the lines to create a digest of the appropriate length.

It is assumed that a traditional compression function[1] of the following form is used:

$$g : \Sigma^n \times \Sigma^b \rightarrow \Sigma^n \qquad \text{where } n < b \leq 4n.$$

The dynamic hash function construction creates digests whose bit length is equal to the security parameter, so $d(s) = s$. To accomplish this, multiple lines are used to create multiple digests that are concatenated together after the message has been processed. Figure 5.1 shows the dynamic hash function construction when $(\ell - 1)n < d(s) \leq \ell n$. By concatenating multiple lines together to form the overall digest, a large compression function is simulated. The extent at which this technique can be exploited is discussed in Section 5.2.

[1]The compression functions of MD5 and SHA-1 satisfy this inequality.

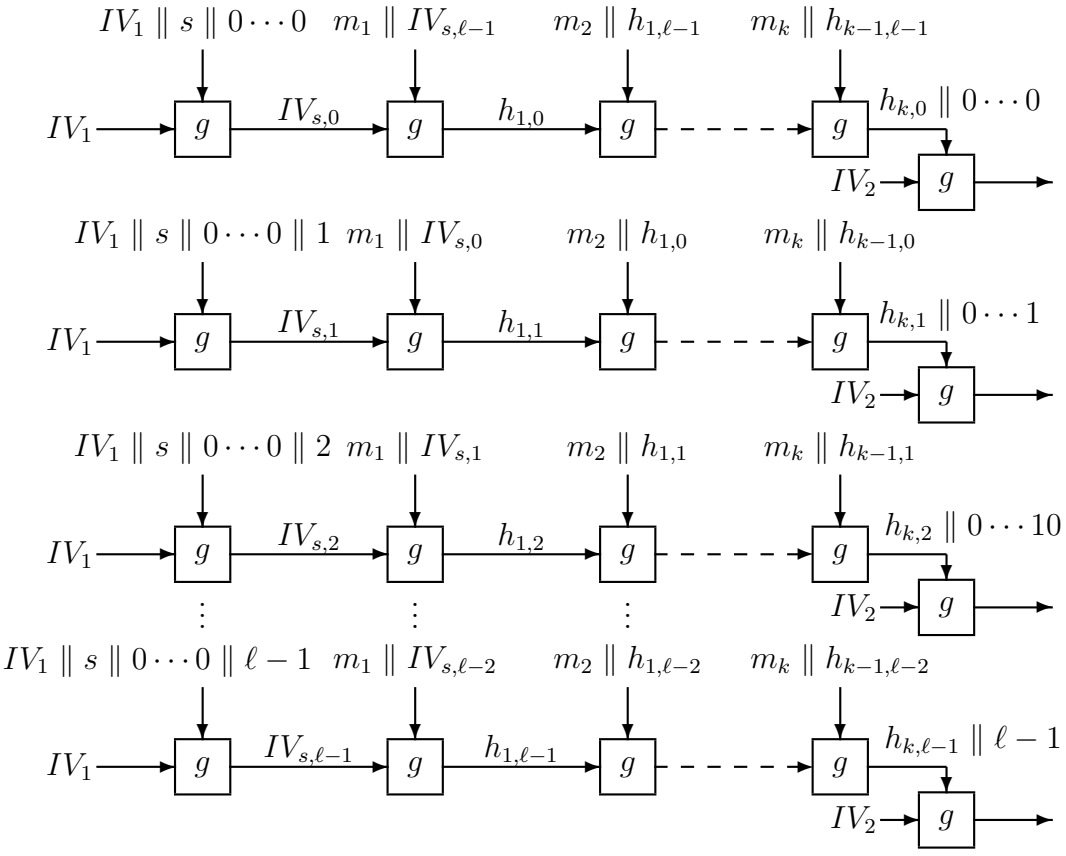

Figure 5.1. The dynamic hash function construction.

5.1.1 Initial Value Creation

The initial value for each line depends on the security parameter used to create the digest and the line the initial value is used on. This ensures that from the beginning the intermediate values of the construction are different on each line, for each security parameter.

The initial value, for security parameter s and line j, is:

$$IV_{s,j} = g(IV_1, IV_1 \parallel s \parallel 0 \cdots 0 \parallel j).$$

It should be noted that this method is a modified version of the one used in [8] to help ensure that two digests of different sizes, computed from the same message, will not result in one being the truncation of the other. This same technique is also used by SHA-224 and SHA-384 so that the digest of a message will not be the truncation of SHA-256 and SHA-512 respectively [56].

Because the initial value is created without processing any bits of the message, initial values can be precomputed. To increase the speed of the dynamic hash function construction, common initial values can be precomputed and stored in a table. When a message is to be hashed with a specific security parameter already in the table the number of calls to the underlying compression function is reduced. For short messages this can greatly increase the relative speed of the function.

5.1.2 Message Processing

A message is processed in much the same way as the Merkle-Damgård construction. The message is broken up into message blocks, $b-n$ bits in length. The message blocks are processed sequentially by the compression function where the output of the last iteration is fed forward into the next. Instead of processing message blocks alone, as is done in the Merkle-Damgård construction, the dynamic hash function construction concatenates the output of one of the other lines to each message block.

By concatenating the output of another line with the message block a larger compression function is simulated. This technique is the same used in [43] to create a larger internal value, preventing the multi-collision attack. The technique is applied to the dynamic hash function construction to ensure that as the security parameter increases, the security of the digest increases too. If an additional line were not included for each multiple of the compression function's output the security parameter increased, then a simple attack against the construction could be launched to cause a collision in less work than expected. A simple attack on a single line construction is as follows.

If only a single line were used in the dynamic hash function construction, then a collision for a digest of size $2n$ could be constructed with only $\mathcal{O}(2^{n/2})$ evaluations of the compression function instead of the expected $\mathcal{O}(2^n)$. The attack works by finding a collision for an internal value which requires only $\mathcal{O}(2^{n/2})$ evaluations of the compression function. Two messages that are exactly the same, except for the colliding message blocks, can be constructed such that $H(M, s) = H(M', s)$ and $|H(M, s)| = 2n$. To accomplish the increase in security, $\lceil s/n \rceil \times n$ lines are needed.

Let $\ell = \lceil s/n \rceil \times n$, or the number of lines in the construction. Let $h_{x,y}$ be the internal value for iteration x, line y. The overall form of the construction is as follows, excluding digest creation.

$$\begin{aligned} IV_{s,j} &= g(IV_1, IV_1 \parallel s \parallel j) \qquad \text{for } j = 0 \ldots \ell - 1 \\ h_{1,j} &= g(IV_{s,j}, m_1 \parallel IV_{s,j-1 \mod \ell}) \qquad \text{for } j = 0 \ldots \ell - 1 \\ h_{i,j} &= g(h_{i-1,j}, m_i \parallel h_{i-1,j-1 \mod \ell}) \qquad \text{for } i = 2 \ldots k, \text{ for } j = 0 \ldots \ell - 1 \end{aligned}$$

5.1.3 Message Padding

The dynamic construction pads a message to a multiple of $b - n$, minus 96 bits. The padding is done by concatenating a single 1 bit to the end of the message followed by enough 0 bits to allow for concatenating the message's size as a 64-bit number and

the security parameter as a 32-bit number to end of the message. The suffix of all messages have the form: $10 \cdots 0 \parallel |M| \parallel s$.

Including the security parameter in the padding of the message is done for much the same reason as including the message's length in the padding of the message. Including the security parameter prevents attacks against the hash function where a collision between two messages is found for different security parameters up to the padding of the message. Including the security parameter in the final block of the message prevents these two messages from colliding under different security parameters. It should be noted that this technique is outlined in [8].

5.1.4 Digest Creation

When the size of the requested digest is equal to the output of the compression function, a single line is used and the output is enveloped using the technique described in [3]. By enveloping the output, multiple properties of the underlying compression function are preserved (see Section 4.3.3). This technique also helps to prevent security parameter collisions because the initial value used in enveloping the output of the compression function is not dependent on the security parameter. The digest is created as follows, assuming the message after padding is k blocks long and $s = n$:

$$H(M, n) = g(IV_2, h_k \parallel 0 \cdots 0).$$

To create digests larger than the output of the compression function, the dynamic hash function construction envelopes the output of each line and the line's number and then concatenates the output of the enveloping step, truncating appropriately. Assuming a digest of $2n$ is requested from the construction, two lines are created, enveloped and then concatenated together to form a $2n$-bit digest. This is done as follows:

$$H(M, 2n) = g(IV_2, h_{k,1} \parallel 0 \cdots 1) \parallel g(IV_2, h_{k,0} \parallel 0 \cdots 0).$$

If the size of the digest requested is between n and $2n$ bits long, the $2n$-bit digest is truncated to the proper length, removing the most significant bits first.

5.1.5 Security Parameter Bounds

In the dynamic hash function construction the size of the digest $d(s)$ is simply the security parameter provided: $d(s) = s$. As for all dynamic hash functions, the range of possible security parameters must be specified. The construction specifies that enveloped lines are concatenated together until a digest that is large enough is created. Therefore, restrictions exist on how small a digest can be. The lower bound of the security parameter for the dynamic hash function construction is $\lambda(l) = n$. This ensures that if a collision resistant compression function is used in the construction, that the overall construction is also collision resistant. This would not necessarily be the case if the output was truncated.

The upper bound on the size of the digest is limited by the technique used to create larger and larger digests. This limitation is investigated in Section 5.2, and turns out to be four times the output size of the compression function. The upper bound on the security parameter is $\upsilon(l) = 4n$.

5.2 The Security of This Construction

All of the properties defined in Chapter 3 are possessed by the dynamic construction with a few common assumptions made about the underlying compression function used. However, as alluded to in Section 5.1.5, there are limitations on the size of a digest the dynamic hash function construction can securely create. Certain attacks become possible as the number of lines in the construction increase. Each of the five properties defined in Chapter 3 are proved to be possessed by the dynamic hash function construction when $\upsilon(l) = 4n$.

Before the properties are investigated, the following definitions and lemmas are provided to aid in proving that the construction is secure. These definitions and lem-

mas are used throughout the rest of this section. It is assumed that the compression function g is a random oracle for this entire section unless explicitly noted otherwise.

Definition 5.2.1 *An* r-way cross collision *occurs when a single message block causes* $2 \leq r \leq \ell$ *lines in the construction to result in the same output:* $g(h_{i,j}, m_i \parallel h_{i,j-1 \mod \ell}) = g(h_{i,j+1 \mod \ell}, m_i \parallel h_{i,j \mod \ell}) = \cdots = g(h_{i,j+r-1 \mod \ell}, m_i \parallel h_{i,j+r-2 \mod \ell})$.

Definition 5.2.2 *An* r-way strict collision *is caused when two different message blocks cause the output of* $1 \leq r \leq \ell$ *lines to be the same:* $g(h_{i,j}, m_i \parallel h_{i,j-1 \mod \ell}) = g(h_{i,j}, m'_i \parallel h_{i,j-1 \mod \ell})$, $\ldots$, $g(h_{i,j+r-1 \mod \ell}, m_i \parallel h_{i,j+r-2 \mod \ell}) = g(h_{i,j+r-1 \mod \ell}, m'_i \parallel h_{i,j+r-2 \mod \ell})$.

Because the compression function g is assumed to be a random oracle with an output size of n, an expected 2^n messages must be tested to find a preimage and an expected $2^{n/2}$ messages must be tested to find a collision. These expected values are generalized and applied to cross collisions and strict collisions in the following Lemma.

Lemma 5.2.1 *If* g *is a random oracle, then the expected number of messages that must be tested before finding a message block that causes a cross collision for* r *lines is* $2^{(r-1)n}$. *The expected number of messages that must be tested before finding two message blocks that cause a strict collision for* r *lines is* $2^{rn/2}$.

Proof To find a cross collision for r lines at iteration i, many messages can be tested comparing the outputs of g at iteration i in the r lines. For each line after the first one, the probability is 2^{-n} that the output of its g is the same as that of g in the first line. Because there are $r-1$ lines after the first, and the lines are independent, the overall probability of an r-way cross collision is $2^{-(r-1)n}$. Therefore, $2^{(r-1)n}$ messages must be tested before a preimage is found for the r lines.

The output size of r lines in the construction is rn. The birthday attack shows that the expected number of messages that must be tested to find a collision is $2^{rn/2}$.

■

One should note that cross collisions can be built up through multiple iterations. In one iteration a cross collision is found for a subset of the lines. In the next iteration other lines are cross collided with the subset that has already been collided. Figure 5.2 shows how cross collisions can be built up through multiple iterations, where capital letters are used to exemplify lines that mimic each other. In the first iteration, a cross collision is found for the first three lines. In the second iteration a cross collision is found for the top, bottom and one of the two middle lines. The other middle line will also collide requiring no additional messages to be tested because the input to the compression function for the two middle lines mimic each other. Building up a cross collision in this manner reduces the overall amount of work required significantly. Before determining how much work is required to build a cross collision iteration by iteration, an invariant of the construction is required.

Theorem 5.2.1 *An r-way cross collision at one iteration will produce at most an $(r-1)$-way cross collision at the next iteration, if no other cross collisions are found.*

Proof Assume that on iteration i r lines cross collide and on iteration $i+1$ at least r lines cross collide. The inputs to each compression function for the r lines that result in the same output for iteration $i+1$ come from pairs of lines $j \mod \ell$ and $j+1 \mod \ell$ by definition of the construction. By definition of modular arithmetic, $j \mod \ell \neq j+1 \mod \ell$. Therefore, at least $r+1$ lines from iteration i are used in constructing the inputs to the compression functions that cross collide for iteration $i+1$. Because only r lines cross collide at iteration i at least one of the inputs to the compression function for one of the lines in iteration $i+1$ must be different. By definition of a random oracle, the output of this compression function will be different from that of the other lines. This breaks our assumption that r lines still collide in iteration $i+1$, proving the theorem correct. ■

Theorem 5.2.1 states that the number of lines that cross collision in one iteration is reduced in each subsequent iteration. This means that if a cross collision is found for four lines, in the next iteration, if no additional work is performed, no more than three

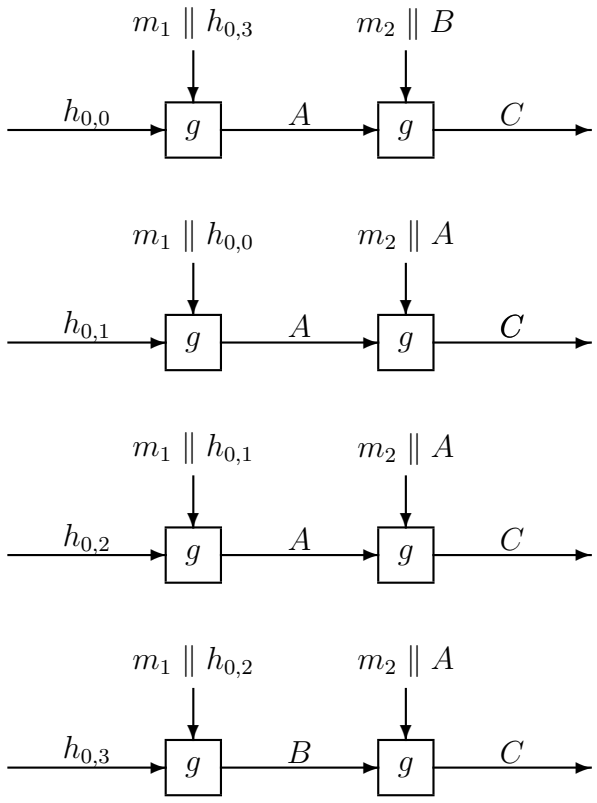

Figure 5.2. Cross collisions built up through multiple iterations.

of the four lines still result in a cross collision. Theorem 5.2.1 is an important fact that is used in the examination of both preimage resistance and collision resistance. Theorem 5.2.1 can be used to aid in calculating the amount of work required to construct a cross collision iteration-by-iteration for a subset of the lines.

Lemma 5.2.2 *The optimal way to find a cross collision for $r \geq 3$ lines is by constructing the cross collision iteration-by-iteration, which requires testing an expected $(r-2)2^{2n}$ messages.*

Proof First it is proved that an expected $(r-2)2^{2n}$ messages must be tested to find a cross collision for $r \geq 3$ lines. By application of Theorem 5.2.1, the smallest r can be so that progress is still made towards combining lines is $r = 3$. When $r = 3$ and the three lines are adjacent to each other, two of the lines will be the same in the next iteration. A cross collision is then found for one of the two lines that still collide and two additional lines. This results in four lines that cross collide, or three lines that still cross collide in the next iteration. Repeating this process and adding up the number of expected messages until all r lines cross collide results in the stated $(r-2)2^{2n}$ messages by application of Theorem 5.2.1 and Lemma 5.2.1.

Any attack that is more efficient than the one described must do less work per iteration or the same amount of work, but in fewer iterations. If less work is performed in each iteration, then only two lines can cross collide. This will result in all lines being different in the next iteration by Theorem 5.2.1. Therefore, no less than 2^{2n} work can be done in each iteration. Any attack that works in fewer steps must cross collide more than one additional line in each iteration. This cannot be done in the same amount of work as cross colliding three lines by Lemma 5.2.1. Therefore, the attack described is the most optimal way to cross collide $r \geq 3$ lines. ■

5.2.1 Dynamic Preimage Resistance

Preimage resistance of the dynamic hash function construction is now considered. To find the expected number of messages that must be tested to find a preimage of any digest is a straightforward calculation.

Lemma 5.2.3 tells the expected number of messages that must be tested when all of the outputs are different. As expected, this scenario is the most costly with respect to testing messages.

Lemma 5.2.3 *If all ℓ output lines of the dynamic hash function construction are different and the compression function g is a random oracle, then an expected $2^{\ell n}$ messages must be tested before finding a preimage for the entire hash function.*

Proof By definition of the dynamic hash function construction, the inputs to the final application of the compression function are different for each line because the line number is part of the last input. An expected 2^n messages must be tested to find a preimage for a single line of the construction by the definition of a random oracle. For ℓ lines, the expected number of messages that must be tested is $2^{\ell n}$, because the inputs to the final compression function are all independent. ■

Lemma 5.2.4 *If the output of $1 < r \leq \ell$ lines of the dynamic hash function construction are the same, then an expected $(r-1)2^{2n}$ messages must be tested to find a preimage for these r lines with a probability greater than $1/2$.*

Proof The most efficient way to find a preimage for r lines is to cross collide $r-1$ of the lines and then search for a preimage for the collided lines an the remaining line. By Lemma 5.2.2, a cross collision for $r-1$ lines can be found by testing an expected $(r-2)2^{2n}$ messages. This will result in $r-1$ lines acting as one line. A preimage can then be found by testing 2^{2n} messages. The total amount of work is the stated $(r-2)2^{2n} + 2^{2n} = (r-1)2^{2n}$. ■

Theorem 5.2.2 *Assuming the compression function g is a random oracle, the expected number of messages that must be tested to find a preimage for the dynamic hash function construction is 2^{cn}, where*

$$c = \ell e^{-\ell 2^{-n}} + 2^{n+1}(1 - (1 - \ell 2^{-n})e^{-\ell 2^{-n}}).$$

In particular, $c \approx \ell$ when $1 \leq \ell \leq 2^{n/2}$, $\ell < c < (2 + 1/e)\ell$ when $2^{n/2} < \ell < 2^n$, and $c \approx 2^{n+1} + 3\ell e^{-\ell 2^{-n}}$ when $\ell \gg 2^n$.

Proof When the outputs of the ℓ lines are all different, then number of messages needed is $2^{\ell n}$ by Lemma 5.2.3. Less work is needed when some lines have the same output. By Lemma 5.2.4 each group of $r \geq 2$ lines with the same output will take $\geq 2^{2n}$ messages (or $(r-1)2^{2n}$) to find a preimage. On the other hand, each line whose output is not repeated requires testing 2^n messages to find a preimage.

Fix one of the 2^n possible line outputs. The probability that it appears exactly r times in a digest having ℓ lines is given by the binomial distribution

$$\binom{\ell}{r}(2^{-n})^r(1-2^{-n})^{\ell-r},$$

and the expected number of line outputs that appear exactly r times is 2^n times this probability.

Because n is at least a few dozen, the probability 2^{-n} is small enough so that this binomial distribution is well approximated by the Poisson distribution (see page 211 of [26])

$$e^{-\lambda}\frac{\lambda^r}{r!},$$

where $\lambda = \ell 2^{-n}$.

Let the total number of messages needed to find a preimage for the entire digest be 2^{cn}. By Lemmas 5.2.3 and 5.2.4,

$$\begin{aligned} c &= (2^n\lambda e^{-\lambda})\cdot 1 + (2^n(1-e^{-\lambda}-\lambda e^{-\lambda}))\cdot 2 \\ &= 2^n\ell 2^{-n}e^{-\ell 2^{-n}} + 2^{n+1}(1-(1-\ell 2^{-n})e^{-\ell 2^{-n}}), \end{aligned}$$

as claimed.

If $1 \leq \ell < 2^{n/2}$, then $0 < \ell 2^{-n} < 2^{-n/2}$, so

$$\begin{aligned} c &\approx \ell e^{-\ell 2^{-n}} + 2^{n+1}(1-(1-\ell 2^{-n})e^{-\ell 2^{-n}}) \\ &\approx \ell + 2^{n+1}(1-(1-\ell 2^{-n})(1-\ell 2^{-n})) \\ &\approx \ell + 2^{n+1}(2\ell^2 2^{-2n}) \\ &\approx \ell + 2^{2-n}\ell^2. \end{aligned}$$

But $\ell^2 < 2^n$, so $2^{2-n}\ell^2$ is small compared to ℓ and thus $c \approx \ell$.

If $\ell = 2^n$, then $\ell 2^{-n} = 1$, so $c = 2^n e^{-1} + 2^{n+1}(1-0) = 2^n(2+1/e)$. As ℓ increases from $2^{n/2}$ to 2^n, the factor $e^{-\ell 2^{-n}}$ decreases smoothly from approximately 1 to e^{-1}, so the first term increases smoothly from $\ell = 2^{n/2}$ to $\ell/e = 2^n/e$. On the other hand, as ℓ increases from $2^{n/2}$ to 2^n, the second term increases smoothly from 0 to $2^{n+1} = 2\ell$. Thus c increases smoothly from ℓ to $(2+1/e)\ell$ as ℓ increases from $2^{n/2}$ to 2^n.

Finally, suppose $\ell = k \cdot 2^n$, where k is large. Then $\ell 2^{-n} = k$ and

$$\begin{aligned} c &= 2^n k e^{-k} + 2^{n+1}(1-(1-k)e^{-k}) \\ &= 2^{n+1} + 2^n(e^{-k}(k-(1-k)2) \\ &= 2^{n+1} + 2^n e^{-k}(3k-2) \\ &\approx 2^{n+1} + 2^n \cdot 3k e^{-k} = 2^{n+1} + 3\ell e^{-\ell 2^{-n}}, \end{aligned}$$

as claimed, completing the proof for Theorem 5.2.2. ■

Corollary 5.2.1 *Assuming the compression function g is a random oracle, the dynamic hash function created from this construction is $(2^{cn}, 1)$-dynamic preimage resistant, where $c = c(\ell) = c(\lceil s/n \rceil)$ is specified in Theorem 5.2.2.*

Proof The adversary's algorithm A tries 2^{cn} messages to find a preimage and the algorithm succeeds by Theorem 5.2.2. ■

Similar corollaries hold for (t, ϵ)-dynamic preimage resistance whenever the ratio $t/\epsilon \approx 2^{cn}$.

Because the size of the output of the compression function, n, is likely to be more than 100, it is true that $\ell < 2^{50} < 2^{n/2}$ for any practical instance of the construction. In this situation, Theorem 5.2.2 gives $c = \ell = \lceil s/n \rceil$. Here the dynamic hash function construction is (t, ϵ)-dynamic preimage resistant so long as $t/\epsilon \approx 2^{cn} = 2^{\lceil s/n \rceil n} \approx 2^s = 2^{d(s)}$. This shows, for example, that if the time t is fixed, then ϵ is multiplied by 2^{-s} when s is doubled, so long as $2s < 2^{n/2}$.

5.2.2 Dynamic Collision Resistance

All attacks against collision resistance that reduce the amount of work required to find a collision have one of two forms. The first method for attacking this construction is to find cross collisions so that multiple lines can be treated as a single line, and then find a strict collision for the subset of lines. The second method is the opposite, finding a strict collision for a subset of lines and then find cross collisions to combine them together. Theorem 5.2.3 proves that causing a strict collision in more than one message block does not help to efficiently attack this construction.

Theorem 5.2.3 *The most efficient attack, excluding the birthday attack, that causes a collision in the dynamic hash function construction produces two messages that differ in only one message block, that is, there is just one strict collision.*

Proof Without loss of generality, assume that for two messages $M \neq M'$ and $H(M, s) = H(M', s)$, that the i^{th} block in each message is the first message block that is different in the two messages. There are two cases.

Case 1: The i^{th} message blocks in each message cause a strict collision for the entire construction. In this case searching for another pair of message blocks that causes a strict collision requires additional, unnecessary, work. Also, this is the birthday attack.

Case 2: The i^{th} message blocks in each message cause a strict collision in a proper subset of the lines. Let $h_i \neq h_i'$ denote the two outputs of some g in a line not part of the strict collision. If a cross collision is not found for the two outputs h_i and h_i', then those differences will propagate to additional lines with each iteration that a cross collision is not found. Letting the two different outputs propagate requires twice as much work to cause a full collision.

If additional strict collisions are used to collide all of the lines, those lines that were not collided and those that were must be considered. Those lines that were not collided count twice in the calculation of the expected number of messages because the inputs for either previous message block must be considered. Let r be the number

of messages that do not collide by the strict collision. Each line that does not collide results in two different inputs to the compression function in the next iteration. These two different inputs require additional work to cause a collision. In the scenario the strict collision causes the first $\ell - r$ lines to collide. This results in $(2r+2)+(\ell-(r+1))$ inputs that must be considered for the next strict collision. By Theorem 5.2.1, the expected number of messages that must be tested to find the two strict collisions is:

$$2^{(\ell-r)n/2} + 2^{((2r+2)+(\ell-(r+1)))n/2}.$$

For two strict collisions to be more efficient than finding cross collisions first, $((2r + 2) + (\ell - (r + 1)))n$ has to be less than $\log_2((\ell - r) - 2) + 2n$. This can never happen with integer values for ℓ, r and n.

Therefore, any attack that finds more than one strict collision requires more work than finding a single strict collision. ■

To find an attack more efficient than the birthday attack, by application of Theorem 5.2.3 only a single strict collision should be found for t of the ℓ lines, where $t < \ell$. To find a strict collision for t lines, $2^{tn/2}$ messages must be tested, by application of Lemma 5.2.1. By application of Lemma 5.2.2 the fastest way to collide the remaining lines is by building cross collisions iteration-by-iteration. However, one should note that each line that does not collide by the strict collision results in $2(\ell - t)$ "lines" that must be collided with a cross collision. This is because there will be two different outputs (one for each message) for each line not collided during the strict collision which are inputs to the next iteration. These different inputs require more work to cross collide the line; so twice as much work. Therefore, it is more beneficial to cross collide lines before a strict collision than after.

Lemma 5.2.5 *Any efficient attack other than the birthday attack that finds a collision for the entire construction uses cross collisions before the strict collision.*

Proof By application of Theorem 5.2.3 any optimal attack other than the birthday attack against the construction uses one strict collision. Let t be the number of lines

that collide after the strict collision. For the $\ell - t$ lines that do not collide after the strict collision, the cross collision(s) must consider $2(\ell - t)$ "lines" by definition of the construction. If the $\ell - t$ lines were collided via a cross collision before the strict collision, less work would be required for the overall attack, by application of Lemma 5.2.1. To collide $\ell - t$ lines before the strict collision, $((\ell - t) - 2)2^{2n}$ messages must be tested. After the strict collision, $(2(\ell - t) - 2)2^{2n}$ messages must be tested. Therefore, finding cross collisions before the strict collision is always more efficient. ■

Because of Lemma 5.2.5, the only efficient attack against this construction is performed by finding a cross collision for a subset of the lines, and then a strict collision for the remaining lines. Using Lemma 5.2.2, finding cross collisions one iteration at a time is the most efficient method for cross colliding lines. The overall number of messages that must be tested is determined by adding the number of messages that must be tested for the cross collisions to the number of messages that must be tested to cause a strict collision for the remaining lines.

Theorem 5.2.4 *If r is the number of lines collided by the cross collision(s), then an expected $(r - 2)2^{2n} + 2^{(\ell - r + 1)n/2}$ messages must be tested before a collision is found for the entire construction.*

Proof First, r cross collisions are found requiring $(r - 2)2^{2n}$ messages to be tested from a direct application of Lemma 5.2.2. Then a strict collision for the remaining lines is found by testing an expected $2^{(\ell - r + 1)n/2}$ messages, which is derived from Lemma 5.2.1 and the fact that even if all the lines are the same at the iteration where the r-way cross collision occurs, a single strict collision must still be found. It is that fact that accounts for the additional $n/2$ term in the exponent. Therefore, these two pieces added together results in the expected number of messages that must be tested. ■

It is clear that when $r = \ell$ the number of messages that must be tested is minimized. When this occurs, the expected number of messages is reduced to $(\ell - 2)2^{2n} +$

$2^{n/2}$. To ensure that the dynamic hash function construction is secure with respect to collision resistance, it is shown that the expected number of messages is $2^{d(s)/2}$ for all valid values of s, that is, for $n \leq s \leq 4n$.

Theorem 5.2.5 *If the compression function g is a random oracle, then the dynamic hash function construction is (t, ϵ)-dynamic collision resistant, for any t and ϵ with $t/\epsilon = 2^{d(s)/2}$.*

Proof To determine whether the dynamic hash function construction is secure with respect to collision resistance, the possible values of ℓ can be tested to ensure the expected number of messages is $2^{\ell n/2}$. Equations (5.1), (5.2), and (5.3) compare the number of messages that must be tested for both the attack described in Theorem 5.2.4 and the birthday attack, when $\ell = 1$, $\ell = 2, 3, 4$, and $\ell = 5$ respectively. If the number of messages that must be tested for the former attack is less than $2^{\ell n/2}$, then the construction is insecure for that value of ℓ. One has

$$2^{n/2} = 2^{n/2} \quad \text{when } \ell = 1 \tag{5.1}$$

$$(\ell - 2)2^{2n} + 2^{n/2} > 2^{\ell n/2} \quad \text{when } \ell = 2, 3, 4 \tag{5.2}$$

$$(5 - 2)2^{2n} + 2^{n/2} < 2^{5n/2} \quad \text{when } \ell = 5. \tag{5.3}$$

Therefore, when the security parameter ($s = d(s) \leq \ell n$) is less than or equal to $4n$ the construction is (t, ϵ)-dynamic collision resistant with $t/\epsilon \approx 2^{d(s)/2}$. ■

Theorem 5.2.5 is the reason the maximum security parameter is limited to $4n$ in Section 5.1.5. Allowing the security parameter to be larger than four allows for a more efficient attack against the construction making the overall construction not (t, ϵ)-dynamic collision resistant, with $t/\epsilon \approx 2^{d(s)/2}$. Section 5.3 discusses possible methods for extending the dynamic hash function construction to create larger digests.

5.2.3 Security Parameter Collision Resistance

The dynamic hash function construction is always security parameter collision resistant. For an attacker to successfully attack the construction with respect to security parameter collision resistance, the attacker must create two digests using different security parameters that are the same length. However, the way the dynamic hash function construction is defined this can never occur because $s = d(s)$. The following theorem states this precisely.

Theorem 5.2.6 *For any compression function g, the dynamic hash function construction is (t, ϵ)-security parameter collision resistant, for any positive t and ϵ.*

Proof By the definition of (t, ϵ)-security parameter collision resistance, an attacker must find a message M and a second security parameter s_2 such that $s_1 \neq s_2$ and $d(s_1) = d(s_2)$. By definition of the dynamic hash function construction, $d(s_1) \neq d(s_2)$ if $s_1 \neq s_2$. Therefore, the dynamic hash function is (t, ϵ)-security parameter collision resistant. ■

5.2.4 Digest Resistance

Theorem 5.2.7 *If the compression function g is a random oracle, then the dynamic hash function construction is (t, ϵ)-digest resistant, for any t and ϵ with $t/\epsilon \geq 2^{cn}$, with c as in Theorem 5.2.2.*

Proof By the definition of digest resistance, an attacker is given s_1 and $H(M, s_1)$, but not M and must find a second security parameter s_2 and the digest of M computed using security parameter s_2. Because the security parameter is part of the input to the last compression function in each line, no input to these compression functions when computing $H(M, s_1)$ can equal any input to these compression functions when computing $H(M, s_2)$. Because g is assumed to be a random oracle, the outputs

of these compression functions are chosen uniformly at random from the set of all possible outputs.

As the total width of these outputs is $d(s) = s \leq \ell n$, the only hope of finding $H(M, s_2)$ from $H(M, s_1)$ is to find a M' with $H(M', s_1) = H(M, s_1)$, choose $s_2 \neq s_1$ in the interval $(\ell - 1)n < s_2 \leq \ell n$, and then compute $H(M', s_2)$. If $M' = M$, then this works. If $M' \neq M$, then one cannot compute $H(M, s_2)$ from $H(M, s_1)$ at all. The effort required to find M' with $H(M', s_1) = H(M, s_1)$ is 2^{cn}, with c as in Theorem 5.2.2. ■

5.3 Expanding the Digest Size Beyond $4n$

One possible method for extending the dynamic hash function construction beyond $4n$-bit digests is by imposing limits on the block size of the compression function. It is clear that $b > n$ for any of the message to be processed with each iteration. However, if $b < 3n$ (or $b - n < 2n$), then the optimal attack described in Section 5.2.2 cannot be launched. The attack requires that 2^{2n} messages be tested for each iteration until enough cross collisions are found such that all of the lines produce the same output. However, if the message block size, $b - n$, is not large enough to make this possible, then this type of an attack will not always work. This occurs when $b - n < 2n$. This prevents the collision finding attack described in Section 5.2, making the dynamic hash function construction secure for all of the properties when $b < 3n$.

It is clear that Theorem 5.2.2 still holds because the number of messages that must be tested in Lemma 5.2.4 increases from $(r-1)2^{2n}$ to $2^{(r-1)n}$ to find the r cross collisions needed. Therefore, the construction is still secure with respect to dynamic preimage resistance.

The construction also remains secure with respect to both security parameter collision resistance and digest resistance. The change in block size does not, in any way, affect Theorem 5.2.6. The effects on Theorem 5.2.7 are the same as on Theorem

5.2.2; therefore, the construction is remains secure with respect to digest resistance. Dynamic collision resistance is shown in Theorem 5.3.1 to be secure.

Theorem 5.3.1 *If $b < 3n$, then the best collision finding attack is the birthday attack.*

Proof The attack described in Section 5.2.2 cannot be launched. Theorem 5.2.1 states that to build cross collision iteration by iteration, at least three of the lines must cross collide. However, the restriction of $b - n < 2n$ limits the number of messages that can be tested to 2^{b-n}, which is less than 2^{2n}, the number required for even three lines to cross collide in a single iteration.

Theorem 5.2.3 and Lemma 5.2.5 still apply, so the most optimal attack is to find a cross collision for all of the lines, and then find a single strict collision. The only way to cause a cross collision for all of the lines is by using multiple message blocks. If multiple message blocks are treated as if they are the input to a single larger compression function, Lemma 5.2.1 states that $2^{(\ell-1)n}$ message blocks must be tested to find a cross collision for all of the lines. However, when $\ell \geq 2$, this clearly requires at least as much work as launching the birthday attack, with an expected $2^{\ell n/2}$ messages being tested.

From this it is clear that as many of the lines as possible must be cross collided and the rest found to collide with a single strict collision. However, at most three lines can collide with a single strict collision because of the restriction on the message block's size. Therefore, the total number of messages that must be tested is $2^{(\ell-2)n} + 2^{3n/2}$. The $2^{(\ell-2)n}$ is derived from the number of messages that must be tested to cross collide $\ell - 1$ lines, which is required because one line is lost due to Theorem 5.2.1. The strict collision of three lines gives the $2^{3n/2}$.

To determine when this attack is successful, each piece can be examined to ensure that it is less than $2^{\ell n/2}$, the amount of work for the birthday attack. For $2^{(\ell-2)n} < 2^{\ell n/2}$, it is required that $\ell < 4$, while $2^{3n/2} < 2^{\ell n/2}$ requires $\ell > 3$. These two restrictions upon ℓ are contradictory. Therefore, no attack against the dynamic hash function construction is more efficient than the birthday attack when $b < 3n$. ■

Corollary 5.3.1 *If $b < 3n$ the dynamic hash function created from this construction is (t, ϵ)-dynamic collision resistant for any t, ϵ with $t/\epsilon \approx 2^{\ell n/2} \approx 2^{s/2}$.*

Proof When $b < 3n$, Theorem 5.3.1 states that the birthday attack is the most efficient way to find a collision. An expected $2^{\ell n/2}$ messages must be tested for the birthday attack to succeed with high probability. ■

The corollary shows, for example, that when s is doubled, the security ratio t/ϵ is squared.

While this restriction creates a construction that can securely scale to any ℓn-bit digest, it imposes a somewhat unreasonable restriction upon the block size of the compression function. The block size of common compression functions, such as the ones used for SHA-1 or RIPEMD-160 is $3.2n$. This means that all of these common and well known compression functions are unusable with the dynamic hash function construction. Filling part of the message block with some fixed value can reduce the actual size of the message block to meet the requirement, however, at the loss of efficiency.

For reasons of efficiency, and lack of a suitable compression function, the message block size restriction was not placed on the dynamic hash function construction. Allowing digests of only four times the output size of the compression function is also suitable for most applications. For example, using SHA-1's compression function a 640-bit digest can be constructed. This is larger than the output of the largest hash function standardized by NIST, SHA-512. Using the SHA-512 compression function results in a 2,048-bit digest, larger than most keys used in digital signatures schemes.

5.4 Additional Properties

In [3] it was proved that the Enveloped Merkle-Damgård construction preserves the properties of collision resistance, pseudorandom oracles and pseudorandom functions. It was proved in Section 5.2.2 that the dynamic hash function construction

preserves collision resistance, but it can also be shown that it preserves pseudorandom oracles and pseudorandom functions.

In fact, the dynamic hash function construction is equivalent to the Enveloped Merkle-Damgård construction with respect to the preservation of pseudorandom oracles and pseudorandom functions. The only difference between the dynamic hash function construction and the Enveloped Merkle-Damgård construction is the concatenation of multiple lines together to form a larger digest.

Theorem 5.4.1 *The dynamic hash function construction is equivalent to the Enveloped Merkle-Damgård construction with respect to preserving pseudorandom oracles and pseudorandom functions.*

Proof When $s = n$ the dynamic hash function construction is exactly the same as the Enveloped Merkle-Damgård construction. When $s > n$ the digest created can be thought of as the concatenation of multiple hash functions all using the Enveloped Merkle-Damgård construction. While the iterations up to the enveloping step are dependent upon each other, the final enveloping step is independent for each line. With respect to the property preservation of pseudorandom oracles and pseudorandom functions, the concatenation of these hash functions preserves the properties. ■

5.5 Provisions for Adding Salt

A salt value helps to prevent precomputation of collisions for a specific hash function. While the security parameter attempts to provide this type of security, the domain of the security parameter may not be large enough to prevent precomputation. However, a 32-bit salt provides enough variation on the same message to prevent such precomputation by today's standards.

A salt can be added to this construction by concatenating the salt to each message block. To increase the security further, the salt, treated as an integer, can be incremented once for each message block that is processed by the construction. A similar

technique is described in [8] to prevent the fixed point attack (see Section 4.3.4). The modified version of the message processing is as follows:

$$\begin{aligned} IV_{s,j} &= g(IV_1, IV_1 \parallel s \parallel j) \qquad \text{for } j = 0 \ldots \ell - 1 \\ h_{1,j} &= g(IV_{s,j}, salt \parallel m_1 \parallel IV_{s,[j+1]}) \qquad \text{for } j = 0 \ldots \ell - 1 \\ h_{i,j} &= g(h_{i-1,j}, salt + i \parallel m_i \parallel h_{i-1,[j+1]}) \qquad \text{for } i = 0 \ldots k, \text{ for } j = 0 \ldots \ell - 1 \end{aligned}$$

It is clear that adding a salt will reduce the efficiency of the construction; however, the increase in security may be worth the trade-off. For example, passwords are commonly precomputed in an attempt to make it easier to break a password file. Adding a salt to a password before it is hashed is the most common way to make such a precomputation infeasible. The amount of time required to hash a password that is a single block long is much less than the time required to hash the same single block password with all possible salt values. However, the extra time needed to hash the contents of a CD or DVD when using a salt will have a large impact and may not make sense.

5.6 Preventing the Multi-Collision Attack

The same technique used by the dynamic hash function construction to increase the security as the security parameter increases can also be applied to preventing the multi-collision attack. In [43] a technique for expanding the size of the internal values was presented which would prevent the multi-collision attack (see Section 4.3.1). The requirement for preventing the attack is that the size of the output of the internal compression function must be at least twice the size of the output of the hash function. In the case of the dynamic hash function construction, the concatenation of the intermediate values is considered the output of the internal compression function.

To achieve this with the dynamic hash function construction more lines will be needed as the size of the digest increases. For example, if $s = n$, only a single line is required by the dynamic hash function construction. However, an attacker can easily launch the multi-collision attack against the construction. If two lines were

used, the attack would not be possible as it would require trying an expected 2^n message blocks to find an internal collision. Expanding this logic, if $s = 2n$, then four lines are required. This way the expected number of message blocks would climb to 2^{2n}, preventing the multi-collision attack. One should note that while this attack is prevented, an exponential increase in work is required. As the size of the digest increases, the number of lines must increase two-fold. As with the salting modification, preventing this type of an attack may only make sense in certain situations, and can only be applied for $s \leq 2n$.

5.7 Implementation Issues

While the dynamic hash function construction appears difficult to implement, there are several features that programming languages can leverage to easily and efficiently implement the dynamic hash function construction. First, the initial values are always created the same way for each security parameter. Time can be saved by storing the initial values for common security parameters in a table so that they do not need to be recomputed each time the function computes a digest. In most architectures a table lookup will be more efficient than a computation of the compression function.

Second, with multiple CPUs becoming more prevalent in computers, the natural parallelism of this construction can be exploited to compute digests more efficiently. Each line in the construction is computed in the same manner and therefore the same code can be used in parallel to compute each line. If multiple processors are not available, as is the case on such restricted architectures as smart cards, each of the computations can be done serially with only a linear increase in work needed for each line required to compute the digest.

Third, while a number of variables are concatenated in this construction, all of them have lengths that facilitate efficient concatenation for most architectures. The lengths of the intermediate values concatenated with the message blocks are multiples

of the word size of the target architecture for the underlying compression function that is used. Individual bits are never manipulated outside of the compression function, with the exception of the padding algorithm, which is only used once per digest.

Finally, the padding algorithm used with the dynamic hash function construction is essentially the same as the padding algorithm used for the Merkle-Damgård construction. Code that implements the padding algorithm for the Merkle-Damgård construction requires only minor modifications to adapt it to work with the dynamic hash function construction. The 64 bits usually required for the message's size can be increased to 96 bits to include the digest size as well. Concatenating the digest size to the end of the message after the message's length can be done with the same code used to concatenate the message's length.

5.7.1 Speed Comparison

To test the relative speed of the dynamic hash function construction described in Section 5.1 compared to the commonly used SHA family of hash functions, both were implemented with the same level of optimization and the running time of each was recorded. Table 5.1 shows the relative time between the SHA functions and the dynamic hash function construction using the SHA-1 compression function when computing the digest of a 1 MB message.

Table 5.1
The relative speed between the SHA functions and the dynamic hash function construction using the SHA-1 compression function when calculating the digest of a 1 MB message.

Digest Size	**Relative Speed**
160	1.42
256	1.38
512	1.85

For a 160-bit digest, the SHA-1 function was 1.42 times faster than the dynamic hash function construction using the SHA-1 compression function. While this might seem like a large decrease in speed when using the dynamic hash function construction, the dynamic hash function construction can only process 352 bits of the message per iteration. In contrast, the SHA-1 function can process 160 bits of the message per iteration. The SHA-1 function is able to process messages with fewer than half of the iterations of the dynamic hash function construction.

When calculating a 256-bit digest, the SHA-256 function is actually slower than the dynamic hash function construction. This is most likely due to the fact that the dynamic hash function construction must run the SHA-1 compression function twice per message block, but the SHA-1 compression function is twice as fast as the SHA-256 compression function. Therefore, the amount of time required to process one message block for a single line compared to two lines increases at approximately the same rate as the SHA-1 function compared to the SHA-256 function. The slight variation in relatives speeds can most likely be attributed to rounding and inaccurate timing of the CPU.

The 512-bit digest is calculated almost twice as slowly by the dynamic hash function construction as the SHA-512 function. The dynamic hash function construction must compute the SHA-1 compression function four times per message block. The SHA-1 compression function is however not even three times faster than the SHA-512 compression function. This attributes to the increase in relative computation time.

Overall the dynamic hash function construction is slower than the SHA family of functions for the three digest sizes that were tested. However, the actual speed is still practical. The dynamic hash function construction only required 37 clock ticks[2] to compute the 512-bit digest for a message of 1 MB in length. For most applications this is a more than adequate amount of time to compute a digest for a message of its size.

[2]See the Linux manual page for the `gettimeofday` system call for a definition of "clock tick."

6 SUMMARY

6.1 Conclusion

This dissertation introduced a new type of cryptographic hash function, the dynamic cryptographic hash function. Dynamic cryptographic hash functions are different from traditional cryptographic hash functions in that a security parameter dictates how the digest is computed. The goal of a dynamic cryptographic hash function is essentially the same as a traditional hash function: provide a cryptographically secure hash function with respect to the properties of preimage resistance, second preimage resistance, and collision resistance. However, because an adversary potentially has control of the way in which the digest is computed, additional security properties are required to ensure a dynamic cryptographic hash function is secure.

Security parameter collision resistance and digest resistance, were introduced and formally defined in Chapter 3. These new properties prevent an adversary from gaining an advantage in attacking the dynamic hash function by controlling the security parameter. For example, digest resistance prevents an attacker from creating a smaller version of the digest for a message, and then searching for a collision for the larger digest. Security parameter collision resistance and digest resistance are what differentiate dynamic cryptographic hash functions from previous cryptographic hash functions that simply create variable size digests.

This dissertation also presented a number of attacks against the Merkle-Damgård construction. The Merkle-Damgård construction is important because most of the cryptographic hash functions used today are built upon this construction. In light of these attacks new constructions by various authors were also presented. Most of these constructions only seek to thwart a specific attack against the Merkle-Damgård con-

struction. While each succeeds in thwarting a specific attack, no single construction thwarts them all.

The dynamic hash function construction presented in this dissertation has the ability to thwart all of the known attacks against the Merkle-Damgård construction and produces a variable size digest. The dynamic hash function construction leverages many of the techniques used by the new constructions, combining them in a secure way. The resulting construction is resistant to most of the attacks against the Merkle-Damgård construction and those that are not can be thwarted by simply increasing the digest's size.

6.2 Future Work

The area of cryptographic hash functions is one of the most active areas in cryptography today. Much research is needed to understand all of the properties required of a hash function to consider it cryptographically secure and how to achieve these properties. The first area for future work is in identifying versions of the properties listed in this dissertation that are potentially useful for cryptographic hash functions to possess. While the implications between properties are important to understand, it is also of great importance to understand what properties protocol and system designers are counting on.

This dissertation presented a construction for creating a dynamic cryptographic hash function from a traditional compression function. More efficient methods are needed to create a viable dynamic cryptographic hash function. The most obvious avenue for creating a more efficient dynamic cryptographic hash function is by developing a dynamic compression function that can be used with the standard Merkle-Damgård construction or one of its improved variants. Such a compression function will allow for the more efficient computation of digests while leveraging all that is known about the Merkle-Damgård construction.

Finally, much more work needs to be done in determining whether a compression function possess a given security property. While there have been a few examples of provably secure compression functions, they are usually based on number theory and are slower than dedicated compression functions. All of the properties proved about the Merkle-Damgård construction and the dynamic cryptographic hash function construction rely on certain properties being possessed by the underlying compression function. Much more work is needed in this area to create efficient and provably secure dedicated compression function.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference*, volume 4117 of *Lecture Notes in Computer Science*, pages 602 – 619, Santa Barbra, California, USA, August 2006. Springer-Verlag.

[2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference*, volume 1109 of *Lecture Notes in Computer Science*, pages 1 – 15, Santa Barbara, California, USA, August 1996. Springer-Verlag.

[3] Mihir Bellare and Thomas Ristenpart. Multi-property-preserving hash domain extension and the EMD transform. In *Advances in Cryptology - ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 299–314, Shanghai, China, December 2006. Springer-Verlag.

[4] Mihir Bellare and Thomas Ristenpart. Multi-property-preserving hash domain extension: The EMD transform. In *The Second Cryptographic Hash Workshop*, Santa Barbara, California, USA, August 2006.

[5] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. Chapter 5: `http://www-cse.ucsd.edu/users/mihir/cse207/w-hash.pdf`, September 2005.

[6] Steven Bellovin and Eric Rescorla. Deploying a new hash algorithm. In *Cryptographic Hash Workshop*, Gaithersburg, Maryland, USA, October 2005. National Institue of Standards and Technology.

[7] Eli Biham and Rafi Chen. Near-collisions of SHA-0. In Matthew K. Franklin, editor, *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference*, Lecture Notes in Computer Science, pages 290–305, Santa Barbara, California, USA, August 2004. Springer.

[8] Eli Biham and Orr Dunkelman. A framework for iterative hash functions – HAIFA. In *The Second Cryptographic Hash Workshop*, Santa Barbara, California, USA, August 2006. National Institue of Standards and Technology.

[9] John Black. *Message Authentication Codes*. Thesis (Ph.D.), University of California Davis, Davis, California, 2000.

[10] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In Moti Yung, editor, *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335, Santa Barbara, California, USA, August 2002. Springer-Verlag.

[11] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, December 2001. `http://eprint.iacr.org/2000/067.pdf`.

[12] J. Lawrence Carter and Mark Wegman. Universal classes of hash functions (extended abstract). In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106 – 112, Boulder, Colorado, USA, 1977. ACM Press.

[13] Florent Chabaud and Antoine Joux. Differential collisions in SHA-0. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference*, Lecture Notes in Computer Science, pages 56–71, Santa Barbara, California, USA, August 1998. Springer.

[14] Denis Charles, Eyal Goren, and Kristin Lauter. Cryptographic hash functions from expander graphs. Cryptology ePrint Archive, Report 2000/021, January 2006. `http://eprint.iacr.org/2000/021.pdf`.

[15] Scott Contini, Arjen K. Lenstra, and Ron Steinfeld. VSH, an efficient and provable collision-resistant hash function. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 4004 of *Lecture Notes in Computer Science*, pages 165 – 182, St. Petersburg, Russia, May 2006. Springer.

[16] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Book Company, second edition, 2002.

[17] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference*, Lecture Notes in Computer Science, pages 430–448, Santa Barbara, California, USA, August 2005. Springer.

[18] Ivan Damgård. *The application of claw free functions in cryptography*. Thesis (Ph.D.), Aarthus University, Mathematical Institute, 1988.

[19] Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference*, Lecture Notes in Computer Science, pages 416–427, Santa Barbara, California, USA, August 1989. Springer.

[20] Bert den Boer and Antoon Bosselaers. An attack on the last two rounds of MD4. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference*, Lecture Notes in Computer Science, pages 194–203, Santa Barbara, California, USA, August 1991. Springer.

[21] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644 – 654, November 1976.

[22] Hans Dobbertin. RIPEMD with two-round compress function is not collision-free. *Journal of Cryptology*, 10(1):51–70, 1997.

[23] Hans Dobbertin. The first two rounds of MD4 are not one-way. In Serge Vaudenay, editor, *Fast Software Encryption, 5th International Workshop, FSE '98*, Lecture Notes in Computer Science, pages 284–292, Paris, France, March 1998. Springer.

[24] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption, Third International Workshop*, volume 1039 of *Lecture Notes in Computer Science*, pages 71 – 82, Cambridge, United Kingdom, February 1996. Springer-Verlag.

[25] Orr Dunkelman. E-Mail correspondence with Orr Dunkelman, November 2006.

[26] W. Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley, New York, 1957.

[27] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley Publishing, Indianapolis, Indiana, USA, 2003.

[28] Réjane Forré. The strict avalanche criterion: Spectral properties of boolean functions and an extended definition. In Shafi Goldwasser, editor, *Advances in Cryptology - CRYPTO '88, 8th Annual International Cryptology Conference*, Lecture Notes in Computer Science, pages 450–468, Santa Barbara, California, USA, August 1988. Springer.

[29] Praveen Gauravaram, William Millan, Juanma Gonzalez Nieto, and Edward Dawson. 3C - A provably secure pseudorandom function and message authentication code. A new mode of operation for cryptographic hash function. Cryptology ePrint Archive, Report 2005/390, 2005. `http://eprint.iacr.org/2005/390.pdf`.

[30] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *2006 IEEE Symposium on Security and Privacy (S&P 2006)*, pages 371–385, Berkeley, California, USA, 2006. IEEE Computer Society.

[31] Shai Halevi and Hugo Krawczyk. The RMX transform and digital signatures. In *The Second Cryptographic Hash Workshop*, Santa Barbara, CA, USA, August 2006. National Institue of Standards and Technology.

[32] Shai Halevi and Hugo Krawczyk. Strengthening digital signatures via randomized hashing. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006: 26th Annual International Cryptology Conference*, volume 4117 of *Lecture Notes in Computer Science*, pages 41 – 59, Santa Barbra, California, USA, August 2006. Springer.

[33] Shoichi Hirose. Provably secure double-block-length hash functions in a black-box model. In Choonsik Park and Seongtaek Chee, editors, *Information Security and Cryptology - ICISC 2004, 7th International Conference*, volume 3506 of *Lecture Notes in Computer Science*, pages 330–342, Seoul, Korea, December 2005. Springer-Verlag.

[34] Walter Hohl, Xuejia Lai, Thomas Meier, and Christian Waldvogel. Security of iterated hash functions based on block ciphers. In Douglas R. Stinson, editor, *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference*, volume 773 of *Lecture Notes in Computer Science*, pages 379–390, Santa Barbara, California, USA, August 1994. Springer-Verlag.

[35] Antoine Joux. Multicollisions in iterated hash functions. Application to cascaded constructions. In Matthew K. Franklin, editor, *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316, Santa Barbara, California, USA, August 2004. Springer.

[36] John Kelsey and Tadayoshi Kohno. Herding hash functions and the Nostradamus attack. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200, St. Petersburg, Russia, May 2006. Springer.

[37] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. Cryptology ePrint Archive, Report 2004/304, November 2004. `http://eprint.iacr.org/2004/304.pdf`.

[38] Lars R. Knudsen and Bart Preneel. Hash functions based on block ciphers and quaternary codes. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology - ASIACRYPT '96, International Conference on the Theory and Applications of Cryptology and Information Security*, volume 1163 of *Lecture Notes in Computer Science*, pages 77–90, Kyongju, Korea, November 1996. Springer-Verlag.

[39] Lars R. Knudsen and Bart Preneel. Fast and secure hashing based on codes. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference*, volume 1294 of *Lecture Notes in Computer Science*, pages 485–498, Santa Barbara, California, USA, August 1997. Springer-Verlag.

[40] Lars R. Knudsen and Bart Preneel. Construction of secure and fast hash functions using nonbinary error-correcting codes. *IEEE Transactions on Information Theory*, 48(9):2524–2539, 2002.

[41] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. Request For Comments 2104, Network Working Group, February 1997.

[42] Xuejia Lai and James L. Massey. Hash functions based on block ciphers. In Rainer A. Rueppel, editor, *Advances in Cryptology - EUROCRYPT '92, Workshop on the Theory and Application of Cryptographic Techniques*, volume 658 of *Lecture Notes in Computer Science*, pages 55–70, Balatonfüred, Hungary, May 1992. Springer.

[43] Stefan Lucks. Design principles for iterated hash functions. Cryptology ePrint Archive, Report 2004/253, September 2004. `http://eprint.iacr.org/2004/253.pdf`.

[44] Stefan Lucks. A failure-friendly design principle for hash functions. In Bimal K. Roy, editor, *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494, Chennai, India, December 2005. Springer.

[45] Ueli Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *Theory of Cryptography - TCC 2004*, volume 2951 of *Lecture Notes in Computer Science*, pages 21 – 39. Springer, February 2004.

[46] Ueli M. Maurer and Johan Sjödin. Single-key AIL-MACs from any FIL-MAC. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 472–484, Lisbon, Portugal, July 2005. Springer.

[47] E. H. Mckinney. Generalized birthday problem. *The American Mathematical Monthly*, 73(4):385 – 387, April 1966.

[48] Alfred Menezes, Paul Van Oorschot, and Scott Vanstone. *Handbook of applied cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press LLC, 2000 N.W. Corporate Blvd., Boca Raton, Florida 33431, 1997.

[49] Ralph C. Merkle. One way hash functions and DES. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference*, Lecture Notes in Computer Science, pages 428–446, Santa Barbara, California, USA, August 1989. Springer.

[50] Ilya Mironov. Hash functions: Theory, attacks, and applications. Microsoft Research Technial Report TR-2005-187, November 2005. `ftp://ftp.research.microsoft.com/pub/tr/TR-2005-187.pdf`.

[51] Mridul Nandi. Towards optimal double-length hash functions. In Subhamoy Maitra, C. E. Veni Madhavan, and Ramarathnam Venkatesan, editors, *Progress in Cryptology - INDOCRYPT 2005, 6th International Conference on Cryptology*, volume 3797 of *Lecture Notes in Computer Science*, pages 77–89, India, Bangalore, December 2005. Springer.

[52] Mridul Nandi, Wonil Lee, Kouichi Sakurai, and Sangjin Lee. Security analysis of a 2/3-rate double length compression function in the black-box model. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 243–254, Paris, France, February 2005. Springer-Verlag.

[53] Mridul Nandi and Douglas Stinson. Multicollision attacks on generalized hash functions. Cryptology ePrint Archive, Report 2004/330, May 2005.

[54] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 33 – 43, Seattle, Washington, USA, May 1989. ACM Press.

[55] NIST. FIPS PUB 197: Advanced encryption standard (AES). Technical report, National Institute for Standards and Technology, Gaithersburg, Maryland, USA, November 2001.

[56] NIST. FIPS PUB 180-2: Secure hash standard. Technical report, National Institute for Standards and Technology, Gaithersburg, Maryland, USA, May 2002.

[57] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In Frances M. Titsworth, editor, *IEEE Symposium on Security and Privacy*, pages 184 – 200, Oakland, California, USA, May 2001. IEEE Computer Society Press.

[58] Josef Pieprzyk and Babak Sadeghiyan. *Design of Hashing Algorithms*. Number 756 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[59] Bart Preneel. *Analysis and Design of Cryptographic Hash Functions*. Thesis (Ph.D.), Katholieke Universiteit Leuven, Leuven, Belgium, January 1993.

[60] Bart Preneel. The state of cryptographic hash functions. In Ivan Damgård, editor, *Lectures on Data Security, Modern Cryptology in Theory and Practice*, volume 1561 of *Lecture Notes in Computer Science*, pages 158–182, Aarhus, Denmark, July 1999. Springer.

[61] Bart Preneel. Hash functions. K.U. Leuven, Version 2.b – `http://homes.esat.kuleuven.be/~preneel/`, January 2004.

[62] Bart Preneel. E-Mail correspondence with Bart Preneel, December 2005.

[63] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: A synthetic approach. In Douglas R. Stinson, editor, *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference*, volume 435, pages 154–163, Santa Barbara, California, USA, August 1994. Springer-Verlag.

[64] Jean-Jacques Quisquater and Marc Girault. $2n$-bit hash-functions using n-bit symmetric block cipher algorithms. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques*, volume 434 of *Lecture Notes in Computer Science*, pages 102–109, Houthalen, Belgium, April 1989. Springer.

[65] Vincent Rijmen and Paulo Barreto. Iso/iec 10118-3:2004 – part 3: Dedicated hash-functions. Technical report, International Organization for Standardization, 2004.

[66] Ron Rivest, Adi Shamir, Bart Preneel, Antoine Joux, and Niels Ferguson. Sha-256 today and maybe something else in a few years: Effects on research and design. Panel Discussion, August 2006.

[67] Ronald Rivest. The MD4 message-digest algorithm. Request For Comments 1320, Internet Activities Board, Internet Pricacy Task Force, April 1992.

[68] Ronald Rivest. The MD5 message-digest algorithm. Request For Comments 1321, Internet Activities Board, Internet Pricacy Task Force, April 1992.

[69] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004*, Lecture Notes in Computer Science, pages 371–388, Delhi, India, February 2004. Springer.

[70] Thomas Shrimpton. E-Mail correspondence, February 2006.

[71] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 2nd edition, 1998.

[72] Douglas Stinson. *Cryptography Theory and Practice*. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, Boca Raton, Florida, USA, third edition, 2006.

[73] Samuel S. Wagstaff, Jr. *Cryptanalysis of Number Theoretic Ciphers*. Chapman & Hall/CRC, 2002.

[74] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference*, Lecture Notes in Computer Science, pages 17–36, Santa Barbara, California, USA, August 2005. Springer.

[75] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Lecture Notes in Computer Science, pages 19–35, Aarhus, Denmark, May 2005. Springer.

[76] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *Journal of Computer System Sciences*, 22(3):265 – 279, 1981.

[77] Gideon Yuval. How to swindle Rabin. *Cryptologia*, 3:187 – 189, 1979.

[78] Yuliang Zheng, Josef Pieprzyk, and Jennifer Seberry. HAVAL – A one-way hashing algorithm with variable length of output. In Jennifer Seberry and Yuliang Zheng, editors, *Advances in Cryptology - ASIACRYPT '92, Workshop on the Theory and Application of Cryptographic Techniques*, Lecture Notes in Computer Science, pages 83–104, Gold Coast, Queensland, Australia, December 1992. Springer.

APPENDICES

A ADDITIONAL EXPERIMENTS FOR DYNAMIC HASH FUNCTION SECURITY PROPERTIES

Stronger Versions of the Traditional Properties

Experiment $\mathbf{Exp}_H^{\text{sSec}}(A)$	Experiment $\mathbf{Exp}_H^{\text{sCol}}(A)$

$K \xleftarrow{\$} \mathcal{K}$
$M_1 \xleftarrow{\$} D$
$(M_2, s) \xleftarrow{\$} A(K, M_1)$
if$(H_K(M_1, s) = H_K(M_2, s)$ **and** $M_1 \neq M_2)$
  return 1
else
  return 0
	$K \xleftarrow{\$} \mathcal{K}$
	$(M_1, M_2, s) \xleftarrow{\$} A(K)$
	if$(H_K(M_1, s) = H_K(M_2, s)$ **and** $M_1 \neq M_2)$
	  return 1
	else
	  return 0

Stronger Versions of the Dynamic Properties

Experiment $\mathbf{Exp}_H^{\text{sdSec}}(A)$	Experiment $\mathbf{Exp}_H^{\text{sdCol}}(A)$

$K \xleftarrow{\$} \mathcal{K}$
$M_1 \xleftarrow{\$} D$
$(M_2, s_1, s_2) \xleftarrow{\$} A(K, M_1)$
if$(H_K(M_1, s_1) = H_K(M_2, s_2)$
and $M_1 \neq M_2$ **and** $d(s_1) = d(s_2))$
  return 1
else
  return 0
	$K \xleftarrow{\$} \mathcal{K}$
	$(M_1, M_2, s_1, s_2) \xleftarrow{\$} A(K)$
	if$(H_K(M_1, s_1) = H_K(M_2, s_2)$
	and $M_1 \neq M_2$ **and** $d(s_1) = d(s_2))$
	  return 1
	else
	  return 0

Weaker Versions of the Dynamic Properties

Experiment $\mathbf{Exp}_H^{\text{wdPre}}(A)$	Experiment $\mathbf{Exp}_H^{\text{wdSec}}(A)$
$K \xleftarrow{\$} \mathcal{K}$	$K \xleftarrow{\$} \mathcal{K}$
$M_1 \xleftarrow{\$} D$	$M_1 \xleftarrow{\$} D$
$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$	$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$s_2 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$	$s_2 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$Y \leftarrow H_K(M_1, s_1)$	$(M_2) \xleftarrow{\$} A(K, M_1, s_1, s_2)$
$(M_2) \xleftarrow{\$} A(K, Y, s_1, s_2)$	**if**$(H_K(M_1, s_1) = H_K(M_2, s_2)$
if$(Y = H_K(M_2, s_2)$ **and** $d(s_1) = d(s_2))$	**and** $M_1 \neq M_2$ **and** $d(s_1) = d(s_2))$
return 1	**return** 1
else	**else**
return 0	**return** 0

Experiment $\mathbf{Exp}_H^{\text{wdCol}}(A)$
$K \xleftarrow{\$} \mathcal{K}$
$l \xleftarrow{\$} \mathbb{N}$
$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$s_2 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$(M_1, M_2) \xleftarrow{\$} A(K, s_1, s_2)$
if$(H_K(M_1, s_1) = H_K(M_2, s_2)$
and $M_1 \neq M_2$ **and** $d(s_1) = d(s_2))$
return 1
else
return 0

Weak and Strong Versions of Security Parameter Collision Resistance

Experiment $\mathbf{Exp}_H^{\text{wPCol}}(A)$	Experiment $\mathbf{Exp}_H^{\text{sPCol}}(A)$
$K \xleftarrow{\$} \mathcal{K}$	$K \xleftarrow{\$} \mathcal{K}$
$l \xleftarrow{\$} \mathbb{N}$	
$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$	$(M, s_1, s_2) \xleftarrow{\$} A(K)$
$s_2 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$	**if**$(H_K(M, s_1) = H_K(M, s_2)$
$(M) \xleftarrow{\$} A(K, s_1, s_2)$	**and** $d(s_1) = d(s_2))$
if$(H_K(M, s_1) = H_K(M, s_2)$	**return** 1
and $d(s_1) = d(s_2))$	**else**
return 1	**return** 0
else	
return 0	

Weak Version of Digest Resistance

Experiment $\mathbf{Exp}_H^{\text{wDig}}(A)$
$K \xleftarrow{\$} \mathcal{K}$
$M \xleftarrow{\$} \mathcal{M}$
$s_1 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$s_2 \xleftarrow{\$} [\lambda(l), \upsilon(l)]$
$Y_1 \leftarrow H_K(M, s_1)$
$(Y_2) \xleftarrow{\$} A(K, Y_1, s_1, s_2)$
if$(Y_2 = H_K(M, s_2)$ **and** $s_1 \neq s_2)$
return 1
else
return 0

B BIRTHDAY ATTACKS

The birthday attack derives its name from the birthday paradox. The birthday paradox is often used in introductory probability courses to demonstrate that probability results are sometimes unexpected [71]. The problem asks the following question: *How many people are needed in a group to have two members of that group have the same birthday (meaning day and month) with a probability greater than 50%?* The relation between this question and hash functions is not obvious.

The digest of a hash function can be thought of as a birthday. The messages provided as input to the hash function can be thought of as the people. The question can then be rephrased as follows: *How many messages must be hashed before two messages are found that have the same digest with a probability great than 50%?* This question asks how many messages must be hashed before collision resistance (describe in Section 2.4.3) is broken.

This line of questioning can be extended to two other problems. The first question asks, how many messages must be hashed before a message is found that results in some target digest. This is the notion of preimage resistance described in Section 2.4.1. The second question extends the original birthday paradox to an arbitrary number of collisions, or people with the same birthday. This is the notion of k-collisions described in Section 4.2.3.

B.1 Preimage

Theorem B.1.1 *For an ideal hash function H that produces an n-bit digest, the expected number of messages hashed before finding a preimage is 2^n.*

Proof The proof is straightforward. There are 2^n possible digests for an n-bit hash function. Selecting messages randomly, there is a 2^{-n} probability of selecting a

message that hashes to a specified digest. Therefore, the expected number of messages that must be hashed before a message is found that hashes to a given digest is 2^n. ■

B.2 Collision

Theorem B.2.1 *For an ideal hash function H that produces an n-bit digest, the expected number of messages hashed before finding two messages that collide is $2^{n/2}$.*

The proof for this theorem is taken, in modified form, from [73].

Proof The probability of k messages being independently chosen from 2^n possible messages and having two that result in the same digest is considered first. Let $P(2^n, k)$ denote this probability.

$$P(2^n, k) = 1 - \frac{2^n(2^n-1)\cdots(2^n-k+1)}{2^{nk}} \tag{B.1}$$

$$= 1 - \left(1 - \frac{1}{2^n}\right)\left(1 - \frac{2}{2^n}\right) \times \cdots \times \left(1 - \frac{k-1}{2^n}\right) \tag{B.2}$$

To find the value of k such that $P(2^n, k)$ is greater than 50%, the fact that $1-x \approx e^{-x}$ is applied to each factor in (B.2).

$$P(2^n, k) \approx 1 - (e^{-1/2^n})(e^{-2/2^n})(e^{-3/2^n}) \times \cdots \times (e^{-(k-1)/2^n}) \tag{B.3}$$

$$= 1 - e^{-(1/2^n+2/2^n+3/2^n+\cdots+(k-1)/2^n)} \tag{B.4}$$

$$= 1 - e^{-k(k-1)/2^{(n+1)}} \tag{B.5}$$

Now that the probability is in an easier form, the probability is set to 1/2 and k is solved for. Another approximation is used to simplify the computations, $k(k-1) \approx k^2$ for large values of k.

$$\begin{aligned}
\frac{1}{2} &= P(2^n, k) && \text{(B.6)}\\
&= 1 - e^{-k(k-1)/2^{(n+1)}} && \text{(B.7)}\\
&= e^{-k^2/2^{(n+1)}} && \text{(B.8)}\\
2 &= e^{k^2/2^{(n+1)}} && \text{(B.9)}\\
\ln 2 &= \frac{k^2}{2^{(n+1)}} && \text{(B.10)}\\
\sqrt{2^{(n+1)}(\ln 2)} &= k && \text{(B.11)}\\
1.18\sqrt{2^n} &\approx k && \text{(B.12)}
\end{aligned}$$

Therefore, to pick messages independently from 2^n possible messages and find two the result in the same digest, an expected $2^{n/2}$ messages must be hashed. ■

B.3 k-Collisions

Finding a k-collision consists of finding k unique messages, $M_1, M_2, \ldots, M_k$, such that the digests of the messages are all the same: $H(M_1) = H(M_2) = \cdots = H(M_k)$. The number of messages that must be hashed before finding k that collide is $2^{n(k-1)/k}$. This is given in Theorem 4.2.2, and is restated below.

Theorem 4.2.2 (k-collisions) *For an ideal hash function H that produces an n-bit digest, an expected $2^{n(k-1/k)}$ messages must be hashed before finding k messages that collide.*

To prove this theorem the following lemma is proved, taken from [53].

Lemma B.3.1 *For a random oracle $g : D \to \Sigma^n$, the birthday attack with complexity q finds a k-way collision with probability $q^k/2^{(k-1)n}$, where D is the domain of g.*

Proof Let $m_1, \ldots, m_q$ be randomly chosen from the domain D, then $g(m_1), \ldots, g(m_q)$ are independent and uniformly distributed over the range R. Thus, for any set $\{i_1, \ldots, i_k\} \subset [1, k]$,

$$\Pr[g(x_{i_1}) = \cdots = g(x_{i_k})] = 1/2^{(r-1)n}.$$

Let $S_1, \ldots, S_j$ be k-subsets of $\{m_1, \ldots, m_q\}$ (which denotes the complete query list of the birthday attack) where $j = \binom{q}{k}$. Let E_i denote the event that S_i is a k-collision set. Thus, the event corresponding to the existence of an k-way collision in the set $\{m_1, \ldots, m_q\}$ is $\bigcup_i E_i$. Hence, the probability that the birthday attack finds a k-collision set is

$$\begin{aligned} \Pr\left[\bigcup_i E_i\right] &\leq \sum_i \Pr[E_i] && \text{(B.13)} \\ &= \frac{\binom{q}{k}}{2^{(k-1)n}} && \text{(B.14)} \\ &= \mathcal{O}\left(\frac{q^k}{2^{(k-1)n}}\right) && \text{(B.15)} \end{aligned}$$

■

By application of Lemma B.3.1, Theorem 4.2.2 is proved as follows.

Proof From Lemma B.3.1, the probability of a k-collision is, $\mathcal{O}\left(\frac{q^k}{2^{(k-1)n}}\right)$. Setting this probability equal to 1 and solving for q gives a lower bound on the expected number of messages that must be hashed before finding a k-collision.

$$\begin{aligned} \frac{q^k}{2^{(k-1)n}} &= 1 && \text{(B.16)} \\ q^k &= 2^{(k-1)n} && \text{(B.17)} \\ q &= \Omega\left(2^{\frac{(k-1)n}{r}}\right) && \text{(B.18)} \end{aligned}$$

■

VITA

VITA

William R. Speirs, II received his Ph.D. degree in Computer Science from Purdue University in 2007 with Samuel S. Wagstaff, Jr. as his adviser. He received his M.S. degree in Computer Science from Purdue University in 2005. He received his B.S. degree in Computer Science and Information Technology from Rensselaer Polytechnic Institute in 2003.

William's interests include all aspects of practical security, specifically cryptographic hash functions, and operating systems. He has worked on an operating system designed for education outside of his academic work at Purdue University. He has worked for numerous companies while pursuing his graduate degrees including Lockheed Martin, Telemus Solutions, Inc., and The Pikewerks Corporation.